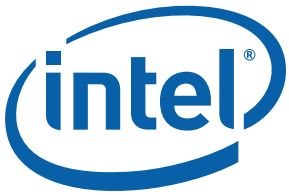


Intel® 64 and IA-32 Architectures Software Developer's Manual

Documentation Changes

January 2011

Notice: The Intel® 64 and IA-32 architectures may contain design defects or errors known as errata that may cause the product to deviate from published specifications. Current characterized errata are documented in the specification updates.

Document Number: 252046-030

Contents

Revision History

Revision	Description	Date
-001	• Initial release	November 2002
-002	• Added 1-10 Documentation Changes. • Removed old Documentation Changes items that already have been incorporated in the published Software Developer's manual	December 2002
-003	• Added 9 -17 Documentation Changes. • Removed Documentation Change #6 - References to bits Gen and Len Deleted. • Removed Documentation Change #4 - VIF Information Added to CLI Discussion	February 2003
-004	• Removed Documentation changes 1-17. • Added Documentation changes 1-24.	June 2003
-005	• Removed Documentation Changes 1-24. • Added Documentation Changes 1-15.	September 2003
-006	• Added Documentation Changes 16- 34.	November 2003
-007	• Updated Documentation changes 14, 16, 17, and 28. • Added Documentation Changes 35-45.	January 2004
-008	• Removed Documentation Changes 1-45. • Added Documentation Changes 1-5.	March 2004
-009	• Added Documentation Changes 7-27.	May 2004
-010	• Removed Documentation Changes 1-27. • Added Documentation Changes 1.	August 2004
-011	• Added Documentation Changes 2-28.	November 2004
-012	• Removed Documentation Changes 1-28. • Added Documentation Changes 1-16.	March 2005
-013	• Updated title. • There are no Documentation Changes for this revision of the document.	July 2005
-014	• Added Documentation Changes 1-21.	September 2005
-015	• Removed Documentation Changes 1-21. • Added Documentation Changes 1-20.	March 9, 2006
-016	• Added Documentation changes 21-23.	March 27, 2006
-017	• Removed Documentation Changes 1-23. • Added Documentation Changes 1-36.	September 2006
-018	• Added Documentation Changes 37-42.	October 2006
-019	• Removed Documentation Changes 1-42. • Added Documentation Changes 1-19.	March 2007
-020	• Added Documentation Changes 20-27.	May 2007
-021	• Removed Documentation Changes 1-27. • Added Documentation Changes 1-6	November 2007
-022	• Removed Documentation Changes 1-6 • Added Documentation Changes 1-6	August 2008
-023	• Removed Documentation Changes 1-6 • Added Documentation Changes 1-21	March 2009

Revision	Description	Date
-024	• Removed Documentation Changes 1-21 • Added Documentation Changes 1-16	June 2009
-025	• Removed Documentation Changes 1-16 • Added Documentation Changes 1-18	September 2009
-026	• Removed Documentation Changes 1-18 • Added Documentation Changes 1-15	December 2009
-027	• Removed Documentation Changes 1-15 • Added Documentation Changes 1-24	March 2010
-028	• Removed Documentation Changes 1-24 • Added Documentation Changes 1-29	June 2010
-029	• Removed Documentation Changes 1-29 • Added Documentation Changes 1-29	September 2010
-030	• Removed Documentation Changes 1-29 • Added Documentation Changes 1-29	January 2011

§

Preface

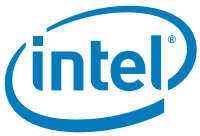
This document is an update to the specifications contained in the Affected Documents table below. This document is a compilation of device and documentation errata, specification clarifications and changes. It is intended for hardware system manufacturers and software developers of applications, operating systems, or tools.

Affected Documents

Document Title	Document Number/Location
Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1: Basic Architecture	253665
Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A: Instruction Set Reference, A-M	253666
Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2B: Instruction Set Reference, N-Z	253667
Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A: System Programming Guide, Part 1	253668
Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3B: System Programming Guide, Part 2	253669

Nomenclature

Documentation Changes include typos, errors, or omissions from the current published specifications. These will be incorporated in any new release of the specification.



Summary Tables of Changes

The following table indicates documentation changes which apply to the Intel® 64 and IA-32 architectures. This table uses the following notations:

Codes Used in Summary Tables

Change bar to left of table row indicates this erratum is either new or modified from the previous version of the document.

Documentation Changes (Sheet 1 of 2)

No.	DOCUMENTATION CHANGES
1	Updates to Chapter 5, Volume 1
2	Updates to Chapter 11, Volume 1
3	New Chapter 13, Volume 1
4	Updates to Chapter 2, Volume 2A
5	Updates to Chapter 3, Volume 2A
6	Updates to Chapter 4, Volume 2B
7	Updates to Chapter 5, Volume 2B
8	Updates to Chapter 6, Volume 2B
9	Updates to Appendix A, Volume 2B
10	Updates to Chapter 2, Volume 3A
11	Updates to Chapter 4, Volume 3A
12	Updates to Chapter 7, Volume 3A
13	Updates to Chapter 8, Volume 3A
14	Updates to Chapter 10, Volume 3A
15	Updates to Chapter 11, Volume 3A
16	Updates to Chapter 13, Volume 3A
17	Updates to Chapter 14, Volume 3A
18	Updates to Chapter 15, Volume 3A
19	Updates to Chapter 16, Volume 3A
20	Updates to Chapter 21, Volume 3B
21	Updates to Chapter 22, Volume 3B
22	Updates to Chapter 23, Volume 3B
23	Updates to Chapter 24, Volume 3B
24	Updates to Chapter 26, Volume 3B
25	Updates to Chapter 28, Volume 3B
26	Updates to Chapter 30, Volume 3B

Documentation Changes (Sheet 2 of 2)

No.	DOCUMENTATION CHANGES
27	Updates to Appendix A, Volume 3B
28	Updates to Appendix B, Volume 3B
29	Updates to Appendix G, Volume 3B

Documentation Changes

1. Updates to Chapter 5, Volume 1

Change bars show changes to Chapter 5 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1:* Basic Architecture.

...

Table 5-1 Instruction Groups and IA-32 Processors

Instruction Set Architecture	Intel 64 and IA-32 Processor Support
General Purpose	All Intel 64 and IA-32 processors
x87 FPU	Intel486, Pentium, Pentium with MMX Technology, Celeron, Pentium Pro, Pentium II, Pentium II Xeon, Pentium III, Pentium III Xeon, Pentium 4, Intel Xeon processors, Pentium M, Intel Core Solo, Intel Core Duo, Intel Core 2 Duo processors, Intel Atom processors
x87 FPU and SIMD State Management	Pentium II, Pentium II Xeon, Pentium III, Pentium III Xeon, Pentium 4, Intel Xeon processors, Pentium M, Intel Core Solo, Intel Core Duo, Intel Core 2 Duo processors, Intel Atom processors
MMX Technology	Pentium with MMX Technology, Celeron, Pentium II, Pentium II Xeon, Pentium III, Pentium III Xeon, Pentium 4, Intel Xeon processors, Pentium M, Intel Core Solo, Intel Core Duo, Intel Core 2 Duo processors, Intel Atom processors
SSE Extensions	Pentium III, Pentium III Xeon, Pentium 4, Intel Xeon processors, Pentium M, Intel Core Solo, Intel Core Duo, Intel Core 2 Duo processors, Intel Atom processors
SSE2 Extensions	Pentium 4, Intel Xeon processors, Pentium M, Intel Core Solo, Intel Core Duo, Intel Core 2 Duo processors, Intel Atom processors
SSE3 Extensions	Pentium 4 supporting HT Technology (built on 90nm process technology), Intel Core Solo, Intel Core Duo, Intel Core 2 Duo processors, Intel Xeon processor 3xxxx, 5xxx, 7xxx Series, Intel Atom processors
SSSE3 Extensions	Intel Xeon processor 3000, 3200, 5100, 5300, 7300 series, Intel Core 2 Extreme processors QX6000 series, Intel Core 2 Duo, Intel Core 2 Quad processors, Intel Pentium Dual-Core processors, Intel Atom processors
SSE4.1 Extensions	Intel Xeon processor 3100, 3300, 5200, 5400, 7400 series, Intel Core 2 Extreme processors QX9000 series, Intel Core 2 Quad processor Q9000 series, Intel Core 2 Duo processors 8000 series, T9000 series.
SSE4.2 Extensions	Intel Core i7 965 processor, Intel Xeon processors 3400, 5500, 7500 series.
AESNI, PCLMULQDQ	Intel Core i7 980X processor, Intel Xeon processors 5600.
Intel AVX	Intel Core i7 2600 processor.
IA-32e mode: 64-bit mode instructions	Intel 64 processors

Table 5-1 Instruction Groups and IA-32 Processors (Continued)

Instruction Set Architecture	Intel 64 and IA-32 Processor Support
System Instructions	Intel 64 and IA-32 processors
VMX Instructions	Intel 64 and IA-32 processors supporting Intel Virtualization Technology
SMX Instructions	Intel Core 2 Duo processor E6x50, E8xxx; Intel Core 2 Quad processor Q9xxx

...

5.12 AESNI AND PCLMULQDQ

Six AESNI instructions operate on XMM registers to provide accelerated primitives for block encryption/decryption using Advanced Encryption Standard (FIPS-197). PCLMULQDQ instruction perform carry-less multiplication for two binary numbers up to 64-bit wide.

AESDEC	Perform an AES decryption round using an 128-bit state and a round key
AESDECLAST	Perform the last AES decryption round using an 128-bit state and a round key
AESENC	Perform an AES encryption round using an 128-bit state and a round key
AESENCLAST	Perform the last AES encryption round using an 128-bit state and a round key
AESIMC	Perform an inverse mix column transformation primitive
AESKEYGENASSIST	Assist the creation of round keys with a key expansion schedule
PCLMULQDQ	Perform carryless multiplication of two 64-bit numbers

5.13 INTEL® ADVANCED VECTOR EXTENSIONS (AVX)

Intel® Advanced Vector Extensions (AVX) promotes legacy 128-bit SIMD instruction sets that operate on XMM register set to use a "vector extension" (VEX) prefix and operates on 256-bit vector registers (YMM). Almost all prior generations of 128-bit SIMD instructions that operates on XMM (but not on MMX registers) are promoted to support three-operand syntax with VEX-128 encoding.

VEX-prefix encoded AVX instructions support 256-bit and 128-bit floating-point operations by extending the legacy 128-bit SIMD floating-point instructions to support three-operand syntax.

Additional functional enhancements are also provided with VEX-encoded AVX instructions.

The list of AVX instructions are listed in the following tables:

- Table 13-2 lists 256-bit and 128-bit floating-point arithmetic instructions promoted from legacy 128-bit SIMD instruction sets.

- Table 13-3 lists 256-bit and 128-bit data movement and processing instructions promoted from legacy 128-bit SIMD instruction sets.
- Table 13-4 lists functional enhancements of 256-bit AVX instructions not available from legacy 128-bit SIMD instruction sets.
- Table 13-5 lists 128-bit integer and floating-point instructions promoted from legacy 128-bit SIMD instruction sets.
- Table 13-6 lists functional enhancements of 128-bit AVX instructions not available from legacy 128-bit SIMD instruction sets.
- Table 13-7 lists 128-bit data movement and processing instructions promoted from legacy instruction sets.

5.14 SYSTEM INSTRUCTIONS

The following system instructions are used to control those functions of the processor that are provided to support for operating systems and executives.

LGDT	Load global descriptor table (GDT) register
SGDT	Store global descriptor table (GDT) register
LLDT	Load local descriptor table (LDT) register
SLDT	Store local descriptor table (LDT) register
LTR	Load task register
STR	Store task register
LIDT	Load interrupt descriptor table (IDT) register
SIDT	Store interrupt descriptor table (IDT) register
MOV	Load and store control registers
LMSW	Load machine status word
SMSW	Store machine status word
CLTS	Clear the task-switched flag
ARPL	Adjust requested privilege level
LAR	Load access rights
LSL	Load segment limit
VERR	Verify segment for reading
VERW	Verify segment for writing
MOV	Load and store debug registers
INVD	Invalidate cache, no writeback
WBINVD	Invalidate cache, with writeback
INVLPG	Invalidate TLB Entry
LOCK (prefix)	Lock Bus
HLT	Halt processor
RSM	Return from system management mode (SMM)
RDMSR	Read model-specific register
WRMSR	Write model-specific register
RDPMC	Read performance monitoring counters
RDTSC	Read time stamp counter
RDTSCP	Read time stamp counter and processor ID

SYSENTER	Fast System Call, transfers to a flat protected mode kernel at CPL = 0
SYSEXIT	Fast System Call, transfers to a flat protected mode kernel at CPL = 3
XSAVE	Save processor extended states to memory
XSAVEOPT	Save processor extended states to memory, optimized
XRSTOR	Restore processor extended states from memory
XGETBV	Reads the state of an extended control register
XSETBV	Writes the state of an extended control register

...

5.17 SAFER MODE EXTENSIONS

The behavior of the GETSEC instruction leaves of the Safer Mode Extensions (SMX) are summarized below:

GETSEC[CAPABILITIES]Returns the available leaf functions of the GETSEC instruction.

GETSEC[ENTERACCS] Loads an authenticated code chipset module and enters authenticated code execution mode.

GETSEC[EXITAC] Exits authenticated code execution mode.

GETSEC[SENTER] Establishes a Measured Launched Environment (MLE) which has its dynamic root of trust anchored to a chipset supporting Intel Trusted Execution Technology.

GETSEC[SEXIT] Exits the MLE.

GETSEC[PARAMETERS]Returns SMX related parameter information.

GETSEC[SMCRTL] SMX mode control.

GETSEC[WAKEUP] Wakes up sleeping logical processors inside an MLE.

...

2. Updates to Chapter 11, Volume 1

Change bars show changes to Chapter 11 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1:* Basic Architecture.

...

11.6.10.3 Caller-Save Recommendation for Procedure and Function Calls

When making procedure (or function) calls from SSE or SSE2 code, a caller-save convention is recommended for saving the state of the calling procedure. Using this convention, any register whose content must survive intact across a procedure call must be stored in memory by the calling procedure prior to executing the call.

The primary reason for using the caller-save convention is to prevent performance degradation. XMM registers can contain packed or scalar double-precision floating-point, packed single-precision floating-point, and 128-bit packed integer data types. The called procedure has no way of knowing the data types in XMM registers following a call; so it

is unlikely to use the correctly typed move instruction to store the contents of XMM registers in memory or to restore the contents of XMM registers from memory.

As described in Section 11.6.9, "Mixing Packed and Scalar Floating-Point and 128-Bit SIMD Integer Instructions and Data," executing a move instruction that does not match the type for the data being moved to/from XMM registers will be carried out correctly, but can lead to a greater instruction latency.

...

3. New Chapter 13, Volume 1

Change bars show the addition of Chapter 13 to the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1:* Basic Architecture.

...

CHAPTER 13
PROGRAMMING WITH AVX

Intel® Advanced Vector Extensions (AVX) introduces 256-bit vector processing capability. The Intel AVX instruction set extends 128-bit SIMD instruction sets by employing a new instruction encoding scheme via a vector extension prefix (VEX). Intel AVX also offers several enhanced features beyond those available in prior generations of 128-bit SIMD extensions. This chapter summarizes the key features of Intel AVX.

13.1 INTEL AVX OVERVIEW

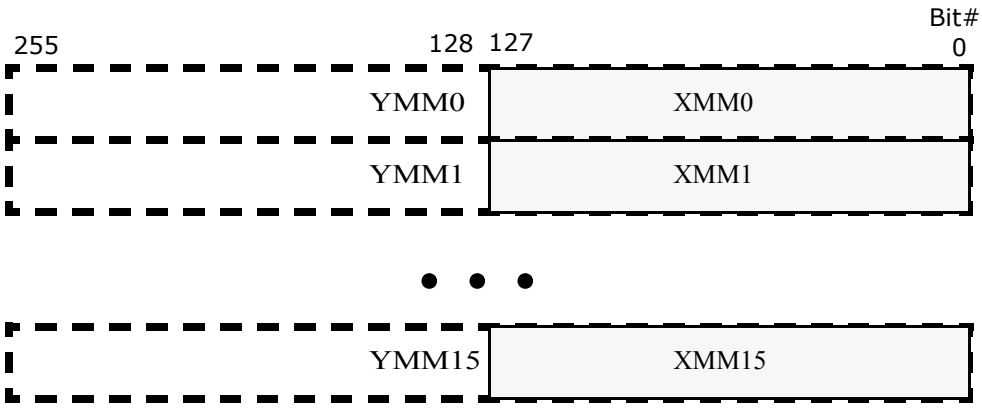
Intel AVX introduces the following architectural enhancements:

- Support for 256-bit wide vectors with the YMM vector register set.
- 256-bit floating-point instruction set enhancement with up to 2X performance gain relative to 128-bit Streaming SIMD extensions.
- Enhancement of legacy 128-bit SIMD instruction extensions to support three-operand syntax and to simplify compiler vectorization of high-level language expressions.
- VEX prefix-encoded instruction syntax support for generalized three-operand syntax to improve instruction programming flexibility and efficient encoding of new instruction extensions.
- Most VEX-encoded 128-bit and 256-bit AVX instructions (with both load and computational operation semantics) are not restricted to 16-byte or 32-byte memory alignment.
- Support flexible deployment of 256-bit AVX code, 128-bit AVX code, legacy 128-bit code and scalar code.

With the exception of SIMD instructions operating on MMX registers, almost all legacy 128-bit SIMD instructions have AVX equivalents that support three operand syntax. 256-bit AVX instructions employ three-operand syntax and some with 4-operand syntax.

13.1.1 256-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.



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 beyond bit 128 of the YMM registers. AVX instructions with a VEX prefix and vector length of 128-bits zeroes the upper bits (above bit 128) of the YMM register.

13.1.2 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 in legacy SSE instruction has

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

128-bit AVX equivalent 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.

13.1.3 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 13.3).

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 2, "Instruction Format," of *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A*.

13.2 FUNCTIONAL OVERVIEW

Intel AVX 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.
- 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.

Table 13-1 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 VPERMILPS
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 VPERMILPS
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
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	

VEX.256 Encoding	VEX.128 Encoding	Group	Instruction	If No, Reason?
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	
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	

VEX.256 Encoding	VEX.128 Encoding	Group	Instruction	If No, Reason?
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	
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

VEX.256 Encoding	VEX.128 Encoding	Group	Instruction	If No, Reason?
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
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

VEX.256 Encoding	VEX.128 Encoding	Group	Instruction	If No, Reason?
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
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	

VEX.256 Encoding	VEX.128 Encoding	Group	Instruction	If No, Reason?
no	yes		MPSADBW	VI
no	yes		PACKUSDW	VI
no	yes		PBLENDVB	VI
no	yes		PBLENDW	VI
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

13.2.1 256-bit Floating-Point Arithmetic Processing Enhancements

Intel AVX provides 35 256-bit floating-point arithmetic instructions, see Table 13-2. 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.

Table 13-2 Promoted 256-Bit and 128-bit Arithmetic AVX Instructions

VEX.256 Encoding	VEX.128 Encoding	Legacy Instruction Mnemonic
yes	yes	SQRTPS, SQRTPD, RSQRTPS, RCPPS
yes	yes	ADDPS, ADDPD, SUBPS, SUBPD
yes	yes	MULPS, MULPD, DIVPS, DIVPD
yes	yes	CVTPS2PD, CVTPD2PS
yes	yes	CVTDQ2PS, CVTPS2DQ
yes	yes	CVTTPS2DQ, CVTTPD2DQ
yes	yes	CVTPD2DQ, CVTDQ2PD
yes	yes	MINPS, MINPD, MAXPS, MAXPD
yes	yes	HADDPD, HADDPS, HSUBPD, HSUBPS
yes	yes	CMPPS, CMPPD
yes	yes	ADDSUBPD, ADDSUBPS, DPPS
yes	yes	ROUNDPD, ROUNDPS

13.2.2 256-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 39 256-bit data movement and processing instructions that are promoted from previous generations of SIMD instruction extensions, ranging from logical, blend, convert, test, unpacking, shuffling, load and stores (see Table 13-3).

Table 13-3 Promoted 256-bit and 128-bit Data Movement AVX Instructions

VEX.256 Encoding	VEX.128 Encoding	Legacy Instruction Mnemonic
yes	yes	MOVAPS, MOVAPD, MOVDQA
yes	yes	MOVUPS, MOVUPD, MOVDQU
yes	yes	MOVMSKPS, MOVMSKPD
yes	yes	LDDQU, MOVNTPS, MOVNTPD, MOVNTDQ, MOVNTDQA
yes	yes	MOVSHDUP, MOVSLDUP, MOVDDUP
yes	yes	UNPCKHPD, UNPCKHPS, UNPCKLPD
yes	yes	BLENDPS, BLENDPD

Table 13-3 Promoted 256-bit and 128-bit Data Movement AVX Instructions

VEX.256 Encoding	VEX.128 Encoding	Legacy Instruction Mnemonic
yes	yes	SHUFPD, SHUFPS, UNPCKLPS
yes	yes	BLENDVPS, BLENDVPD
yes	yes	PTEST, MOVMSKPD, MOVMSKPS
yes	yes	XORPS, XORPD, ORPS, ORPD
yes	yes	ANDNPD, ANDNPS, ANDPD, ANDPS

AVX introduces 18 new data processing instructions that operate on 256-bit vectors, Table 13-4. 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.

Table 13-4 256-bit AVX Instruction Enhancement

Instruction	Description
VBROADCASTF128 ymm1, m128	Broadcast 128-bit floating-point values in mem to low and high 128-bits in ymm1.
VBROADCASTSD ymm1, m64	Broadcast double-precision floating-point element in mem to four locations in ymm1.
VBROADCASTSS ymm1, m32	Broadcast single-precision floating-point element in mem to eight locations in ymm1.
VEXTRACTF128 xmm1/m128, ymm2, imm8	Extracts 128-bits of packed floating-point values from ymm2 and store results in xmm1/mem.
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
VMASKMOVPS ymm1, ymm2, m256	Load packed single-precision values from mem using mask in ymm2 and store in ymm1

Table 13-4 256-bit AVX Instruction Enhancement

Instruction	Description
VMASKMOVPD ymm1, ymm2, m256	Load packed double-precision values from mem using mask in ymm2 and store in ymm1
VMASKMOVPS m256, ymm1, ymm2	Store packed single-precision values from ymm2 mask in ymm1
VMASKMOVPD m256, ymm1, ymm2	Store packed double-precision values from ymm2 using mask in ymm1
VPERMILPD ymm1, ymm2, ymm3/m256	Permute Double-Precision Floating-Point values in ymm2 using controls from xmm3/mem and store result in ymm1
VPERMILPD ymm1, ymm2/m256 imm8	Permute Double-Precision Floating-Point values in ymm2/mem using controls from imm8 and store result in ymm1
VPERMILPS ymm1, ymm2, ymm/m256	Permute Single-Precision Floating-Point values in ymm2 using controls from ymm3/mem and store result in ymm1
VPERMILPS ymm1, ymm2/m256, imm8	Permute Single-Precision Floating-Point values in ymm2/mem using controls from imm8 and store result in ymm1
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
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.
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.
VZEROALL	Zero all YMM registers
VZEROUPPER	Zero upper 128 bits of all YMM registers

13.2.3 Arithmetic Primitives for 128-bit Vector and Scalar processing

Intel AVX provides a full compliment of 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. Table 13-5 lists the state of promotion of legacy SIMD arithmetic ISA to VEX-128 encoding. Legacy SIMD floating-point arithmetic ISA promoted to VEX-256 encoding also support VEX-128 encoding (see Table 13-2).

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.

Table 13-5 Promotion of Legacy SIMD ISA to 128-bit Arithmetic AVX instruction

VEX.256 Encoding	VEX.128 Encoding	Instruction	Reason Not Promoted
no	no	CVTPI2PS, CVTPI2PD, CVTPD2PI	MMX
no	no	CVTTPS2PI, CVTTPD2PI, CVTPS2PI	MMX
no	yes	CVTSI2SS, CVTSI2SD, CVTSD2SI	scalar
no	yes	CVTTSS2SI, CVTTSD2SI, CVTSS2SI	scalar
no	yes	COMISD, RSQRTSS, RCPSS	scalar
no	yes	UCOMISS, UCOMISD, COMISS,	scalar
no	yes	ADDSS, ADDSD, SUBSS, SUBSD	scalar
no	yes	MULSS, MULSD, DIVSS, DIVSD	scalar
no	yes	SQRTSS, SQRTSD	scalar
no	yes	CVTSS2SD, CVTSD2SS	scalar
no	yes	MINSS, MINSD, MAXSS, MAXSD	scalar
no	yes	PAND, PANDN, POR, PXOR	VI
no	yes	PCMPGTB, PCMPGTW, PCMPGTD	VI
no	yes	PMADDWD, PMADDUBSW	VI
no	yes	PAVGB, PAVGW, PMULUDQ	VI
no	yes	PCMPEQB, PCMPEQW, PCMPEQD	VI
no	yes	PMULLW, PMULHUW, PMULHW	VI
no	yes	PSUBSW, PADDSW, PSADBW	VI
no	yes	PADDUSB, PADDUSW, PADDSB	VI
no	yes	PSUBUSB, PSUBUSW, PSUBSB	VI
no	yes	PMINUB, PMINSW	VI
no	yes	PMAXUB, PMAXSW	VI
no	yes	PADDB, PADDW, PADDD, PADDQ	VI
no	yes	PSUBB, PSUBW, PSUBD, PSUBQ	VI
no	yes	PSLLW, PSLLD, PSLLQ, PSRAW	VI
no	yes	PSRLW, PSRLD, PSRLQ, PSRAD	VI
CPUID.SSSE3			
no	yes	PHSUBW, PHSUBD, PHSUBSW	VI
no	yes	PHADDW, PHADDD, PHADDSW	VI
no	yes	PMULHRSW	VI
no	yes	PSIGNB, PSIGNW, PSIGND	VI
no	yes	PABSB, PABSW, PABSD	VI
CPUID.SSE4_1			
no	yes	DPPD	

Table 13-5 Promotion of Legacy SIMD ISA to 128-bit Arithmetic AVX instruction

VEX.256 Encoding	VEX.128 Encoding	Instruction	Reason Not Promoted
no	yes	PHMINPOSUW, MPSADBW	VI
no	yes	PMAXSB, PMAXSD, PMAXUD	VI
no	yes	PMINSB, PMINSD, PMINUD	VI
no	yes	PMAXUW, PMINUW	VI
no	yes	PMOVSXxx, PMOVZXxx	VI
no	yes	PMULDQ, PMULLD	VI
no	yes	ROUNDSD, ROUNDSS	scalar
CPUID.POPCNT			
no	yes	POPCNT	integer
CPUID.SSE4_2			
no	yes	PCMPGTQ	VI
no	no	CRC32	integer
no	yes	PCMPESTRI, PCMPESTRM	VI
no	yes	PCMPISTRI, PCMPISTRM	VI
CPUID.CLMUL			
no	yes	PCLMULQDQ	VI
CPUID.AESNI			
no	yes	AESDEC, AESDECLAST	VI
no	yes	AESENC, AESENCLAST	VI
no	yes	AESIMX, AESKEYGENASSIST	VI

Description of Column "Reason not promoted?"

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

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

Intel AVX provides a full compliment of 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.

A subset of new functionalities listed in Table 13-4 is also extended via VEX.128 encoding. These enhancements in AVX on 128-bit data processing primitives include 11 new instructions (see Table 13-6) 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.

Table 13-6 128-bit AVX Instruction Enhancement

Instruction	Description
VBROADCASTSS xmm1, m32	Broadcast single-precision floating-point element in mem to four locations in xmm1.
VMASKMOVPS xmm1, xmm2, m128	Load packed single-precision values from mem using mask in xmm2 and store in xmm1
VMASKMOVPD xmm1, xmm2, m128	Load packed double-precision values from mem using mask in xmm2 and store in xmm1
VMASKMOVPS m128, xmm1, xmm2	Store packed single-precision values from xmm2 using mask in xmm1
VMASKMOVPD m128, xmm1, xmm2	Store packed double-precision values from xmm2 using mask in xmm1
VPERMILPD xmm1, xmm2, xmm3/m128	Permute Double-Precision Floating-Point values in xmm2 using controls from xmm3/mem and store result in xmm1
VPERMILPD xmm1, xmm2/m128, imm8	Permute Double-Precision Floating-Point values in xmm2/mem using controls from imm8 and store result in xmm1
VPERMILPS xmm1, xmm2, xmm3/m128	Permute Single-Precision Floating-Point values in xmm2 using controls from xmm3/mem and store result in xmm1
VPERMILPS xmm1, xmm2/m128, imm8	Permute Single-Precision Floating-Point values in xmm2/mem using controls from imm8 and store result in xmm1
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.
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.

The 128-bit data processing instructions in AVX cover floating-point and integer data movement primitives. Legacy SIMD non-arithmetic ISA promoted to VEX-256 encoding also support VEX-128 encoding (see Table 13-3). Table 13-7 lists the state of promotion of the remaining legacy SIMD non-arithmetic ISA to VEX-128 encoding.

Table 13-7 Promotion of Legacy SIMD ISA to 128-bit Non-Arithmetic AVX instruction

VEX.256 Encoding	VEX.128 Encoding	Instruction	Reason Not Promoted
no	no	MOVQ2DQ, MOVDQ2Q	MMX
no	yes	LDMXCSR, STMXCSR	
no	yes	MOVSS, MOVSD, CMPSS, CMPSD	scalar
no	yes	MOVHPS, MOVHPD	Note 1
no	yes	MOVLPS, MOVLPD	Note 1
no	yes	MOVLHPS, MOVHLPS	Redundant with VPER-MILPS
no	yes	MOVQ, MOVD	scalar
no	yes	PACKUSWB, PACKSSDW, PACKSSWB	VI
no	yes	PUNPCKHBW, PUNPCKHWD	VI
no	yes	PUNPCKLBW, PUNPCKLWD	VI
no	yes	PUNPCKHDQ, PUNPCKLDQ	VI
no	yes	PUNPCKLQDQ, PUNPCKHQDQ	VI
no	yes	PSHUFHW, PSHUFLW, PSHUFD	VI
no	yes	PMOVMSKB, MASKMOVDQU	VI
no	yes	PAND, PANDN, POR, PXOR	VI
no	yes	PINSRW, PEXTRW,	VI
CPUID.SSSE3			
no	yes	PALIGNR, PSHUFB	VI
CPUID.SSE4_1			
no	yes	EXTRACTPS, INSERTPS	Note 3
no	yes	PACKUSDW, PCMPEQQ	VI
no	yes	PBLENDVB, PBLENDW	VI
no	yes	PEXTRW, PEXTRB, PEXTRD, PEXTRQ	VI
no	yes	PINSRB, PINSRD, PINSRQ	VI

Description of Column "Reason not promoted?"

MMX: Instructions referencing MMX registers do not support VEX

Scalar: Scalar instructions are not promoted to 256-bit

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 3: It is expected that using 128-bit INSERTPS followed by a VINSERTF128 would be better than promoting INSERTPS to 256-bit (for example).

13.3 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 13-9.

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 instructions can 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 13-8 for details.

Certain AVX instructions always require 16- or 32-byte alignment (see the complete list of such instructions in Table 13-9). 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 13-8 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 13-10)	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 13-9)	#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 13-10)	no fault	no fault
		op XMM, m128	#GP(0)	#GP(0)
		"explicitly aligned" loads and stores (see Table 13-9)	#GP(0)	#GP(0)
		2, 4, or 8-byte loads and stores	no fault	#AC(0)

Table 13-9 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
(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 13-10 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

Table 13-10 Instructions Not Requiring Explicit Memory Alignment

(V)MOVDQU xmm, m128
(V)MOVDQU m128, m128
(V)MOVUPS xmm, m128
(V)MOVUPS m128, xmm
VMOVUPD ymm, m256
VMOVUPD m256, ymm

13.4 SIMD FLOATING-POINT EXCEPTIONS

AVX 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).

The detailed exception conditions for AVX instructions and legacy SIMD instructions (excluding instructions that operates on MMX registers) are described in a number of exception class types, depending on the operand syntax and memory operation characteristics. The complete list of SIMD instruction exception class types are defined in Chapter 2, "Instruction Format," of *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A*.

13.5 DETECTION OF AVX INSTRUCTIONS

Intel AVX operates on the 256-bit YMM register state. Application detection of new instruction extensions operating on the YMM state follows the general procedural flow in Figure 13-1.

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 XCR0[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)

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

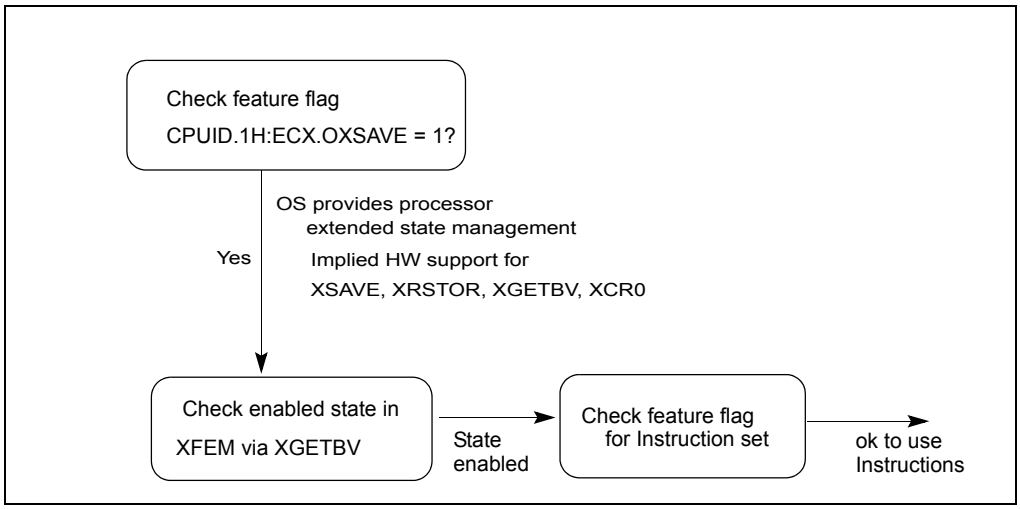


Figure 13-1 General Procedural Flow of Application Detection of AVX

The following pseudocode illustrates this recommended application AVX detection process:

Example 13-1 Detection of AVX Instruction

```
INT supports_AVX()
{    mov       eax, 1
     cpuid
     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 XCR0 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.

13.5.1 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.

Example 13-2 Detection of VEX-Encoded AESNI Instructions

```
INT supports_VAESNI()
{   mov     eax, 1
    cpuid
    and     ecx, 01A000000H
    cmp     ecx, 01A000000H; check OSXSAVE AVX and AESNI feature flags
     jne    not_supported
    ; processor supports AVX and VEX-encoded AESNI and XGETBV is enabled by OS
    mov     ecx, 0; specify 0 for XCR0 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:
```

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:

Example 13-3 Detection of VEX-Encoded AESNI Instructions

```
INT supports_VPCLMULQDQ)
{   mov     eax, 1
    cpuid
    and     ecx, 018000002H
    cmp     ecx, 018000002H; check OSXSAVE AVX and PCLMULQDQ feature flags
     jne    not_supported
    ; processor supports AVX and VEX-encoded PCLMULQDQ and XGETBV is enabled by OS
    mov     ecx, 0; specify 0 for XCR0 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
```

Example 13-3 Detection of VEX-Encoded AESNI Instructions

```
    mov       eax, 1
    jmp       done
NOT_SUPPORTED:
    mov       eax, 0
done:
```

13.6 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.

If an operating system wishes to emulate AVX instructions, set XCR0[2:1] to zero. This will cause AVX instructions to #UD.

13.7 WRITING AVX FLOATING-POINT EXCEPTION HANDLERS

AVX 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)," 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.

...

4. Updates to Chapter 2, Volume 2A

Change bars show changes to Chapter 2 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A:* Instruction Set Reference, A-M.

...

2.3 INTEL® ADVANCED VECTOR EXTENSIONS (INTEL® AVX)

Intel AVX instructions are encoded using an encoding scheme that combines prefix bytes, opcode extension field, operand encoding fields, and vector length encoding capability into a new prefix, referred to as VEX. In the VEX encoding scheme, the VEX prefix may be two or three bytes long, depending on the instruction semantics. Despite the two-byte or three-byte length of the VEX prefix, the VEX encoding format provides a more compact representation/packing of the components of encoding an instruction in Intel 64 architecture. The VEX encoding scheme also allows more headroom for future growth of Intel 64 architecture.

2.3.1 Instruction Format

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 2-8 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.

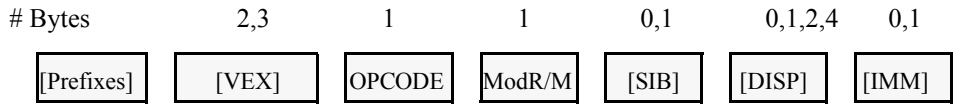


Figure 2-8 Instruction Encoding Format with VEX Prefix

2.3.2 VEX and the LOCK prefix

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

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

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

2.3.4 VEX and the REX prefix

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

2.3.5 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 2-9..

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-encoded 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.

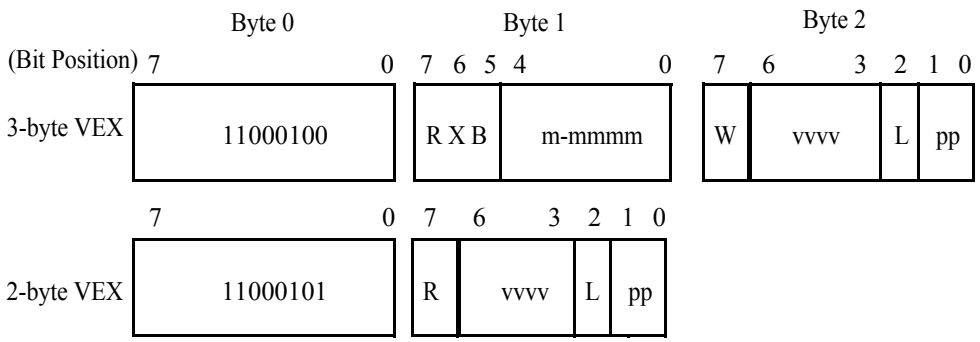
The VEX prefix is required to be the last prefix and immediately precedes the opcode bytes. It must follow any other prefixes. If VEX prefix is present a REX prefix is not supported.

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. Note, certain new instruction functionality can only be encoded with the VEX prefix.

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 2-9. VEX bitfields

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

2.3.5.1 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.

2.3.5.2 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.

2.3.5.3 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.

2.3.5.4 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.

2.3.5.5 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.

2.3.5.6 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 2-8 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 2-8 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.

2.3.6 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 2-9. The notation in the "Opcode" column in Table 2-9 is described in detail in section 3.1.1.

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

Table 2-9 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.

2.3.6.1 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 2-10 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.

2.3.6.2 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 2-11 VEX.L interpretation

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

2.3.6.3 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 2-12 VEX.pp interpretation

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

2.3.7 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.

2.3.8 The MODRM, SIB, and Displacement Bytes

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

2.3.9 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.

2.3.10 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 bits (above bit 128) of the equivalent YMM register . Legacy SSE instructions without VEX preserve the upper bits.

2.3.11 AVX Instruction Length

The AVX 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.

2.4 INSTRUCTION EXCEPTION SPECIFICATION

To look up the exceptions of legacy 128-bit SIMD instruction, 128-bit VEX-encoded instructions, and 256-bit VEX-encoded instruction, Table 2-13 summarizes the exception behavior into separate classes, with detailed exception conditions defined in sub-sections 2.4.1 through 2.4.9. For example, ADDPS contains the entry:

"See Exceptions Type 2"

In this entry, *"Type2"* can be looked up in Table 2-13.

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.

NOTE

Instructions that operate only with MMX, X87, or general-purpose registers are not covered by the exception classes defined in this section.

Table 2-13 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, Legacy SSE	16/32 byte not explicitly aligned	yes
Type 3	AVX, 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

See Table 2-14 for lists of instructions in each exception class.

Table 2-14 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*, (V)HADDPD, (V)HADDPS, (V)HSUBPD, (V)HSUBPS, (V)MAXPD, (V)MAXPS, (V)MINPD, (V)MINPS, (V)MULPD, (V)MULPS, (V)ROUNDPD, (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, (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

(*) - 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 2-14 classifies exception behaviors for AVX instructions. Within each class of exception conditions that are listed in Table 2-17 through Table 2-23, certain subsets of AVX instructions may be subject to #UD exception depending on the encoded value of the VEX.L field. Table 2-16 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 2-15 #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		
Type 9		

Table 2-16 #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

Exception Class	#UD If VEX.L = 0	#UD If VEX.L = 1
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

2.4.1 Exceptions Type 1 (Aligned memory reference)

Table 2-17 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 XCR0[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

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

Table 2-18 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 XCR0[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

2.4.3 Exceptions Type 3 (<16 Byte memory argument)

Table 2-19 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 XCR0[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

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

Table 2-20 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 XCR0[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

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

Table 2-21 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 XCR0[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.

2.4.6 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 2-22 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 XCR0[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.

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

Table 2-23 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 XCR0[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

2.4.8 Exceptions Type 8 (AVX and no memory argument)

Table 2-24 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 XCR0[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.

2.4.9 Exception Type 9 (Intel AVX)

Table 2-25 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.

...

5. Updates to Chapter 3, Volume 2A

Change bars show changes to Chapter 3 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A:* Instruction Set Reference, A-M.

...

This chapter describes the instruction set for the Intel 64 and IA-32 architectures (A-M) in IA-32e, protected, Virtual-8086, and real modes of operation. The set includes general-purpose, x87 FPU, MMX, SSE/SSE2/SSE3/SSSE3/SSE4, AESNI/PCLMULQDQ, AVX, and system instructions. See also Chapter 4, "Instruction Set Reference, N-Z," in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2B*.

...

3.1.1.1 Opcode Column in the Instruction Summary Table (Instructions without VEX prefix)

The "Opcode" column in the table above shows the object code produced for each form of the instruction. When possible, codes are given as hexadecimal bytes in the same order in which they appear in memory. Definitions of entries other than hexadecimal bytes are as follows:

- **REX.W** — Indicates the use of a REX prefix that affects operand size or instruction semantics. The ordering of the REX prefix and other optional/mandatory instruction prefixes are discussed Chapter 2. Note that REX prefixes that promote legacy instructions to 64-bit behavior are not listed explicitly in the opcode column.
- **/digit** — A digit between 0 and 7 indicates that the ModR/M byte of the instruction uses only the r/m (register or memory) operand. The reg field contains the digit that provides an extension to the instruction's opcode.
- **/r —** Indicates that the ModR/M byte of the instruction contains a register operand and an r/m operand.
- **cb, cw, cd, cp, co, ct —** A 1-byte (cb), 2-byte (cw), 4-byte (cd), 6-byte (cp), 8-byte (co) or 10-byte (ct) value following the opcode. This value is used to specify a code offset and possibly a new value for the code segment register.
- **ib, iw, id, io —** A 1-byte (ib), 2-byte (iw), 4-byte (id) or 8-byte (io) immediate operand to the instruction that follows the opcode, ModR/M bytes or scale-indexing bytes. The opcode determines if the operand is a signed value. All words, doublewords and quadwords are given with the low-order byte first.
- **+rb, +rw, +rd, +ro —** A register code, from 0 through 7, added to the hexadecimal byte given at the left of the plus sign to form a single opcode byte. See Table 3-1 for the codes. The +ro columns in the table are applicable only in 64-bit mode.
- +**i —** A number used in floating-point instructions when one of the operands is ST(i) from the FPU register stack. The number i (which can range from 0 to 7) is added to the hexadecimal byte given at the left of the plus sign to form a single opcode byte.

...

3.1.1.2 Opcode Column in the Instruction Summary Table (Instructions with VEX prefix)

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 2.3 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 values map 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 2.3 on the subfield definitions within VEX.

— **WIG:** can use C5H form (if not requiring VEX.mmmmm) 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].
- 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 2.3.

3.1.1.3 Instruction Column in the Opcode Summary Table

The "Instruction" column gives the syntax of the instruction statement as it would appear in an ASM386 program. The following is a list of the symbols used to represent operands in the instruction statements:

- **rel8** — A relative address in the range from 128 bytes before the end of the instruction to 127 bytes after the end of the instruction.
- **rel16, rel32, rel64** — A relative address within the same code segment as the instruction assembled. The rel16 symbol applies to instructions with an operand-size attribute of 16 bits; the rel32 symbol applies to instructions with an operand-size attribute of 32 bits; the rel64 symbol applies to instructions with an operand-size attribute of 64 bits.

…

- **<XMM0>**— indicates implied use of the XMM0 register.

  When there is ambiguity, xmm1 indicates the first source operand using an XMM register and xmm2 the second source operand using an XMM register.

  Some instructions use the XMM0 register as the third source operand, indicated by <XMM0>. The use of the third XMM register operand is implicit in the instruction encoding and does not affect the ModR/M encoding.

- **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 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.

3.1.1.4 Operand Encoding Column in the Instruction Summary Table

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

NOTES

- The letters in the Op/En column of an instruction apply ONLY to the encoding definition table immediately following the instruction summary table.
- In the encoding definition table, the letter 'r' within a pair of parenthesis denotes the content of the operand will be read by the processor. The letter 'w' within a pair of parenthesis denotes the content of the operand will be updated by the processor.

3.1.1.5 64/32-bit Mode Column in the Instruction Summary Table

The "64/32-bit Mode" column indicates whether the 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.

3.1.1.6 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/AESNI/PCLMULQDQ/AVX support) that indicate processor support for the instruction. If the corresponding flag is '0', the instruction will #UD.

3.1.1.7 Description Column in the Instruction Summary Table

The "Description" column briefly explains forms of the instruction.

3.1.1.8 Description Section

Each instruction is then described by number of information sections. The "Description" section describes the purpose of the instructions and required operands in more detail.

Summary of terms that may be used in the description section:

- **Legacy SSE:** Refers to SSE, SSE2, SSE3, SSSE3, SSE4, AESNI, PCLMULQDQ and any future instruction sets referencing XMM registers and encoded without 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

3.1.1.9 Operation Section

The "Operation" section contains an algorithm description (frequently written in pseudo-code) for the instruction. Algorithms are composed of the following elements:

- Comments are enclosed within the symbol pairs "(*" and "*)".
- Compound statements are enclosed in keywords, such as: IF, THEN, ELSE and FI for an if statement; DO and OD for a do statement; or CASE... OF for a case statement.
- A register name implies the contents of the register. A register name enclosed in brackets implies the contents of the location whose address is contained in that

register. For example, ES:[DI] indicates the contents of the location whose ES segment relative address is in register DI. [SI] indicates the contents of the address contained in register SI relative to the SI register's default segment (DS) or the overridden segment.

- Parentheses around the "E" in a general-purpose register name, such as (E)SI, indicates that the offset is read from the SI register if the address-size attribute is 16, from the ESI register if the address-size attribute is 32. Parentheses around the "R" in a general-purpose register name, (R)SI, in the presence of a 64-bit register definition such as (R)SI, indicates that the offset is read from the 64-bit RSI register if the address-size attribute is 64.
- Brackets are used for memory operands where they mean that the contents of the memory location is a segment-relative offset. For example, [SRC] indicates that the content of the source operand is a segment-relative offset.
- A ← B indicates that the value of B is assigned to A.
- The symbols =, ≠, >, <, ≥, and ≤ are relational operators used to compare two values: meaning equal, not equal, greater or equal, less or equal, respectively. A relational expression such as A ← B is TRUE if the value of A is equal to B; otherwise it is FALSE.
- The expression "« COUNT" and "» COUNT" indicates that the destination operand should be shifted left or right by the number of bits indicated by the count operand.

The following identifiers are used in the algorithmic descriptions:

- **OperandSize and AddressSize** — The OperandSize identifier represents the operand-size attribute of the instruction, which is 16, 32 or 64-bits. The AddressSize identifier represents the address-size attribute, which is 16, 32 or 64-bits. For example, the following pseudo-code indicates that the operand-size attribute depends on the form of the MOV instruction used.

```
IF Instruction ← MOVW
    THEN OperandSize = 16;
ELSE
    IF Instruction ← MOVD
        THEN OperandSize = 32;
    ELSE
        IF Instruction ← MOVQ
            THEN OperandSize = 64;
        FI;
    FI;
FI;
```

  See "Operand-Size and Address-Size Attributes" in Chapter 3 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*, for guidelines on how these attributes are determined.
- **StackAddrSize** — Represents the stack address-size attribute associated with the instruction, which has a value of 16, 32 or 64-bits. See "Address-Size Attribute for Stack" in Chapter 6, "Procedure Calls, Interrupts, and Exceptions," of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*.
- **SRC** — Represents the source operand.
- **DEST** — Represents the destination operand.
- **VLMAX** — The maximum vector register width pertaining to the instruction. This is not the vector-length encoding in the instruction's prefix but is instead determined

by the current value of XCR0. For existing processors, VLMAX is 256 whenever XCR0.YMM[bit 2] is 1. Future processors may defined new bits in XCR0 whose setting may imply other values for VLMAX.

VLMAX Definition

XCR0 Component	VLMAX
XCR0.YMM	256

The following functions are used in the algorithmic descriptions:

...

ADC—Add with Carry

Opcode	Instruction	Op/ En	64-bit Mode	Compat/ Leg Mode	Description
14 *ib*	ADC AL, *imm8*	C	Valid	Valid	Add with carry *imm8* to AL.
15 *iw*	ADC AX, *imm16*	C	Valid	Valid	Add with carry *imm16* to AX.
15 *id*	ADC EAX, *imm32*	C	Valid	Valid	Add with carry *imm32* to EAX.
REX.W + 15 *id*	ADC RAX, *imm32*	C	Valid	N.E.	Add with carry *imm32 sign extended to 64-bits* to RAX.
80 /2 *ib*	ADC *r/m8, imm8*	B	Valid	Valid	Add with carry *imm8* to *r/m8.*
REX + 80 /2 *ib*	ADC *r/m8**, *imm8*	B	Valid	N.E.	Add with carry *imm8* to *r/m8.*
81 /2 *iw*	ADC *r/m16, imm16*	B	Valid	Valid	Add with carry *imm16* to *r/m16.*
81 /2 *id*	ADC *r/m32, imm32*	B	Valid	Valid	Add with CF *imm32* to *r/m32.*
REX.W + 81 /2 *id*	ADC *r/m64, imm32*	B	Valid	N.E.	Add with CF *imm32* sign extended to 64-bits to *r/m64.*
83 /2 *ib*	ADC *r/m16, imm8*	B	Valid	Valid	Add with CF sign-extended *imm8* to *r/m16.*
83 /2 *ib*	ADC *r/m32, imm8*	B	Valid	Valid	Add with CF sign-extended *imm8* into *r/m32.*
REX.W + 83 /2 *ib*	ADC *r/m64, imm8*	B	Valid	N.E.	Add with CF sign-extended *imm8* into *r/m64.*
10 /*r*	ADC *r/m8, r8*	A	Valid	Valid	Add with carry byte register to *r/m8.*
REX + 10 /*r*	ADC *r/m8**, *r8**	A	Valid	N.E.	Add with carry byte register to *r/m64.*
11 /*r*	ADC *r/m16, r16*	A	Valid	Valid	Add with carry *r16* to *r/m16.*
11 /*r*	ADC *r/m32, r32*	A	Valid	Valid	Add with CF *r32* to *r/m32.*

Opcode	Instruction	Op/ En	64-bit Mode	Compat/ Leg Mode	Description
REX.W + 11 /r	ADC *r/m64, r64*	A	Valid	N.E.	Add with CF *r64* to *r/m64.*
12 /r	ADC *r8, r/m8*	A	Valid	Valid	Add with carry *r/m8* to byte register.
REX + 12 /r	ADC *r8**, *r/m8**	A	Valid	N.E.	Add with carry *r/m64* to byte register.
13 /r	ADC *r16, r/m16*	A	Valid	Valid	Add with carry *r/m16* to *r16.*
13 /r	ADC *r32, r/m32*	A	Valid	Valid	Add with CF *r/m32* to *r32.*
REX.W + 13 /r	ADC *r64, r/m64*	A	Valid	N.E.	Add with CF *r/m64* to *r64.*

NOTES:
*In 64-bit mode, r/m8 can not be encoded to access the following byte registers if a REX prefix is used: AH, BH, CH, DH.

…

ADDPD—Add Packed Double-Precision Floating-Point Values

Opcode/ Instruction	Op/ En	64/32-bit Mode	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/m128* to *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

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
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 two packed double-precision floating-point values from the source operand (second operand) and the destination operand (first operand), and stores the packed double-precision floating-point results in the destination operand.

In 64-bit mode, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

128-bit Legacy SSE version: The second source 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 (VLMAX-1:128) of the corresponding YMM register destination are unmodified. See Chapter 11 in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*, for an overview of SIMD double-precision floating-point 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 (VLMAX-1:128) of the corresponding YMM register destination are zeroed.

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

Operation

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[VLMAX-1:128] (Unmodified)

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

VADDPD (VEX.256 encoded version)
DEST[63:0] ← SRC1[63:0] + SRC2[63:0]
DEST[127:64] ← SRC1[127:64] + SRC2[127:64]
DEST[191:128] ← SRC1[191:128] + SRC2[191:128]
DEST[255:192] ← SRC1[255:192] + SRC2[255:192]

.Intel C/C++ Compiler Intrinsic Equivalent

ADDPD __m128d _mm_add_pd (__m128d a, __m128d b)

VADDPD __m256d _mm256_add_pd (__m256d a, __m256d 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	CPUID Feature Flag	Description
0F 58 /r ADDPS *xmm1, xmm2/m128*	A	V/V	SSE	Add packed single-precision floating-point values from *xmm2/m128* 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

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
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 four packed single-precision floating-point values from the source operand (second operand) and the destination operand (first operand), and stores the packed single-precision floating-point results in the destination operand.

In 64-bit mode, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

128-bit Legacy SSE version: The second source 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 (VLMAX-1:128) of the corresponding YMM register destination are unmodified. See Chapter 10 in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*, for an overview of SIMD single-precision floating-point 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 (VLMAX-1:128) of the corresponding YMM register destination are zeroed.

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

Operation

ADDPS (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[VLMAX-1:128] (Unmodified)

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[VLMAX-1:128] ← 0

VADDPS (VEX.256 encoded version)
DEST[31:0] ← SRC1[31:0] + SRC2[31:0]
DEST[63:32] ← SRC1[63:32] + SRC2[63:32]
DEST[95:64] ← SRC1[95:64] + SRC2[95:64]
DEST[127:96] ← SRC1[127:96] + SRC2[127:96]
DEST[159:128] ← SRC1[159:128] + SRC2[159:128]
DEST[191:160]← SRC1[191:160] + SRC2[191:160]
DEST[223:192] ← SRC1[223:192] + SRC2[223:192]
DEST[255:224] ← SRC1[255:224] + SRC2[255:224]

Intel C/C++ Compiler Intrinsic Equivalent

ADDPS __m128 _mm_add_ps(__m128 a, __m128 b)

VADDPS __m256 _mm256_add_ps (__m256 a, __m256 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	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/m64* to *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

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
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 source operand (second operand) and the destination operand (first operand), and stores the double-precision floating-point result in the destination operand.

The source operand can be an XMM register or a 64-bit memory location. The destination operand is an XMM register. 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.

In 64-bit mode, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

128-bit Legacy SSE version: Bits (VLMAX-1:64) 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 (VLMAX-1:128) of the destination YMM register are zeroed.

Operation

ADDSD (128-bit Legacy SSE version)
DEST[63:0] ← DEST[63:0] + SRC[63:0]
DEST[VLMAX-1:64] (Unmodified)
VADDSD (VEX.128 encoded version)
DEST[63:0] ← SRC1[63:0] + SRC2[63:0]
DEST[127:64] ← SRC1[127:64]
DEST[VLMAX-1:128] ← 0

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	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/m32* to *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

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
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 source operand (second operand) and the destination operand (first operand), and stores the single-precision floating-point result in the destination operand.

The source operand can be an XMM register or a 32-bit memory location. The destination operand is an XMM register. 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.

In 64-bit mode, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

128-bit Legacy SSE version: Bits (VLMAX-1:32) 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 (VLMAX-1:128) of the destination YMM register are zeroed.

Operation

ADDSS DEST, SRC (128-bit Legacy SSE version)
DEST[31:0] ← DEST[31:0] + SRC[31:0];
DEST[VLMAX-1:32] (Unmodified)

VADDSS DEST, SRC1, SRC2 (VEX.128 encoded version)
DEST[31:0] ← SRC1[31:0] + SRC2[31:0]
DEST[127:32] ← SRC1[127:32]
DEST[VLMAX-1:128] ← 0

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	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

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
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.

In 64-bit mode, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

128-bit Legacy SSE version: The second source 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 (VLMAX-1:128) of the corresponding YMM register destination are unmodified. See Figure 3-3.

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 (VLMAX-1:128) of the corresponding YMM register destination are zeroed.

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

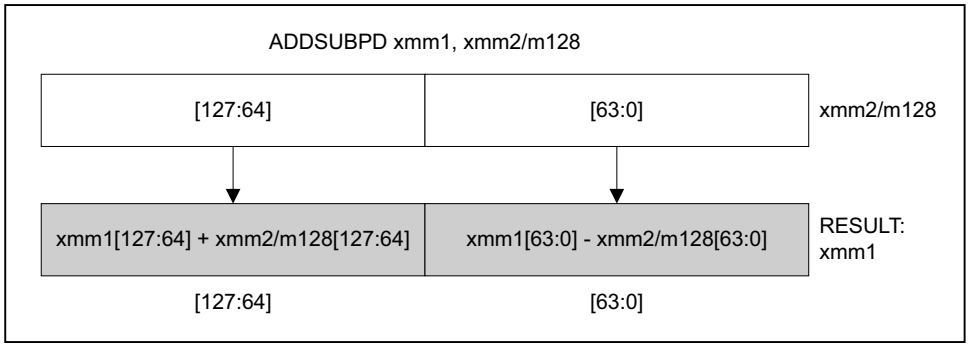


Figure 3-3 ADDSUBPD—Packed Double-FP Add/Subtract

Operation

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[VLMAX-1:128] (Unmodified)

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

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]

Intel C/C++ Compiler Intrinsic Equivalent

ADDSUBPD __m128d _mm_addsub_pd(__m128d a, __m128d b)

VADDSUBPD __m256d _mm256_addsub_pd (__m256d a, __m256d b)

Exceptions

When the source operand is a memory operand, it must be aligned on a 16-byte boundary or a general-protection exception (#GP) will be generated.

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	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

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
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.

In 64-bit mode, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

128-bit Legacy SSE version: The second source 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 (VLMAX-1:128) of the corresponding YMM register destination are unmodified. See Figure 3-4.

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 (VLMAX-1:128) of the corresponding YMM register destination are zeroed.

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

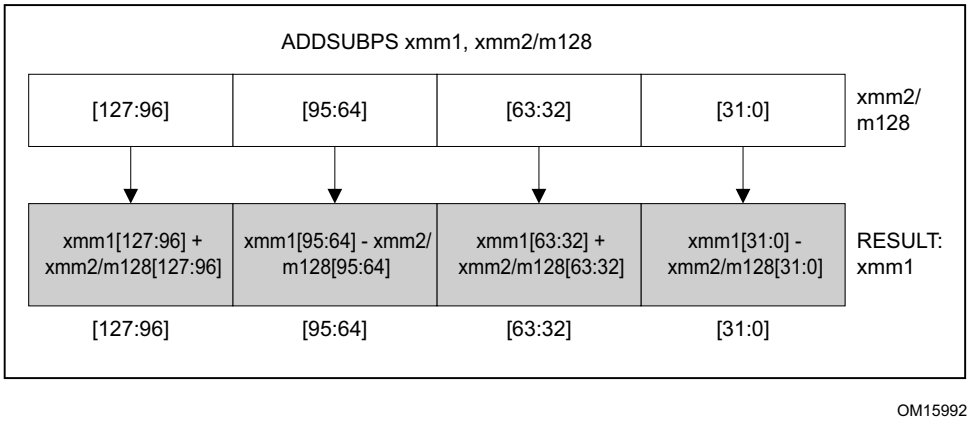


Figure 3-4 ADDSUBPS—Packed Single-FP Add/Subtract

Operation

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[VLMAX-1:128] (Unmodified)

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[VLMAX-1:128] ← 0

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].

Intel C/C++ Compiler Intrinsic Equivalent

ADDSUBPS __m128 _mm_addsub_ps(__m128 a, __m128 b)

VADDSUBPS __m256 _mm256_addsub_ps (__m256 a, __m256 b)

Exceptions

When the source operand is a memory operand, the operand must be aligned on a 16-byte boundary or a general-protection exception (#GP) will be generated.

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal.

Other Exceptions

See Exceptions Type 2.

...

AESDEC—Perform One Round of an AES Decryption Flow

Opcode/ Instruction	Op/ En	64/32-bit Mode	CPUID Feature Flag	Description
66 0F 38 DE /r AESDEC xmm1, xmm2/m128	A	V/V	AES	Perform one round of an AES decryption flow, using the Equivalent Inverse Cipher, 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.

Instruction Operand Encoding

Op/En	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

This instruction performs 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.

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 (VLMAX-1:128) of the corresponding YMM destination register remain unchanged.

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

Operation

AESDEC

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

VAESDEC

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

Intel C/C++ Compiler Intrinsic Equivalent

(V)AESDEC __m128i _mm_aesdec (__m128i, __m128i)

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type 4.

...

AESDECLAST—Perform Last Round of an AES Decryption Flow

Opcode	Instruction	Op/ En	64/32-bit Mode	CPUID Feature Flag	Description
66 0F 38 DF /r	AESDECLAST xmm1, xmm2/m128	A	V/V	AES	Perform the last round of an AES decryption flow, using the Equivalent Inverse Cipher, operating on a 128-bit data (state) from xmm1 with a 128-bit round key from xmm2/m128.
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

Op/En	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

This instruction performs the last 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.

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 (VLMAX-1:128) of the corresponding YMM destination register remain unchanged.

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

Operation

AESDECLAST

STATE ← SRC1;

RoundKey ← SRC2;
STATE ← InvShiftRows(STATE);
STATE ← InvSubBytes(STATE);
DEST[127:0] ← STATE XOR RoundKey;
DEST[VLMAX-1:128] (Unmodified)

VAESDECLAST
STATE ← SRC1;
RoundKey ← SRC2;
STATE ← InvShiftRows(STATE);
STATE ← InvSubBytes(STATE);
DEST[127:0] ← STATE XOR RoundKey;
DEST[VLMAX-1:128] ← 0

Intel C/C++ Compiler Intrinsic Equivalent

(V)AESDECLAST __m128i _mm_aesdeclast (__m128i, __m128i)

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type 4.

...

AESENC—Perform One Round of an AES Encryption Flow

Opcode	Instruction	Op/ En	64/32-bit Mode	CPUID Feature Flag	Description
66 0F 38 DC /r	AESENC xmm1, xmm2/m128	A	V/V	AES	Perform one round of an AES encryption flow, 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.

Instruction Operand Encoding

Op/En	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

This instruction performs 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.

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 (VLMAX-1:128) of the corresponding YMM destination register remain unchanged.

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

Operation

AESENC

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

VAESENC

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

Intel C/C++ Compiler Intrinsic Equivalent

(V)AESENC __m128i _mm_aesenc (__m128i, __m128i)

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type 4.

...

AESENCLAST—Perform Last Round of an AES Encryption Flow

Opcode/ Instruction	Op/ En	64/32-bit Mode	CPUID Feature Flag	Description
66 0F 38 DD /r AESENCLAST xmm1, xmm2/m128	A	V/V	AES	Perform the last round of an AES encryption flow, operating on a 128-bit data (state) from xmm1 with a 128-bit round key from xmm2/m128.
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

Op/En	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

This instruction performs the last 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.

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 (VLMAX-1:128) of the corresponding YMM destination register remain unchanged.

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

Operation

AESENCLAST

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

VAESENCLAST
STATE ← SRC1;
RoundKey ← SRC2;
STATE ← ShiftRows(STATE);
STATE ← SubBytes(STATE);
DEST[127:0] ← STATE XOR RoundKey;
DEST[VLMAX-1:128] ← 0

Intel C/C++ Compiler Intrinsic Equivalent

(V)AESENCLAST __m128i _mm_aesenclast (__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	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

Op/En	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).

128-bit Legacy SSE version: Bits (VLMAX-1:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: Bits (VLMAX-1:128) of the destination YMM register are zeroed.

Operation

AESIMC
DEST[127:0] ← InvMixColumns(SRC);
DEST[VLMAX-1:128] (Unmodified)

VAESIMC
DEST[127:0] ← InvMixColumns(SRC);
DEST[VLMAX-1:128] ← 0;

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	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

Op/En	Operand 1	Operand2	Operand3	Operand4
A	ModRM:reg (w)	ModRM:r/m (r)	imm8	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.

128-bit Legacy SSE version:Bits (VLMAX-1:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: Bits (VLMAX-1:128) of the destination YMM register are zeroed.

Operation

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[VLMAX-1:128] (Unmodified)
```

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[VLMAX-1:128] ← 0;
```

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.

...

AND—Logical AND

Opcode	Instruction	Op/ En	64-bit Mode	Compat/ Leg Mode	Description
24 *ib*	AND AL, *imm8*	C	Valid	Valid	AL AND *imm8.*
25 *iw*	AND AX, *imm16*	C	Valid	Valid	AX AND i*mm16.*
25 *id*	AND EAX, *imm32*	C	Valid	Valid	EAX AND *imm32.*
REX.W + 25 *id*	AND RAX, *imm32*	C	Valid	N.E.	RAX AND *imm32 sign-extended to 64-bits.*
80 /4 *ib*	AND *r/m8, imm8*	B	Valid	Valid	*r/m8* AND *imm8.*
REX + 80 /4 *ib*	AND *r/m8**, *imm8*	B	Valid	N.E.	*r/m8* AND *imm8.*
81 /4 *iw*	AND *r/m16, imm16*	B	Valid	Valid	*r/m16* AND *imm16.*
81 /4 *id*	AND *r/m32, imm32*	B	Valid	Valid	*r/m32* AND *imm32.*
REX.W + 81 /4 *id*	AND *r/m64, imm32*	B	Valid	N.E.	*r/m64* AND *imm32 sign extended to 64-bits.*
83 /4 *ib*	AND *r/m16, imm8*	B	Valid	Valid	*r/m16* AND *imm8 (sign-extended).*
83 /4 *ib*	AND *r/m32, imm8*	B	Valid	Valid	*r/m32* AND *imm8 (sign-extended).*
REX.W + 83 /4 *ib*	AND *r/m64, imm8*	B	Valid	N.E.	*r/m64* AND *imm8 (sign-extended).*
20 /*r*	AND *r/m8, r8*	A	Valid	Valid	*r/m8* AND *r8.*
REX + 20 /*r*	AND *r/m8**, *r8**	A	Valid	N.E.	*r/m64* AND *r8 (sign-extended).*
21 /*r*	AND *r/m16, r16*	A	Valid	Valid	*r/m16* AND *r16.*
21 /*r*	AND *r/m32, r32*	A	Valid	Valid	*r/m32* AND *r32.*
REX.W + 21 /*r*	AND *r/m64, r64*	A	Valid	N.E.	*r/m64* AND *r32.*
22 /*r*	AND *r8, r/m8*	A	Valid	Valid	*r8* AND *r/m8.*
REX + 22 /*r*	AND *r8**, *r/m8**	A	Valid	N.E.	*r/m64* AND *r8 (sign-extended).*
23 /*r*	AND *r16, r/m16*	A	Valid	Valid	*r16* AND *r/m16.*
23 /*r*	AND *r32, r/m32*	A	Valid	Valid	*r32* AND *r/m32.*
REX.W + 23 /*r*	AND *r64, r/m64*	A	Valid	N.E.	*r64* AND *r/m64.*

NOTES:

*In 64-bit mode, r/m8 can not be encoded to access the following byte registers if a REX prefix is used: AH, BH, CH, DH.

...

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

Opcode/ Instruction	Op/ En	64/32-bit Mode	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/m128*.
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

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
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 packed double-precision floating-point values from the source operand (second operand) and the destination operand (first operand), and stores the result in the destination operand.

In 64-bit mode, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

128-bit Legacy SSE version: The second source 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 (VLMAX-1:128) of the corresponding YMM register destination are unmodified.

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 (VLMAX-1:128) of the corresponding YMM register destination are zeroed.

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

Operation

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[VLMAX-1:128] (Unmodified)

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[VLMAX-1:128] ← 0

VANDPD (VEX.256 encoded version)
DEST[63:0] ← SRC1[63:0] BITWISE AND SRC2[63:0]
DEST[127:64] ← SRC1[127:64] BITWISE AND SRC2[127:64]
DEST[191:128] ← SRC1[191:128] BITWISE AND SRC2[191:128]
DEST[255:192] ← SRC1[255:192] BITWISE AND SRC2[255:192]

Intel C/C++ Compiler Intrinsic Equivalent

ANDPD __m128d _mm_and_pd(__m128d a, __m128d b)

VANDPD __m256d _mm256_and_pd (__m256d a, __m256d 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	CPUID Feature Flag	Description
0F 54 /r ANDPS *xmm1, xmm2/m128*	A	V/V	SSE	Bitwise logical AND of *xmm2/m128* and *xmm1*.
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

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
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.

In 64-bit mode, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

128-bit Legacy SSE version: The second source 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 (VLMAX-1:128) of the corresponding YMM register destination are unmodified.

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 (VLMAX-1:128) of the corresponding YMM register destination are zeroed.

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

Operation

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[VLMAX-1:128] (Unmodified)
```

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[VLMAX-1:128] ← 0
```

VANDPS (VEX.256 encoded version)

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

Intel C/C++ Compiler Intrinsic Equivalent

ANDPS __m128 _mm_and_ps(__m128 a, __m128 b)

VANDPS __m256 _mm256_and_ps (__m256 a, __m256 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	CPUID Feature Flag	Description
66 0F 55 /r ANDNPD *xmm1*, *xmm2/m128*	A	V/V	SSE2	Bitwise logical AND NOT of *xmm2/m128* and *xmm1*.
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

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
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.

In 64-bit mode, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

128-bit Legacy SSE version: The second source 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 (VLMAX-1:128) of the corresponding YMM register destination are unmodified.

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 (VLMAX-1:128) of the corresponding YMM register destination are zeroed.

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

Operation

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[VLMAX-1:128] (Unmodified)

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[VLMAX-1:128] ← 0

VANDNPD (VEX.256 encoded version)
DEST[63:0] ← (NOT(SRC1[63:0])) BITWISE AND SRC2[63:0]
DEST[127:64] ← (NOT(SRC1[127:64])) BITWISE AND SRC2[127:64]
DEST[191:128] ← (NOT(SRC1[191:128])) BITWISE AND SRC2[191:128]
DEST[255:192] ← (NOT(SRC1[255:192])) BITWISE AND SRC2[255:192]

Intel C/C++ Compiler Intrinsic Equivalent

ANDNPD __m128d _mm_andnot_pd(__m128d a, __m128d b)

VANDNPD __m256d _mm256_andnot_pd (__m256d a, __m256d 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	CPUID Feature Flag	Description
0F 55 /r ANDNPS *xmm1, xmm2/m128*	A	V/V	SSE	Bitwise logical AND NOT of *xmm2/m128* and *xmm1*.
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

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
A	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
B	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA

Description

Inverts the bits of the four packed single-precision floating-point values in the destination operand (first operand), performs a bitwise logical AND of the four packed single-precision floating-point values in the source operand (second operand) and the temporary inverted result, and stores the result in the destination operand.

In 64-bit mode, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

128-bit Legacy SSE version: The second source 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 (VLMAX-1:128) of the corresponding YMM register destination are unmodified.

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 (VLMAX-1:128) of the corresponding YMM register destination are zeroed.

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

Operation

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[VLMAX-1:128] (Unmodified)

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[VLMAX-1:128] ← 0

VANDNPS (VEX.256 encoded version)
DEST[31:0] ← (NOT(SRC1[31:0])) BITWISE AND SRC2[31:0]
DEST[63:32] ← (NOT(SRC1[63:32])) BITWISE AND SRC2[63:32]
DEST[95:64] ← (NOT(SRC1[95:64])) BITWISE AND SRC2[95:64]
DEST[127:96] ← (NOT(SRC1[127:96])) BITWISE AND SRC2[127:96]
DEST[159:128] ← (NOT(SRC1[159:128])) BITWISE AND SRC2[159:128]
DEST[191:160]← (NOT(SRC1[191:160])) BITWISE AND SRC2[191:160]
DEST[223:192] ← (NOT(SRC1[223:192])) BITWISE AND SRC2[223:192]
DEST[255:224] ← (NOT(SRC1[255:224])) BITWISE AND SRC2[255:224].

Intel C/C++ Compiler Intrinsic Equivalent

ANDNPS __m128 _mm_andnot_ps(__m128 a, __m128 b)

VANDNPS __m256 _mm256_andnot_ps (__m256 a, __m256 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	CPUID Feature Flag	Description
66 0F 3A 0D /r ib BLENDPD *xmm1, xmm2/m128, imm8*	A	V/V	SSE4_1	Select packed DP-FP values from *xmm1* and *xmm2/m128* from mask specified in imm8 and store the values into *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

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
A	ModRM:reg (r, w)	ModRM:r/m (r)	imm8	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.

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 (VLMAX-1:128) of the corresponding YMM register destination are unmodified.

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 (VLMAX-1:128) of the corresponding YMM register destination are zeroed.

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

Operation

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[VLMAX-1:128] (Unmodified)

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[VLMAX-1:128] ← 0

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

Intel C/C++ Compiler Intrinsic Equivalent

BLENDPD __m128d _mm_blend_pd (__m128d v1, __m128d v2, const int mask);

VBLENDPD __m256d _mm256_blend_pd (__m256d a, __m256d 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	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 specified in *imm8* and store the values into *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

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
A	ModRM:reg (r, w)	ModRM:r/m (r)	imm8	NA
B	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	imm8

Description

Packed 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.

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 (VLMAX-1:128) of the corresponding YMM register destination are unmodified.

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 (VLMAX-1:128) of the corresponding YMM register destination are zeroed.

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

Operation

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[VLMAX-1:128] (Unmodified)
```

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[VLMAX-1:128] ← 0
```

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.
```

Intel C/C++ Compiler Intrinsic Equivalent

BLENDPS __m128 _mm_blend_ps (__m128 v1, __m128 v2, const int mask);

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

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type 4.

...

BLENDVPD — Variable Blend Packed Double Precision Floating-Point Values

Opcode/ Instruction	Op/ En	64/32-bit Mode	CPUID Feature Flag	Description
66 0F 38 15 /r BLENDVPD *xmm1, xmm2/m128 , <XMM0>*	A	V/V	SSE4_1	Select packed DP FP values from *xmm1* and *xmm2* from mask specified in *XMM0* and store the values in *xmm1*.
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

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
A	ModRM:reg (r, w)	ModRM:r/m (r)	implicit XMM0	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 (VLMAX-1:128) 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 (VLMAX-1:128) 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

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[VLMAX-1:128] (Unmodified)
```

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[VLMAX-1:128] ← 0
```

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
```

Intel C/C++ Compiler Intrinsic Equivalent

BLENDVPD __m128d _mm_blendv_pd(__m128d v1, __m128d v2, __m128d v3);

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

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

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type 4; additionally

#UD	If VEX.W = 1.

...

BLENDVPS — Variable Blend Packed Single Precision Floating-Point Values

Opcode/ Instruction	Op/ En	64/32-bit Mode	CPUID Feature Flag	Description
66 0F 38 14 /r BLENDVPS *xmm1, xmm2/m128, <XMM0>*	A	V/V	SSE4_1	Select packed single precision floating-point values from *xmm1* and *xmm2/m128* from mask specified in *XMM0* and store the values into *xmm1*.
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

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
A	ModRM:reg (r, w)	ModRM:r/m (r)	implicit XMM0	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 (VLMAX-1:128) 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 (VLMAX-1:128) 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

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[VLMAX-1:128] (Unmodified)
```

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[VLMAX-1:128] ← 0
```

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
```

Intel C/C++ Compiler Intrinsic Equivalent

BLENDVPS __m128 _mm_blendv_ps(__m128 v1, __m128 v2, __m128 v3);

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

VBLENDVPS __m256 _mm256_blendv_ps (__m256 a, __m256 b, __m256 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	CPUID Feature Flag	Description
VEX.128.66.0F38.W0 18 /r VBROADCASTSS xmm1, m32	A	I/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

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
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.

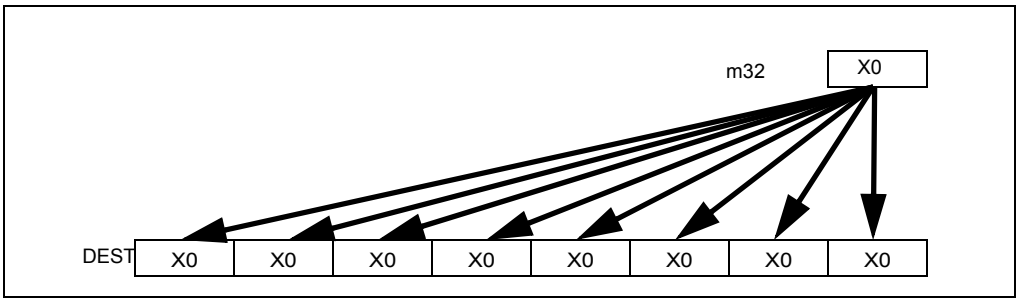


Figure 3-5 VBROADCASTSS Operation (VEX.256 encoded version)

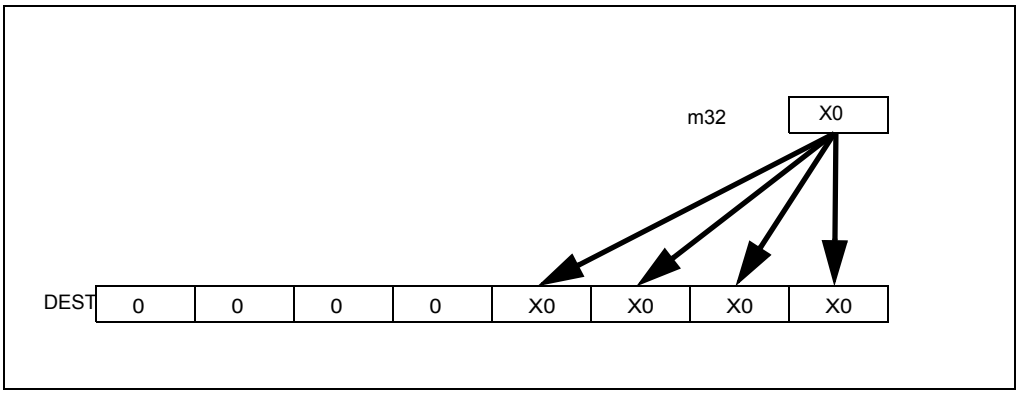


Figure 3-6 VBROADCASTSS Operation (128-bit version)

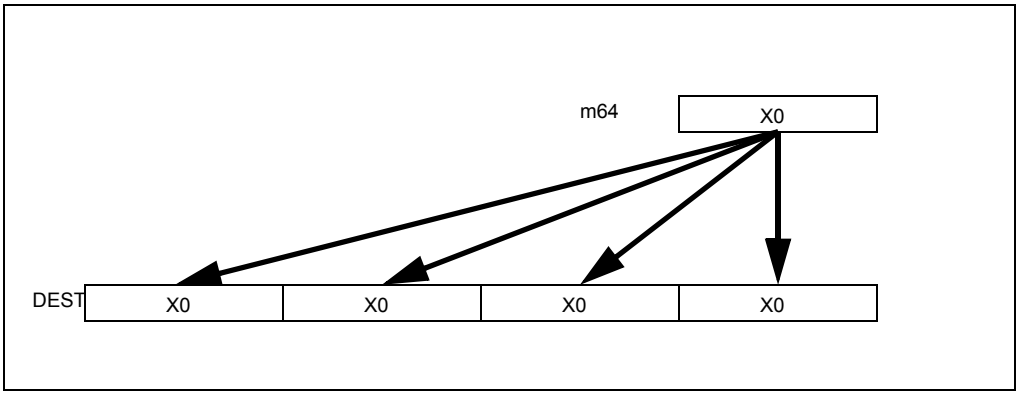


Figure 3-7 VBROADCASTSD Operation

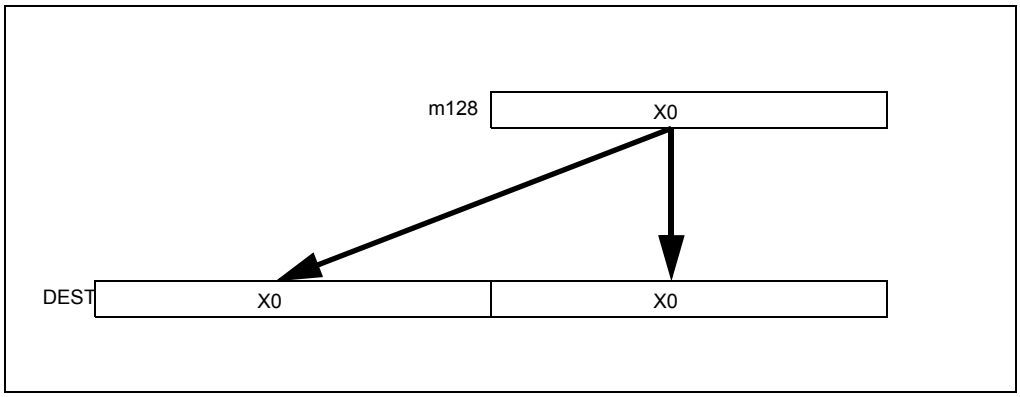


Figure 3-8 VBROADCASTF128 Operation

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[VLMAX-1:128] ← 0

VBROADCASTSS (VEX.256 encoded version)
temp ← SRC[31:0]
DEST[31:0] ← temp
DEST[63:32] ← temp
DEST[95:64] ← temp
DEST[127:96] ← temp
DEST[159:128] ← temp
DEST[191:160] ← temp
DEST[223:192] ← temp
DEST[255:224] ← temp

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[VLMAX-1:128] ← 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);

Flags Affected

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.

...

CMOV*cc*—Conditional Move

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
0F 47 /r	CMOVA *r16, r/m16*	A	Valid	Valid	Move if above (CF=0 and ZF=0).
0F 47 /r	CMOVA *r32, r/m32*	A	Valid	Valid	Move if above (CF=0 and ZF=0).
REX.W + 0F 47 /r	CMOVA *r64, r/m64*	A	Valid	N.E.	Move if above (CF=0 and ZF=0).
0F 43 /r	CMOVAE *r16, r/m16*	A	Valid	Valid	Move if above or equal (CF=0).
0F 43 /r	CMOVAE *r32, r/m32*	A	Valid	Valid	Move if above or equal (CF=0).
REX.W + 0F 43 /r	CMOVAE *r64, r/m64*	A	Valid	N.E.	Move if above or equal (CF=0).
0F 42 /r	CMOVB *r16, r/m16*	A	Valid	Valid	Move if below (CF=1).
0F 42 /r	CMOVB *r32, r/m32*	A	Valid	Valid	Move if below (CF=1).
REX.W + 0F 42 /r	CMOVB *r64, r/m64*	A	Valid	N.E.	Move if below (CF=1).
0F 46 /r	CMOVBE *r16, r/m16*	A	Valid	Valid	Move if below or equal (CF=1 or ZF=1).
0F 46 /r	CMOVBE *r32, r/m32*	A	Valid	Valid	Move if below or equal (CF=1 or ZF=1).
REX.W + 0F 46 / r	CMOVBE *r64, r/m64*	A	Valid	N.E.	Move if below or equal (CF=1 or ZF=1).

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
0F 42 /r	CMOVC *r16, r/m16*	A	Valid	Valid	Move if carry (CF=1).
0F 42 /r	CMOVC *r32, r/m32*	A	Valid	Valid	Move if carry (CF=1).
REX.W + 0F 42 /r	CMOVC *r64, r/m64*	A	Valid	N.E.	Move if carry (CF=1).
0F 44 /r	CMOVE *r16, r/m16*	A	Valid	Valid	Move if equal (ZF=1).
0F 44 /r	CMOVE *r32, r/m32*	A	Valid	Valid	Move if equal (ZF=1).
REX.W + 0F 44 /r	CMOVE *r64, r/m64*	A	Valid	N.E.	Move if equal (ZF=1).
0F 4F /r	CMOVG *r16, r/m16*	A	Valid	Valid	Move if greater (ZF=0 and SF=OF).
0F 4F /r	CMOVG *r32, r/m32*	A	Valid	Valid	Move if greater (ZF=0 and SF=OF).
REX.W + 0F 4F /r	CMOVG *r64, r/m64*	A	V/N.E.	NA	Move if greater (ZF=0 and SF=OF).
0F 4D /r	CMOVGE *r16, r/m16*	A	Valid	Valid	Move if greater or equal (SF=OF).
0F 4D /r	CMOVGE *r32, r/m32*	A	Valid	Valid	Move if greater or equal (SF=OF).
REX.W + 0F 4D /r	CMOVGE *r64, r/m64*	A	Valid	N.E.	Move if greater or equal (SF=OF).
0F 4C /r	CMOVL *r16, r/m16*	A	Valid	Valid	Move if less (SF≠ OF).
0F 4C /r	CMOVL *r32, r/m32*	A	Valid	Valid	Move if less (SF≠ OF).
REX.W + 0F 4C /r	CMOVL *r64, r/m64*	A	Valid	N.E.	Move if less (SF≠ OF).
0F 4E /r	CMOVLE *r16, r/m16*	A	Valid	Valid	Move if less or equal (ZF=1 or SF≠ OF).
0F 4E /r	CMOVLE *r32, r/m32*	A	Valid	Valid	Move if less or equal (ZF=1 or SF≠ OF).
REX.W + 0F 4E /r	CMOVLE *r64, r/m64*	A	Valid	N.E.	Move if less or equal (ZF=1 or SF≠ OF).
0F 46 /r	CMOVNA *r16, r/m16*	A	Valid	Valid	Move if not above (CF=1 or ZF=1).
0F 46 /r	CMOVNA *r32, r/m32*	A	Valid	Valid	Move if not above (CF=1 or ZF=1).
REX.W + 0F 46 /r	CMOVNA *r64, r/m64*	A	Valid	N.E.	Move if not above (CF=1 or ZF=1).
0F 42 /r	CMOVNAE *r16, r/m16*	A	Valid	Valid	Move if not above or equal (CF=1).
0F 42 /r	CMOVNAE *r32, r/m32*	A	Valid	Valid	Move if not above or equal (CF=1).
REX.W + 0F 42 /r	CMOVNAE *r64, r/m64*	A	Valid	N.E.	Move if not above or equal (CF=1).
0F 43 /r	CMOVNB *r16, r/m16*	A	Valid	Valid	Move if not below (CF=0).

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
0F 43 /r	CMOVNB *r32, r/m32*	A	Valid	Valid	Move if not below (CF=0).
REX.W + 0F 43 /r	CMOVNB *r64, r/m64*	A	Valid	N.E.	Move if not below (CF=0).
0F 47 /r	CMOVNBE *r16, r/m16*	A	Valid	Valid	Move if not below or equal (CF=0 and ZF=0).
0F 47 /r	CMOVNBE *r32, r/m32*	A	Valid	Valid	Move if not below or equal (CF=0 and ZF=0).
REX.W + 0F 47 /r	CMOVNBE *r64, r/m64*	A	Valid	N.E.	Move if not below or equal (CF=0 and ZF=0).
0F 43 /r	CMOVNC *r16, r/m16*	A	Valid	Valid	Move if not carry (CF=0).
0F 43 /r	CMOVNC *r32, r/m32*	A	Valid	Valid	Move if not carry (CF=0).
REX.W + 0F 43 /r	CMOVNC *r64, r/m64*	A	Valid	N.E.	Move if not carry (CF=0).
0F 45 /r	CMOVNE *r16, r/m16*	A	Valid	Valid	Move if not equal (ZF=0).
0F 45 /r	CMOVNE *r32, r/m32*	A	Valid	Valid	Move if not equal (ZF=0).
REX.W + 0F 45 /r	CMOVNE *r64, r/m64*	A	Valid	N.E.	Move if not equal (ZF=0).
0F 4E /r	CMOVNG *r16, r/m16*	A	Valid	Valid	Move if not greater (ZF=1 or SF≠ OF).
0F 4E /r	CMOVNG *r32, r/m32*	A	Valid	Valid	Move if not greater (ZF=1 or SF≠ OF).
REX.W + 0F 4E /r	CMOVNG *r64, r/m64*	A	Valid	N.E.	Move if not greater (ZF=1 or SF≠ OF).
0F 4C /r	CMOVNGE *r16, r/m16*	A	Valid	Valid	Move if not greater or equal (SF≠ OF).
0F 4C /r	CMOVNGE *r32, r/m32*	A	Valid	Valid	Move if not greater or equal (SF≠ OF).
REX.W + 0F 4C /r	CMOVNGE *r64, r/m64*	A	Valid	N.E.	Move if not greater or equal (SF≠ OF).
0F 4D /r	CMOVNL *r16, r/m16*	A	Valid	Valid	Move if not less (SF=OF).
0F 4D /r	CMOVNL *r32, r/m32*	A	Valid	Valid	Move if not less (SF=OF).
REX.W + 0F 4D /r	CMOVNL *r64, r/m64*	A	Valid	N.E.	Move if not less (SF=OF).
0F 4F /r	CMOVNLE *r16, r/m16*	A	Valid	Valid	Move if not less or equal (ZF=0 and SF=OF).
0F 4F /r	CMOVNLE *r32, r/m32*	A	Valid	Valid	Move if not less or equal (ZF=0 and SF=OF).
REX.W + 0F 4F /r	CMOVNLE *r64, r/m64*	A	Valid	N.E.	Move if not less or equal (ZF=0 and SF=OF).
0F 41 /r	CMOVNO *r16, r/m16*	A	Valid	Valid	Move if not overflow (OF=0).
0F 41 /r	CMOVNO *r32, r/m32*	A	Valid	Valid	Move if not overflow (OF=0).

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
REX.W + 0F 41 /r	CMOVNO *r64, r/m64*	A	Valid	N.E.	Move if not overflow (OF=0).
0F 4B /r	CMOVNP *r16, r/m16*	A	Valid	Valid	Move if not parity (PF=0).
0F 4B /r	CMOVNP *r32, r/m32*	A	Valid	Valid	Move if not parity (PF=0).
REX.W + 0F 4B /r	CMOVNP *r64, r/m64*	A	Valid	N.E.	Move if not parity (PF=0).
0F 49 /r	CMOVNS *r16, r/m16*	A	Valid	Valid	Move if not sign (SF=0).
0F 49 /r	CMOVNS *r32, r/m32*	A	Valid	Valid	Move if not sign (SF=0).
REX.W + 0F 49 /r	CMOVNS *r64, r/m64*	A	Valid	N.E.	Move if not sign (SF=0).
0F 45 /r	CMOVNZ *r16, r/m16*	A	Valid	Valid	Move if not zero (ZF=0).
0F 45 /r	CMOVNZ *r32, r/m32*	A	Valid	Valid	Move if not zero (ZF=0).
REX.W + 0F 45 /r	CMOVNZ *r64, r/m64*	A	Valid	N.E.	Move if not zero (ZF=0).
0F 40 /r	CMOVO *r16, r/m16*	A	Valid	Valid	Move if overflow (OF=1).
0F 40 /r	CMOVO *r32, r/m32*	A	Valid	Valid	Move if overflow (OF=1).
REX.W + 0F 40 /r	CMOVO *r64, r/m64*	A	Valid	N.E.	Move if overflow (OF=1).
0F 4A /r	CMOVP *r16, r/m16*	A	Valid	Valid	Move if parity (PF=1).
0F 4A /r	CMOVP *r32, r/m32*	A	Valid	Valid	Move if parity (PF=1).
REX.W + 0F 4A /r	CMOVP *r64, r/m64*	A	Valid	N.E.	Move if parity (PF=1).
0F 4A /r	CMOVPE *r16, r/m16*	A	Valid	Valid	Move if parity even (PF=1).
0F 4A /r	CMOVPE *r32, r/m32*	A	Valid	Valid	Move if parity even (PF=1).
REX.W + 0F 4A /r	CMOVPE *r64, r/m64*	A	Valid	N.E.	Move if parity even (PF=1).
0F 4B /r	CMOVPO *r16, r/m16*	A	Valid	Valid	Move if parity odd (PF=0).
0F 4B /r	CMOVPO *r32, r/m32*	A	Valid	Valid	Move if parity odd (PF=0).
REX.W + 0F 4B /r	CMOVPO *r64, r/m64*	A	Valid	N.E.	Move if parity odd (PF=0).
0F 48 /r	CMOVS *r16, r/m16*	A	Valid	Valid	Move if sign (SF=1).
0F 48 /r	CMOVS *r32, r/m32*	A	Valid	Valid	Move if sign (SF=1).
REX.W + 0F 48 /r	CMOVS *r64, r/m64*	A	Valid	N.E.	Move if sign (SF=1).
0F 44 /r	CMOVZ *r16, r/m16*	A	Valid	Valid	Move if zero (ZF=1).
0F 44 /r	CMOVZ *r32, r/m32*	A	Valid	Valid	Move if zero (ZF=1).
REX.W + 0F 44 /r	CMOVZ *r64, r/m64*	A	Valid	N.E.	Move if zero (ZF=1).

…

Operation

```
temp ← SRC

IF condition TRUE
    THEN
        DEST ← temp;
    FI;
ELSE
    IF (OperandSize = 32 and IA-32e mode active)
        THEN
            DEST[63:32] ← 0;
    FI;
FI;

...
```

CMPPD—Compare Packed Double-Precision Floating-Point Values

Opcode/ Instruction	Op/ En	64/32-bit Mode	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 imm8 as 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

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
A	ModRM:reg (r, w)	ModRM:r/m (r)	imm8	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 source operand (second operand) and the destination operand (first operand) and returns the results of the comparison to the destination operand. The comparison predicate operand (third operand) 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).

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. The comparison predicate operand is an 8-bit immediate, bits 2:0 of the immediate define the type of comparison to be performed (see Table 3-7). Bits 7:3 of the immediate is reserved. Bits (VLMAX-1:128) of the corresponding YMM destination register remain unchanged. Two comparisons are performed with results written to bits 127:0 of the destination operand.

Table 3-7 Comparison Predicate for CMPPD and CMPPS Instructions

Predi-cate	imm8 Encod-ing	Description	Relation where: A Is 1st Operand B Is 2nd Operand	Emulation	Result if NaN Operand	QNaN Oper-and Signals Invalid
EQ	000B	Equal	A = B		False	No
LT	001B	Less-than	A < B		False	Yes
LE	010B	Less-than-or-equal	A ≤ B		False	Yes
		Greater than	A > B	Swap Operands, Use LT	False	Yes
		Greater-than-or-equal	A ≥ B	Swap Operands, Use LE	False	Yes
UNORD	011B	Unordered	A, B = Unordered		True	No
NEQ	100B	Not-equal	A ≠ B		True	No
NLT	101B	Not-less-than	NOT(A < B)		True	Yes
NLE	110B	Not-less-than-or-equal	NOT(A ≤ B)		True	Yes
		Not-greater-than	NOT(A > B)	Swap Operands, Use NLT	True	Yes
		Not-greater-than-or-equal	NOT(A ≥ B)	Swap Operands, Use NLE	True	Yes
ORD	111B	Ordered	A , B = Ordered		False	No

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 the 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. 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 Table 3-7 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 3-8. Compiler should treat reserved Imm8 values as illegal syntax.

Table 3-8 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.)

In 64-bit mode, use of the REX.R prefix permits this instruction to access additional registers (XMM8-XMM15).

Enhanced Comparison Predicate for VEX-Encoded VCMPPD

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 (VLMAX-1:128) of the destination YMM register are zeroed. Two comparisons are performed with results written to bits 127:0 of the destination operand.

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

The 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 3-9). Bits 5 through 7 of the immediate are reserved.

Table 3-9 Comparison Predicate for VCMPPD and VCMPPS 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

Table 3-9 Comparison Predicate for VCMPPD and VCMPPS 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]	
LT_OS (LT)	1H	Less-than (ordered, signaling)	False	True	False	False	Yes
LE_OS (LE)	2H	Less-than-or-equal (ordered, signaling)	False	True	True	False	Yes
UNORD_Q (UNORD)	3H	Unordered (non-signaling)	False	False	False	True	No
NEQ_UQ (NEQ)	4H	Not-equal (unordered, non-signaling)	True	True	False	True	No
NLT_US (NLT)	5H	Not-less-than (unordered, signaling)	True	False	True	True	Yes
NLE_US (NLE)	6H	Not-less-than-or-equal (unordered, signaling)	True	False	False	True	Yes
ORD_Q (ORD)	7H	Ordered (non-signaling)	True	True	True	False	No
EQ_UQ	8H	Equal (unordered, non-signaling)	False	False	True	True	No
NGE_US (NGE)	9H	Not-greater-than-or-equal (unordered, signaling)	False	True	False	True	Yes
NGT_US (NGT)	AH	Not-greater-than (unordered, signaling)	False	True	True	True	Yes
FALSE_OQ(FALSE)	BH	False (ordered, non-signaling)	False	False	False	False	No
NEQ_OQ	CH	Not-equal (ordered, non-signaling)	True	True	False	False	No
GE_OS (GE)	DH	Greater-than-or-equal (ordered, signaling)	True	False	True	False	Yes
GT_OS (GT)	EH	Greater-than (ordered, signaling)	True	False	False	False	Yes
TRUE_UQ(TRUE)	FH	True (unordered, non-signaling)	True	True	True	True	No
EQ_OS	10H	Equal (ordered, signaling)	False	False	True	False	Yes
LT_OQ	11H	Less-than (ordered, nonsignaling)	False	True	False	False	No

Table 3-9 Comparison Predicate for VCMPPD and VCMPPS 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]	
LE_OQ	12H	Less-than-or-equal (ordered, nonsignaling)	False	True	True	False	No
UNORD_S	13H	Unordered (signaling)	False	False	False	True	Yes
NEQ_US	14H	Not-equal (unordered, signaling)	True	True	False	True	Yes
NLT_UQ	15H	Not-less-than (unordered, nonsignaling)	True	False	True	True	No
NLE_UQ	16H	Not-less-than-or-equal (unordered, nonsignaling)	True	False	False	True	No
ORD_S	17H	Ordered (signaling)	True	True	True	False	Yes
EQ_US	18H	Equal (unordered, signaling)	False	False	True	True	Yes
NGE_UQ	19H	Not-greater-than-or-equal (unordered, nonsignaling)	False	True	False	True	No
NGT_UQ	1AH	Not-greater-than (unordered, nonsignaling)	False	True	True	True	No
FALSE_OS	1BH	False (ordered, signaling)	False	False	False	False	Yes
NEQ_OS	1CH	Not-equal (ordered, signaling)	True	True	False	False	Yes
GE_OQ	1DH	Greater-than-or-equal (ordered, nonsignaling)	True	False	True	False	No
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.

Processors with "CPUID.1H:ECX.AVX =1" implement the full complement of 32 predicates shown in Table 3-9, 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 3-10, 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 predefined constants to support a simpler intrinsic interface.

Table 3-10 Pseudo-Op and VCMPPD Implementation

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

Operation

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

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[VLMAX-1:128] (Unmodified)
```

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[VLMAX-1:128] ← 0
```

VCMPPD (VEX.256 encoded version)

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

Intel C/C++ Compiler Intrinsic Equivalents

CMPPD for equality __m128d _mm_cmpeq_pd(__m128d a, __m128d b)

CMPPD for less-than__m128d _mm_cmplt_pd(__m128d a, __m128d b)

CMPPD for less-than-or-equal__m128d _mm_cmple_pd(__m128d a, __m128d b)

CMPPD for greater-than__m128d _mm_cmpgt_pd(__m128d a, __m128d b)

CMPPD for greater-than-or-equal__m128d _mm_cmpge_pd(__m128d a, __m128d b)

CMPPD for inequality __m128d _mm_cmpneq_pd(__m128d a, __m128d b)

CMPPD for not-less-than __m128d _mm_cmpnlt_pd(__m128d a, __m128d b)

CMPPD for not-greater-than __m128d _mm_cmpngt_pd(__m128d a, __m128d b)

CMPPD for not-greater-than-or-equal__m128d _mm_cmpnge_pd(__m128d a, __m128d b)

CMPPD for ordered __m128d _mm_cmpord_pd(__m128d a, __m128d b)

CMPPD for unordered__m128d _mm_cmpunord_pd(__m128d a, __m128d b)

CMPPD for not-less-than-or-equal__m128d _mm_cmpnle_pd(__m128d a, __m128d b)

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 above table, Denormal.

Other Exceptions

See Exceptions Type 2.

...

CMPPS—Compare Packed Single-Precision Floating-Point Values

Opcode/ Instruction	Op/ En	64/32-bit Mode	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/mem* and *xmm1* using *imm8* as 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

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
A	ModRM:reg (r, w)	ModRM:r/m (r)	imm8	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 source operand (second operand) and the destination operand (first operand) and returns the results of the comparison to the destination operand. The comparison predicate operand (third operand) specifies the type of comparison performed on each of the pairs of packed values. The result of each comparison is a doubleword mask of all 1s (comparison true) or all 0s (comparison false).

128-bit Legacy SSE version: The first source and destination operand (first operand) is an XMM register. The second source operand (second operand) can be an XMM register or 128-bit memory location. The comparison predicate operand is an 8-bit immediate, bits 2:0 of the immediate define the type of comparison to be performed (see Table 3-7). Bits 7:3 of the immediate is reserved. Bits (VLMAX-1:128) of the corresponding YMM

destination register remain unchanged. Four comparisons are performed with results written to bits 127:0 of the destination operand.

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 a fault, 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 Table 3-7 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 3-11. Compiler should treat reserved Imm8 values as illegal syntax.

In 64-bit mode, use of the REX.R prefix permits this instruction to access additional registers (XMM8-XMM15).

Table 3-11 Pseudo-Ops and CMPPS

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

The greater-than relations not implemented by processor 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.)

Enhanced Comparison Predicate for VEX-Encoded VCMPPS

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 (VLMAX-1:128) of the destination YMM register are zeroed. Four comparisons are performed with results written to bits 127:0 of the destination operand.

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

location. The destination operand (first operand) is a YMM register. Eight comparisons are performed with results written to the destination operand.

The 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 3-9). Bits 5 through 7 of the immediate are reserved.

Processors with "CPUID.1H:ECX.AVX =1" implement the full complement of 32 predicates shown in Table 3-9, 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 3-12, 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 3-12 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*
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*

Table 3-12 Pseudo-Op and VCMPPS Implementation

Pseudo-Op	CMPPS Implementation
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
EASC;
```

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[VLMAX-1:128] (Unmodified)
```

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[VLMAX-1:128] ← 0
```

VCMPPS (VEX.256 encoded version)

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

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

Intel C/C++ Compiler Intrinsic Equivalents

CMPPS for equality __m128 _mm_cmpeq_ps(__m128 a, __m128 b)

CMPPS for less-than__m128 _mm_cmplt_ps(__m128 a, __m128 b)

CMPPS for less-than-or-equal__m128 _mm_cmple_ps(__m128 a, __m128 b)

CMPPS for greater-than __m128 _mm_cmpgt_ps(__m128 a, __m128 b)

CMPPS for greater-than-or-equal__m128 _mm_cmpge_ps(__m128 a, __m128 b)

CMPPS for inequality __m128 _mm_cmpneq_ps(__m128 a, __m128 b)

CMPPS for not-less-than __m128 _mm_cmpnlt_ps(__m128 a, __m128 b)

CMPPS for not-greater-than __m128 _mm_cmpngt_ps(__m128 a, __m128 b)

CMPPS for not-greater-than-or-equal__m128 _mm_cmpnge_ps(__m128 a, __m128 b)

CMPPS for ordered__m128 _mm_cmpord_ps(__m128 a, __m128 b)

CMPPS for unordered__m128 _mm_cmpunord_ps(__m128 a, __m128 b)

CMPPS for not-less-than-or-equal__m128 _mm_cmpnle_ps(__m128 a, __m128 b)

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 above table, Denormal.

Other Exceptions

See Exceptions Type 2.

...

CMPSD—Compare Scalar Double-Precision Floating-Point Values

Opcode/ Instruction	Op/ En	64/32-bit Mode	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 *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

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
A	ModRM:reg (r, w)	ModRM:r/m (r)	imm8	NA
B	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA

Description

Compares the low double-precision floating-point values in the source operand (second operand) and the destination operand (first operand) and returns the results of the comparison to the destination operand. The comparison predicate operand (third operand) specifies the type of comparison performed. The comparison result is a quad-word 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. The comparison predicate operand is an 8-bit immediate, bits 2:0 of the immediate define the type of comparison to be performed (see Table 3-7). Bits 7:3 of the immediate is reserved. Bits (VLMAX-1:64) of the corresponding YMM destination register remain unchanged.

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 a fault, 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 operand), and then perform the compare using a different predicate. The predicate to be used for these emulations is listed in Table 3-7 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 3-13. Compiler should treat reserved Imm8 values as illegal syntax.

Table 3-13 Pseudo-Ops and CMPSD

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

The greater-than relations not implemented in the processor 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.)

In 64-bit mode, use of the REX.R prefix permits this instruction to access additional registers (XMM8-XMM15).

Enhanced Comparison Predicate for VEX-Encoded VCMPSD

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. Bits (VLMAX-1:128) 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 3-9). Bits 5 through 7 of the immediate are reserved.

Processors with "CPUID.1H:ECX.AVX =1" implement the full complement of 32 predicates shown in Table 3-9, 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 3-14, 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 3-14 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*

Table 3-14 Pseudo-Op and VCMPSD Implementation

Pseudo-Op	CMPSD Implementation
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*
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[VLMAX-1:64] (Unmodified)

VCMPSD (VEX.128 encoded version)
CMP0 ← SRC1[63:0] OP5 SRC2[63:0];
IF CMP0 = TRUE
THEN DEST[63:0] ← FFFFFFFFFFFFFFFFH;
ELSE DEST[63:0] ← 0000000000000000H; FI;
DEST[127:64] ← SRC1[127:64]
DEST[VLMAX-1:128] ← 0

Intel C/C++ Compiler Intrinsic Equivalents

CMPSD for equality__m128d _mm_cmpeq_sd(__m128d a, __m128d b)

CMPSD for less-than__m128d _mm_cmplt_sd(__m128d a, __m128d b)

CMPSD for less-than-or-equal__m128d _mm_cmple_sd(__m128d a, __m128d b)

CMPSD for greater-than__m128d _mm_cmpgt_sd(__m128d a, __m128d b)

CMPSD for greater-than-or-equal__m128d _mm_cmpge_sd(__m128d a, __m128d b)

CMPSD for inequality__m128d _mm_cmpneq_sd(__m128d a, __m128d b)

CMPSD for not-less-than__m128d _mm_cmpnlt_sd(__m128d a, __m128d b)

CMPSD for not-greater-than__m128d _mm_cmpngt_sd(__m128d a, __m128d b)

CMPSD for not-greater-than-or-equal__m128d _mm_cmpnge_sd(__m128d a, __m128d b)

CMPSD for ordered__m128d _mm_cmpord_sd(__m128d a, __m128d b)

CMPSD for unordered__m128d _mm_cmpunord_sd(__m128d a, __m128d b)

CMPSD for not-less-than-or-equal__m128d _mm_cmpnle_sd(__m128d a, __m128d b)

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 above table, Denormal.

Other Exceptions

See Exceptions Type 3.

...

CMPSS—Compare Scalar Single-Precision Floating-Point Values

Opcode/ Instruction	Op/ En	64/32-bit Mode	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 *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

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
A	ModRM:reg (r, w)	ModRM:r/m (r)	imm8	NA
B	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA

Description

Compares the low single-precision floating-point values in the source operand (second operand) and the destination operand (first operand) and returns the results of the comparison to the destination operand. The comparison predicate operand (third

operand) specifies the type of comparison performed. The comparison result is a double-word 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. The comparison predicate operand is an 8-bit immediate, bits 2:0 of the immediate define the type of comparison to be performed (see Table 3-7). Bits 7:3 of the immediate is reserved. Bits (VLMAX-1:32) of the corresponding YMM destination register remain unchanged.

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 a fault, since 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 operand), and then perform the compare using a different predicate. The predicate to be used for these emulations is listed in Table 3-7 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 3-15. Compiler should treat reserved Imm8 values as illegal syntax.

Table 3-15 Pseudo-Ops and CMPSS

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

The greater-than relations not implemented in the processor 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.)

In 64-bit mode, use of the REX.R prefix permits this instruction to access additional registers (XMM8-XMM15).

Enhanced Comparison Predicate for VEX-Encoded VCMPSD

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. Bits (VLMAX-1:128) 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 3-9). Bits 5 through 7 of the immediate are reserved.

Processors with "CPUID.1H:ECX.AVX =1" implement the full complement of 32 predicates shown in Table 3-9, 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 3-16, 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 predefined constants to support a simpler intrinsic interface.

Table 3-16 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*
VCMPNGE_UQSS r*eg1, reg2, reg3*	VCMPSS r*eg1, reg2, reg3, 19H*

Table 3-16 Pseudo-Op and VCMPSS Implementation

Pseudo-Op	CMPSS Implementation
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[VLMAX-1:32] (Unmodified)
```

VCMPSS (VEX.128 encoded version)

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

Intel C/C++ Compiler Intrinsic Equivalents

CMPSS for equality__m128 _mm_cmpeq_ss(__m128 a, __m128 b)

CMPSS for less-than__m128 _mm_cmplt_ss(__m128 a, __m128 b)

CMPSS for less-than-or-equal__m128 _mm_cmple_ss(__m128 a, __m128 b)

CMPSS for greater-than__m128 _mm_cmpgt_ss(__m128 a, __m128 b)

CMPSS for greater-than-or-equal__m128 _mm_cmpge_ss(__m128 a, __m128 b)

CMPSS for inequality__m128 _mm_cmpneq_ss(__m128 a, __m128 b)

CMPSS for not-less-than__m128 _mm_cmpnlt_ss(__m128 a, __m128 b)

CMPSS for not-greater-than__m128 _mm_cmpngt_ss(__m128 a, __m128 b)

CMPSS for not-greater-than-or-equal__m128 _mm_cmpnge_ss(__m128 a, __m128 b)

CMPSS for ordered__m128 _mm_cmpord_ss(__m128 a, __m128 b)

CMPSS for unordered__m128 _mm_cmpunord_ss(__m128 a, __m128 b)

CMPSS for not-less-than-or-equal__m128 _mm_cmpnle_ss(__m128 a, __m128 b)

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 above table, 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	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

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
A	ModRM:reg (r)	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.

In 64-bit mode, use of the REX.R prefix permits this instruction to access additional registers (XMM8-XMM15).

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

Operation

```
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 Equivalents

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	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

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

Description

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

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

The 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.

In 64-bit mode, use of the REX.R prefix permits this instruction to access additional registers (XMM8-XMM15).

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

Operation

```
RESULT ← OrderedCompare(SRC1[31:0] <> SRC2[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 Equivalents

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.

...

CPUID—CPU Identification

...

Table 3-17 Information Returned by CPUID Instruction

Initial EAX Value	Information Provided about the Processor	
	Basic CPUID Information	
0H	EAX EBX ECX EDX	Maximum Input Value for Basic CPUID Information (see Table 3-18) "Genu" "ntel" "inel"
01H	EAX EBX ECX EDX	Version Information: Type, Family, Model, and Stepping ID (see Figure 3-9) Bits 07-00: Brand Index Bits 15-08: 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 Feature Information (see Figure 3-10 and Table 3-20) Feature Information (see Figure 3-11 and Table 3-21) **NOTES:** * The nearest power-of-2 integer that is not smaller than EBX[23:16] is the number of unique initial APIC IDs reserved for addressing different logical processors in a physical package.
02H	EAX EBX ECX EDX	Cache and TLB Information (see Table Table 3-17) Cache and TLB Information Cache and TLB Information Cache and TLB Information
03H	EAX EBX ECX EDX	Reserved. Reserved. Bits 00-31 of 96 bit processor serial number. (Available in Pentium III processor only; otherwise, the value in this register is reserved.) 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_ENABLE.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 2-144.	

Initial EAX Value	Information Provided about the Processor	
	EAX	Bits 04-00: Cache Type Field 0 = Null - No more caches 1 = Data Cache 2 = Instruction Cache 3 = Unified Cache 4-31 = Reserved Bits 07-05: Cache Level (starts at 1) Bit 08: Self Initializing cache level (does not need SW initialization) Bit 09: Fully Associative cache Bits 13-10: Reserved Bits 25-14: Maximum number of addressable IDs for logical processors sharing this cache*, ** Bits 31-26: Maximum number of addressable IDs for processor cores in the physical package*, ***, ****
	EBX	Bits 11-00: L = System Coherency Line Size* Bits 21-12: P = Physical Line partitions* Bits 31-22: W = Ways of associativity*
	ECX	Bits 31-00: S = Number of Sets*
	EDX	Bit 0: Write-Back Invalidate/Invalidate 0 = WBINVD/INVD from threads sharing this cache acts upon lower level caches for threads sharing this cache. 1 = WBINVD/INVD is not guaranteed to act upon lower level caches of non-originating threads sharing this cache. Bit 1: Cache Inclusiveness 0 = Cache is not inclusive of lower cache levels. 1 = Cache is inclusive of lower cache levels. Bit 2: Complex Cache Indexing 0 = Direct mapped cache. 1 = A complex function is used to index the cache, potentially using all address bits. Bits 31-03: Reserved = 0 **NOTES:** * Add one to the return value to get the result. ** The nearest power-of-2 integer that is not smaller than (1 + EAX[25:14]) is the number of unique initial APIC IDs reserved for addressing different logical processors sharing this cache *** The nearest power-of-2 integer that is not smaller than (1 + EAX[31:26]) is the number of unique Core_IDs reserved for addressing different processor cores in a physical package. Core ID is a subset of bits of the initial APIC ID. ****The returned value is constant for valid initial values in ECX. Valid ECX values start from 0.
	MONITOR/MWAIT Leaf	
05H	EAX	Bits 15-00: Smallest monitor-line size in bytes (default is processor's monitor granularity) Bits 31-16: Reserved = 0

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	Bit 00: Enumeration of Monitor-Mwait extensions (beyond EAX and EBX registers) supported Bit 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		
06H	EAX	Bit 00: Digital temperature sensor is supported if set Bit 01: Intel Turbo Boost Technology Available (see description of IA32_MISC_ENABLE[38]). Bit 02: ARAT. APIC-Timer-always-running feature is supported if set. Bit 03: Reserved Bit 04: PLN. Power limit notification controls are supported if set. Bit 05: ECMD. Clock modulation duty cycle extension is supported if set. Bit 06: PTM. Package thermal management is supported if set. Bits 31 - 07: Reserved
	EBX	Bits 03 - 00: Number of Interrupt Thresholds in Digital Thermal Sensor Bits 31 - 04: Reserved
	ECX	Bit 00: Hardware Coordination Feedback Capability (Presence of IA32_MPERF and IA32_APERF). 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
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

Initial EAX Value	Information Provided about the Processor	
	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
	EBX	Bit 00: Core cycle event not available if 1 Bit 01: Instruction retired event not available if 1 Bit 02: Reference cycles event not available if 1 Bit 03: Last-level cache reference event not available if 1 Bit 04: Last-level cache misses event not available if 1 Bit 05: Branch instruction retired event not available if 1 Bit 06: Branch mispredict retired event not available if 1 Bits 31- 07: Reserved = 0
	ECX	Reserved = 0
	EDX	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		**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. Leaf 0BH exists if EBX[15:0] is not zero.
	EAX	Bits 04-00: Number of bits to shift right on x2APIC ID to get a unique topology ID of the next level type*. All logical processors with the same next level ID share current level. Bits 31-05: Reserved.
	EBX	Bits 15 - 00: Number of logical processors at this level type. The number reflects configuration as shipped by Intel**. Bits 31- 16: Reserved.
	ECX	Bits 07 - 00: Level number. Same value in ECX input Bits 15 - 08: Level type***. Bits 31 - 16:: Reserved.
	EDX	Bits 31- 00: x2APIC ID the current logical processor.
		NOTES: * Software should use this field (EAX[4:0]) to enumerate processor topology of the system.

Initial EAX Value	Information Provided about the Processor
	** 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 31-00: Reports the valid bit fields of the lower 32 bits of XCR0. If a bit is 0, the corresponding bit field in XCR0 is reserved. EBX — Bits 31-00: Maximum size (bytes, from the beginning of the XSAVE/XRSTOR save area) required by enabled features in XCR0. May be different than ECX if some features at the end of the XSAVE save area are not enabled. ECX — Bit 31-00: Maximum size (bytes, from the beginning of the XSAVE/XRSTOR save area) of the XSAVE/XRSTOR save area required by all supported features in the processor, i.e all the valid bit fields in XCR0. EDX — Bit 31-00: Reports the valid bit fields of the upper 32 bits of XCR0. If a bit is 0, the corresponding bit field in XCR0 is reserved.
Processor Extended State Enumeration Sub-leaf (EAX = 0DH, ECX = 1)	
	EAX — Reserved *EBX* — Reserved *ECX* — Reserved *EDX* — Reserved
Processor Extended State Enumeration Sub-leaves (EAX = 0DH, ECX = n, n > 1)	
0DH	**NOTES:** Leaf 0DH output depends on the initial value in ECX. If ECX contains an invalid sub leaf index, EAX/EBX/ECX/EDX return 0. Each valid sub-leaf index maps to a valid bit in the XCR0 register starting at bit position 2 EAX — Bits 31-0: The size in bytes (from the offset specified in EBX) of the save area for an extended state feature associated with a valid sub-leaf index, *n*. This field reports 0 if the sub-leaf index, *n*, is invalid*.

Initial EAX Value	Information Provided about the Processor	
	EBX	Bits 31-0: The offset in bytes of this extended state component's save area from the beginning of the XSAVE/XRSTOR area. This field reports 0 if the sub-leaf index, *n*, is invalid*.
	ECX	This field reports 0 if the sub-leaf index, *n*, is invalid*; otherwise it is reserved.
	EDX	This field reports 0 if the sub-leaf index, *n*, is invalid*; otherwise it is reserved.
Unimplemented CPUID Leaf Functions		
40000000H - 4FFFFFFFH		Invalid. No existing or future CPU will return processor identification or feature information if the initial EAX value is in the range 40000000H to 4FFFFFFFH.
Extended Function CPUID Information		
80000000H	EAX	Maximum Input Value for Extended Function CPUID Information (see Table 3-18).
	EBX	Reserved
	ECX	Reserved
	EDX	Reserved
80000001H	EAX	Extended Processor Signature and Feature Bits.
	EBX	Reserved
	ECX	Bit 00: LAHF/SAHF available in 64-bit mode Bits 31-01 Reserved
	EDX	Bits 10-00: Reserved Bit 11: SYSCALL/SYSRET available (when in 64-bit mode) Bits 19-12: Reserved = 0 Bit 20: Execute Disable Bit available Bits 25-21: Reserved = 0 Bit 26: 1-GByte pages are available if 1 Bit 27: RDTSCP and IA32_TSC_AUX are available if 1 Bits 28: 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

Initial EAX Value	Information Provided about the Processor
80000005H	EAX Reserved = 0 EBX Reserved = 0 ECX Reserved = 0 EDX Reserved = 0
80000006H	EAX Reserved = 0 EBX Reserved = 0 ECX Bits 07-00: Cache Line size in bytes Bits 11-08: Reserved Bits 15-12: L2 Associativity field * Bits 31-16: Cache size in 1K units EDX Reserved = 0 **NOTES:** * L2 associativity field encodings: 00H - Disabled 01H - Direct mapped 02H - 2-way 04H - 4-way 06H - 8-way 08H - 16-way 0FH - Fully associative
80000007H	EAX Reserved = 0 EBX Reserved = 0 ECX Reserved = 0 EDX Bits 07-00: Reserved = 0 Bit 08: Invariant TSC available if 1 Bits 31-09: Reserved = 0
80000008H	EAX Linear/Physical Address size Bits 07-00: #Physical Address Bits* Bits 15-8: #Linear 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.

...

Table 3-14 Processor Type Field

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

NOTE

See Chapter 14 in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*, for information on identifying earlier IA-32 processors.

...

Table 3-15 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	**PCLMULQDQ.** A value of 1 indicates the processor supports the PCLMULQDQ instruction
2	DTES64	**64-bit DS Area.** A value of 1 indicates the processor supports DS area using 64-bit layout
3	MONITOR	**MONITOR/MWAIT.** A value of 1 indicates the processor supports this feature.
4	DS-CPL	**CPL Qualified Debug Store.** A value of 1 indicates the processor supports the extensions to the Debug Store feature to allow for branch message storage qualified by CPL.
5	VMX	**Virtual Machine Extensions.** A value of 1 indicates that the processor supports this technology
6	SMX	**Safer Mode Extensions.** A value of 1 indicates that the processor supports this technology. See Chapter 6, "Safer Mode Extensions Reference".
7	EIST	**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

Bit #	Mnemonic	Description
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. See the "CMPXCHG8B/CMPXCHG16B—Compare and Exchange Bytes" section in this chapter for a description.
14	xTPR Update Control	**xTPR Update Control.** A value of 1 indicates that the processor supports changing IA32_MISC_ENABLE[bit 23].
15	PDCM	**Perfmon and Debug Capability:** A value of 1 indicates the processor supports the performance and debug feature indication MSR IA32_PERF_CAPABILITIES.
16	Reserved	Reserved
17	PCID	**Process-context identifiers.** A value of 1 indicates that the processor supports PCIDs and that software may set CR4.PCIDE to 1.
18	DCA	A value of 1 indicates the processor supports the ability to prefetch data from a memory mapped device.
19	SSE4.1	A value of 1 indicates that the processor supports SSE4.1.
20	SSE4.2	A value of 1 indicates that the processor supports SSE4.2.
21	x2APIC	A value of 1 indicates that the processor supports x2APIC feature.
22	MOVBE	A value of 1 indicates that the processor supports MOVBE instruction.
23	POPCNT	A value of 1 indicates that the processor supports the POPCNT instruction.
24	TSC-Deadline	A value of 1 indicates that the processor's local APIC timer supports one-shot operation using a TSC deadline value.
25	AESNI	A value of 1 indicates that the processor supports the AESNI instruction extensions.
26	XSAVE	A value of 1 indicates that the processor supports the XSAVE/ XRSTOR processor extended states feature, the XSETBV/ XGETBV instructions, and XCR0.
27	OSXSAVE	A value of 1 indicates that the OS has enabled XSETBV/XGETBV instructions to access XCR0, and support for processor extended state management using XSAVE/XRSTOR.
28	AVX	A value of 1 indicates the processor supports the AVX instruction extensions.
30 - 29	Reserved	Reserved
31	Not Used	Always returns 0

…

Table 3-17 Encoding of CPUID Leaf 2 Descriptors

Value	Type	Description
00H	General	Null descriptor, this byte contains no information
01H	TLB	Instruction TLB: 4 KByte pages, 4-way set associative, 32 entries
02H	TLB	Instruction TLB: 4 MByte pages, fully associative, 2 entries

Table 3-17 Encoding of CPUID Leaf 2 Descriptors (Continued)

Value	Type	Description
03H	TLB	Data TLB: 4 KByte pages, 4-way set associative, 64 entries
04H	TLB	Data TLB: 4 MByte pages, 4-way set associative, 8 entries
05H	TLB	Data TLB1: 4 MByte pages, 4-way set associative, 32 entries
06H	Cache	1st-level instruction cache: 8 KBytes, 4-way set associative, 32 byte line size
08H	Cache	1st-level instruction cache: 16 KBytes, 4-way set associative, 32 byte line size
09H	Cache	1st-level instruction cache: 32KBytes, 4-way set associative, 64 byte line size
0AH	Cache	1st-level data cache: 8 KBytes, 2-way set associative, 32 byte line size
0BH	TLB	Instruction TLB: 4 MByte pages, 4-way set associative, 4 entries
0CH	Cache	1st-level data cache: 16 KBytes, 4-way set associative, 32 byte line size
0DH	Cache	1st-level data cache: 16 KBytes, 4-way set associative, 64 byte line size
0EH	Cache	1st-level data cache: 24 KBytes, 6-way set associative, 64 byte line size
21H	Cache	2nd-level cache: 256 KBytes, 8-way set associative, 64 byte line size
22H	Cache	3rd-level cache: 512 KBytes, 4-way set associative, 64 byte line size, 2 lines per sector
23H	Cache	3rd-level cache: 1 MBytes, 8-way set associative, 64 byte line size, 2 lines per sector
25H	Cache	3rd-level cache: 2 MBytes, 8-way set associative, 64 byte line size, 2 lines per sector
29H	Cache	3rd-level cache: 4 MBytes, 8-way set associative, 64 byte line size, 2 lines per sector
2CH	Cache	1st-level data cache: 32 KBytes, 8-way set associative, 64 byte line size
30H	Cache	1st-level instruction cache: 32 KBytes, 8-way set associative, 64 byte line size
40H	Cache	No 2nd-level cache or, if processor contains a valid 2nd-level cache, no 3rd-level cache
41H	Cache	2nd-level cache: 128 KBytes, 4-way set associative, 32 byte line size
42H	Cache	2nd-level cache: 256 KBytes, 4-way set associative, 32 byte line size
43H	Cache	2nd-level cache: 512 KBytes, 4-way set associative, 32 byte line size
44H	Cache	2nd-level cache: 1 MByte, 4-way set associative, 32 byte line size
45H	Cache	2nd-level cache: 2 MByte, 4-way set associative, 32 byte line size
46H	Cache	3rd-level cache: 4 MByte, 4-way set associative, 64 byte line size
47H	Cache	3rd-level cache: 8 MByte, 8-way set associative, 64 byte line size
48H	Cache	2nd-level cache: 3MByte, 12-way set associative, 64 byte line size
49H	Cache	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	Cache	3rd-level cache: 6MByte, 12-way set associative, 64 byte line size
4BH	Cache	3rd-level cache: 8MByte, 16-way set associative, 64 byte line size
4CH	Cache	3rd-level cache: 12MByte, 12-way set associative, 64 byte line size

Table 3-17 Encoding of CPUID Leaf 2 Descriptors (Continued)

Value	Type	Description
4DH	Cache	3rd-level cache: 16MByte, 16-way set associative, 64 byte line size
4EH	Cache	2nd-level cache: 6MByte, 24-way set associative, 64 byte line size
4FH	TLB	Instruction TLB: 4 KByte pages, 32 entries
50H	TLB	Instruction TLB: 4 KByte and 2-MByte or 4-MByte pages, 64 entries
51H	TLB	Instruction TLB: 4 KByte and 2-MByte or 4-MByte pages, 128 entries
52H	TLB	Instruction TLB: 4 KByte and 2-MByte or 4-MByte pages, 256 entries
55H	TLB	Instruction TLB: 2-MByte or 4-MByte pages, fully associative, 7 entries
56H	TLB	Data TLB0: 4 MByte pages, 4-way set associative, 16 entries
57H	TLB	Data TLB0: 4 KByte pages, 4-way associative, 16 entries
59H	TLB	Data TLB0: 4 KByte pages, fully associative, 16 entries
5AH	TLB	Data TLB0: 2-MByte or 4 MByte pages, 4-way set associative, 32 entries
5BH	TLB	Data TLB: 4 KByte and 4 MByte pages, 64 entries
5CH	TLB	Data TLB: 4 KByte and 4 MByte pages,128 entries
5DH	TLB	Data TLB: 4 KByte and 4 MByte pages,256 entries
60H	Cache	1st-level data cache: 16 KByte, 8-way set associative, 64 byte line size
66H	Cache	1st-level data cache: 8 KByte, 4-way set associative, 64 byte line size
67H	Cache	1st-level data cache: 16 KByte, 4-way set associative, 64 byte line size
68H	Cache	1st-level data cache: 32 KByte, 4-way set associative, 64 byte line size
70H	Cache	Trace cache: 12 K-µop, 8-way set associative
71H	Cache	Trace cache: 16 K-µop, 8-way set associative
72H	Cache	Trace cache: 32 K-µop, 8-way set associative
76H	TLB	Instruction TLB: 2M/4M pages, fully associative, 8 entries
78H	Cache	2nd-level cache: 1 MByte, 4-way set associative, 64byte line size
79H	Cache	2nd-level cache: 128 KByte, 8-way set associative, 64 byte line size, 2 lines per sector
7AH	Cache	2nd-level cache: 256 KByte, 8-way set associative, 64 byte line size, 2 lines per sector
7BH	Cache	2nd-level cache: 512 KByte, 8-way set associative, 64 byte line size, 2 lines per sector
7CH	Cache	2nd-level cache: 1 MByte, 8-way set associative, 64 byte line size, 2 lines per sector
7DH	Cache	2nd-level cache: 2 MByte, 8-way set associative, 64byte line size
7FH	Cache	2nd-level cache: 512 KByte, 2-way set associative, 64-byte line size
80H	Cache	2nd-level cache: 512 KByte, 8-way set associative, 64-byte line size
82H	Cache	2nd-level cache: 256 KByte, 8-way set associative, 32 byte line size
83H	Cache	2nd-level cache: 512 KByte, 8-way set associative, 32 byte line size
84H	Cache	2nd-level cache: 1 MByte, 8-way set associative, 32 byte line size
85H	Cache	2nd-level cache: 2 MByte, 8-way set associative, 32 byte line size
86H	Cache	2nd-level cache: 512 KByte, 4-way set associative, 64 byte line size

Table 3-17 Encoding of CPUID Leaf 2 Descriptors (Continued)

Value	Type	Description
87H	Cache	2nd-level cache: 1 MByte, 8-way set associative, 64 byte line size
B0H	TLB	Instruction TLB: 4 KByte pages, 4-way set associative, 128 entries
B1H	TLB	Instruction TLB: 2M pages, 4-way, 8 entries or 4M pages, 4-way, 4 entries
B2H	TLB	Instruction TLB: 4KByte pages, 4-way set associative, 64 entries
B3H	TLB	Data TLB: 4 KByte pages, 4-way set associative, 128 entries
B4H	TLB	Data TLB1: 4 KByte pages, 4-way associative, 256 entries
BAH	TLB	Data TLB1: 4 KByte pages, 4-way associative, 64 entries
C0H	TLB	Data TLB: 4 KByte and 4 MByte pages, 4-way associative, 8 entries
CAH	STLB	Shared 2nd-Level TLB: 4 KByte pages, 4-way associative, 512 entries
D0H	Cache	3rd-level cache: 512 KByte, 4-way set associative, 64 byte line size
D1H	Cache	3rd-level cache: 1 MByte, 4-way set associative, 64 byte line size
D2H	Cache	3rd-level cache: 2 MByte, 4-way set associative, 64 byte line size
D6H	Cache	3rd-level cache: 1 MByte, 8-way set associative, 64 byte line size
D7H	Cache	3rd-level cache: 2 MByte, 8-way set associative, 64 byte line size
D8H	Cache	3rd-level cache: 4 MByte, 8-way set associative, 64 byte line size
DCH	Cache	3rd-level cache: 1.5 MByte, 12-way set associative, 64 byte line size
DDH	Cache	3rd-level cache: 3 MByte, 12-way set associative, 64 byte line size
DEH	Cache	3rd-level cache: 6 MByte, 12-way set associative, 64 byte line size
E2H	Cache	3rd-level cache: 2 MByte, 16-way set associative, 64 byte line size
E3H	Cache	3rd-level cache: 4 MByte, 16-way set associative, 64 byte line size
E4H	Cache	3rd-level cache: 8 MByte, 16-way set associative, 64 byte line size
EAH	Cache	3rd-level cache: 12MByte, 24-way set associative, 64 byte line size
EBH	Cache	3rd-level cache: 18MByte, 24-way set associative, 64 byte line size
ECH	Cache	3rd-level cache: 24MByte, 24-way set associative, 64 byte line size
F0H	Prefetch	64-Byte prefetching
F1H	Prefetch	128-Byte prefetching
FFH	General	CPUID leaf 2 does not report cache descriptor information, use CPUID leaf 4 to query cache parameters

...

INPUT EAX = 04H: Returns Deterministic Cache Parameters for Each Level

When CPUID executes with EAX set to 04H 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 3-12.

This Cache Size in Bytes

= (Ways + 1) * (Partitions + 1) * (Line_Size + 1) * (Sets + 1)

= (EBX[31:22] + 1) * (EBX[21:12] + 1) * (EBX[11:0] + 1) * (ECX + 1)

...

CRC32 — Accumulate CRC32 Value

...

Operation

Notes:

BIT_REFLECT64: DST[63-0] = SRC[0-63]
BIT_REFLECT32: DST[31-0] = SRC[0-31]
BIT_REFLECT16: DST[15-0] = SRC[0-15]
BIT_REFLECT8: DST[7-0] = SRC[0-7]
MOD2: Remainder from Polynomial division modulus 2

CRC32 instruction for 64-bit source operand and 64-bit destination operand:

TEMP1[63-0] ← BIT_REFLECT64 (SRC[63-0])
TEMP2[31-0] ← BIT_REFLECT32 (DEST[31-0])
TEMP3[95-0] ← TEMP1[63-0] « 32
TEMP4[95-0] ← TEMP2[31-0] « 64
TEMP5[95-0] ← TEMP3[95-0] XOR TEMP4[95-0]
TEMP6[31-0] ← TEMP5[95-0] MOD2 11EDC6F41H
DEST[31-0] ← BIT_REFLECT (TEMP6[31-0])
DEST[63-32] ← 00000000H

CRC32 instruction for 32-bit source operand and 32-bit destination operand:

TEMP1[31-0] ← BIT_REFLECT32 (SRC[31-0])
TEMP2[31-0] ← BIT_REFLECT32 (DEST[31-0])
TEMP3[63-0] ← TEMP1[31-0] « 32
TEMP4[63-0] ← TEMP2[31-0] « 32
TEMP5[63-0] ← TEMP3[63-0] XOR TEMP4[63-0]
TEMP6[31-0] ← TEMP5[63-0] MOD2 11EDC6F41H
DEST[31-0] ← BIT_REFLECT (TEMP6[31-0])

CRC32 instruction for 16-bit source operand and 32-bit destination operand:

TEMP1[15-0] ← BIT_REFLECT16 (SRC[15-0])
TEMP2[31-0] ← BIT_REFLECT32 (DEST[31-0])
TEMP3[47-0] ← TEMP1[15-0] « 32
TEMP4[47-0] ← TEMP2[31-0] « 16
TEMP5[47-0] ← TEMP3[47-0] XOR TEMP4[47-0]
TEMP6[31-0] ← TEMP5[47-0] MOD2 11EDC6F41H
DEST[31-0] ← BIT_REFLECT (TEMP6[31-0])

CRC32 instruction for 8-bit source operand and 64-bit destination operand:

TEMP1[7-0] ← BIT_REFLECT8(SRC[7-0])
TEMP2[31-0] ← BIT_REFLECT32 (DEST[31-0])
TEMP3[39-0] ← TEMP1[7-0] « 32

```
TEMP4[39-0] ← TEMP2[31-0] « 8
TEMP5[39-0] ← TEMP3[39-0] XOR TEMP4[39-0]
TEMP6[31-0] ← TEMP5[39-0] MOD2 11EDC6F41H
DEST[31-0] ← BIT_REFLECT (TEMP6[31-0])
DEST[63-32] ← 00000000H
```

CRC32 instruction for 8-bit source operand and 32-bit destination operand:

```
TEMP1[7-0] ← BIT_REFLECT8(SRC[7-0])
TEMP2[31-0] ← BIT_REFLECT32 (DEST[31-0])
TEMP3[39-0] ← TEMP1[7-0] « 32
TEMP4[39-0] ← TEMP2[31-0] « 8
TEMP5[39-0] ← TEMP3[39-0] XOR TEMP4[39-0]
TEMP6[31-0] ← TEMP5[39-0] MOD2 11EDC6F41H
DEST[31-0] ← BIT_REFLECT (TEMP6[31-0])
```

Flags Affected

None

Intel C/C++ Compiler Intrinsic Equivalent

unsigned int _mm_crc32_u8(unsigned int crc, unsigned char data)
unsigned int _mm_crc32_u16(unsigned int crc, unsigned short data)
unsigned int _mm_crc32_u32(unsigned int crc, unsigned int data)
unsigned __int64 _mm_crc32_u64(unsigned __int64 crc, unsigned __int64 data)

SIMD Floating Point Exceptions

None

Protected Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS or GS segments.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF (fault-code)	For a page fault.
#UD	If CPUID.01H:ECX.SSE4_2 [Bit 20] = 0.
	If LOCK prefix is used.

Real Mode Exceptions

#GP(0)	If any part of the operand lies outside of the effective address space from 0 to 0FFFFH.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#UD	If CPUID.01H:ECX.SSE4_2 [Bit 20] = 0.
	If LOCK prefix is used.

Virtual 8086 Mode Exceptions

#GP(0)	If any part of the operand lies outside of the effective address space from 0 to 0FFFFH.

#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF (fault-code)	For a page fault.
#UD	If CPUID.01H:ECX.SSE4_2 [Bit 20] = 0.
	If LOCK prefix is used.

Compatibility Mode Exceptions

Same exceptions as in Protected Mode.

64-Bit Mode Exceptions

#GP(0)	If the memory address is in a non-canonical form.
#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#PF (fault-code)	For a page fault.
#UD	If CPUID.01H:ECX.SSE4_2 [Bit 20] = 0.
	If LOCK prefix is used.

...

CVTDQ2PD—Convert Packed Dword Integers to Packed Double-Precision FP Values

Opcode/ Instruction	Op/ En	64/32-bit Mode	CPUID Feature Flag	Description
F3 0F E6 CVTDQ2PD *xmm1, xmm2/m64*	A	V/V	SSE2	Convert two packed signed doubleword integers from *xmm2/m128* 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

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

Description

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

In 64-bit mode, use of the REX.R prefix permits this instruction to access additional registers (XMM8-XMM15).

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 (VLMAX-1:128) of the corresponding YMM register destination are unmodified.

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 (VLMAX-1:128) of the corresponding YMM register destination are zeroed.

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

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

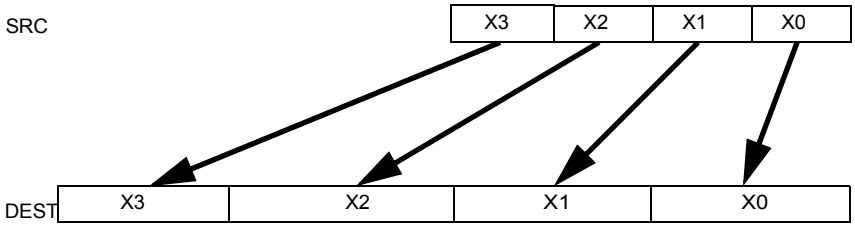


Figure 3-14 CVTDQ2PD (VEX.256 encoded version)

Operation

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[VLMAX-1:128] (unmodified)

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[VLMAX-1:128] ← 0

VCVTDQ2PD (VEX.256 encoded version)
DEST[63:0] ← Convert_Integer_To_Double_Precision_Floating_Point(SRC[31:0])
DEST[127:64] ← Convert_Integer_To_Double_Precision_Floating_Point(SRC[63:32])
DEST[191:128] ← Convert_Integer_To_Double_Precision_Floating_Point(SRC[95:64])
DEST[255:192] ← Convert_Integer_To_Double_Precision_Floating_Point(SRC[127:96)

Intel C/C++ Compiler Intrinsic Equivalent

CVTDQ2PD __m128d _mm_cvtepi32_pd(__m128i a)

VCVTDQ2PD __m256d _mm256_cvtepi32_pd (__m128i src)

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Exceptions Type 5; additionally

#UD	If VEX.vvvv != 1111B.

...

CVTDQ2PS—Convert Packed Dword Integers to Packed Single-Precision FP Values

Opcode/ Instruction	Op/ En	64/32-bit Mode	CPUID Feature Flag	Description
0F 5B /r CVTDQ2PS *xmm1, xmm2/m128*	A	V/V	SSE2	Convert four packed signed doubleword integers from *xmm2/m128* 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

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

Description

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

In 64-bit mode, use of the REX.R prefix permits this instruction to access additional registers (XMM8-XMM15).

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 (VLMAX-1:128) of the corresponding YMM register destination are unmodified.

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 (VLMAX-1:128) of the corresponding YMM register destination are zeroed.

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

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

Operation

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[VLMAX-1:128] (unmodified)

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[VLMAX-1:128] ← 0

VCVTDQ2PS (VEX.256 encoded version)
DEST[31:0] ← Convert_Integer_To_Single_Precision_Floating_Point(SRC[31:0])
DEST[63:32] ← Convert_Integer_To_Single_Precision_Floating_Point(SRC[63:32])
DEST[95:64] ← Convert_Integer_To_Single_Precision_Floating_Point(SRC[95:64])
DEST[127:96] ← Convert_Integer_To_Single_Precision_Floating_Point(SRC[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)

Intel C/C++ Compiler Intrinsic Equivalent

CVTDQ2PS __m128 _mm_cvtepi32_ps(__m128i a)

VCVTDQ2PS __m256 _mm256_cvtepi32_ps (__m256i src)

SIMD Floating-Point Exceptions

Precision.

Other Exceptions

See Exceptions Type 2; additionally

#UD	If VEX.vvvv != 1111B.

...

CVTPD2DQ—Convert Packed Double-Precision FP Values to Packed Dword Integers

Opcode/ Instruction	Op/ En	64/32-bit Mode	CPUID Feature Flag	Description
F2 0F E6 CVTPD2DQ *xmm1, xmm2/m128*	A	V/V	SSE2	Convert two packed double-precision floating-point values from *xmm2/m128* to two packed 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

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

Description

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

The source operand can be an XMM register or a 128-bit memory location. The destination operand is an XMM register. The result is stored in the low quadword of the destination operand and the high quadword is cleared to all 0s.

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.

In 64-bit mode, use of the REX.R prefix permits this instruction to access additional registers (XMM8-XMM15).

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 (VLMAX-1:128) of the corresponding YMM register destination are unmodified.

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 (VLMAX-1:64) of the corresponding YMM register destination are zeroed.

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.

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

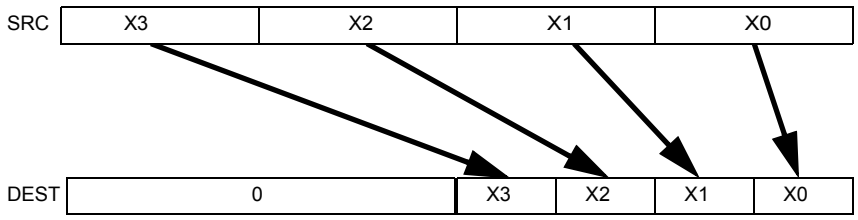


Figure 3-15 VCVTPD2DQ (VEX.256 encoded version)

Operation

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[VLMAX-1:128] (unmodified)

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[VLMAX-1:64] ← 0

VCVTPD2DQ (VEX.256 encoded version)
DEST[31:0] ← Convert_Double_Precision_Floating_Point_To_Integer(SRC[63:0])
DEST[63:32] ← Convert_Double_Precision_Floating_Point_To_Integer(SRC[127:64])
DEST[95:64] ← Convert_Double_Precision_Floating_Point_To_Integer(SRC[191:128])
DEST[127:96] ← Convert_Double_Precision_Floating_Point_To_Integer(SRC[255:192)
DEST[255:128]← 0

Intel C/C++ Compiler Intrinsic Equivalent

CVTPD2DQ __m128i _mm_cvtpd_epi32 (__m128d src)

CVTPD2DQ __m128i _mm256_cvtpd_epi32 (__m256d src)

SIMD Floating-Point Exceptions

Invalid, Precision.

Other Exceptions

See Exceptions Type 2; additionally

#UD	If VEX.vvvv != 1111B.

...

CVTPD2PS—Convert Packed Double-Precision FP Values to Packed Single-Precision FP Values

Opcode/ Instruction	Op/ En	64/32-bit Mode	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/m128* to two packed 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

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

Description

Converts two packed double-precision floating-point values in the source operand (second operand) to two 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.

In 64-bit mode, use of the REX.R prefix permits this instruction to access additional registers (XMM8-XMM15).

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 (VLMAX-1:128) of the corresponding YMM register destination are unmodified.

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 (VLMAX-1:64) of the corresponding YMM register destination are zeroed.

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.

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

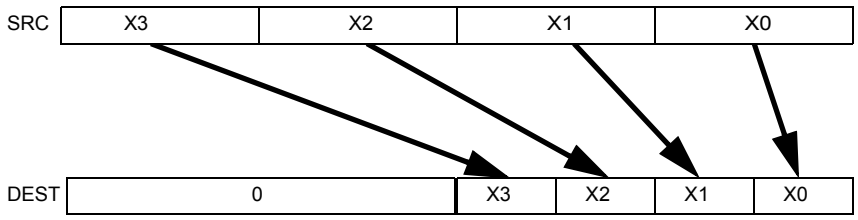


Figure 2-1. VCVTPD2PS (VEX.256 encoded version)

Operation

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[VLMAX-1:128] (unmodified)

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[VLMAX-1:64] ← 0

VCVTPD2PS (VEX.256 encoded version)
DEST[31:0] ← Convert_Double_Precision_To_Single_Precision_Floating_Point(SRC[63:0])
DEST[63:32] ← Convert_Double_Precision_To_Single_Precision_Floating_Point(SRC[127:64])
DEST[95:64] ← Convert_Double_Precision_To_Single_Precision_Floating_Point(SRC[191:128])
DEST[127:96] ← Convert_Double_Precision_To_Single_Precision_Floating_Point(SRC[255:192)
DEST[255:128]← 0

Intel C/C++ Compiler Intrinsic Equivalent

CVTPD2PS __m128 _mm_cvtpd_ps(__m128d a)

CVTPD2PS __m256 _mm256_cvtpd_ps (__m256d a)

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal.

Other Exceptions

See Exceptions Type 2; additionally

#UD	If VEX.vvvv != 1111B.

...

CVTPS2DQ—Convert Packed Single-Precision FP Values to Packed Dword Integers

Opcode/ Instruction	Op/ En	64/32-bit Mode	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/m128* to four packed signed doubleword integers 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

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
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.

In 64-bit mode, use of the REX.R prefix permits this instruction to access additional registers (XMM8-XMM15).

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 (VLMAX-1:128) of the corresponding YMM register destination are unmodified.

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 (VLMAX-1:128) of the corresponding YMM register destination are zeroed.

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

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

Operation

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[VLMAX-1:128] (unmodified)

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[VLMAX-1:128] ← 0

VCVTPS2DQ (VEX.256 encoded version)
DEST[31:0] ← Convert_Single_Precision_Floating_Point_To_Integer(SRC[31:0])
DEST[63:32] ← Convert_Single_Precision_Floating_Point_To_Integer(SRC[63:32])
DEST[95:64] ← Convert_Single_Precision_Floating_Point_To_Integer(SRC[95:64])
DEST[127:96] ← Convert_Single_Precision_Floating_Point_To_Integer(SRC[127:96)
DEST[159:128] ← Convert_Single_Precision_Floating_Point_To_Integer(SRC[159:128])
DEST[191:160] ← Convert_Single_Precision_Floating_Point_To_Integer(SRC[191:160])
DEST[223:192] ← Convert_Single_Precision_Floating_Point_To_Integer(SRC[223:192])
DEST[255:224] ← Convert_Single_Precision_Floating_Point_To_Integer(SRC[255:224])

Intel C/C++ Compiler Intrinsic Equivalent

CVTPS2DQ __m128i _mm_cvtps_epi32(__m128 a)

VCVTPS2DQ __ m256i _mm256_cvtps_epi32 (__m256 a)

SIMD Floating-Point Exceptions

Invalid, Precision.

Other Exceptions

See Exceptions Type 2; additionally

#UD	If VEX.vvvv != 1111B.

...

CVTPS2PD—Convert Packed Single-Precision FP Values to Packed Double-

Precision FP Values

Opcode/ Instruction	Op/ En	64/32-bit Mode	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/m64* to two packed double-precision floating-point values in *xmm1*.
VEX.128.0F.WIG 5A /r VCVTPS2PD xmm1, xmm2/m64	A	V/V	AVX	Convert two packed single-precision floating-point values in xmm2/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

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
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).

In 64-bit mode, use of the REX.R prefix permits this instruction to access additional registers (XMM8-XMM15).

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 (VLMAX-1:128) of the corresponding YMM register destination are unmodified.

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 (VLMAX-1:128) of the corresponding YMM register destination are zeroed.

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

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

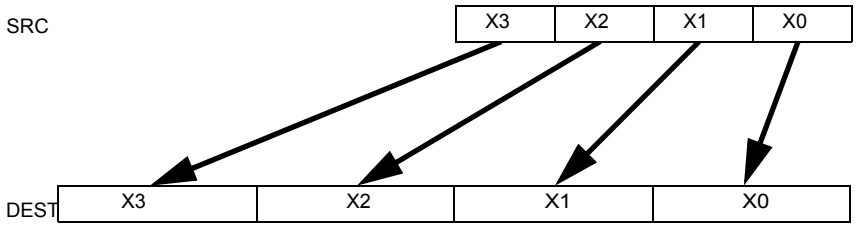


Figure 3-17 CVTPS2PD (VEX.256 encoded version)

Operation

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[VLMAX-1:128] (unmodified)

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[VLMAX-1:128] ← 0

VCVTPS2PD (VEX.256 encoded version)
DEST[63:0] ← Convert_Single_Precision_To_Double_Precision_Floating_Point(SRC[31:0])
DEST[127:64] ← Convert_Single_Precision_To_Double_Precision_Floating_Point(SRC[63:32])
DEST[191:128] ← Convert_Single_Precision_To_Double_Precision_Floating_Point(SRC[95:64])
DEST[255:192] ← Convert_Single_Precision_To_Double_Precision_Floating_Point(SRC[127:96)

Intel C/C++ Compiler Intrinsic Equivalent

CVTPS2PD __m128d _mm_cvtps_pd(__m128 a)

VCVTPS2PD __m256d _mm256_cvtps_pd (__m128 a)

SIMD Floating-Point Exceptions

Invalid, Denormal.

Other Exceptions

See Exceptions Type 3; additionally

#UDIf VEX.vvvv != 1111B.

...

CVTSD2SI—Convert Scalar Double-Precision FP Value to Integer

Opcode/ Instruction	Op/ En	64/32-bit Mode	CPUID Feature Flag	Description
F2 0F 2D /*r* CVTSD2SI *r32, xmm/m64*	A	V/V	SSE2	Convert one double-precision floating-point value from *xmm/m64* to one signed doubleword integer *r32*.
F2 REX.W 0F 2D /*r* CVTSD2SI *r64, xmm/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

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
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.

In 64-bit mode, the instruction can access additional registers (XMM8-XMM15, R8-R15) when used with a REX.R prefix. 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.

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

```
IF 64-Bit Mode and OperandSize = 64
    THEN
        DEST[63:0] ← Convert_Double_Precision_Floating_Point_To_Integer64(SRC[63:0]);
    ELSE
        DEST[31:0] ← Convert_Double_Precision_Floating_Point_To_Integer32(SRC[63:0]);
FI;
```

Intel C/C++ Compiler Intrinsic Equivalent

int _mm_cvtsd_si32(__m128d a)

__int64 _mm_cvtsd_si64(__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 FP Value to Scalar Single-Precision FP Value

Opcode/ Instruction	Op/ En	64/32-bit Mode	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

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
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 source operand (second operand) to a single-precision floating-point value in the destination operand (first operand).

The source operand can be an XMM register or a 64-bit memory location. The destination operand is an XMM register. When the 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 left unchanged. When the conversion is inexact, the value returned is rounded according to the rounding control bits in the MXCSR register.

In 64-bit mode, use of the REX.R prefix permits this instruction to access additional registers (XMM8-XMM15).

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

Operation

CVTSD2SS (128-bit Legacy SSE version)

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

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[VLMAX-1:128] ← 0
```

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 Dword Integer to Scalar Double-Precision FP Value

Opcode/ Instruction	Op/ En	64/32-bit Mode	CPUID Feature Flag	Description
F2 0F 2A /*r* CVTSI2SD *xmm, r/m32*	A	V/V	SSE2	Convert one signed doubleword integer from *r/m32* to one double-precision floating-point value in *xmm*.
F2 REX.W 0F 2A /*r* CVTSI2SD *xmm, r/m64*	A	V/N.E.	SSE2	Convert one signed quadword integer from *r/m64* to one double-precision floating-point value in *xmm*.
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

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
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 (VLMAX-1:64) 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 (VLMAX-1:128) of the destination YMM register are zeroed.

Operation

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[VLMAX-1:64] (Unmodified)
```

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[VLMAX-1:128] ← 0
```

Intel C/C++ Compiler Intrinsic Equivalent

CVTSI2SD __m128d _mm_cvtsi32_sd(__m128d a, int b)

CVTSI2SD __m128d _mm_cvtsi64_sd(__m128d a, __int64 b)

SIMD Floating-Point Exceptions

Precision.

Other Exceptions

See Exceptions Type 3.

...

CVTSI2SS—Convert Dword Integer to Scalar Single-Precision FP Value

Opcode/ Instruction	Op/ En	64/32-bit Mode	CPUID Feature Flag	Description
F3 0F 2A /*r* CVTSI2SS *xmm, r/m32*	A	V/V	SSE	Convert one signed doubleword integer from *r/m32* to one single-precision floating-point value in *xmm*.
F3 REX.W 0F 2A /*r* CVTSI2SS *xmm, r/m64*	A	V/N.E.	SSE	Convert one signed quadword integer from *r/m64* to one single-precision floating-point value in *xmm*.
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

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
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 double-words 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, the instruction can access additional registers (XMM8-XMM15, R8-R15) when used with a REX.R prefix. 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.

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

Operation

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[VLMAX-1:32] (Unmodified)
```

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[VLMAX-1:128] ← 0
```

Intel C/C++ Compiler Intrinsic Equivalent

CVTSI2SS __m128 _mm_cvtsi32_ss(__m128 a, int b)

CVTSI2SS __m128 _mm_cvtsi64_ss(__m128 a, __int64 b)

SIMD Floating-Point Exceptions

Precision.

Other Exceptions

See Exceptions Type 3.

...

CVTSS2SD—Convert Scalar Single-Precision FP Value to Scalar Double-Precision FP Value

Opcode/ Instruction	Op/ En	64/32-bit Mode	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

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
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 source operand (second operand) to a double-precision floating-point value in the destination operand (first operand). The source operand can be an XMM register or a 32-bit memory location. The destination operand is an XMM register. When the 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 left unchanged.

In 64-bit mode, use of the REX.R prefix permits this instruction to access additional registers (XMM8-XMM15).

128-bit Legacy SSE version: The destination and first source operand are the same. Bits (VLMAX-1:64) 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 (VLMAX-1:128) of the destination YMM register are zeroed.

Operation

CVTSS2SD (128-bit Legacy SSE version)

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

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[VLMAX-1:128] ← 0
```

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 FP Value to Dword Integer

Opcode/ Instruction	Op/ En	64/32-bit Mode	CPUID Feature Flag	Description
F3 0F 2D /*r* CVTSS2SI *r32, xmm/m32*	A	V/V	SSE	Convert one single-precision floating-point value from *xmm/m32* to one signed doubleword integer in *r32*.
F3 REX.W 0F 2D /*r* CVTSS2SI *r64, xmm/m32*	A	V/N.E.	SSE	Convert one single-precision floating-point value from *xmm/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

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
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.

In 64-bit mode, the instruction can access additional registers (XMM8-XMM15, R8-R15) when used with a REX.R prefix. 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.

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

```
IF 64-bit Mode and OperandSize = 64
    THEN
        DEST[64:0] ← 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(__m128d a)

__int64 _mm_cvtss_si64(__m128d a)

SIMD Floating-Point Exceptions

Invalid, Precision.

Other Exceptions

See Exceptions Type 3; additionally

#UFD	If VEX.vvvv != 1111B.

...

CVTTPD2DQ—Convert with Truncation Packed Double-Precision FP Values to Packed Dword Integers

Opcode/ Instruction	Op/ En	64/32-bit Mode	CPUID Feature Flag	Description
66 0F E6 CVTTPD2DQ *xmm1, xmm2/m128*	A	V/V	SSE2	Convert two packed double-precision floating-point values from *xmm2/m128* to two packed 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

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
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.

In 64-bit mode, use of the REX.R prefix permits this instruction to access additional registers (XMM8-XMM15).

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 (VLMAX-1:128) of the corresponding YMM register destination are unmodified.

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 (VLMAX-1:128) of the corresponding YMM register destination are zeroed.

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.

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

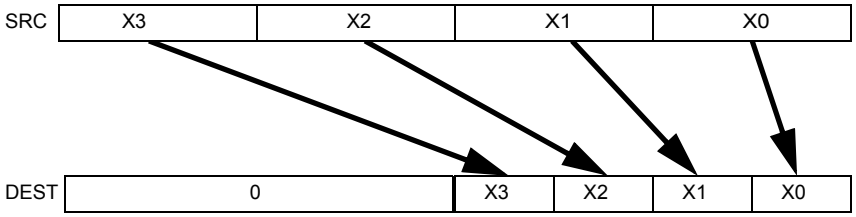


Figure 3-18 VCVTTPD2DQ (VEX.256 encoded version)

Operation

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[VLMAX-1:128] (unmodified)

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[VLMAX-1:64] ← 0

VCVTTPD2DQ (VEX.256 encoded version)
DEST[31:0] ← Convert_Double_Precision_Floating_Point_To_Integer_Truncate(SRC[63:0])
DEST[63:32] ← Convert_Double_Precision_Floating_Point_To_Integer_Truncate(SRC[127:64])
DEST[95:64] ← Convert_Double_Precision_Floating_Point_To_Integer_Truncate(SRC[191:128])
DEST[127:96] ← Convert_Double_Precision_Floating_Point_To_Integer_Truncate(SRC[255:192)
DEST[255:128]← 0

Intel C/C++ Compiler Intrinsic Equivalent

CVTTPD2DQ __m128i _mm_cvttpd_epi32(__m128d a)

VCVTTPD2DQ __m128i _mm256_cvttpd_epi32 (__m256d src)

SIMD Floating-Point Exceptions

Invalid, Precision.

Other Exceptions

See Exceptions Type 2; additionally

#UFD	If VEX.vvvv != 1111B.

...

CVTTPS2DQ—Convert with Truncation Packed Single-Precision FP Values to Packed Dword Integers

Opcode/ Instruction	Op/ En	64/32-bit Mode	CPUID Feature Flag	Description
F3 0F 5B /*r* CVTTPS2DQ *xmm1, xmm2/m128*	A	V/V	SSE2	Convert four single-precision floating-point values from *xmm2/m128* to four signed doubleword integers 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

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
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.

In 64-bit mode, use of the REX.R prefix permits this instruction to access additional registers (XMM8-XMM15).

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 (VLMAX-1:128) of the corresponding YMM register destination are unmodified.

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 (VLMAX-1:128) of the corresponding YMM register destination are zeroed.

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

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

Operation

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[VLMAX-1:128] (unmodified)

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[VLMAX-1:128] ← 0

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

Intel C/C++ Compiler Intrinsic Equivalent

CVTTPS2DQ __m128i _mm_cvttps_epi32(__m128 a)

VCVTTPS2DQ __m256i _mm256_cvttps_epi32 (__m256 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 FP Value to Signed Integer

Opcode/ Instruction	Op/ En	64/32-bit Mode	CPUID Feature Flag	Description
F2 0F 2C /*r* CVTTSD2SI *r32, xmm/m64*	A	V/V	SSE2	Convert one double-precision floating-point value from *xmm/m64* to one signed doubleword integer in *r32* using truncation.
F2 REX.W 0F 2C /*r* CVTTSD2SI *r64, xmm/m64*	A	V/N.E.	SSE2	Convert one double precision floating-point value from *xmm/m64* to one signedquadword 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

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
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

```
IF 64-Bit Mode and OperandSize = 64
    THEN
        DEST[63:0] ← Convert_Double_Precision_Floating_Point_To_
                        Integer64_Truncate(SRC[63:0]);
    ELSE
        DEST[31:0] ← Convert_Double_Precision_Floating_Point_To_
                        Integer32_Truncate(SRC[63:0]);
FI;
```

Intel C/C++ Compiler Intrinsic Equivalent

int _mm_cvttsd_si32(__m128d a)

__int64 _mm_cvttsd_si64(__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 FP Value to Dword Integer

Opcode/ Instruction	Op/ En	64/32-bit Mode	CPUID Feature Flag	Description
F3 0F 2C /r CVTTSS2SI *r32, xmm/m32*	A	V/V	SSE	Convert one single-precision floating-point value from *xmm/m32* to one signed doubleword integer in *r32* using truncation.
F3 REX.W 0F 2C /r CVTTSS2SI *r64, xmm/m32*	A	V/N.E.	SSE	Convert one single-precision floating-point value from *xmm/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

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
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, the instruction can access additional registers (XMM8-XMM15, R8-R15) when used with a REX.R prefix. 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

```
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(__m128d a)

__int64 _mm_cvttss_si64(__m128d 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	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 *xmm2/m128.*
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

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
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. See Chapter 11 in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*, for an overview of a SIMD double-precision floating-point operation.

In 64-bit mode, use of the REX.R prefix permits this instruction to access additional registers (XMM8-XMM15).

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 (VLMAX-1:128) of the corresponding YMM register destination are unmodified.

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 (VLMAX-1:128) of the corresponding YMM register destination are zeroed.

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

Operation

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[VLMAX-1:128] (Unmodified)

VDIVPD (VEX.128 encoded version)
DEST[63:0] ← SRC1[63:0] / SRC2[63:0]
DEST[127:64] ← SRC1[127:64] / SRC2[127:64]
DEST[VLMAX-1:128] ← 0

VDIVPD (VEX.256 encoded version)
DEST[63:0] ← SRC1[63:0] / SRC2[63:0]
DEST[127:64] ← SRC1[127:64] / SRC2[127:64]
DEST[191:128] ← SRC1[191:128] / SRC2[191:128]
DEST[255:192] ← SRC1[255:192] / SRC2[255:192]

Intel C/C++ Compiler Intrinsic Equivalent

DIVPD __m128d _mm_div_pd(__m128d a, __m128d b)

VDIVPD __m256d _mm256_div_pd (__m256d a, __m256d 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	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 single-precision floating-point values *xmm2/m128*.
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

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
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. See Chapter 10 in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*, for an overview of a SIMD single-precision floating-point operation.

In 64-bit mode, use of the REX.R prefix permits this instruction to access additional registers (XMM8-XMM15).

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 (VLMAX-1:128) of the corresponding YMM register destination are unmodified.

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 (VLMAX-1:128) of the corresponding YMM register destination are zeroed.

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

Operation

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[VLMAX-1:128] (Unmodified)

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[VLMAX-1:128] ← 0

VDIVPS (VEX.256 encoded version)
DEST[31:0] ← SRC1[31:0] / SRC2[31:0]
DEST[63:32] ← SRC1[63:32] / SRC2[63:32]
DEST[95:64] ← SRC1[95:64] / SRC2[95:64]
DEST[127:96] ← SRC1[127:96] / SRC2[127:96]
DEST[159:128] ← SRC1[159:128] / SRC2[159:128]
DEST[191:160]← SRC1[191:160] / SRC2[191:160]
DEST[223:192] ← SRC1[223:192] / SRC2[223:192]
DEST[255:224] ← SRC1[255:224] / SRC2[255:224].

Intel C/C++ Compiler Intrinsic Equivalent

DIVPS __m128 _mm_div_ps(__m128 a, __m128 b)

VDIVPS __m256 _mm256_div_ps (__m256 a, __m256 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	CPUID Feature Flag	Description
F2 0F 5E /r DIVSD *xmm1, xmm2/m64*	A	V/V	SSE2	Divide low double-precision floating-point value n *xmm1* by low double-precision floating-point value in *xmm2/mem64.*
VEX.NDS.LIG.F2.0F.WIG 5E /r VDIVSD xmm1, xmm2, xmm3/m64	A	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

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
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.

In 64-bit mode, use of the REX.R prefix permits this instruction to access additional registers (XMM8-XMM15).

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

VEX.128 encoded version: Bits (VLMAX-1:128) of the destination YMM register are zeroed.

Operation

DIVSD (128-bit Legacy SSE version)
DEST[63:0] ← DEST[63:0] / SRC[63:0]
DEST[VLMAX-1:64] (Unmodified)

VDIVSD (VEX.128 encoded version)
DEST[63:0] ← SRC1[63:0] / SRC2[63:0]
DEST[127:64] ← SRC1[127:64]
DEST[VLMAX-1:128] ← 0

Intel C/C++ Compiler 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	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

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
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.

In 64-bit mode, use of the REX.R prefix permits this instruction to access additional registers (XMM8-XMM15).

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

VEX.128 encoded version: Bits (VLMAX-1:128) of the destination YMM register are zeroed.

Operation

DIVSS (128-bit Legacy SSE version)
DEST[31:0] ← DEST[31:0] / SRC[31:0]
DEST[VLMAX-1:32] (Unmodified)

VDIVSS (VEX.128 encoded version)
DEST[31:0] ← SRC1[31:0] / SRC2[31:0]
DEST[127:32] ← SRC1[127:32]
DEST[VLMAX-1:128] ← 0

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	CPUID Feature Flag	Description
66 0F 3A 41 /r ib DPPD *xmm1, xmm2/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

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
A	ModRM:reg (r, w)	ModRM:r/m (r)	imm8	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 [5:4] of the immediate operand (third operand). If a condition mask bit is zero, the corresponding multiplication is replaced by a value of 0.0.

The two resulting double-precision values are summed into an intermediate result. The intermediate result is conditionally broadcasted to the destination using a broadcast mask specified by bits [1:0] of the immediate byte.

If a broadcast mask bit is "1", the intermediate result is copied to the corresponding qword element in the destination operand. If a broadcast mask bit is zero, the corresponding element in the destination is set to zero.

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 computationally generated NaNs will have at least one NaN propagated to the destination.

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 (VLMAX-1:128) of the corresponding YMM register destination are unmodified.

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 (VLMAX-1:128) of the corresponding YMM register destination are zeroed.

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] ← DEST[63:0] * SRC[63:0];
    ELSE Temp1[63:0] ← +0.0; FI;
IF (imm8[5] = 1)
    THEN Temp1[127:64] ← DEST[127:64] * SRC[127:64];
    ELSE Temp1[127:64] ← +0.0; FI;
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; FI;
IF (imm8[1] = 1)
    THEN DEST[127:64] ← Temp2[63:0];
```

ELSE DEST[127:64] ← +0.0; FI;

DPPD (128-bit Legacy SSE version)
DEST[127:0]←DP_Primitive(SRC1[127:0], SRC2[127:0]);
DEST[VLMAX-1:128] (Unmodified)

VDPPD (VEX.128 encoded version)
DEST[127:0]←DP_Primitive(SRC1[127:0], SRC2[127:0]);
DEST[VLMAX-1:128] ← 0

Flags Affected

None

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	CPUID Feature Flag	Description
66 0F 3A 40 /r ib DPPS *xmm1, xmm2/m128, imm8*	A	V/V	SSE4_1	Selectively multiply packed SP floating-point values from *xmm1* with packed SP floating-point values from *xmm2*, add and selectively store the packed SP floating-point values or zero values 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

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
A	ModRM:reg (r, w)	ModRM:r/m (r)	imm8	NA
B	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA

Description

Conditionally multiplies the packed single precision floating-point values in the destination operand (first operand) with the packed single-precision floats in the source (second operand) depending on a mask extracted from the high 4 bits of the immediate byte (third operand). If a condition mask bit in Imm8[7:4] is zero, the corresponding multiplication is replaced by a value of 0.0.

The four resulting single-precision values are summed into an intermediate result. The intermediate result is conditionally broadcasted to the destination using a broadcast mask specified by bits [3:0] of the immediate byte.

If a broadcast mask bit is "1", the intermediate result is copied to the corresponding dword element in the destination operand. If a broadcast mask bit is zero, the corresponding element in the destination is set to zero.

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 computationally generated NaNs will have at least one NaN propagated to the destination.

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 (VLMAX-1:128) of the corresponding YMM register destination are unmodified.

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 (VLMAX-1:128) of the corresponding YMM register destination are zeroed.

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

Operation

DP_primitive (SRC1, SRC2)

```
IF (imm8[4] = 1)
    THEN Temp1[31:0] ← DEST[31:0] * SRC[31:0];
    ELSE Temp1[31:0] ← +0.0; FI;
IF (imm8[5] = 1)
    THEN Temp1[63:32] ← DEST[63:32] * SRC[63:32];
    ELSE Temp1[63:32] ← +0.0; FI;
IF (imm8[6] = 1)
    THEN Temp1[95:64] ← DEST[95:64] * SRC[95:64];
    ELSE Temp1[95:64] ← +0.0; FI;
IF (imm8[7] = 1)
    THEN Temp1[127:96] ← DEST[127:96] * SRC[127:96];
    ELSE Temp1[127:96] ← +0.0; FI;

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; FI;
IF (imm8[1] = 1)
    THEN DEST[63:32] ← Temp4[31:0];
    ELSE DEST[63:32] ← +0.0; FI;
IF (imm8[2] = 1)
    THEN DEST[95:64] ← Temp4[31:0];
    ELSE DEST[95:64] ← +0.0; FI;
IF (imm8[3] = 1)
    THEN DEST[127:96] ← Temp4[31:0];
    ELSE DEST[127:96] ← +0.0; FI;
```

DPP (128-bit Legacy SSE version)

```
DEST[127:0]←DP_Primitive(SRC1[127:0], SRC2[127:0]);
DEST[VLMAX-1:128] (Unmodified)
```

VDPPS (VEX.128 encoded version)
DEST[127:0]←DP_Primitive(SRC1[127:0], SRC2[127:0]);
DEST[VLMAX-1:128] ← 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]);

Intel C/C++ Compiler Intrinsic Equivalent

(V)DPPS __m128 _mm_dp_ps (__m128 a, __m128 b, const int mask);

VDPPS __m256 _mm256_dp_ps (__m256 a, __m256 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	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

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
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[VLMAX-1:128] ← 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 Single Precision Floating-Point Value

Opcode/ Instruction	Op/ En	64/32-bit Mode	CPUID Feature Flag	Description
66 0F 3A 17 /r ib EXTRACTPS *reg/m32, xmm2, imm8*	A	V/V	SSE4_1	Extract a single-precision floating-point value from *xmm2* at the source offset specified by *imm8* and store the result to *reg or m32*. The upper 32 bits of r64 is zeroed if reg is r64.
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

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
A	ModRM:r/m (w)	ModRM:reg (r)	imm8	NA

Description

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

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

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

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.

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.

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

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
```

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
```

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.

...

FXRSTOR—Restore x87 FPU, MMX , XMM, and MXCSR State

...

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 memory operand is not aligned on a 16-byte boundary, regardless of segment. For an attempt to set reserved bits in MXCSR.
#PF(fault-code)	For a page fault.
#NM	If CR0.TS[bit 3] = 1. If CR0.EM[bit 2] = 1.
#UD	If CPUID.01H:EDX.FXSR[bit 24] = 0. If instruction is preceded by a LOCK prefix.
#AC	If this exception is disabled a general protection exception (#GP) is signaled if the memory operand is not aligned on a 16-byte boundary, as described above. If the alignment check exception (#AC) is enabled (and the CPL is 3), signaling of #AC is not guaranteed and may vary with implementation, as follows. In all implementations where #AC is not signaled, a general protection exception is signaled in its place. In addition, the width of the alignment check may also vary with implementation. For instance, for a given implementation, an alignment check exception might be signaled for a 2-byte misalignment, whereas a general protection exception might be signaled for all other misalignments (4-, 8-, or 16-byte misalignments).

...

FXSAVE—Save x87 FPU, MMX Technology, and SSE State

...

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 memory operand is not aligned on a 16-byte boundary, regardless of segment.

#PF(fault-code)	For a page fault.
#NM	If CR0.TS[bit 3] = 1. If CR0.EM[bit 2] = 1.
#UD	If CPUID.01H:EDX.FXSR[bit 24] = 0. If the LOCK prefix is used.
#AC	If this exception is disabled a general protection exception (#GP) is signaled if the memory operand is not aligned on a 16-byte boundary, as described above. If the alignment check exception (#AC) is enabled (and the CPL is 3), signaling of #AC is not guaranteed and may vary with implementation, as follows. In all implementations where #AC is not signaled, a general protection exception is signaled in its place. In addition, the width of the alignment check may also vary with implementation. For instance, for a given implementation, an alignment check exception might be signaled for a 2-byte misalignment, whereas a general protection exception might be signaled for all other misalignments (4-, 8-, or 16-byte misalignments).

...

HADDPD—Packed Double-FP Horizontal Add

Opcode/ Instruction	Op/ En	64/32-bit Mode	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 *xmm2/m128* to *xmm1*.
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

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
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 double-precision floating-point values in the high and low quadwords of the destination operand and stores the result in the low quadword of the destination operand.

Adds the double-precision floating-point values in the high and low quadwords of the source operand and stores the result in the high quadword of the destination operand.

In 64-bit mode, use of the REX.R prefix permits this instruction to access additional registers (XMM8-XMM15).

See Figure 3-19 for HADDPD; see Figure 3-20 for VHADDPD.

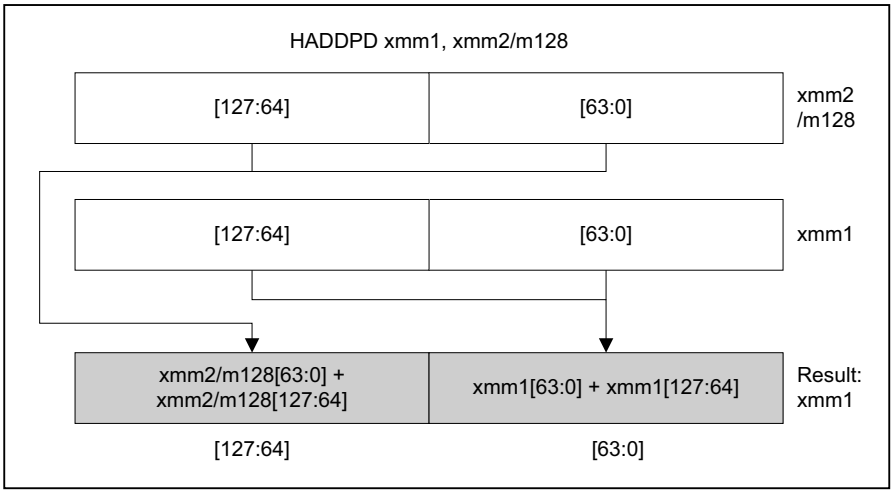


Figure 3-19 HADDPD—Packed Double-FP Horizontal Add

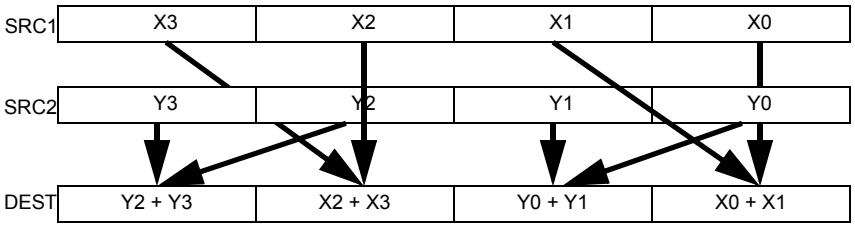


Figure 3-20 VHADDPD operation

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 (VLMAX-1:128) of the corresponding YMM register destination are unmodified.

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 (VLMAX-1:128) of the corresponding YMM register destination are zeroed.

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

Operation

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[VLMAX-1:128] (Unmodified)

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

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]

Intel C/C++ Compiler Intrinsic Equivalent

VHADDPD __m256d _mm256_hadd_pd (__m256d a, __m256d b);

HADDPD __m128d _mm_hadd_pd (__m128d a, __m128d b);

Exceptions

When the source operand is a memory operand, the operand must be aligned on a 16-byte boundary or a general-protection exception (#GP) will be generated.

Numeric Exceptions

Overflow, Underflow, Invalid, Precision, Denormal.

Other Exceptions

See Exceptions Type 2.

...

HADDPS—Packed Single-FP Horizontal Add

Opcode/ Instruction	Op/ En	64/32-bit Mode	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 *xmm2/m128* to *xmm1*.
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

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
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 single-precision floating-point values in the first and second dwords of the destination operand and stores the result in the first dword of the destination operand.

Adds single-precision floating-point values in the third and fourth dword of the destination operand and stores the result in the second dword of the destination operand.

Adds single-precision floating-point values in the first and second dword of the source operand and stores the result in the third dword of the destination operand.

Adds single-precision floating-point values in the third and fourth dword of the source operand and stores the result in the fourth dword of the destination operand.

In 64-bit mode, use of the REX.R prefix permits this instruction to access additional registers (XMM8-XMM15).

See Figure 3-21 for HADDPS; see Figure 3-22 for VHADDPS.

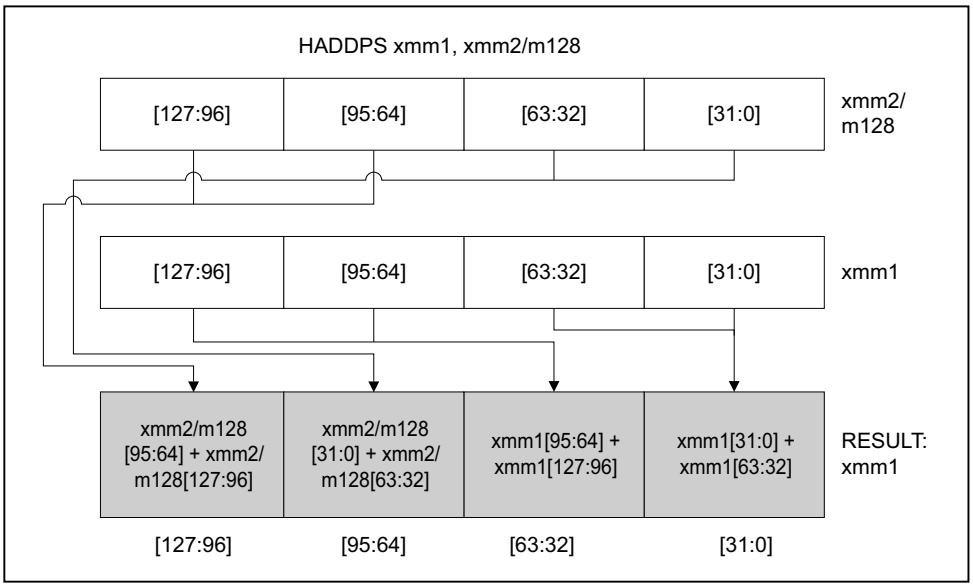


Figure 3-21 HADDPS—Packed Single-FP Horizontal Add

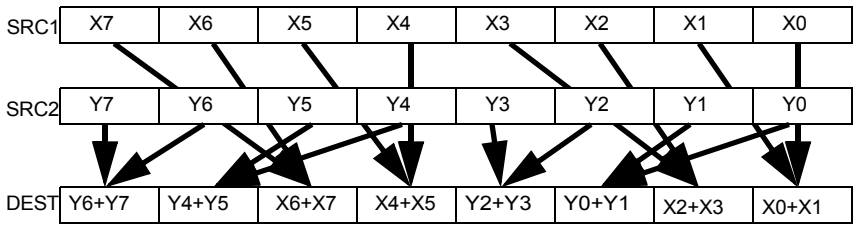


Figure 3-22 VHADDPS operation

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 (VLMAX-1:128) of the corresponding YMM register destination are unmodified.

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 (VLMAX-1:128) of the corresponding YMM register destination are zeroed.

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

Operation

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[VLMAX-1:128] (Unmodified)

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[VLMAX-1:128] ← 0

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]

Intel C/C++ Compiler Intrinsic Equivalent

HADDPS __m128 _mm_hadd_ps (__m128 a, __m128 b);

VHADDPS __m256 _mm256_hadd_ps (__m256 a, __m256 b);

Exceptions

When the source operand is a memory operand, the operand must be aligned on a 16-byte boundary or a general-protection exception (#GP) will be generated.

Numeric Exceptions

Overflow, Underflow, Invalid, Precision, Denormal.

Other Exceptions

See Exceptions Type 2.

...

HSUBPD—Packed Double-FP Horizontal Subtract

Opcode/ Instruction	Op/ En	64/32-bit Mode	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 *xmm2/m128* to *xmm1*.
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

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
A	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
B	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA

Description

The HSUBPD instruction subtracts horizontally the packed DP FP numbers of both operands.

Subtracts the double-precision floating-point value in the high quadword of the destination operand from the low quadword of the destination operand and stores the result in the low quadword of the destination operand.

Subtracts the double-precision floating-point value in the high quadword of the source operand from the low quadword of the source operand and stores the result in the high quadword of the destination operand.

In 64-bit mode, use of the REX.R prefix permits this instruction to access additional registers (XMM8-XMM15).

See Figure 3-23 for HSUBPD; see Figure 3-24 for VHSUBPD.

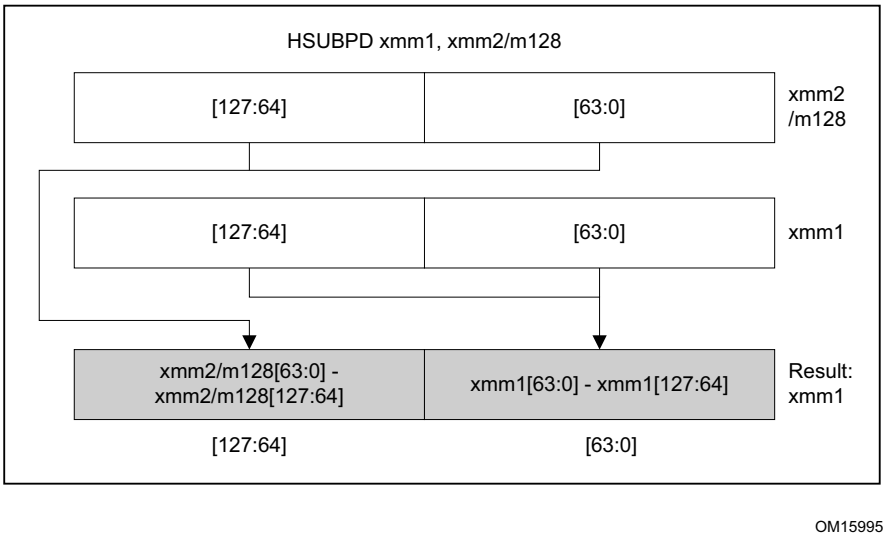


Figure 3-23 HSUBPD—Packed Double-FP Horizontal Subtract

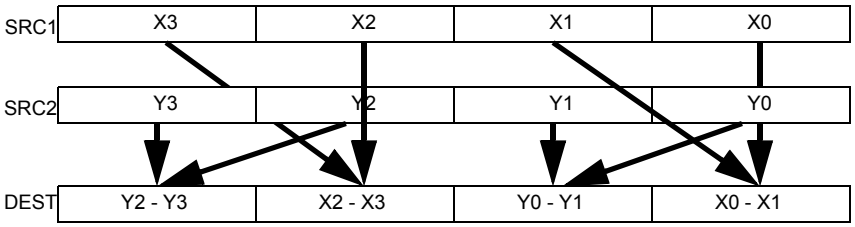


Figure 3-24 VHSUBPD operation

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 (VLMAX-1:128) of the corresponding YMM register destination are unmodified.

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 (VLMAX-1:128) of the corresponding YMM register destination are zeroed.

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

Operation

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[VLMAX-1:128] (Unmodified)

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

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]

Intel C/C++ Compiler Intrinsic Equivalent

HSUBPD __m128d _mm_hsub_pd(__m128d a, __m128d b)

VHSUBPD __m256d _mm256_hsub_pd (__m256d a, __m256d b);

Exceptions

When the source operand is a memory operand, the operand must be aligned on a 16-byte boundary or a general-protection exception (#GP) will be generated.

Numeric Exceptions

Overflow, Underflow, Invalid, Precision, Denormal.

Other Exceptions

See Exceptions Type 2.

...

HSUBPS—Packed Single-FP Horizontal Subtract

Opcode/ Instruction	Op/ En	64/32-bit Mode	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 *xmm2/m128* to *xmm1*.
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

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
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 single-precision floating-point value in the second dword of the destination operand from the first dword of the destination operand and stores the result in the first dword of the destination operand.

Subtracts the single-precision floating-point value in the fourth dword of the destination operand from the third dword of the destination operand and stores the result in the second dword of the destination operand.

Subtracts the single-precision floating-point value in the second dword of the source operand from the first dword of the source operand and stores the result in the third dword of the destination operand.

Subtracts the single-precision floating-point value in the fourth dword of the source operand from the third dword of the source operand and stores the result in the fourth dword of the destination operand.

In 64-bit mode, use of the REX.R prefix permits this instruction to access additional registers (XMM8-XMM15).

See Figure 3-25 for HSUBPS; see Figure 3-26 for VHSUBPS.

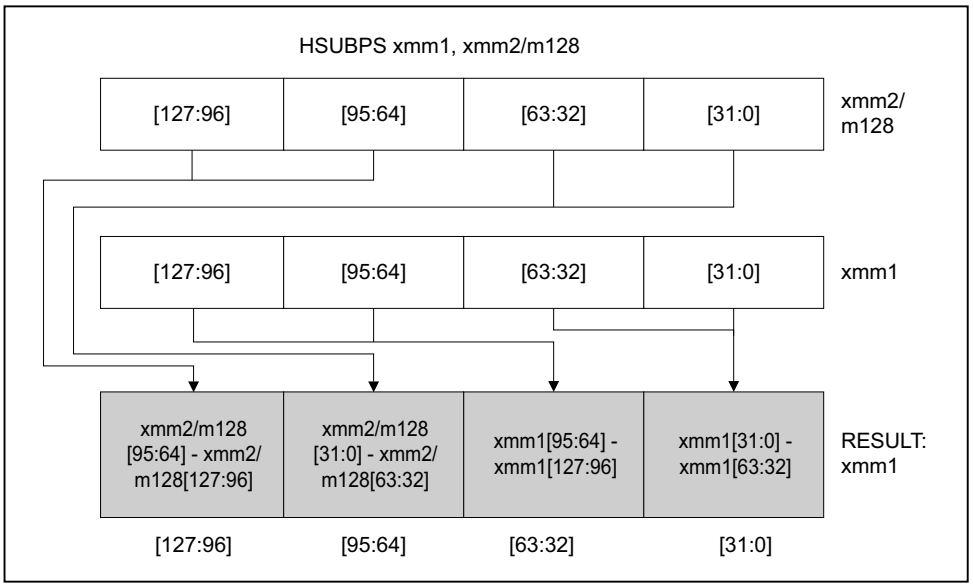


Figure 3-25 HSUBPS—Packed Single-FP Horizontal Subtract

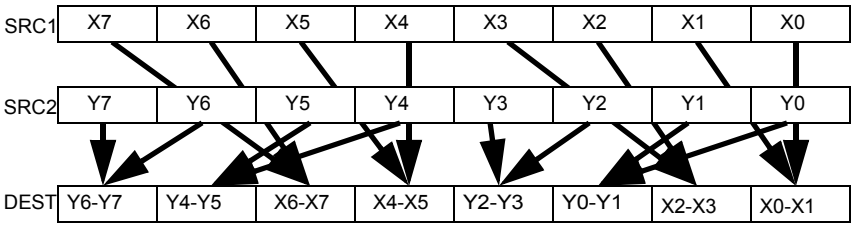


Figure 3-26 VHSUBPS operation

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 (VLMAX-1:128) of the corresponding YMM register destination are unmodified.

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 (VLMAX-1:128) of the corresponding YMM register destination are zeroed.

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

Operation

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[VLMAX-1:128] (Unmodified)

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[VLMAX-1:128] ← 0

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]

Intel C/C++ Compiler Intrinsic Equivalent

HSUBPS __m128 _mm_hsub_ps(__m128 a, __m128 b);

VHSUBPS __m256 _mm256_hsub_ps (__m256 a, __m256 b);

Exceptions

When the source operand is a memory operand, the operand must be aligned on a 16-byte boundary or a general-protection exception (#GP) will be generated.

Numeric 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	CPUID Feature Flag	Description
VEX.NDS.256.66.0F3A.W0 18 /r ib VINSERTF128 ymm1, ymm2, xmm3/m128, imm8	A	V/V	AVX	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*.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
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

```
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 Packed Single Precision Floating-Point Value

Opcode/ Instruction	Op/ En	64/32-bit Mode	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

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
A	ModRM:reg (w)	ModRM:r/m (r)	imm8	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 (VLMAX-1:128) 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 (VLMAX-1:128) 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

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[VLMAX-1:128] (Unmodified)
```

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[VLMAX-1:128] ← 0
```

Intel C/C++ Compiler Intrinsic Equivalent

INSERTPS __m128 _mm_insert_ps(__m128 dst, __m128 src, const int ndx);

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type 5.

...

INT *n*/INTO/INT 3—Call to Interrupt Procedure

...

Description

The INT *n* instruction generates a call to the interrupt or exception handler specified with the destination operand (see the section titled "Interrupts and Exceptions" in Chapter 6 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*). The destination operand specifies an interrupt vector number from 0 to 255, encoded as an

8-bit unsigned intermediate value. Each interrupt vector number provides an index to a gate descriptor in the IDT. The first 32 interrupt vector numbers are reserved by Intel for system use. Some of these interrupts are used for internally generated exceptions.

...

Operation

The following operational description applies not only to the INT *n* and INTO instructions, but also to external interrupts, nonmaskable interrupts (NMIs), and exceptions. Some of these events push onto the stack an error code.

The operational description specifies numerous checks whose failure may result in delivery of a nested exception. In these cases, the original event is not delivered.

The operational description specifies the error code delivered by any nested exception. In some cases, the error code is specified with a pseudofunction error_code(num,idt,ext), where idt and ext are bit values. The pseudofunction produces an error code as follows: (1) if idt is 0, the error code is (num & FCH) | ext; (2) if idt is 1, the error code is (num « 3) | 2 | ext.

In many cases, the pseudofunction error_code is invoked with a pseudovariable EXT. The value of EXT depends on the nature of the event whose delivery encountered a nested exception: if that event is a software interrupt, EXT is 0; otherwise, EXT is 1.

```
IF PE = 0
    THEN
        GOTO REAL-ADDRESS-MODE;
    ELSE (* PE = 1 *)
        IF (VM = 1 and IOPL < 3 AND INT n)
            THEN
                #GP(0); (* Bit 0 of error code is 0 because INT n *)
            ELSE (* Protected mode, IA-32e mode, or virtual-8086 mode interrupt *)
                IF (IA32_EFER.LMA = 0)
                    THEN (* Protected mode, or virtual-8086 mode interrupt *)
                        GOTO PROTECTED-MODE;
                ELSE (* IA-32e mode interrupt *)
                    GOTO IA-32e-MODE;
                FI;
        FI;
FI;
REAL-ADDRESS-MODE:
    IF ((vector_number « 2) + 3) is not within IDT limit
        THEN #GP; FI;
    IF stack not large enough for a 6-byte return information
        THEN #SS; FI;
    Push (EFLAGS[15:0]);
    IF ← 0; (* Clear interrupt flag *)
    TF ← 0; (* Clear trap flag *)
    AC ← 0; (* Clear AC flag *)
    Push(CS);
    Push(IP);
    (* No error codes are pushed in real-address mode*)
    CS ← IDT(Descriptor (vector_number « 2), selector));
```

```
    EIP ← IDT(Descriptor (vector_number « 2), offset)); (* 16 bit offset AND 0000FFFFH *)
END;
PROTECTED-MODE:
    IF ((vector_number « 3) + 7) is not within IDT limits
    or selected IDT descriptor is not an interrupt-, trap-, or task-gate type
        THEN #GP(error_code(vector_number,1,EXT)); FI;
        (* idt operand to error_code set because vector is used *)
    IF software interrupt (* Generated by INT n, INT3, or INTO *)
        THEN
            IF gate DPL < CPL (* PE = 1, DPL < CPL, software interrupt *)
                THEN #GP(error_code(vector_number,1,0)); FI;
                (* idt operand to error_code set because vector is used *)
                (* ext operand to error_code is 0 because INT n, INT3, or INTO*)
    FI;
    IF gate not present
        THEN #NP(error_code(vector_number,1,EXT)); FI;
        (* idt operand to error_code set because vector is used *)
    IF task gate (* Specified in the selected interrupt table descriptor *)
        THEN GOTO TASK-GATE;
        ELSE GOTO TRAP-OR-INTERRUPT-GATE; (* PE = 1, trap/interrupt gate *)
    FI;
END;
IA-32e-MODE:
    IF INTO and CS.L = 1 (64-bit mode)
        THEN #UD;
    FI;
    IF ((vector_number « 4) + 15) is not in IDT limits
    or selected IDT descriptor is not an interrupt-, or trap-gate type
        THEN #GP(error_code(vector_number,1,EXT));
        (* idt operand to error_code set because vector is used *)
    FI;
    IF software interrupt (* Generated by INT n, INT 3, or INTO *)
        THEN
            IF gate DPL < CPL (* PE = 1, DPL < CPL, software interrupt *)
                THEN #GP(error_code(vector_number,1,0));
                (* idt operand to error_code set because vector is used *)
                (* ext operand to error_code is 0 because INT n, INT3, or INTO*)
            FI;
    FI;
    IF gate not present
        THEN #NP(error_code(vector_number,1,EXT));
        (* idt operand to error_code set because vector is used *)
    FI;
    GOTO TRAP-OR-INTERRUPT-GATE; (* Trap/interrupt gate *)
END;
TASK-GATE: (* PE = 1, task gate *)
    Read TSS selector in task gate (IDT descriptor);
        IF local/global bit is set to local or index not within GDT limits
            THEN #GP(error_code(TSS selector,0,EXT)); FI;
            (* idt operand to error_code is 0 because selector is used *)
```

```
            Access TSS descriptor in GDT;
            IF TSS descriptor specifies that the TSS is busy (low-order 5 bits set to 00001)
                THEN #GP(TSS selector,0,EXT)); FI;
                (* idt operand to error_code is 0 because selector is used *)
            IF TSS not present
                THEN #NP(TSS selector,0,EXT)); FI;
                (* idt operand to error_code is 0 because selector is used *)
        SWITCH-TASKS (with nesting) to TSS;
        IF interrupt caused by fault with error code
            THEN
                IF stack limit does not allow push of error code
                    THEN #SS(EXT); FI;
                Push(error code);
        FI;
        IF EIP not within code segment limit
            THEN #GP(EXT); FI;
END;

...
```

Jcc—Jump if Condition Is Met

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
77 *cb*	JA *rel8*	A	Valid	Valid	Jump short if above (CF=0 and ZF=0).
73 *cb*	JAE *rel8*	A	Valid	Valid	Jump short if above or equal (CF=0).
72 *cb*	JB *rel8*	A	Valid	Valid	Jump short if below (CF=1).
76 *cb*	JBE *rel8*	A	Valid	Valid	Jump short if below or equal (CF=1 or ZF=1).
72 *cb*	JC *rel8*	A	Valid	Valid	Jump short if carry (CF=1).
E3 *cb*	JCXZ *rel8*	A	N.E.	Valid	Jump short if CX register is 0.
E3 *cb*	JECXZ *rel8*	A	Valid	Valid	Jump short if ECX register is 0.
E3 *cb*	JRCXZ *rel8*	A	Valid	N.E.	Jump short if RCX register is 0.
74 *cb*	JE *rel8*	A	Valid	Valid	Jump short if equal (ZF=1).
7F *cb*	JG *rel8*	A	Valid	Valid	Jump short if greater (ZF=0 and SF=OF).
7D *cb*	JGE *rel8*	A	Valid	Valid	Jump short if greater or equal (SF=OF).
7C *cb*	JL *rel8*	A	Valid	Valid	Jump short if less (SF≠ OF).
7E *cb*	JLE *rel8*	A	Valid	Valid	Jump short if less or equal (ZF=1 or SF≠ OF).
76 *cb*	JNA *rel8*	A	Valid	Valid	Jump short if not above (CF=1 or ZF=1).

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
72 *cb*	JNAE *rel8*	A	Valid	Valid	Jump short if not above or equal (CF=1).
73 *cb*	JNB *rel8*	A	Valid	Valid	Jump short if not below (CF=0).
77 *cb*	JNBE *rel8*	A	Valid	Valid	Jump short if not below or equal (CF=0 and ZF=0).
73 *cb*	JNC *rel8*	A	Valid	Valid	Jump short if not carry (CF=0).
75 *cb*	JNE *rel8*	A	Valid	Valid	Jump short if not equal (ZF=0).
7E *cb*	JNG *rel8*	A	Valid	Valid	Jump short if not greater (ZF=1 or SF≠ OF).
7C *cb*	JNGE *rel8*	A	Valid	Valid	Jump short if not greater or equal (SF≠ OF).
7D *cb*	JNL *rel8*	A	Valid	Valid	Jump short if not less (SF=OF).
7F *cb*	JNLE *rel8*	A	Valid	Valid	Jump short if not less or equal (ZF=0 and SF=OF).
71 *cb*	JNO *rel8*	A	Valid	Valid	Jump short if not overflow (OF=0).
7B *cb*	JNP *rel8*	A	Valid	Valid	Jump short if not parity (PF=0).
79 *cb*	JNS *rel8*	A	Valid	Valid	Jump short if not sign (SF=0).
75 *cb*	JNZ *rel8*	A	Valid	Valid	Jump short if not zero (ZF=0).
70 *cb*	JO *rel8*	A	Valid	Valid	Jump short if overflow (OF=1).
7A *cb*	JP *rel8*	A	Valid	Valid	Jump short if parity (PF=1).
7A *cb*	JPE *rel8*	A	Valid	Valid	Jump short if parity even (PF=1).
7B *cb*	JPO *rel8*	A	Valid	Valid	Jump short if parity odd (PF=0).
78 *cb*	JS *rel8*	A	Valid	Valid	Jump short if sign (SF=1).
74 *cb*	JZ *rel8*	A	Valid	Valid	Jump short if zero (ZF ← 1).
0F 87 *cw*	JA *rel16*	A	N.S.	Valid	Jump near if above (CF=0 and ZF=0). Not supported in 64-bit mode.
0F 87 *cd*	JA *rel32*	A	Valid	Valid	Jump near if above (CF=0 and ZF=0).
0F 83 *cw*	JAE *rel16*	A	N.S.	Valid	Jump near if above or equal (CF=0). Not supported in 64-bit mode.

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
0F 83 *cd*	JAE *rel32*	A	Valid	Valid	Jump near if above or equal (CF=0).
0F 82 *cw*	JB *rel16*	A	N.S.	Valid	Jump near if below (CF=1). Not supported in 64-bit mode.
0F 82 *cd*	JB *rel32*	A	Valid	Valid	Jump near if below (CF=1).
0F 86 *cw*	JBE *rel16*	A	N.S.	Valid	Jump near if below or equal (CF=1 or ZF=1). Not supported in 64-bit mode.
0F 86 *cd*	JBE *rel32*	A	Valid	Valid	Jump near if below or equal (CF=1 or ZF=1).
0F 82 *cw*	JC *rel16*	A	N.S.	Valid	Jump near if carry (CF=1). Not supported in 64-bit mode.
0F 82 *cd*	JC *rel32*	A	Valid	Valid	Jump near if carry (CF=1).
0F 84 *cw*	JE *rel16*	A	N.S.	Valid	Jump near if equal (ZF=1). Not supported in 64-bit mode.
0F 84 *cd*	JE *rel32*	A	Valid	Valid	Jump near if equal (ZF=1).
0F 84 *cw*	JZ *rel16*	A	N.S.	Valid	Jump near if 0 (ZF=1). Not supported in 64-bit mode.
0F 84 *cd*	JZ *rel32*	A	Valid	Valid	Jump near if 0 (ZF=1).
0F 8F *cw*	JG *rel16*	A	N.S.	Valid	Jump near if greater (ZF=0 and SF=OF). Not supported in 64-bit mode.
0F 8F *cd*	JG *rel32*	A	Valid	Valid	Jump near if greater (ZF=0 and SF=OF).
0F 8D *cw*	JGE *rel16*	A	N.S.	Valid	Jump near if greater or equal (SF=OF). Not supported in 64-bit mode.
0F 8D *cd*	JGE *rel32*	A	Valid	Valid	Jump near if greater or equal (SF=OF).
0F 8C *cw*	JL *rel16*	A	N.S.	Valid	Jump near if less (SF≠ OF). Not supported in 64-bit mode.
0F 8C *cd*	JL *rel32*	A	Valid	Valid	Jump near if less (SF≠ OF).
0F 8E *cw*	JLE *rel16*	A	N.S.	Valid	Jump near if less or equal (ZF=1 or SF≠ OF). Not supported in 64-bit mode.
0F 8E *cd*	JLE *rel32*	A	Valid	Valid	Jump near if less or equal (ZF=1 or SF≠ OF).

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
0F 86 *cw*	JNA *rel16*	A	N.S.	Valid	Jump near if not above (CF=1 or ZF=1). Not supported in 64-bit mode.
0F 86 *cd*	JNA *rel32*	A	Valid	Valid	Jump near if not above (CF=1 or ZF=1).
0F 82 *cw*	JNAE *rel16*	A	N.S.	Valid	Jump near if not above or equal (CF=1). Not supported in 64-bit mode.
0F 82 *cd*	JNAE *rel32*	A	Valid	Valid	Jump near if not above or equal (CF=1).
0F 83 *cw*	JNB *rel16*	A	N.S.	Valid	Jump near if not below (CF=0). Not supported in 64-bit mode.
0F 83 *cd*	JNB *rel32*	A	Valid	Valid	Jump near if not below (CF=0).
0F 87 *cw*	JNBE *rel16*	A	N.S.	Valid	Jump near if not below or equal (CF=0 and ZF=0). Not supported in 64-bit mode.
0F 87 *cd*	JNBE *rel32*	A	Valid	Valid	Jump near if not below or equal (CF=0 and ZF=0).
0F 83 *cw*	JNC *rel16*	A	N.S.	Valid	Jump near if not carry (CF=0). Not supported in 64-bit mode.
0F 83 *cd*	JNC *rel32*	A	Valid	Valid	Jump near if not carry (CF=0).
0F 85 *cw*	JNE *rel16*	A	N.S.	Valid	Jump near if not equal (ZF=0). Not supported in 64-bit mode.
0F 85 *cd*	JNE *rel32*	A	Valid	Valid	Jump near if not equal (ZF=0).
0F 8E *cw*	JNG *rel16*	A	N.S.	Valid	Jump near if not greater (ZF=1 or SF≠ OF). Not supported in 64-bit mode.
0F 8E *cd*	JNG *rel32*	A	Valid	Valid	Jump near if not greater (ZF=1 or SF≠ OF).
0F 8C *cw*	JNGE *rel16*	A	N.S.	Valid	Jump near if not greater or equal (SF≠ OF). Not supported in 64-bit mode.
0F 8C *cd*	JNGE *rel32*	A	Valid	Valid	Jump near if not greater or equal (SF≠ OF).
0F 8D *cw*	JNL *rel16*	A	N.S.	Valid	Jump near if not less (SF=OF). Not supported in 64-bit mode.

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
0F 8D *cd*	JNL *rel32*	A	Valid	Valid	Jump near if not less (SF=OF).
0F 8F *cw*	JNLE *rel16*	A	N.S.	Valid	Jump near if not less or equal (ZF=0 and SF=OF). Not supported in 64-bit mode.
0F 8F *cd*	JNLE *rel32*	A	Valid	Valid	Jump near if not less or equal (ZF=0 and SF=OF).
0F 81 *cw*	JNO *rel16*	A	N.S.	Valid	Jump near if not overflow (OF=0). Not supported in 64-bit mode.
0F 81 *cd*	JNO *rel32*	A	Valid	Valid	Jump near if not overflow (OF=0).
0F 8B *cw*	JNP *rel16*	A	N.S.	Valid	Jump near if not parity (PF=0). Not supported in 64-bit mode.
0F 8B *cd*	JNP *rel32*	A	Valid	Valid	Jump near if not parity (PF=0).
0F 89 *cw*	JNS *rel16*	A	N.S.	Valid	Jump near if not sign (SF=0). Not supported in 64-bit mode.
0F 89 *cd*	JNS *rel32*	A	Valid	Valid	Jump near if not sign (SF=0).
0F 85 *cw*	JNZ *rel16*	A	N.S.	Valid	Jump near if not zero (ZF=0). Not supported in 64-bit mode.
0F 85 *cd*	JNZ *rel32*	A	Valid	Valid	Jump near if not zero (ZF=0).
0F 80 *cw*	JO *rel16*	A	N.S.	Valid	Jump near if overflow (OF=1). Not supported in 64-bit mode.
0F 80 *cd*	JO *rel32*	A	Valid	Valid	Jump near if overflow (OF=1).
0F 8A *cw*	JP *rel16*	A	N.S.	Valid	Jump near if parity (PF=1). Not supported in 64-bit mode.
0F 8A *cd*	JP *rel32*	A	Valid	Valid	Jump near if parity (PF=1).
0F 8A *cw*	JPE *rel16*	A	N.S.	Valid	Jump near if parity even (PF=1). Not supported in 64-bit mode.
0F 8A *cd*	JPE *rel32*	A	Valid	Valid	Jump near if parity even (PF=1).
0F 8B *cw*	JPO *rel16*	A	N.S.	Valid	Jump near if parity odd (PF=0). Not supported in 64-bit mode.

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
0F 8B *cd*	JPO *rel32*	A	Valid	Valid	Jump near if parity odd (PF=0).
0F 88 *cw*	JS *rel16*	A	N.S.	Valid	Jump near if sign (SF=1). Not supported in 64-bit mode.
0F 88 *cd*	JS *rel32*	A	Valid	Valid	Jump near if sign (SF=1).
0F 84 *cw*	JZ *rel16*	A	N.S.	Valid	Jump near if 0 (ZF=1). Not supported in 64-bit mode.
0F 84 *cd*	JZ *rel32*	A	Valid	Valid	Jump near if 0 (ZF=1).

…

JMP—Jump

…

Operation

```
IF near jump
    IF 64-bit Mode
        THEN
            IF near relative jump
             THEN
                tempRIP ← RIP + DEST; (* RIP is instruction following JMP instruction*)
             ELSE (* Near absolute jump *)
                tempRIP ← DEST;
            FI;
        ELSE
            IF near relative jump
             THEN
                tempEIP ← EIP + DEST; (* EIP is instruction following JMP instruction*)
             ELSE (* Near absolute jump *)
                tempEIP ← DEST;
            FI;
    FI;
    IF (IA32_EFER.LMA = 0 or target mode = Compatibility mode)
    and tempEIP outside code segment limit
        THEN #GP(0); FI
    IF 64-bit mode and tempRIP is not canonical
        THEN #GP(0);
    FI;
    IF OperandSize = 32
         THEN
            EIP ← tempEIP;
         ELSE
            IF OperandSize = 16
                THEN (* OperandSize = 16 *)
```

```
                    EIP ← tempEIP AND 0000FFFFH;
                ELSE (* OperandSize = 64)
                    RIP ← tempRIP;
            FI;
    FI;
FI;
IF far jump and (PE = 0 or (PE = 1 AND VM = 1)) (* Real-address or virtual-8086 mode *)
    THEN
        tempEIP ← DEST(Offset); (* DEST is ptr16:32 or [m16:32] *)
        IF tempEIP is beyond code segment limit
            THEN #GP(0); FI;
        CS ← DEST(segment selector); (* DEST is ptr16:32 or [m16:32] *)
        IF OperandSize = 32
            THEN
                EIP ← tempEIP; (* DEST is ptr16:32 or [m16:32] *)
            ELSE (* OperandSize = 16 *)
                EIP ← tempEIP AND 0000FFFFH; (* Clear upper 16 bits *)
        FI;
FI;
IF far jump and (PE = 1 and VM = 0)
(* IA-32e mode or protected mode, not virtual-8086 mode *)
    THEN
        IF effective address in the CS, DS, ES, FS, GS, or SS segment is illegal
        or segment selector in target operand NULL
                THEN #GP(0); FI;
        IF segment selector index not within descriptor table limits
            THEN #GP(new selector); FI;
        Read type and access rights of segment descriptor;
        IF (EFER.LMA = 0)
            THEN
                IF segment type is not a conforming or nonconforming code
                segment, call gate, task gate, or TSS
                    THEN #GP(segment selector); FI;
            ELSE
                IF segment type is not a conforming or nonconforming code segment
                call gate
                    THEN #GP(segment selector); FI;
        FI;
        Depending on type and access rights:
            GO TO CONFORMING-CODE-SEGMENT;
            GO TO NONCONFORMING-CODE-SEGMENT;
            GO TO CALL-GATE;
            GO TO TASK-GATE;
            GO TO TASK-STATE-SEGMENT;
    ELSE
        #GP(segment selector);
FI;
CONFORMING-CODE-SEGMENT:
    IF L-Bit = 1 and D-BIT = 1 and IA32_EFER.LMA = 1
        THEN GP(new code segment selector); FI;
```

```
    IF DPL > CPL
        THEN #GP(segment selector); FI;
    IF segment not present
        THEN #NP(segment selector); FI;
    tempEIP ← DEST(Offset);
    IF OperandSize = 16
        THEN tempEIP ← tempEIP AND 0000FFFFH;
    FI;
    IF (IA32_EFER.LMA = 0 or target mode = Compatibility mode) and
    tempEIP outside code segment limit
        THEN #GP(0); FI
    IF tempEIP is non-canonical
        THEN #GP(0); FI;
    CS ← DEST[segment selector]; (* Segment descriptor information also loaded *)
    CS(RPL) ← CPL
    EIP ← tempEIP;
END;
NONCONFORMING-CODE-SEGMENT:
    IF L-Bit = 1 and D-BIT = 1 and IA32_EFER.LMA = 1
        THEN GP(new code segment selector); FI;
    IF (RPL > CPL) OR (DPL ≠ CPL)
        THEN #GP(code segment selector); FI;
    IF segment not present
        THEN #NP(segment selector); FI;
    tempEIP ← DEST(Offset);
    IF OperandSize = 16
        THEN tempEIP ← tempEIP AND 0000FFFFH; FI;
    IF (IA32_EFER.LMA = 0 OR target mode = Compatibility mode)
    and tempEIP outside code segment limit
        THEN #GP(0); FI
    IF tempEIP is non-canonical THEN #GP(0); FI;
    CS ← DEST[segment selector]; (* Segment descriptor information also loaded *)
    CS(RPL) ← CPL;
    EIP ← tempEIP;
END;

CALL-GATE:
    IF call gate DPL < CPL
    or call gate DPL < call gate segment-selector RPL
            THEN #GP(call gate selector); FI;
    IF call gate not present
        THEN #NP(call gate selector); FI;
    IF call gate code-segment selector is NULL
        THEN #GP(0); FI;
    IF call gate code-segment selector index outside descriptor table limits
        THEN #GP(code segment selector); FI;
    Read code segment descriptor;
    IF code-segment segment descriptor does not indicate a code segment
    or code-segment segment descriptor is conforming and DPL > CPL
    or code-segment segment descriptor is non-conforming and DPL ≠ CPL
```

```
                THEN #GP(code segment selector); FI;
    IF IA32_EFER.LMA = 1 and (code-segment descriptor is not a 64-bit code segment
    or code-segment segment descriptor has both L-Bit and D-bit set)
                THEN #GP(code segment selector); FI;
    IF code segment is not present
          THEN #NP(code-segment selector); FI;
     IF instruction pointer is not within code-segment limit
          THEN #GP(0); FI;
     tempEIP ← DEST(Offset);
     IF GateSize = 16
           THEN tempEIP ← tempEIP AND 0000FFFFH; FI;
    IF (IA32_EFER.LMA = 0 OR target mode = Compatibility mode) AND tempEIP
    outside code segment limit
          THEN #GP(0); FI
    CS ← DEST[SegmentSelector); (* Segment descriptor information also loaded *)
    CS(RPL) ← CPL;
    EIP ← tempEIP;
END;
TASK-GATE:
    IF task gate DPL < CPL
    or task gate DPL < task gate segment-selector RPL
          THEN #GP(task gate selector); FI;
    IF task gate not present
          THEN #NP(gate selector); FI;
    Read the TSS segment selector in the task-gate descriptor;
    IF TSS segment selector local/global bit is set to local
    or index not within GDT limits
    or TSS descriptor specifies that the TSS is busy
          THEN #GP(TSS selector); FI;
     IF TSS not present
          THEN #NP(TSS selector); FI;
     SWITCH-TASKS to TSS;
     IF EIP not within code segment limit
          THEN #GP(0); FI;
END;
TASK-STATE-SEGMENT:
    IF TSS DPL < CPL
    or TSS DPL < TSS segment-selector RPL
    or TSS descriptor indicates TSS not available
          THEN #GP(TSS selector); FI;
    IF TSS is not present
          THEN #NP(TSS selector); FI;
    SWITCH-TASKS to TSS;
    IF EIP not within code segment limit
          THEN #GP(0); FI;
END;

...
```

LAR—Load Access Rights Byte

…

Operation

```
IF Offset(SRC) > descriptor table limit
    THEN

        ZF = 0;
    ELSE

        IF SegmentDescriptor(Type) ≠ conforming code segment
        and (CPL > DPL) or (RPL > DPL)
        or segment type is not valid for instruction
            THEN
                ZF ← 0
            ELSE

                TEMP ← Read segment descriptor ;
                IF OperandSize = 64
                    THEN
                        DEST ← (ACCESSRIGHTWORD(TEMP) AND 00000000_00FxFF00H);
                    ELSE (* OperandSize = 32*)
                        DEST ← (ACCESSRIGHTWORD(TEMP) AND 00FxFF00H);
                    ELSE (* OperandSize = 16 *)
                        DEST ← (ACCESSRIGHTWORD(TEMP) AND FF00H);
                FI;
        FI;
FI;
```

…

LDDQU—Load Unaligned Integer 128 Bits

Opcode/ Instruction	Op/ En	64/32-bit Mode	CPUID Feature Flag	Description
F2 0F F0 /*r* LDDQU *xmm1*, *mem*	A	V/V	SSE3	Load unaligned data from *mem* and return double quadword in *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

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

Description

The instruction is *functionally similar* to (V)MOVDQU ymm/xmm, m256/m128 for loading from memory. That is: 32/16 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 32/16-byte boundary. Up to 64/32 bytes may be loaded from memory; this is implementation dependent.

This instruction may improve performance relative to (V)MOVDQU if the source operand crosses a cache line boundary. In situations that require the data loaded by (V)LDDQU be modified and stored to the same location, use (V)MOVDQU or (V)MOVDQA instead of (V)LDDQU. To move a double quadword to or from memory locations that are known to be aligned on 16-byte boundaries, use the (V)MOVDQA instruction.

Implementation Notes

- If the source is aligned to a 32/16-byte boundary, based on the implementation, the 32/16 bytes may be loaded more than once. For that reason, the usage of (V)LDDQU should be avoided when using uncached or write-combining (WC) memory regions. For uncached or WC memory regions, keep using (V)MOVDQU.
- This instruction is a replacement for (V)MOVDQU (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 (V)MOVDQA store-load pairs when data is 256/128-bit aligned or (V)MOVDQU store-load pairs when data is 256/128-bit unaligned.
- If the memory address is not aligned on 32/16-byte boundary, some implementations may load up to 64/32 bytes and return 32/16 bytes in the destination. Some processor implementations may issue multiple loads to access the appropriate 32/16 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.
- If alignment checking is enabled (CR0.AM = 1, RFLAGS.AC = 1, and CPL = 3), an alignment-check exception (#AC) may or may not be generated (depending on processor implementation) when the memory address is not aligned on an 8-byte boundary.

In 64-bit mode, use of the REX.R prefix permits this instruction to access additional registers (XMM8-XMM15).

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

Operation

LDDQU (128-bit Legacy SSE version)

DEST[127:0] ← SRC[127:0]
DEST[VLMAX-1:128] (Unmodified)

VLDDQU (VEX.128 encoded version)

DEST[127:0] ← SRC[127:0]
DEST[VLMAX-1:128] ← 0

VLDDQU (VEX.256 encoded version)
DEST[255:0] ← SRC[255:0]

Intel C/C++ Compiler Intrinsic Equivalent

LDDQU __m128i _mm_lddqu_si128 (__m128i * p);

LDDQU __m256i _mm256_lddqu_si256 (__m256i * p);

Numeric Exceptions

None.

Other Exceptions

See Exceptions Type 4;
Note treatment of #AC varies.

...

LDMXCSR—Load MXCSR Register

Opcode/ Instruction	Op/ En	64/32-bit Mode	CPUID Feature Flag	Description
0F,AE,/2 LDMXCSR *m32*	A	V/V	SSE	Load MXCSR register from *m32.*
VEX.LZ.0F.WIG AE /2 VLDMXCSR *m32*	A	V/V	AVX	Load MXCSR register from *m32.*

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
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. See "MXCSR Control and Status Register" in Chapter 10, of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*, for a description of the MXCSR register and its contents.

The LDMXCSR instruction is typically used in conjunction with the (V)STMXCSR instruction, which stores the contents of the MXCSR register in memory.

The default MXCSR value at reset is 1F80H.

If a (V)LDMXCSR 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)

Numeric Exceptions

None.

Other Exceptions

See Exceptions Type 9; additionally

#GP	For an attempt to set reserved bits in MXCSR.

...

LODS/LODSB/LODSW/LODSD/LODSQ—Load String

...

Description

Loads a byte, word, or doubleword from the source operand into the AL, AX, or EAX register, respectively. The source operand is a memory location, the address of which is read from the DS:ESI or the DS:SI registers (depending on the address-size attribute of the instruction, 32 or 16, respectively). The DS segment may be overridden with a segment override prefix.

...

MASKMOVDQU—Store Selected Bytes of Double Quadword

Opcode/ Instruction	Op/ En	64/32-bit Mode	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: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]

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
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 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 *Intel® 64 and IA-32 Architectures 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 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).

1. ModRM.MOD = 011B required

- 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.

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 byte-mask without allocating old data prior to the store.

In 64-bit mode, use of the REX.R prefix permits this instruction to access additional registers (XMM8-XMM15).

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

```
IF (MASK[7] = 1)
    THEN DEST[DI/EDI] ← SRC[7:0] ELSE (* Memory location unchanged *); FI;
IF (MASK[15] = 1)
    THEN DEST[DI/EDI +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[DI/EDI +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)

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	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.
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

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
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 -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[VLMAX-1:128] ← 0
```

VMASKMOVPS - 256-bit load

```
DEST[31:0] ← IF (SRC1[31]) Load_32(mem) ELSE 0
DEST[63:32] ← IF (SRC1[63]) Load_32(mem + 4) ELSE 0
DEST[95:64] ← IF (SRC1[95]) Load_32(mem + 8) ELSE 0
DEST[127:96] ← IF (SRC1[127]) Load_32(mem + 12) ELSE 0
DEST[159:128] ← IF (SRC1[159]) Load_32(mem + 16) ELSE 0
DEST[191:160] ← IF (SRC1[191]) Load_32(mem + 20) ELSE 0
DEST[223:192] ← IF (SRC1[223]) Load_32(mem + 24) ELSE 0
DEST[255:224] ← IF (SRC1[255]) Load_32(mem + 28) ELSE 0
```

VMASKMOVPD - 128-bit load

```
DEST[63:0] ← IF (SRC1[63]) Load_64(mem) ELSE 0
DEST[127:64] ← IF (SRC1[127]) Load_64(mem + 16) ELSE 0
DEST[VLMAX-1:128] ← 0
```

VMASKMOVPD - 256-bit load
DEST[63:0] ← IF (SRC1[63]) Load_64(mem) ELSE 0
DEST[127:64] ← IF (SRC1[127]) Load_64(mem + 8) ELSE 0
DEST[195:128] ← IF (SRC1[191]) Load_64(mem + 16) ELSE 0
DEST[255:196] ← IF (SRC1[255]) Load_64(mem + 24) ELSE 0

VMASKMOVPS - 128-bit store
IF (SRC1[31]) DEST[31:0] ← SRC2[31:0]
IF (SRC1[63]) DEST[63:32] ← SRC2[63:32]
IF (SRC1[95]) DEST[95:64] ← SRC2[95:64]
IF (SRC1[127]) DEST[127:96] ← SRC2[127:96]

VMASKMOVPS - 256-bit store
IF (SRC1[31]) DEST[31:0] ← SRC2[31:0]
IF (SRC1[63]) DEST[63:32] ← SRC2[63:32]
IF (SRC1[95]) DEST[95:64] ← SRC2[95:64]
IF (SRC1[127]) DEST[127:96] ← SRC2[127:96]
IF (SRC1[159]) DEST[159:128] ←SRC2[159:128]
IF (SRC1[191]) DEST[191:160] ← SRC2[191:160]
IF (SRC1[223]) DEST[223:192] ← SRC2[223:192]
IF (SRC1[255]) DEST[255:224] ← SRC2[255:224]

VMASKMOVPD - 128-bit store
IF (SRC1[63]) DEST[63:0] ← SRC2[63:0]
IF (SRC1[127]) DEST[127:64] ←SRC2[127:64]

VMASKMOVPD - 256-bit store
IF (SRC1[63]) DEST[63:0] ← SRC2[63:0]
IF (SRC1[127]) DEST[127:64] ←SRC2[127:64]
IF (SRC1[191]) DEST[191:128] ← SRC2[191:128]
IF (SRC1[255]) DEST[255:192] ← SRC2[255:192]

Intel C/C++ Compiler Intrinsic Equivalent

__m256 _mm256_maskload_ps(float const *a, __m256i mask)

void _mm256_maskstore_ps(float *a, __m256i mask, __m256 b)

__m256d _mm256_maskload_pd(double *a, __m256i mask);

void _mm256_maskstore_pd(double *a, __m256i mask, __m256d b);

__m128 _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—Return Maximum Packed Double-Precision Floating-Point Values

Opcode/ Instruction	Op/ En	64/32-bit Mode	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 *xmm2/m128* and *xmm1*.
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

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
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.

In 64-bit mode, use of the REX.R prefix permits this instruction to access additional registers (XMM8-XMM15).

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 (VLMAX-1:128) of the corresponding YMM register destination are unmodified.

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 (VLMAX-1:128) of the corresponding YMM register destination are zeroed.

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

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;
}
```

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[VLMAX-1:128] (Unmodified)
```

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[VLMAX-1:128] ← 0
```

VMAXPD (VEX.256 encoded version)

```
DEST[63:0] ← MAX(SRC1[63:0], SRC2[63:0])
DEST[127:64] ← MAX(SRC1[127:64], SRC2[127:64])
DEST[191:128] ← MAX(SRC1[191:128], SRC2[191:128])
DEST[255:192] ← MAX(SRC1[255:192], SRC2[255:192])
```

Intel C/C++ Compiler Intrinsic Equivalent

MAXPD __m128d _mm_max_pd(__m128d a, __m128d b);

VMAXPD __m256d _mm256_max_pd (__m256d a, __m256d b);

SIMD Floating-Point Exceptions

Invalid (including QNaN source operand), Denormal.

Other Exceptions

See Exceptions Type 2.

...

MAXPS—Return Maximum Packed Single-Precision Floating-Point Values

Opcode/ Instruction	Op/ En	64/32-bit Mode	CPUID Feature Flag	Description
0F 5F /r MAXPS *xmm1, xmm2/m128*	A	V/V	SSE	Return the maximum single-precision floating-point values between *xmm2/m128* and *xmm1*.
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

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
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.

In 64-bit mode, use of the REX.R prefix permits this instruction to access additional registers (XMM8-XMM15).

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 (VLMAX-1:128) of the corresponding YMM register destination are unmodified.

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 (VLMAX-1:128) of the corresponding YMM register destination are zeroed.

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

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;
}
```

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[VLMAX-1:128] (Unmodified)
```

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[VLMAX-1:128] ← 0
```

VMAXPS (VEX.256 encoded version)

```
DEST[31:0] ← MAX(SRC1[31:0], SRC2[31:0])
DEST[63:32] ← MAX(SRC1[63:32], SRC2[63:32])
DEST[95:64] ← MAX(SRC1[95:64], SRC2[95:64])
DEST[127:96] ← MAX(SRC1[127:96], SRC2[127:96])
DEST[159:128] ← MAX(SRC1[159:128], SRC2[159:128])
DEST[191:160] ← MAX(SRC1[191:160], SRC2[191:160])
DEST[223:192] ← MAX(SRC1[223:192], SRC2[223:192])
DEST[255:224] ← MAX(SRC1[255:224], SRC2[255:224])
```

Intel C/C++ Compiler Intrinsic Equivalent

MAXPS __m128 _mm_max_ps (__m128 a, __m128 b);

VMAXPS __m256 _mm256_max_ps (__m256 a, __m256 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	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

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
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 quadword of the destination operand. The second source operand can be an XMM register or a 64-bit memory location. The first source and destination operands are XMM registers. When the second source operand is a memory operand, only 64 bits are accessed. 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.

In 64-bit mode, use of the REX.R prefix permits this instruction to access additional registers (XMM8-XMM15).

128-bit Legacy SSE version: The destination and first source operand are the same. Bits (VLMAX-1:64) 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 (VLMAX-1:128) 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;
}
```

MAXSD (128-bit Legacy SSE version)

```
DEST[63:0] ←MAX(DEST[63:0], SRC[63:0])
DEST[VLMAX-1:64] (Unmodified)
```

VMAXSD (VEX.128 encoded version)

```
DEST[63:0] ←MAX(SRC1[63:0], SRC2[63:0])
DEST[127:64] ←SRC1[127:64]
DEST[VLMAX-1:128] ← 0
```

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	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

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
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 doubleword of the destination operand.

If the values being compared are both 0.0s (of either sign), the value in the second source operand is returned. If a value in the second source operand is an SNaN, that SNaN is returned unchanged to the destination (that is, a QNaN version of the SNaN is not returned).

If only one value is a NaN (SNaN or QNaN) for this instruction, the second source operand, either a NaN or a valid floating-point value, is written to the result. If instead of this behavior, it is required that the NaN from either source operand be returned, the action of MAXSS can be emulated using a sequence of instructions, such as, a comparison followed by AND, ANDN and OR.

The second source operand can be an XMM register or a 32-bit memory location. The first source and destination operands are XMM registers.

In 64-bit mode, use of the REX.R prefix permits this instruction to access additional registers (XMM8-XMM15).

128-bit Legacy SSE version: The destination and first source operand are the same. Bits (VLMAX-1:32) 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 (VLMAX-1:128) 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;
}
```

MAXSS (128-bit Legacy SSE version)
DEST[31:0] ←MAX(DEST[31:0], SRC[31:0])
DEST[VLMAX-1:32] (Unmodified)

VMAXSS (VEX.128 encoded version)
DEST[31:0] ←MAX(SRC1[31:0], SRC2[31:0])
DEST[127:32] ←SRC1[127:32]
DEST[VLMAX-1:128] ← 0

Intel C/C++ Compiler Intrinsic Equivalent

__m128d _mm_max_ss(__m128d a, __m128d b)

SIMD Floating-Point Exceptions

Invalid (including QNaN source operand), Denormal.

Other Exceptions

See Exceptions Type 3.

...

MINPD—Return Minimum Packed Double-Precision Floating-Point Values

Opcode/ Instruction	Op/ En	64/32-bit Mode	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 *xmm2/m128* and *xmm1*.
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

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
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.

In 64-bit mode, use of the REX.R prefix permits this instruction to access additional registers (XMM8-XMM15).

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 (VLMAX-1:128) of the corresponding YMM register destination are unmodified.

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 (VLMAX-1:128) of the corresponding YMM register destination 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;
}
```

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[VLMAX-1:128] (Unmodified)
```

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[VLMAX-1:128] ← 0

VMINPD (VEX.256 encoded version)
DEST[63:0] ← MIN(SRC1[63:0], SRC2[63:0])
DEST[127:64] ← MIN(SRC1[127:64], SRC2[127:64])
DEST[191:128] ← MIN(SRC1[191:128], SRC2[191:128])
DEST[255:192] ← MIN(SRC1[255:192], SRC2[255:192])

Intel C/C++ Compiler Intrinsic Equivalent

MINPD __m128d _mm_min_pd(__m128d a, __m128d b);

VMINPD __m256d _mm256_min_pd (__m256d a, __m256d b);

SIMD Floating-Point Exceptions

Invalid (including QNaN source operand), Denormal.

Other Exceptions

See Exceptions Type 2.

...

MINPS—Return Minimum Packed Single-Precision Floating-Point Values

Opcode/ Instruction	Op/ En	64/32-bit Mode	CPUID Feature Flag	Description
0F 5D /r MINPS *xmm1, xmm2/m128*	A	V/V	SSE	Return the minimum single-precision floating-point values between *xmm2/m128* and *xmm1*.
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

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
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.

In 64-bit mode, use of the REX.R prefix permits this instruction to access additional registers (XMM8-XMM15).

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 (VLMAX-1:128) of the corresponding YMM register destination are unmodified.

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 (VLMAX-1:128) of the corresponding YMM register destination are zeroed.

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

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;
}
```

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[VLMAX-1:128] (Unmodified)
```

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[VLMAX-1:128] ← 0
```

VMINPS (VEX.256 encoded version)

```
DEST[31:0] ← MIN(SRC1[31:0], SRC2[31:0])
```

DEST[63:32] ← MIN(SRC1[63:32], SRC2[63:32])
DEST[95:64] ← MIN(SRC1[95:64], SRC2[95:64])
DEST[127:96] ← MIN(SRC1[127:96], SRC2[127:96])
DEST[159:128] ← MIN(SRC1[159:128], SRC2[159:128])
DEST[191:160] ← MIN(SRC1[191:160], SRC2[191:160])
DEST[223:192] ← MIN(SRC1[223:192], SRC2[223:192])
DEST[255:224] ← MIN(SRC1[255:224], SRC2[255:224])

Intel C/C++ Compiler Intrinsic Equivalent

MINPS __m128d _mm_min_ps(__m128d a, __m128d b);

VMINPS __m256 _mm256_min_ps (__m256 a, __m256 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	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

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
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.

In 64-bit mode, use of the REX.R prefix permits this instruction to access additional registers (XMM8-XMM15).

128-bit Legacy SSE version: The destination and first source operand are the same. Bits (VLMAX-1:64) 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 (VLMAX-1:128) 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 (128-bit Legacy SSE version)

```
DEST[63:0] ← MIN(SRC1[63:0], SRC2[63:0])
DEST[VLMAX-1:64] (Unmodified)
```

MINSD (VEX.128 encoded version)

```
DEST[63:0] ← MIN(SRC1[63:0], SRC2[63:0])
DEST[127:64] ← SRC1[127:64]
DEST[VLMAX-1:128] ← 0
```

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	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

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
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 doubleword of the destination operand.

If the values being compared are both 0.0s (of either sign), the value in the second source operand is returned. If a value in the second operand is an SNaN, that SNaN is returned unchanged to the destination (that is, a QNaN version of the SNaN is not returned).

If only one value is a NaN (SNaN or QNaN) for this instruction, the second source operand, either a NaN or a valid floating-point value, is written to the result. If instead of this behavior, it is required that the NaN in either source operand be returned, the action of MINSD can be emulated using a sequence of instructions, such as, a comparison followed by AND, ANDN and OR.

The second source operand can be an XMM register or a 32-bit memory location. The first source and destination operands are XMM registers.

In 64-bit mode, use of the REX.R prefix permits this instruction to access additional registers (XMM8-XMM15).

128-bit Legacy SSE version: The destination and first source operand are the same. Bits (VLMAX-1:32) 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 (VLMAX-1:128) 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;
}
```

MINSS (128-bit Legacy SSE version)
DEST[31:0] ← MIN(SRC1[31:0], SRC2[31:0])
DEST[VLMAX-1:32] (Unmodified)

VMINSS (VEX.128 encoded version)
DEST[31:0] ← MIN(SRC1[31:0], SRC2[31:0])
DEST[127:32] ← SRC1[127:32]
DEST[VLMAX-1:128] ← 0

Intel C/C++ Compiler Intrinsic Equivalent

MINSS __m128d _mm_min_ss(__m128d a, __m128d b)

SIMD Floating-Point Exceptions

Invalid (including QNaN source operand), Denormal.

Other Exceptions

See Exceptions Type 3.

...

MONITOR—Set Up Monitor Address

...

Description

The MONITOR instruction arms address monitoring hardware using an address specified in EAX (the address range that the monitoring hardware checks for store operations can be determined by using CPUID). A store to an address within the specified address range triggers the monitoring hardware. The state of monitor hardware is used by MWAIT.

The content of EAX is an effective address (in 64-bit mode, RAX is used). By default, the DS segment is used to create a linear address that is monitored. Segment overrides can be used.

ECX and EDX are also used. They communicate other information to MONITOR. ECX specifies optional extensions. EDX specifies optional hints; it does not change the architectural behavior of the instruction. For the Pentium 4 processor (family 15, model 3), no extensions or hints are defined. Undefined hints in EDX are ignored by the processor; undefined extensions in ECX raises a general protection fault.

The address range must use memory of the write-back type. Only write-back memory will correctly trigger the monitoring hardware. Additional information on determining what address range to use in order to prevent false wake-ups is described in Chapter 8,

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

The MONITOR instruction is ordered as a load operation with respect to other memory transactions. The instruction is subject to the permission checking and faults associated with a byte load. Like a load, MONITOR sets the A-bit but not the D-bit in page tables.

The MONITOR CPUID feature flag (ECX bit 3; CPUID executed EAX = 1) indicates the availability of MONITOR and MWAIT in the processor. When set, MONITOR may be executed only at privilege level 0 (use at any other privilege level results in an invalid-opcode exception). The operating system or system BIOS may disable this instruction by using the IA32_MISC_ENABLE MSR; disabling MONITOR clears the CPUID feature flag and causes execution to generate an illegal opcode exception.

The instruction's operation is the same in non-64-bit modes and 64-bit mode.

...

MOVAPD—Move Aligned Packed Double-Precision Floating-Point Values

Opcode/ Instruction	Op/ En	64/32-bit Mode	CPUID Feature Flag	Description
66 0F 28 /r MOVAPD *xmm1, xmm2/m128*	A	V/V	SSE2	Move packed double-precision floating-point values from *xmm2/m128* to *xmm1*.
66 0F 29 /r MOVAPD *xmm2/m128, xmm1*	B	V/V	SSE2	Move packed double-precision floating-point values from *xmm1* to *xmm2/m128*.
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

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
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 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 double-precision floating-point values to and from unaligned memory locations, use the (V)MOVUPD instruction.

In 64-bit mode, use of the REX.R prefix permits this instruction to access additional registers (XMM8-XMM15).

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

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 (VLMAX-1:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: Bits (VLMAX-1:128) of the destination YMM register destination are zeroed.

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.

Operation

MOVAPD (128-bit load- and register-copy- form Legacy SSE version)

DEST[127:0] ← SRC[127:0]

DEST[VLMAX-1:128] (Unmodified)

(V)MOVAPD (128-bit store-form version)
DEST[127:0] ← SRC[127:0]

VMOVAPD (VEX.128 encoded version)
DEST[127:0] ← SRC[127:0]
DEST[VLMAX-1:128] ← 0

VMOVAPD (VEX.256 encoded version)
DEST[255:0] ← SRC[255:0]

Intel C/C++ Compiler Intrinsic Equivalent

MOVAPD __m128d _mm_load_pd (double const * p);

MOVAPD _mm_store_pd(double * p, __m128d a);

VMOVAPD __m256d _mm256_load_pd (double const * p);

VMOVAPD _mm256_store_pd(double * p, __m256d a);

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Exceptions Type 1.SSE2; additionally

#UD	If VEX.vvvv != 1111B.

...

MOVAPS—Move Aligned Packed Single-Precision Floating-Point Values

Opcode/ Instruction	Op/ En	64/32-bit Mode	CPUID Feature Flag	Description
0F 28 /r MOVAPS *xmm1, xmm2/m128*	A	V/V	SSE	Move packed single-precision floating-point values from *xmm2/m128* to *xmm1*.
0F 29 /r MOVAPS *xmm2/m128, xmm1*	B	V/V	SSE	Move packed single-precision floating-point values from *xmm1* to *xmm2/m128*.
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

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
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 (V)MOVUPS instruction.

In 64-bit mode, use of the REX.R prefix permits this instruction to access additional registers (XMM8-XMM15).

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

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 (VLMAX-1:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: Bits (VLMAX-1:128) of the destination YMM register are zeroed.

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.

Operation

MOVAPS (128-bit load- and register-copy- form Legacy SSE version)
DEST[127:0] ← SRC[127:0]
DEST[VLMAX-1:128] (Unmodified)

(V)MOVAPS (128-bit store form)
DEST[127:0] ← SRC[127:0]

VMOVAPS (VEX.128 encoded version)
DEST[127:0] ← SRC[127:0]
DEST[VLMAX-1:128] ← 0

VMOVAPS (VEX.256 encoded version)
DEST[255:0] ← SRC[255:0]

Intel C/C++ Compiler Intrinsic Equivalent

MOVAPS __m128 _mm_load_ps (float const * p);

MOVAPS _mm_store_ps(float * p, __m128 a);

VMOVAPS __m256 _mm256_load_ps (float const * p);

VMOVAPS _mm256_store_ps(float * p, __m256 a);

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Exceptions Type 1.SSE; additionally

#UD	If VEX.vvvv != 1111B.

...

MOVD/MOVQ—Move Doubleword/Move Quadword

Opcode/ Instruction	Op/ En	64/32-bit Mode	CPUID Feature Flag	Description
0F 6E /*r* MOVD *mm, r/m32*	A	V/V	SSE2	Move doubleword from *r/m32* to *mm*.
REX.W + 0F 6E /*r* MOVQ *mm, r/m64*	A	V/N.E.	SSE2	Move quadword from *r/m64* to *mm*.
0F 7E /*r* MOVD *r/m32, mm*	B	V/V	SSE2	Move doubleword from *mm* to *r/m32*.
REX.W + 0F 7E /*r* MOVQ *r/m64, mm*	B	V/N.E.	SSE2	Move quadword from *mm* to *r/m64*.
VEX.128.66.0F.W0 6E / VMOVD xmm1, r32/m32	A	V/V	AVX	Move doubleword from r/m32 to xmm1.
VEX.128.66.0F.W1 6E /r VMOVQ xmm1, r64/m64	A	V/N.E.	AVX	Move quadword from r/m64 to xmm1.
66 0F 6E /*r* MOVD *xmm, r/m32*	A	V/V	SSE2	Move doubleword from *r/m32* to *xmm*.
66 REX.W 0F 6E /*r* MOVQ *xmm, r/m64*	A	V/N.E.	SSE2	Move quadword from *r/m64* to *xmm*.
66 0F 7E /*r* MOVD *r/m32, xmm*	B	V/V	SSE2	Move doubleword from *xmm* register to *r/m32*.
66 REX.W 0F 7E /*r* MOVQ *r/m64, xmm*	B	V/N.E.	SSE2	Move quadword from *xmm* register to *r/m64*.
VEX.128.66.0F.W0 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

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

Description

Copies a doubleword from the source operand (second operand) to the destination operand (first operand). The source and destination operands can be general-purpose registers, MMX technology registers, XMM registers, or 32-bit memory locations. This instruction can be used to move a doubleword to and from the low doubleword of an MMX technology register and a general-purpose register or a 32-bit memory location, or to and from the low doubleword of an XMM register and a general-purpose register or a 32-bit memory location. The instruction cannot be used to transfer data between MMX technology registers, between XMM registers, between general-purpose registers, or between memory locations.

When the destination operand is an MMX technology register, the source operand is written to the low doubleword of the register, and the register is zero-extended to 64 bits. When the destination operand is an XMM register, the source operand is written to the low doubleword of the register, and the register is zero-extended to 128 bits.

In 64-bit mode, the instruction's default operation size is 32 bits. Use of the REX.R prefix permits access to additional registers (R8-R15). Use of the REX.W prefix promotes operation to 64 bits. See the summary chart at the beginning of this section for encoding data and limits.

Operation

MOVD (when destination operand is MMX technology register)

```
DEST[31:0] ← SRC;
DEST[63:32] ← 00000000H;
```

MOVD (when destination operand is XMM register)

```
DEST[31:0] ← SRC;
DEST[127:32] ← 00000000000000000000000000H;
DEST[VLMAX-1:128] (Unmodified)
```

MOVD (when source operand is MMX technology or XMM register)

```
DEST ← SRC[31:0];
```

VMOVD (VEX-encoded version when destination is an XMM register)

```
DEST[31:0] ← SRC[31:0]
DEST[VLMAX-1:32] ← 0
```

MOVQ (when destination operand is XMM register)

```
DEST[63:0] ← SRC[63:0];
DEST[127:64] ← 0000000000000000H;
DEST[VLMAX-1:128] (Unmodified)
```

MOVQ (when destination operand is r/m64)

```
DEST[63:0] ← SRC[63:0];
```

MOVQ (when source operand is XMM register or r/m64)

DEST ← SRC[63:0];

VMOVQ (VEX-encoded version when destination is an XMM register)
DEST[63:0] ← SRC[63:0]
DEST[VLMAX-1:64] ← 0

Intel C/C++ Compiler Intrinsic Equivalent

MOVD __m64 _mm_cvtsi32_si64 (int i)

MOVD int _mm_cvtsi64_si32 (__m64m)

MOVD __m128i _mm_cvtsi32_si128 (int a)

MOVD int _mm_cvtsi128_si32 (__m128i a)

Flags Affected

None.

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Exceptions Type 5; additionally

#UD	If VEX.L = 1.
	If VEX.vvvv != 1111B.

...

MOVDDUP—Move One Double-FP and Duplicate

Opcode/ Instruction	Op/ En	64/32-bit Mode	CPUID Feature Flag	Description
F2 0F 12 /*r* MOVDDUP *xmm1*, *xmm2/m64*	A	V/V	SSE3	Move one double-precision floating-point value from the lower 64-bit operand in *xmm2/m64* to *xmm1* and duplicate.
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

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

Description

The linear address corresponds to the address of the least-significant byte of the referenced memory data. When a memory address is indicated, the 8 bytes of data at memory location m64 are loaded. When the register-register form of this operation is used, the lower half of the 128-bit source register is duplicated and copied into the 128-bit destination register. See Figure 3-27.

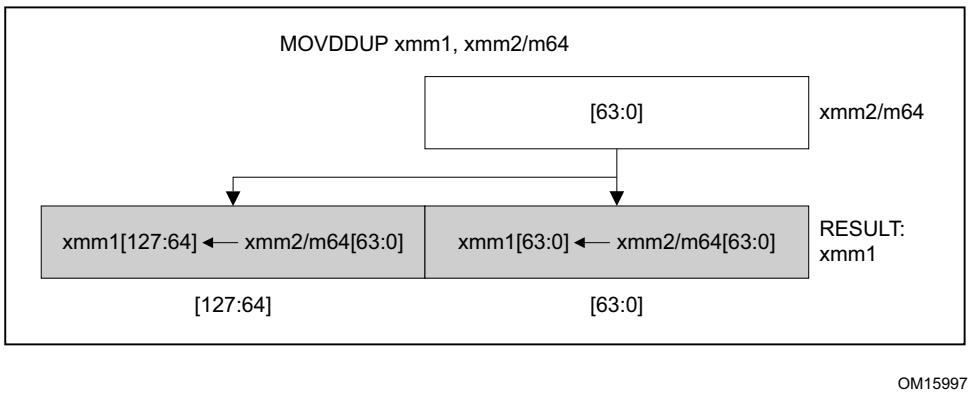


Figure 3-27 MOVDDUP—Move One Double-FP and Duplicate

In 64-bit mode, use of the REX.R prefix permits this instruction to access additional registers (XMM8-XMM15).

Operation

```
IF (Source = m64)
    THEN
        (* Load instruction *)
        xmm1[63:0] = m64;
        xmm1[127:64] = m64;
    ELSE
        (* Move instruction *)
        xmm1[63:0] = xmm2[63:0];
        xmm1[127:64] = xmm2[63:0];
FI;
```

MOVDDUP (128-bit Legacy SSE version)

```
DEST[63:0] ← SRC[63:0]
DEST[127:64] ← SRC[63:0]
DEST[VLMAX-1:128] (Unmodified)
```

VMOVDDUP (VEX.128 encoded version)
DEST[63:0] ← SRC[63:0]
DEST[127:64] ← SRC[63:0]
DEST[VLMAX-1:128] ← 0

VMOVDDUP (VEX.256 encoded version)
DEST[63:0] ← SRC[63:0]
DEST[127:64] ← SRC[63:0]
DEST[191:128] ← SRC[191:128]
DEST[255:192] ← SRC[191:128]

Intel C/C++ Compiler Intrinsic Equivalent

MOVDDUP __m128d _mm_movedup_pd(__m128d a)

MOVDDUP __m128d _mm_loaddup_pd(double const * dp)

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type 5; additionally

#UD	If VEX.vvvv != 1111B.

...

MOVDQA—Move Aligned Double Quadword

Opcode/ Instruction	Op/ En	64/32-bit Mode	CPUID Feature Flag	Description
66 0F 6F /*r* MOVDQA *xmm1, xmm2/m128*	A	V/V	SSE2	Move aligned double quadword from *xmm2/m128* to *xmm1*.
66 0F 7F /*r* MOVDQA *xmm2/m128, xmm1*	B	V/V	SSE2	Move aligned double quadword from *xmm1* to *xmm2/m128*.
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

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

Description

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.

In 64-bit mode, use of the REX.R prefix permits this instruction to access additional registers (XMM8-XMM15).

128-bit Legacy SSE version: Bits (VLMAX-1:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: Bits (VLMAX-1:128) of the destination YMM register are zeroed.

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.

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

Operation

MOVDQA (128-bit load- and register- form Legacy SSE version)

```
DEST[127:0] ← SRC[127:0]
DEST[VLMAX-1:128] (Unmodified)
(* #GP if SRC or DEST unaligned memory operand *)
```

(V)MOVDQA (128-bit store forms)

```
DEST[127:0] ← SRC[127:0]
```

VMOVDQA (VEX.128 encoded version)

```
DEST[127:0] ← SRC[127:0]
DEST[VLMAX-1:128] ← 0
```

VMOVDQA (VEX.256 encoded version)
DEST[255:0] ← SRC[255:0]

Intel C/C++ Compiler Intrinsic Equivalent

MOVDQA __m128i _mm_load_si128 (__m128i *p)

MOVDQA void _mm_store_si128 (__m128i *p, __m128i a)

VMOVDQA __m256i _mm256_load_si256 (__m256i * p);

VMOVDQA _mm256_store_si256(_m256i *p, __m256i a);

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Exceptions Type 1.SSE2; additionally

#UD	If VEX.vvvv != 1111B.

...

MOVDQU—Move Unaligned Double Quadword

Opcode/ Instruction	Op/ En	64/32-bit Mode	CPUID Feature Flag	Description
F3 0F 6F /*r* MOVDQU *xmm1, xmm2/m128*	A	V/V	SSE2	Move unaligned double quadword from *xmm2/m128* to *xmm1*.
F3 0F 7F /*r* MOVDQU *xmm2/m128, xmm1*	B	V/V	SSE2	Move unaligned double quadword from *xmm1* to *xmm2/m128*.
VEX.128.F3.0F.WIG 6F /r VMOVDQU xmm1, xmm2/m128	A	V/V	AVX	Move unaligned packed integer values from xmm2/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

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

Description

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 may be unaligned on a 16-byte boundary without causing a general-protection exception (#GP) to be generated.[1]

To move a double quadword to or from memory locations that are known to be aligned on 16-byte boundaries, use the MOVDQA instruction.

While executing in 16-bit addressing mode, a linear address for a 128-bit data access that overlaps the end of a 16-bit segment is not allowed and is defined as reserved behavior. A specific processor implementation may or may not generate a general-protection exception (#GP) in this situation, and the address that spans the end of the segment may or may not wrap around to the beginning of the segment.

In 64-bit mode, use of the REX.R prefix permits this instruction to access additional registers (XMM8-XMM15).

128-bit Legacy SSE version: Bits (VLMAX-1:128) 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 (VLMAX-1:128) of the destination YMM register are zeroed.

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.

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

Operation

MOVDQU load and register copy (128-bit Legacy SSE version)

DEST[127:0] ← SRC[127:0]

DEST[VLMAX-1:128] (Unmodified)

1. If alignment checking is enabled (CR0.AM = 1, RFLAGS.AC = 1, and CPL = 3), an alignment-check exception (#AC) may or may not be generated (depending on processor implementation) when the operand is not aligned on an 8-byte boundary.

(V)MOVDQU 128-bit store-form versions
DEST[127:0] ← SRC[127:0]

VMOVDQU (VEX.128 encoded version)
DEST[127:0] ← SRC[127:0]
DEST[VLMAX-1:128] ← 0

VMOVDQU (VEX.256 encoded version)
DEST[255:0] ← SRC[255:0]

Intel C/C++ Compiler Intrinsic Equivalent

MOVDQU void _mm_storeu_si128 (__m128i *p, __m128i a)

MOVDQU __m128i _mm_loadu_si128 (__m128i *p)

VMOVDQU __m256i _mm256_loadu_si256 (__m256i * p);

VMOVDQU _mm256_storeu_si256(_m256i *p, __m256i 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	CPUID Feature Flag	Description
0F 12 /r MOVHLPS *xmm1, xmm2*	A	V/V	SSE3	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

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
A	ModRM:reg (w)	ModRM:reg (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. Bits (VLMAX-1:64) 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). Bits (VLMAX-1:128) of the destination YMM register are zeroed.

In 64-bit mode, use of the REX.R prefix permits this instruction to access additional registers (XMM8-XMM15).

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

Operation

MOVHLPS (128-bit two-argument form)

```
DEST[63:0] ← SRC[127:64]
DEST[VLMAX-1:64] (Unmodified)
```

VMOVHLPS (128-bit three-argument form)

```
DEST[63:0] ← SRC2[127:64]
DEST[127:64] ← SRC1[127:64]
DEST[VLMAX-1:128] ← 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 Value

Opcode/ Instruction	Op/ En	64/32-bit Mode	CPUID Feature Flag	Description
66 0F 16 /r MOVHPD *xmm, m64*	A	V/V	SSE2	Move double-precision floating-point value from *m64* to high quadword of *xmm*.
66 0F 17 /r MOVHPD *m64, xmm*	B	V/V	SSE2	Move double-precision floating-point value from high quadword of *xmm* to *m64*.
VEX.NDS.128.66.0F.WIG 16 /r VMOVHPD xmm2, xmm1, m64	C	V/V	AVX	Merge double-precision floating-point value from m64 and the low quadword of xmm1.
VEX128.66.0F.WIG 17/r VMOVHPD m64, xmm1	B	V/V	AVX	Move double-precision floating-point values from high quadword of xmm1 to m64.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
A	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
B	ModRM:r/m (w)	ModRM:reg (r)	NA	NA
C	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	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.

In 64-bit mode, use of the REX.R prefix permits this instruction to access additional registers (XMM8-XMM15).

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[VLMAX-1:128] (Unmodified)

VMOVHPD (VEX.128 encoded load)
DEST[63:0] ← SRC1[63:0]
DEST[127:64] ← SRC2[63:0]
DEST[VLMAX-1:128] ← 0

VMOVHPD (store)
DEST[63:0] ← SRC[127:64]

Intel C/C++ Compiler Intrinsic Equivalent

MOVHPD __m128d _mm_loadh_pd (__m128d a, double *p)

MOVHPD void _mm_storeh_pd (double *p, __m128d a)

SIMD Floating-Point Exceptions

None.

Other Exceptions

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	CPUID Feature Flag	Description
0F 16 /r MOVHPS *xmm, m64*	A	V/V	SSE	Move two packed single-precision floating-point values from *m64* to high quadword of *xmm*.
0F 17 /r MOVHPS *m64, xmm*	B	V/V	SSE	Move two packed single-precision floating-point values from high quadword of *xmm* to *m64*.
VEX.NDS.128.0F.WIG 16 /r VMOVHPS xmm2, xmm1, m64	C	V/V	AVX	Merge two packed single-precision floating-point values from m64 and the low quadword of xmm1.
VEX.128.0F.WIG 17/r VMOVHPS m64, xmm1	B	V/V	AVX	Move two packed single-precision floating-point values from high quadword of xmm1to m64.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
A	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
B	ModRM:r/m (w)	ModRM:reg (r)	NA	NA
C	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	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.

In 64-bit mode, use of the REX.R prefix permits this instruction to access additional registers (XMM8-XMM15).

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[VLMAX-1:128] (Unmodified)

VMOVHPS (VEX.128 encoded load)
DEST[63:0] ← SRC1[63:0]
DEST[127:64] ← SRC2[63:0]
DEST[VLMAX-1:128] ← 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	CPUID Feature Flag	Description
0F 16 /*r* MOVLHPS *xmm1, xmm2*	A	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	B	V/V	AVX	Merge two packed single-precision floating-point values from low quadword of xmm3 and low quadword of xmm2.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
A	ModRM:reg (w)	ModRM:reg (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.

In 64-bit mode, use of the REX.R prefix permits this instruction to access additional registers (XMM8-XMM15).

Operation

MOVLHPS (128-bit two-argument form)

```
DEST[63:0] (Unmodified)
DEST[127:64] ← SRC[63:0]
DEST[VLMAX-1:128] (Unmodified)
```

VMOVLHPS (128-bit three-argument form)

DEST[63:0] ← SRC1[63:0]
DEST[127:64] ← SRC2[63:0]
DEST[VLMAX-1:128] ← 0

Intel C/C++ Compiler Intrinsic Equivalent

MOVHLPS __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 Value

Opcode/ Instruction	Op/ En	64/32-bit Mode	CPUID Feature Flag	Description
66 0F 12 /r MOVLPD *xmm, m64*	A	V/V	SSE2	Move double-precision floating-point value from *m64* to low quadword of *xmm* register.
66 0F 13 /r MOVLPD *m64, xmm*	B	V/V	SSE2	Move double-precision floating-point nvalue from low quadword of *xmm* register to *m64*.
VEX.NDS.128.66.0F.WIG 12 /r VMOVLPD xmm2, xmm1, m64	C	V/V	AVX	Merge double-precision floating-point value from m64 and the high quadword of xmm1.
VEX.128.66.0F.WIG 13/r VMOVLPD m64, xmm1	B	V/V	AVX	Move double-precision floating-point values from low quadword of xmm1 to m64.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
A	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
B	ModRM:r/m (w)	ModRM:reg (r)	NA	NA
C	ModRM:r/m (r)	ModRM:reg (r, w)	VEX.vvvv (r)	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.

In 64-bit mode, use of the REX.R prefix permits this instruction to access additional registers (XMM8-XMM15).

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[VLMAX-1:64] (Unmodified)
```

VMOVLPD (VEX.128 encoded load)

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

VMOVLPD (store)

```
DEST[63:0] ← SRC[63:0]
```

Intel C/C++ Compiler Intrinsic Equivalent

MOVLPD __m128d _mm_loadl_pd (__m128d a, double *p)

MOVLPD void _mm_storel_pd (double *p, __m128d a)

SIMD Floating-Point Exceptions

None.

Other Exceptions

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	CPUID Feature Flag	Description
0F 12 /r MOVLPS *xmm, m64*	A	V/V	SSE	Move two packed single-precision floating-point values from *m64* to low quadword of *xmm*.
0F 13 /r MOVLPS *m64, xmm*	B	V/V	SSE	Move two packed single-precision floating-point values from low quadword of *xmm* to *m64*.
VEX.NDS.128.0F.WIG 12 /r VMOVLPS xmm2, xmm1, m64	C	V/V	AVX	Merge two packed single-precision floating-point values from m64 and the high quadword of xmm1.
VEX.128.0F.WIG 13/r VMOVLPS m64, xmm1	B	V/V	AVX	Move two packed single-precision floating-point values from low quadword of xmm1 to m64.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
A	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
B	ModRM:r/m (w)	ModRM:reg (r)	NA	NA
C	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	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.

In 64-bit mode, use of the REX.R prefix permits this instruction to access additional registers (XMM8-XMM15).

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[VLMAX-1:64] (Unmodified)

VMOVLPS (VEX.128 encoded load)
DEST[63:0] ← SRC2[63:0]
DEST[127:64] ← SRC1[127:64]
DEST[VLMAX-1:128] ← 0

VMOVLPS (store)
DEST[63:0] ← SRC[63:0]

Intel C/C++ Compiler Intrinsic Equivalent

MOVLPS __m128 _mm_loadl_pi (__m128 a, __m64 *p)

MOVLPS void _mm_storel_pi (__m64 *p, __m128 a)

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Exceptions Type 5; additionally

#UD	If VEX.L= 1.
	If VEX.vvvv != 1111B.

...

MOVMSKPD—Extract Packed Double-Precision Floating-Point Sign Mask

Opcode/ Instruction	Op/ En	64/32-bit Mode	CPUID Feature Flag	Description
66 0F 50 /r MOVMSKPD *reg, xmm*	A	V/V	SSE2	Extract 2-bit sign mask from *xmm* and store in *reg*. The upper bits of r32 or r64 are filled with zeros.
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 zeroed.
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 zeroed.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
A	ModRM:reg (w)	ModRM:reg (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-bit mask, and stores the mask in the destination operand (first operand). The source operand is an XMM register, and the destination operand is a general-purpose register. The mask is stored in the 2 low-order bits of the destination operand. Zero-extend the upper bits of the destination.

In 64-bit mode, the instruction can access additional registers (XMM8-XMM15, R8-R15) when used with a REX.R prefix. The default operand size is 64-bit in 64-bit mode.

128-bit versions: The source operand is a YMM register. The destination operand is a general purpose register.

VEX.256 encoded version: 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

(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
```

VMOVMSKPD (VEX.256 encoded version)

```
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
```

Intel C/C++ Compiler Intrinsic Equivalent

MOVMSKPD 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 Packed Single-Precision Floating-Point Sign Mask

Opcode/ Instruction	Op/ En	64/32-bit Mode	CPUID Feature Flag	Description
0F 50 /r MOVMSKPS *reg, xmm*	A	V/V	SSE	Extract 4-bit sign mask from *xmm* and store in *reg*. The upper bits of r32 or r64 are filled with zeros.
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 zeroed.
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 zeroed.

Instruction Operand Encoding[1]

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
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

1. ModRM.MOD = 011B required

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 instruction can access additional registers (XMM8-XMM15, R8-R15) when used with a REX.R prefix. The default operand size is 64-bit in 64-bit mode.

128-bit versions: The source operand is a YMM register. The destination operand is a general purpose register.

VEX.256 encoded version: 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

```
DEST[0] ← SRC[31];
DEST[1] ← SRC[63];
DEST[2] ← SRC[95];
DEST[3] ← SRC[127];

IF DEST = r32
    THEN DEST[31:4] ← ZeroExtend;
    ELSE DEST[63:4] ← ZeroExtend;
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
```

VMOVMSKPS (VEX.256 encoded version)

```
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
```

Intel C/C++ Compiler Intrinsic Equivalent

int _mm_movemask_ps(__m128 a)

int _mm256_movemask_ps(__m256 a)

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Exceptions Type 7; additionally

#UD	If VEX.vvvv != 1111B.

...

MOVNTDQA — Load Double Quadword Non-Temporal Aligned Hint

Opcode/ Instruction	Op/ En	64/32-bit Mode	CPUID Feature Flag	Description
66 0F 38 2A /r MOVNTDQA *xmm1, m128*	A	V/V	SSE4_1	Move double quadword from *m128* to *xmm* 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

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
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. A processor implementation may make use of the non-temporal hint associated with this instruction if the memory source is WC (write combining) memory type. An implementation may also make use of the non-temporal hint associated with this instruction if the memory source is WB (write back) memory type.

A processor's implementation of the non-temporal hint does not override the effective memory type semantics, 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. Another implementation of the hint for WC memory type may optimize data transfer throughput of WC reads. A third implementation may optimize cache reads generated by MOVNTDQA on WB memory type to reduce cache evictions.

WC Streaming Load Hint

For WC memory type in particular, the processor never appears to read the data into the cache hierarchy. Instead, the non-temporal 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 memory currently residing in 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 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 Chapter 11, "Memory Cache Control" of *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*.

Because the WC protocol uses a weakly-ordered memory consistency model, an MFENCE or locked instruction should be used in conjunction with MOVNTDQA instructions if multiple processors might reference the same WC memory locations or in order to synchronize reads of a processor with writes by other agents in the system. Because of the speculative nature of fetching due to MOVNTDQA, Streaming loads must not be used to reference memory addresses that are mapped to I/O devices having side effects or when reads to these devices are destructive. For additional information on MOVNTDQA usages, see Section 12.10.3 in Chapter 12, "Programming with SSE3, SSSE3 and SSE4" of *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*.

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[VLMAX-1:128] (Unmodified)
```

VMOVNTDQA (VEX.128 encoded form)

```
DEST ← SRC
DEST[VLMAX-1:128] ← 0
```

Intel C/C++ Compiler Intrinsic Equivalent

MOVNTDQA __m128i _mm_stream_load_si128 (__m128i *p);

Flags Affected

None

Other Exceptions

See Exceptions Type 1.SSE4.1; additionally

#UD	If VEX.L= 1.
	If VEX.vvvv != 1111B.

...

MOVNTDQ—Store Double Quadword Using Non-Temporal Hint

Opcode/ Instruction	Op/ En	64/32-bit Mode	CPUID Feature Flag	Description
66 0F E7 /r MOVNTDQ *m128*, xmm	A	V/V	SSE2	Move double quadword from *xmm* 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]

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
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 *Intel® 64 and IA-32 Architectures 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 MOVNTDQ instructions if multiple processors might use different memory types to read/write the destination memory locations.

1. ModRM.MOD = 011B required

In 64-bit mode, use of the REX.R prefix permits this instruction to access additional registers (XMM8-XMM15).

Note: In VEX-128 encoded versions, VEX.vvvv is reserved and must be 1111b, VEX.L must be 0; otherwise instructions will #UD.

Operation

DEST ← SRC;

Intel C/C++ Compiler Intrinsic Equivalent

MOVNTDQ void _mm_stream_pd(double* p, __m128d a)

VMOVNTDQ void _mm256_stream_si256 (__m256i * p, __m256i a);

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Exceptions Type 1.SSE2; additionally

#UD	If VEX.vvvv != 1111B.

...

MOVNTPD—Store Packed Double-Precision Floating-Point Values Using Non-Temporal Hint

Opcode/ Instruction	Op/ En	64/32-bit Mode	CPUID Feature Flag	Description
66 0F 2B /r MOVNTPD *m128, xmm*	A	V/V	SSE2	Move packed double-precision floating-point values from *xmm* 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]

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
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 Temporal vs. Non-Temporal Data" in Chapter 10 in the *Intel® 64 and IA-32 Architectures 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.

In 64-bit mode, use of the REX.R prefix permits this instruction to access additional registers (XMM8-XMM15).

Note: In VEX-128 encoded versions, VEX.vvvv is reserved and must be 1111b, VEX.L must be 0; otherwise instructions will #UD.

Operation

```
DEST ← SRC;
```

Intel C/C++ Compiler Intrinsic Equivalent

MOVNTPD void _mm_stream_pd(double *p, __m128d a)

VMOVNTPD void _mm256_stream_pd (double * p, __m256d a);

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Exceptions Type 1.SSE2; additionally

#UD	If VEX.vvvv != 1111B.

...

1. ModRM.MOD = 011B required

MOVNTPS—Store Packed Single-Precision Floating-Point Values Using Non-Temporal Hint

Opcode/ Instruction	Op/ En	64/32-bit Mode	CPUID Feature Flag	Description
0F 2B /r MOVNTPS *m128, xmm*	A	V/V	SSE	Move packed single-precision floating-point values from *xmm* to *m128* 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]

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
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 Temporal vs. Non-Temporal Data" in Chapter 10 in the *Intel® 64 and IA-32 Architectures 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.

In 64-bit mode, use of the REX.R prefix permits this instruction to access additional registers (XMM8-XMM15).

1. ModRM.MOD = 011B required

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

Operation

DEST ← SRC;

Intel C/C++ Compiler Intrinsic Equivalent

MOVNTDQ void _mm_stream_ps(float * p, __m128 a)

VMOVNTPS void _mm256_stream_ps (float * p, __m256 a);

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Exceptions Type 1.SSE; additionally

#UD	If VEX.vvvv != 1111B.

...

MOVSD—Move Scalar Double-Precision Floating-Point Value

Opcode/ Instruction	Op/ En	64/32-bit Mode	CPUID Feature Flag	Description
F2 0F 10 /r MOVSD *xmm1, xmm2/m64*	A	V/V	SSE2	Move scalar double-precision floating-point value from *xmm2/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

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
A	ModRM:reg (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 instruction can be used to move a double-precision floating-point value to and from the low quadword of an XMM register and a 64-bit memory location, or to move a double-precision floating-point value between the low quadwords of two XMM registers. The instruction cannot be used to transfer data between memory locations.

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.

In 64-bit mode, use of the REX.R prefix permits this instruction to access additional registers (XMM8-XMM15).

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[VLMAX-1:64] (Unmodified)

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[VLMAX-1:128] (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[VLMAX-1:128] ← 0

VMOVSD (VEX.NDS.128.F2.0F 10 /r: VMOVSD xmm1, xmm2, xmm3)
DEST[63:0] ← SRC2[63:0]
DEST[127:64] ← SRC1[127:64]
DEST[VLMAX-1:128] ← 0

VMOVSD (VEX.NDS.128.F2.0F 10 /r: VMOVSD xmm1, m64)
DEST[63:0] ← SRC[63:0]
DEST[VLMAX-1:64] ← 0

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_store_sd (__m128d a, __m128d b)

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Exceptions Type 5; additionally

#UD	If VEX.vvvv != 1111B.

...

MOVSHDUP—Move Packed Single-FP High and Duplicate

Opcode/ Instruction	Op/ En	64/32-bit Mode	CPUID Feature Flag	Description
F3 0F 16 /r MOVSHDUP *xmm1, xmm2/m128*	A	V/V	SSE3	Move two single-precision floating-point values from the higher 32-bit operand of each qword in *xmm2/m128* to *xmm1* and duplicate each 32-bit operand to the lower 32-bits of each qword.
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

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

Description

The linear address corresponds to the address of the least-significant byte of the referenced memory data. When a memory address is indicated, the 16 bytes of data at memory location m128 are loaded and the single-precision elements in positions 1 and 3 are duplicated. When the register-register form of this operation is used, the same operation is performed but with data coming from the 128-bit source register. See Figure 3-28.

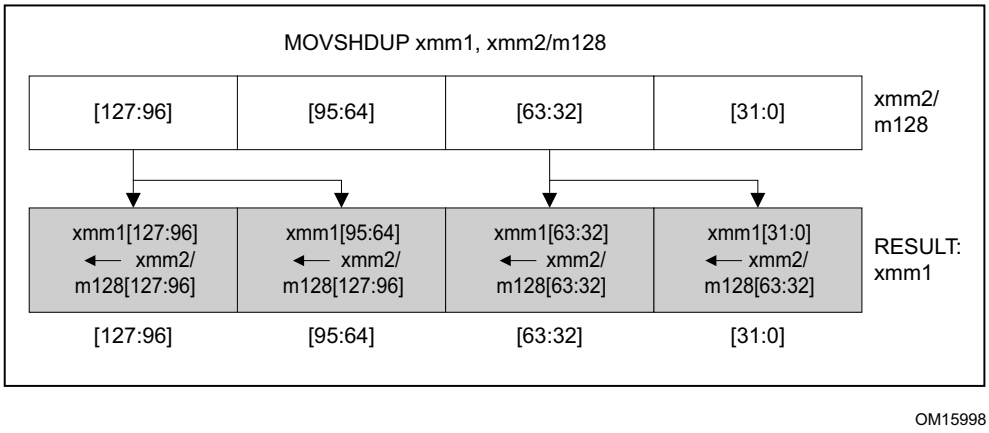


Figure 3-28 MOVSHDUP—Move Packed Single-FP High and Duplicate

In 64-bit mode, use of the REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

128-bit Legacy SSE version: Bits (VLMAX-1:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: Bits (VLMAX-1:128) of the destination YMM register are zeroed.

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

Operation

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[VLMAX-1:128] (Unmodified)

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[VLMAX-1:128] ← 0

VMOVSHDUP (VEX.256 encoded version)
DEST[31:0] ← SRC[63:32]
DEST[63:32] ← SRC[63:32]
DEST[95:64] ← SRC[127:96]
DEST[127:96] ← SRC[127:96]
DEST[159:128] ← SRC[191:160]
DEST[191:160] ← SRC[191:160]
DEST[223:192] ← SRC[255:224]
DEST[255:224] ← SRC[255:224]

Intel C/C++ Compiler Intrinsic Equivalent

(V)MOVSHDUP __m128 _mm_movehdup_ps(__m128 a)

VMOVSHDUP __m256 _mm256_movehdup_ps (__m256 a);

Exceptions

General protection exception if not aligned on 16-byte boundary, regardless of segment.

Numeric Exceptions

None

Other Exceptions

See Exceptions Type 2.

...

MOVSLDUP—Move Packed Single-FP Low and Duplicate

Opcode/ Instruction	Op/ En	64/32-bit Mode	CPUID Feature Flag	Description
F3 0F 12 /r MOVSLDUP *xmm1, xmm2/m128*	A	V/V	SSE3	Move two single-precision floating-point values from the lower 32-bit operand of each qword in *xmm2/m128* to *xmm1* and duplicate each 32-bit operand to the higher 32-bits of each qword.
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

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

Description

The linear address corresponds to the address of the least-significant byte of the referenced memory data. When a memory address is indicated, the 16 bytes of data at memory location m128 are loaded and the single-precision elements in positions 0 and 2 are duplicated. When the register-register form of this operation is used, the same operation is performed but with data coming from the 128-bit source register.

See Figure 3-29.

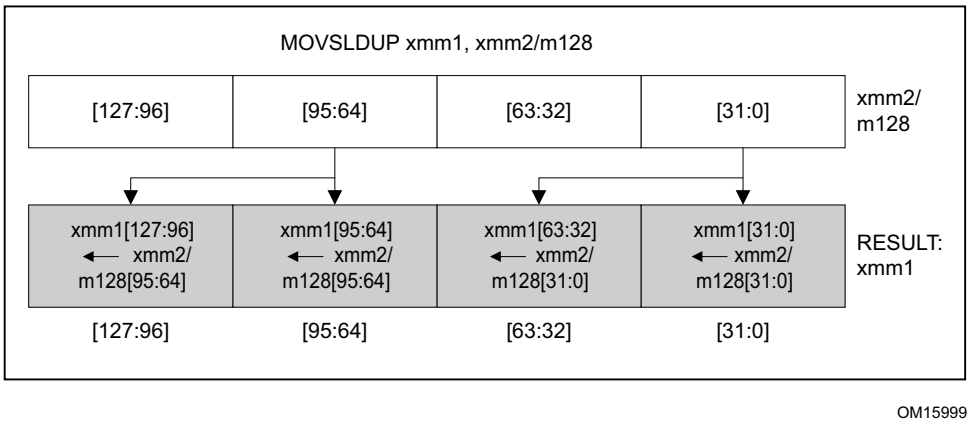


Figure 3-29 MOVSLDUP—Move Packed Single-FP Low and Duplicate

In 64-bit mode, use of the REX.R prefix permits this instruction to access additional registers (XMM8-XMM15).

128-bit Legacy SSE version: Bits (VLMAX-1:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: Bits (VLMAX-1:128) of the destination YMM register are zeroed.

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

Operation

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[VLMAX-1:128] (Unmodified)

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[VLMAX-1:128] ← 0

VMOVSLDUP (VEX.256 encoded version)
DEST[31:0] ← SRC[31:0]
DEST[63:32] ← SRC[31:0]
DEST[95:64] ← SRC[95:64]

DEST[127:96] ← SRC[95:64]
DEST[159:128] ← SRC[159:128]
DEST[191:160] ← SRC[159:128]
DEST[223:192] ← SRC[223:192]
DEST[255:224] ← SRC[223:192]

Intel C/C++ Compiler Intrinsic Equivalent

(V)MOVSLDUP __m128 _mm_moveldup_ps(__m128 a)

VMOVSLDUP __m256 _mm256_moveldup_ps (__m256 a);

Exceptions

General protection exception if not aligned on 16-byte boundary, regardless of segment.

Numeric Exceptions

None.

Other Exceptions

See Exceptions Type 4; additionally

#UD	If VEX.vvvv != 1111B.

...

MOVSS—Move Scalar Single-Precision Floating-Point Values

Opcode/ Instruction	Op/ En	64/32-bit Mode	CPUID Feature Flag	Description
F3 0F 10 /r MOVSS *xmm1, xmm2/m32*	A	V/V	SSE	Move scalar single-precision floating-point value from *xmm2/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, xmm*	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

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
A	ModRM:reg (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

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.

In 64-bit mode, use of the REX.R prefix permits this instruction to access additional registers (XMM8-XMM15).

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.

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.

Operation

MOVSS (Legacy SSE version when the source and destination operands are both XMM registers)
DEST[31:0] ← SRC[31:0]
DEST[VLMAX-1:32] (Unmodified)

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[VLMAX-1:128] (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[VLMAX-1:128] ← 0

VMOVSS (VEX.NDS.128.F3.0F 10 /r where the source and destination are XMM registers)
DEST[31:0] ← SRC2[31:0]
DEST[127:32] ← SRC1[127:32]
DEST[VLMAX-1:128] ← 0

VMOVSS (VEX.NDS.128.F3.0F 10 /r when the source operand is memory and the destination is an XMM register)
DEST[31:0] ← SRC[31:0]
DEST[VLMAX-1:32] ← 0

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	CPUID Feature Flag	Description
66 0F 10 /*r* MOVUPD *xmm1, xmm2/m128*	A	V/V	SSE2	Move packed double-precision floating-point values from *xmm2/m128* to xmm1.
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.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.
66 0F 11 /*r* MOVUPD *xmm2/m128, xmm*	B	V/V	SSE2	Move packed double-precision floating-point values from xmm1 to *xmm2/m128*.
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 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

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

Description

128-bit versions:

Moves a double quadword containing two 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,

store the contents of an XMM register into a 128-bit memory location, or move data between two XMM registers.

128-bit Legacy SSE version: Bits (VLMAX-1:128) 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.[1]

To move double-precision floating-point values to and from memory locations that are known to be aligned on 16-byte boundaries, use the MOVAPD instruction.

While executing in 16-bit addressing mode, a linear address for a 128-bit data access that overlaps the end of a 16-bit segment is not allowed and is defined as reserved behavior. A specific processor implementation may or may not generate a general-protection exception (#GP) in this situation, and the address that spans the end of the segment may or may not wrap around to the beginning of the segment.

In 64-bit mode, use of the REX.R prefix permits this instruction to access additional registers (XMM8-XMM15).

VEX.128 encoded version: Bits (VLMAX-1:128) of the destination YMM register are zeroed.

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.

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

Operation

MOVUPD (128-bit load and register-copy form Legacy SSE version)
DEST[127:0] ← SRC[127:0]
DEST[VLMAX-1:128] (Unmodified)

(V)MOVUPD (128-bit store form)
DEST[127:0] ← SRC[127:0]

VMOVUPD (VEX.128 encoded version)
DEST[127:0] ← SRC[127:0]
DEST[VLMAX-1:128] ← 0

VMOVUPD (VEX.256 encoded version)
DEST[255:0] ← SRC[255:0]

1. If alignment checking is enabled (CR0.AM = 1, RFLAGS.AC = 1, and CPL = 3), an alignment-check exception (#AC) may or may not be generated (depending on processor implementation) when the operand is not aligned on an 8-byte boundary.

Intel C/C++ Compiler Intrinsic Equivalent

MOVUPD __m128 _mm_loadu_pd(double * p)

MOVUPD void _mm_storeu_pd(double *p, __m128 a)

VMOVUPD __m256d _mm256_loadu_pd (__m256d * p);

VMOVUPD _mm256_storeu_pd(_m256d *p, __m256d 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	CPUID Feature Flag	Description
0F 10 /*r* MOVUPS *xmm1, xmm2/m128*	A	V/V	SSE	Move packed single-precision floating-point values from *xmm2/m128* to xmm1.
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.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.
0F 11 /*r* MOVUPS *xmm2/m128, xmm1*	B	V/V	SSE	Move packed single-precision floating-point values from xmm1 to *xmm2/m128*.
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 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

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

Description

128-bit versions:

Moves a double quadword containing four 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, store the contents of an XMM register into a 128-bit memory location, or move data between two XMM registers.

128-bit Legacy SSE version: Bits (VLMAX-1:128) 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.[1]

To move packed single-precision floating-point values to and from memory locations that are known to be aligned on 16-byte boundaries, use the MOVAPS instruction.

While executing in 16-bit addressing mode, a linear address for a 128-bit data access that overlaps the end of a 16-bit segment is not allowed and is defined as reserved behavior. A specific processor implementation may or may not generate a general-protection exception (#GP) in this situation, and the address that spans the end of the segment may or may not wrap around to the beginning of the segment.

In 64-bit mode, use of the REX.R prefix permits this instruction to access additional registers (XMM8-XMM15).

VEX.128 encoded version: Bits (VLMAX-1:128) of the destination YMM register are zeroed.

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.

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

Operation

MOVUPS (128-bit load and register-copy form Legacy SSE version)

DEST[127:0] ← SRC[127:0]

DEST[VLMAX-1:128] (Unmodified)

1. If alignment checking is enabled (CR0.AM = 1, RFLAGS.AC = 1, and CPL = 3), an alignment-check exception (#AC) may or may not be generated (depending on processor implementation) when the operand is not aligned on an 8-byte boundary.

(V)MOVUPS (128-bit store form)
DEST[127:0] ← SRC[127:0]

VMOVUPS (VEX.128 encoded load-form)
DEST[127:0] ← SRC[127:0]
DEST[VLMAX-1:128] ← 0

VMOVUPS (VEX.256 encoded version)
DEST[255:0] ← SRC[255:0]

Intel C/C++ Compiler Intrinsic Equivalent

MOVUPS __m128 _mm_loadu_ps(double * p)

MOVUPS void _mm_storeu_ps(double *p, __m128 a)

VMOVUPS __m256 _mm256_loadu_ps (__m256 * p);

VMOVUPS _mm256_storeu_ps(_m256 *p, __m256 a);

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Exceptions Type 4

Note treatment of #AC varies; additionally

#UD	If VEX.vvvv != 1111B.

...

MPSADBW — Compute Multiple Packed Sums of Absolute Difference

Opcode/ Instruction	Op/ En	64/32-bit Mode	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

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
A	ModRM:reg (r, w)	ModRM:r/m (r)	imm8	NA
B	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA

Description

MPSADBW sums the absolute difference (SAD) of a pair of unsigned bytes for a group of 4 byte pairs, and produces 8 SAD results (one for each 4 byte-pairs) stored as 8 word integers in the destination operand (first operand). Each 4 byte pairs are selected from the source operand (first operand) and the destination according to the bit fields specified in the immediate byte (third operand).

The immediate byte provides two bit fields:

SRC_OFFSET: the value of Imm8[1:0]*32 specifies the offset of the 4 sequential source bytes in the source operand.

DEST_OFFSET: the value of Imm8[2]*32 specifies the offset of the first of 8 groups of 4 sequential destination bytes in the destination operand. The next four destination bytes starts at DEST_OFFSET + 8, etc.

The SAD operation is repeated 8 times, each time using the same 4 source bytes but selecting the next group of 4 destination bytes starting at the next higher byte in the destination. Each 16-bit sum is written to destination.

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

VEX.128 encoded version: Bits (VLMAX-1:128) 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

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[VLMAX-1:128] (Unmodified)
```

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[VLMAX-1:128] ← 0
```

Intel C/C++ Compiler Intrinsic Equivalent

MPSADBW __m128i _mm_mpsadbw_epu8 (__m128i s1, __m128i s2, const int mask);

Flags Affected

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	CPUID Feature Flag	Description
66 0F 59 /r MULPD *xmm1*, *xmm2/m128*	A	V/V	SSE2	Multiply packed double-precision floating-point values in *xmm2/m128* by *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

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
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 source operand (second operand) and the destination operand (first operand), and stores the packed double-precision floating-point results in the destination operand. The source operand can be an XMM register or a 128-bit memory location. The destination operand is an XMM register. See Figure 11-3 in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*, for an illustration of a SIMD double-precision floating-point operation.

In 64-bit mode, use of the REX.R prefix permits this instruction to access additional registers (XMM8-XMM15).

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 (VLMAX-1:128) of the corresponding YMM register destination are unmodified.

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

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

Operation

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[VLMAX-1:128] (Unmodified)

VMULPD (VEX.128 encoded version)
DEST[63:0] ← SRC1[63:0] * SRC2[63:0]
DEST[127:64] ← SRC1[127:64] * SRC2[127:64]
DEST[VLMAX-1:128] ← 0

VMULPD (VEX.256 encoded version)
DEST[63:0] ← SRC1[63:0] * SRC2[63:0]
DEST[127:64] ← SRC1[127:64] * SRC2[127:64]
DEST[191:128] ← SRC1[191:128] * SRC2[191:128]
DEST[255:192] ← SRC1[255:192] * SRC2[255:192]

Intel C/C++ Compiler Intrinsic Equivalent

MULPD __m128d _mm_mul_pd (m128d a, m128d b)

VMULPD __m256d _mm256_mul_pd (__m256d a, __m256d 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	CPUID Feature Flag	Description
0F 59 /r MULPS *xmm1, xmm2/m128*	A	V/V	SSE	Multiply packed single-precision floating-point values in *xmm2/mem* by *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

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
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 four packed single-precision floating-point values from the source operand (second operand) and the destination operand (first operand), and stores the packed single-precision floating-point results in the destination operand. See Figure 10-5 in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*, for an illustration of a SIMD single-precision floating-point operation.

In 64-bit mode, use of the REX.R prefix permits this instruction to access additional registers (XMM8-XMM15).

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 (VLMAX-1:128) of the corresponding YMM register destination are unmodified.

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

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

Operation

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[VLMAX-1:128] (Unmodified)

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[VLMAX-1:128] ← 0

VMULPS (VEX.256 encoded version)
DEST[31:0] ← SRC1[31:0] * SRC2[31:0]
DEST[63:32] ← SRC1[63:32] * SRC2[63:32]
DEST[95:64] ← SRC1[95:64] * SRC2[95:64]
DEST[127:96] ← SRC1[127:96] * SRC2[127:96]
DEST[159:128] ← SRC1[159:128] * SRC2[159:128]
DEST[191:160]← SRC1[191:160] * SRC2[191:160]
DEST[223:192] ← SRC1[223:192] * SRC2[223:192]
DEST[255:224] ← SRC1[255:224] * SRC2[255:224].

Intel C/C++ Compiler Intrinsic Equivalent

MULPS __m128 _mm_mul_ps(__m128 a, __m128 b)

VMULPS __m256 _mm256_mul_ps (__m256 a, __m256 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	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

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
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 source operand (second operand) by the low double-precision floating-point value in the destination operand (first operand), and stores the double-precision floating-point result in the destination operand. The source operand can be an XMM register or a 64-bit memory location. The destination operand is an XMM register. 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.

In 64-bit mode, use of the REX.R prefix permits this instruction to access additional registers (XMM8-XMM15).

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

VEX.128 encoded version: Bits (VLMAX-1:128) of the destination YMM register are zeroed.

Operation

MULSD (128-bit Legacy SSE version)
DEST[63:0] ← DEST[63:0] * SRC[63:0]
DEST[VLMAX-1:64] (Unmodified)

VMULSD (VEX.128 encoded version)
DEST[63:0] ← SRC1[63:0] * SRC2[63:0]
DEST[127:64] ← SRC1[127:64]

DEST[VLMAX-1:128] ← 0

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	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

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
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 source operand (second operand) by the low single-precision floating-point value in the destination operand (first operand), and stores the single-precision floating-point result in the destination operand. The source operand can be an XMM register or a 32-bit memory location. The destination operand 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.

In 64-bit mode, use of the REX.R prefix permits this instruction to access additional registers (XMM8-XMM15).

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

VEX.128 encoded version: Bits (VLMAX-1:128) of the destination YMM register are zeroed.

Operation

MULSS (128-bit Legacy SSE version)
DEST[31:0] ← DEST[31:0] * SRC[31:0]
DEST[VLMAX-1:32] (Unmodified)

VMULSS (VEX.128 encoded version)
DEST[31:0] ← SRC1[31:0] * SRC2[31:0]
DEST[127:32] ← SRC1[127:32]
DEST[VLMAX-1:128] ← 0

Intel C/C++ Compiler Intrinsic Equivalent

MULSS __m128 _mm_mul_ss(__m128 a, __m128 b)

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal.

Other Exceptions

See Exceptions Type 3

...

MWAIT—Monitor Wait

...

Description

MWAIT instruction provides hints to allow the processor to enter an implementation-dependent optimized state. There are two principal targeted usages: address-range monitor and advanced power management. Both usages of MWAIT require the use of the MONITOR instruction.

A CPUID feature flag (ECX bit 3; CPUID executed EAX = 1) indicates the availability of MONITOR and MWAIT in the processor. When set, MWAIT may be executed only at privilege level 0 (use at any other privilege level results in an invalid-opcode exception). The operating system or system BIOS may disable this instruction by using the IA32_MISC_ENABLE MSR; disabling MWAIT clears the CPUID feature flag and causes execution to generate an illegal opcode exception.

This instruction's operation is the same in non-64-bit modes and 64-bit mode.

...

6. Updates to Chapter 4, Volume 2B

Change bars show changes to Chapter 4 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2B:* Instruction Set Reference, N-Z.

...

ORPD—Bitwise Logical OR of 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	Bitwise OR of *xmm2/m128* and *xmm1*.
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

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
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

In 64-bit mode, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

128-bit Legacy SSE version: The second source 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 (VLMAX-1:128) of the corresponding YMM register destination are unmodified.

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

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

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

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[VLMAX-1:128] (Unmodified)

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[VLMAX-1:128] ← 0

VORPD (VEX.256 encoded version)
DEST[63:0] ← SRC1[63:0] BITWISE OR SRC2[63:0]
DEST[127:64] ← SRC1[127:64] BITWISE OR SRC2[127:64]
DEST[191:128] ← SRC1[191:128] BITWISE OR SRC2[191:128]
DEST[255:192] ← SRC1[255:192] BITWISE OR SRC2[255:192]

Intel® C/C++ Compiler Intrinsic Equivalent

ORPD __m128d _mm_or_pd(__m128d a, __m128d b);

VORPD __m256d _mm256_or_pd (__m256d a, __m256d b);

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Exceptions Type 4; additionally

#UD	If VEX.L = 1.

...

ORPS—Bitwise Logical OR of 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	Bitwise OR of *xmm1* and *xmm2/m128.*
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

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
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.

In 64-bit mode, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

128-bit Legacy SSE version: The second source 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 (VLMAX-1:128) of the corresponding YMM register destination are unmodified.

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

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

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

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[VLMAX-1:128] (Unmodified)

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[VLMAX-1:128] ← 0

VORPS (VEX.256 encoded version)
DEST[31:0] ← SRC1[31:0] BITWISE OR SRC2[31:0]
DEST[63:32] ← SRC1[63:32] BITWISE OR SRC2[63:32]
DEST[95:64] ← SRC1[95:64] BITWISE OR SRC2[95:64]
DEST[127:96] ← SRC1[127:96] BITWISE OR SRC2[127:96]
DEST[159:128] ← SRC1[159:128] BITWISE OR SRC2[159:128]
DEST[191:160]← SRC1[191:160] BITWISE OR SRC2[191:160]
DEST[223:192] ← SRC1[223:192] BITWISE OR SRC2[223:192]
DEST[255:224] ← SRC1[255:224] BITWISE OR SRC2[255:224].

Intel C/C++ Compiler Intrinsic Equivalent

ORPS __m128 _mm_or_ps (__m128 a, __m128 b);

VORPS __m256 _mm256_or_ps (__m256 a, __m256 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
0F 38 1C /r[1] PABSB mm1, mm2/m64	A	V/V	SSSE3	Compute the absolute value of bytes in mm2/m64 and store UNSIGNED result in mm1.
66 0F 38 1C /r PABSB xmm1, xmm2/m128	A	V/V	SSSE3	Compute the absolute value of bytes in xmm2/m128 and store UNSIGNED result in xmm1.
0F 38 1D /r[1] PABSW mm1, mm2/m64	A	V/V	SSSE3	Compute the absolute value of 16-bit integers in mm2/m64 and store UNSIGNED result in mm1.
66 0F 38 1D /r PABSW xmm1, xmm2/m128	A	V/V	SSSE3	Compute the absolute value of 16-bit integers in xmm2/m128 and store UNSIGNED result in xmm1.
0F 38 1E /r[1] PABSD mm1, mm2/m64	A	V/V	SSSE3	Compute the absolute value of 32-bit integers in mm2/m64 and store UNSIGNED result in mm1.
66 0F 38 1E /r PABSD xmm1, xmm2/m128	A	V/V	SSSE3	Compute the absolute value of 32-bit integers in xmm2/m128 and store UNSIGNED result in xmm1.
VEX.128.66.0F38.WIG 1C /r VPABSB xmm1, xmm2/m128	A	V/V	AVX	Compute the absolute value of bytes in xmm2/m128 and store UNSIGNED result in xmm1.
VEX.128.66.0F38.WIG 1D /r VPABSW xmm1, xmm2/m128	A	V/V	AVX	Compute the absolute value of 16- bit integers in xmm2/m128 and store UNSIGNED result in xmm1.
VEX.128.66.0F38.WIG 1E /r VPABSD xmm1, xmm2/m128	A	V/V	AVX	Compute the absolute value of 32- bit integers in xmm2/m128 and store UNSIGNED result in xmm1.

NOTES:

1. See note in Section 2.4, “Instruction Exception Specification”.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
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 (the second operand) and stores the UNSIGNED results in the destination operand (the first operand). PABSB operates on signed bytes, PABSW operates on 16-bit words, and PABSD operates on signed 32-bit integers. The source operand can be an MMX register or a 64-bit memory location, or it can be an XMM register or a 128-bit memory location. The destination operand can be an MMX or an XMM register. Both operands can be MMX register or XMM registers. When the source operand is a 128-bit memory operand, the operand must be aligned on a 16byte boundary or a general-protection exception (#GP) will be generated.

In 64-bit mode, use the REX prefix to access additional registers.

128-bit Legacy SSE version: Bits (VLMAX-1:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: Bits (VLMAX-1:128) 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

PABSB (with 64 bit operands)

Unsigned DEST[7:0] ← ABS(SRC[7:0])
Repeat operation for 2nd through 7th bytes
Unsigned DEST[63:56] ← ABS(SRC[63:56])

PABSB (with 128 bit operands)

Unsigned DEST[7:0] ← ABS(SRC[7:.0])
Repeat operation for 2nd through 15th bytes
Unsigned DEST[127:120] ← ABS(SRC[127:120])

PABSW (with 64 bit operands)

Unsigned DEST[15:0] ← ABS(SRC[15:0])
Repeat operation for 2nd through 3rd 16-bit words
Unsigned DEST[63:48] ← ABS(SRC[63:48])

PABSW (with 128 bit operands)

Unsigned DEST[15:0] ← ABS(SRC[15:0])
Repeat operation for 2nd through 7th 16-bit words
Unsigned DEST[127:112] ← ABS(SRC[127:112])

PABSD (with 64 bit operands)

Unsigned DEST[31:0] ← ABS(SRC[31:0])
Unsigned DEST[63:32] ← ABS(SRC[63:32])

PABSD (with 128 bit operands)

Unsigned DEST[31:0] ← ABS(SRC[31:0])
Repeat operation for 2nd through 3rd 32-bit double words
Unsigned DEST[127:96] ← ABS(SRC[127:96])

PABSB (128-bit Legacy SSE version)
DEST[127:0] ← BYTE_ABS(SRC)
DEST[VLMAX-1:128] (Unmodified)

VPABSB (VEX.128 encoded version)
DEST[127:0] ← BYTE_ABS(SRC)
DEST[VLMAX-1:128] ← 0

PABSW (128-bit Legacy SSE version)
DEST[127:0] ← WORD_ABS(SRC)
DEST[VLMAX-1:128] (Unmodified)

VPABSW (VEX.128 encoded version)
DEST[127:0] ← WORD_ABS(SRC)
DEST[VLMAX-1:128] ← 0

PABSD (128-bit Legacy SSE version)
DEST[127:0] ← DWORD_ABS(SRC)
DEST[VLMAX-1:128] (Unmodified)

VPABSD (VEX.128 encoded version)
DEST[127:0] ← DWORD_ABS(SRC)
DEST[VLMAX-1:128] ← 0

Intel C/C++ Compiler Intrinsic Equivalents

PABSB __m64 _mm_abs_pi8 (__m64 a)

PABSB __m128i _mm_abs_epi8 (__m128i a)

PABSW __m64 _mm_abs_pi16 (__m64 a)

PABSW __m128i _mm_abs_epi16 (__m128i a)

PABSD __m64 _mm_abs_pi32 (__m64 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
0F 63 /r[1] PACKSSWB *mm1, mm2/m64*	A	V/V	MMX	Converts 4 packed signed word integers from *mm1* and from *mm2/m64* into 8 packed signed byte integers in *mm1* using signed saturation.
66 0F 63 /*r* PACKSSWB *xmm1, xmm2/m128*	A	V/V	SSE2	Converts 8 packed signed word integers from *xmm1* and from *xxm2/m128* into 16 packed signed byte integers in *xxm1* using signed saturation.
0F 6B /r[1] PACKSSDW *mm1, mm2/m64*	A	V/V	MMX	Converts 2 packed signed doubleword integers from *mm1* and from *mm2/m64* into 4 packed signed word integers in *mm1* using signed saturation.
66 0F 6B /*r* PACKSSDW *xmm1, xmm2/m128*	A	V/V	SSE2	Converts 4 packed signed doubleword integers from *xmm1* and from *xxm2/m128* into 8 packed signed word integers in *xxm1* 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.

NOTES:

1. See note in Section 2.4, “Instruction Exception Specification”.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
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 4-2 for an example of the packing operation.

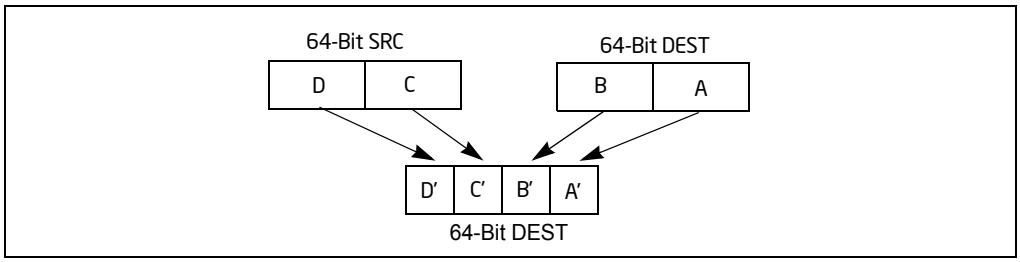


Figure 4-2 Operation of the PACKSSDW Instruction Using 64-bit Operands

The PACKSSWB instruction converts 4 or 8 signed word integers from the destination operand (first operand) and 4 or 8 signed word integers from the source operand (second operand) into 8 or 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 2 or 4 signed doublewords from the destination operand (first operand) and 2 or 4 signed doublewords from the source operand (second operand) into 4 or 8 signed words in the destination operand (see Figure 4-2). 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.

The PACKSSWB and PACKSSDW instructions operate on either 64-bit or 128-bit operands. When operating on 64-bit operands, the destination operand must be an MMX technology register and the source operand can be either an MMX technology register or a 64-bit memory location. When operating on 128-bit operands, the destination operand must be an XMM register and the source operand can be either an XMM register or a 128-bit memory location.

In 64-bit mode, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

128-bit Legacy SSE version: Bits (VLMAX-1:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: Bits (VLMAX-1:128) of the destination YMM register are zeroed. VEX.L must be 0, otherwise the instruction will #UD.

Operation

PACKSSWB (with 64-bit operands)

DEST[7:0] ← SaturateSignedWordToSignedByte DEST[15:0];
DEST[15:8] ← SaturateSignedWordToSignedByte DEST[31:16];
DEST[23:16] ← SaturateSignedWordToSignedByte DEST[47:32];
DEST[31:24] ← SaturateSignedWordToSignedByte DEST[63:48];
DEST[39:32] ← SaturateSignedWordToSignedByte SRC[15:0];
DEST[47:40] ← SaturateSignedWordToSignedByte SRC[31:16];
DEST[55:48] ← SaturateSignedWordToSignedByte SRC[47:32];
DEST[63:56] ← SaturateSignedWordToSignedByte SRC[63:48];

PACKSSDW (with 64-bit operands)

DEST[15:0] ← SaturateSignedDoublewordToSignedWord DEST[31:0];
DEST[31:16] ← SaturateSignedDoublewordToSignedWord DEST[63:32];
DEST[47:32] ← SaturateSignedDoublewordToSignedWord SRC[31:0];
DEST[63:48] ← SaturateSignedDoublewordToSignedWord SRC[63:32];

PACKSSWB (with 128-bit operands)

DEST[7:0]← SaturateSignedWordToSignedByte (DEST[15:0]);
DEST[15:8] ← SaturateSignedWordToSignedByte (DEST[31:16]);
DEST[23:16] ← SaturateSignedWordToSignedByte (DEST[47:32]);
DEST[31:24] ← SaturateSignedWordToSignedByte (DEST[63:48]);
DEST[39:32] ← SaturateSignedWordToSignedByte (DEST[79:64]);
DEST[47:40] ← SaturateSignedWordToSignedByte (DEST[95:80]);
DEST[55:48] ← SaturateSignedWordToSignedByte (DEST[111:96]);
DEST[63:56] ← SaturateSignedWordToSignedByte (DEST[127:112]);
DEST[71:64] ← SaturateSignedWordToSignedByte (SRC[15:0]);
DEST[79:72] ← SaturateSignedWordToSignedByte (SRC[31:16]);
DEST[87:80] ← SaturateSignedWordToSignedByte (SRC[47:32]);
DEST[95:88] ← SaturateSignedWordToSignedByte (SRC[63:48]);
DEST[103:96] ← SaturateSignedWordToSignedByte (SRC[79:64]);
DEST[111:104] ← SaturateSignedWordToSignedByte (SRC[95:80]);
DEST[119:112] ← SaturateSignedWordToSignedByte (SRC[111:96]);
DEST[127:120] ← SaturateSignedWordToSignedByte (SRC[127:112]);

PACKSSDW (with 128-bit operands)

DEST[15:0] ← SaturateSignedDwordToSignedWord (DEST[31:0]);
DEST[31:16] ← SaturateSignedDwordToSignedWord (DEST[63:32]);
DEST[47:32] ← SaturateSignedDwordToSignedWord (DEST[95:64]);
DEST[63:48] ← SaturateSignedDwordToSignedWord (DEST[127:96]);
DEST[79:64] ← SaturateSignedDwordToSignedWord (SRC[31:0]);
DEST[95:80] ← SaturateSignedDwordToSignedWord (SRC[63:32]);
DEST[111:96] ← SaturateSignedDwordToSignedWord (SRC[95:64]);
DEST[127:112] ← SaturateSignedDwordToSignedWord (SRC[127:96]);

PACKSSDW

DEST[127:0] ← SATURATING_PACK_DW(DEST, SRC)
DEST[VLMAX-1:128] (Unmodified)

VPACKSSDW

DEST[127:0] ← SATURATING_PACK_DW(DEST, SRC)
DEST[VLMAX-1:128] ← 0

PACKSSWB
DEST[127:0] ← SATURATING_PACK_WB(DEST, SRC)
DEST[VLMAX-1:128] (Unmodified)

VPACKSSWB
DEST[127:0] ← SATURATING_PACK_WB(DEST, SRC)
DEST[VLMAX-1:128] ← 0

Intel C/C++ Compiler Intrinsic Equivalents

PACKSSWB __m64 _mm_packs_pi16(__m64 m1, __m64 m2)

PACKSSWB __m128i _mm_packs_epi16(__m128i m1, __m128i m2)

PACKSSDW __m64 _mm_packs_pi32 (__m64 m1, __m64 m2)

PACKSSDW __m128i _mm_packs_epi32(__m128i m1, __m128i m2)

Flags Affected

None.

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Exceptions Type 4; additionally

#UD	If VEX.L = 1.

...

PACKUSDW — Pack with Unsigned Saturation

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 38 2B /r PACKUSDW *xmm1, xmm2/m128*	A	V/V	SSE4_1	Convert 4 packed signed doubleword integers from *xmm1* and 4 packed signed doubleword integers from *xmm2/m128* into 8 packed unsigned word integers in *xmm1* using unsigned saturation.
VEX.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

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
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 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, is stored in the destination.

128-bit Legacy SSE version: Bits (VLMAX-1:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: Bits (VLMAX-1:128) of the destination YMM register are zeroed. VEX.L must be 0, otherwise the instruction will #UD.

Operation

```
TMP[15:0] ← (DEST[31:0] < 0) ? 0 : DEST[15:0];
DEST[15:0] ← (DEST[31:0] > FFFFH) ? FFFFH : TMP[15:0] ;
TMP[31:16] ← (DEST[63:32] < 0) ? 0 : DEST[47:32];
DEST[31:16] ← (DEST[63:32] > FFFFH) ? FFFFH : TMP[31:16] ;
TMP[47:32] ← (DEST[95:64] < 0) ? 0 : DEST[79:64];
DEST[47:32] ← (DEST[95:64] > FFFFH) ? FFFFH : TMP[47:32] ;
TMP[63:48] ← (DEST[127:96] < 0) ? 0 : DEST[111:96];
DEST[63:48] ← (DEST[127:96] > FFFFH) ? FFFFH : TMP[63:48] ;
TMP[63:48] ← (DEST[127:96] < 0) ? 0 : DEST[111:96];
```

```
DEST[63:48] ← (DEST[127:96] > FFFFH) ? FFFFH : TMP[63:48] ;
TMP[79:64] ← (SRC[31:0] < 0) ? 0 : SRC[15:0];
DEST[63:48] ← (SRC[31:0] > FFFFH) ? FFFFH : TMP[79:64] ;
TMP[95:80] ← (SRC[63:32] < 0) ? 0 : SRC[47:32];
DEST[95:80] ← (SRC[63:32] > FFFFH) ? FFFFH : TMP[95:80] ;
TMP[111:96] ← (SRC[95:64] < 0) ? 0 : SRC[79:64];
DEST[111:96] ← (SRC[95:64] > FFFFH) ? FFFFH : TMP[111:96] ;
TMP[127:112] ← (SRC[127:96] < 0) ? 0 : SRC[111:96];
DEST[128:112] ← (SRC[127:96] > FFFFH) ? FFFFH : TMP[127:112] ;
```

PACKUSDW (128-bit Legacy SSE version)

```
DEST[127:0] ← UNSIGNED_SATURATING_PACK_DW(DEST, SRC)
DEST[VLMAX-1:128] (Unmodified)
```

VPACKUSDW (VEX.128 encoded version)

```
DEST[127:0] ← UNSIGNED_SATURATING_PACK_DW(SRC1, SRC2)
DEST[VLMAX-1:128] ← 0
```

Intel C/C++ Compiler Intrinsic Equivalent

PACKUSDW __m128i _mm_packus_epi32(__m128i m1, __m128i m2);

Flags Affected

None.

SIMD Exceptions

None.

Other Exceptions

See Exceptions Type 4; additionally

#UD	If VEX.L = 1.

...

PACKUSWB—Pack with Unsigned Saturation

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
0F 67 /r[1] PACKUSWB *mm, mm/m64*	A	V/V	MMX	Converts 4 signed word integers from *mm* and 4 signed word integers from *mm/m64* into 8 unsigned byte integers in *mm* using unsigned saturation.
66 0F 67 /r PACKUSWB *xmm1, xmm2/m128*	A	V/V	SSE2	Converts 8 signed word integers from *xmm1* and 8 signed word integers from *xmm2/m128* into 16 unsigned byte integers in *xmm1* using unsigned saturation.
VEX.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.

NOTES:

1. See note in Section 2.4, “Instruction Exception Specification”.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
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 4 or 8 signed word integers from the destination operand (first operand) and 4 or 8 signed word integers from the source operand (second operand) into 8 or 16 unsigned byte integers and stores the result in the destination operand. (See Figure 4-2 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 PACKUSWB instruction operates on either 64-bit or 128-bit operands. When operating on 64-bit operands, the destination operand must be an MMX technology register and the source operand can be either an MMX technology register or a 64-bit memory location. When operating on 128-bit operands, the destination operand must be an XMM register and the source operand can be either an XMM register or a 128-bit memory location.

In 64-bit mode, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

Operation

PACKUSWB (with 64-bit operands)

```
DEST[7:0] ← SaturateSignedWordToUnsignedByte DEST[15:0];
DEST[15:8] ← SaturateSignedWordToUnsignedByte DEST[31:16];
DEST[23:16] ← SaturateSignedWordToUnsignedByte DEST[47:32];
DEST[31:24] ← SaturateSignedWordToUnsignedByte DEST[63:48];
DEST[39:32] ← SaturateSignedWordToUnsignedByte SRC[15:0];
DEST[47:40] ← SaturateSignedWordToUnsignedByte SRC[31:16];
DEST[55:48] ← SaturateSignedWordToUnsignedByte SRC[47:32];
DEST[63:56] ← SaturateSignedWordToUnsignedByte SRC[63:48];
```

PACKUSWB (with 128-bit operands)

```
DEST[7:0]← SaturateSignedWordToUnsignedByte (DEST[15:0]);
DEST[15:8] ← SaturateSignedWordToUnsignedByte (DEST[31:16]);
DEST[23:16] ← SaturateSignedWordToUnsignedByte (DEST[47:32]);
DEST[31:24] ← SaturateSignedWordToUnsignedByte (DEST[63:48]);
DEST[39:32] ← SaturateSignedWordToUnsignedByte (DEST[79:64]);
DEST[47:40] ← SaturateSignedWordToUnsignedByte (DEST[95:80]);
DEST[55:48] ← SaturateSignedWordToUnsignedByte (DEST[111:96]);
DEST[63:56] ← SaturateSignedWordToUnsignedByte (DEST[127:112]);
DEST[71:64] ← SaturateSignedWordToUnsignedByte (SRC[15:0]);
DEST[79:72] ← SaturateSignedWordToUnsignedByte (SRC[31:16]);
DEST[87:80] ← SaturateSignedWordToUnsignedByte (SRC[47:32]);
DEST[95:88] ← SaturateSignedWordToUnsignedByte (SRC[63:48]);
DEST[103:96] ← SaturateSignedWordToUnsignedByte (SRC[79:64]);
DEST[111:104] ← SaturateSignedWordToUnsignedByte (SRC[95:80]);
DEST[119:112] ← SaturateSignedWordToUnsignedByte (SRC[111:96]);
DEST[127:120] ← SaturateSignedWordToUnsignedByte (SRC[127:112]);
```

PACKUSWB (128-bit Legacy SSE version)

```
DEST[127:0] ← UNSIGNED_SATURATING_PACK_WB(DEST, SRC)
DEST[VLMAX-1:128] (Unmodified)
```

VPACKUSWB (VEX.128 encoded version)

```
DEST[127:0] ← UNSIGNED_SATURATING_PACK_WB(SRC1, SRC2)
DEST[VLMAX-1:128] ← 0
```

Intel C/C++ Compiler Intrinsic Equivalent

PACKUSWB __m64 _mm_packs_pu16(__m64 m1, __m64 m2)

PACKUSWB __m128i _mm_packus_epi16(__m128i m1, __m128i m2)

Flags Affected

None.

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Exceptions Type 4; additionally

#UD	If VEX.L = 1.

...

PADDB/PADDW/PADDD—Add Packed Integers

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
0F FC /r[1] PADDB *mm, mm/m64*	A	V/V	MMX	Add packed byte integers from *mm/m64* and *mm*.
66 0F FC /r PADDB *xmm1, xmm2/m128*	A	V/V	SSE2	Add packed byte integers from *xmm2/m128* and *xmm1*.
0F FD /r[1] PADDW *mm, mm/m64*	A	V/V	MMX	Add packed word integers from *mm/m64* and *mm*.
66 0F FD /r PADDW *xmm1, xmm2/m128*	A	V/V	SE2	Add packed word integers from *xmm2/m128* and *xmm1*.
0F FE /r[1] PADDD *mm, mm/m64*	A	V/V	MMX	Add packed doubleword integers from *mm/m64* and *mm*.
66 0F FE /r PADDD *xmm1, xmm2/m128*	A	V/V	SSE2	Add packed doubleword integers from *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.

NOTES:

1. See note in Section 2.4, “Instruction Exception Specification”.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
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 integers from the source operand (second operand) and the destination operand (first operand), and stores the packed integer results in the destination operand. See Figure 9-4 in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*, for an illustration of a SIMD operation. Overflow is handled with wraparound, as described in the following paragraphs.

These instructions can operate on either 64-bit or 128-bit operands. When operating on 64-bit operands, the destination operand must be an MMX technology register and the source operand can be either an MMX technology register or a 64-bit memory location. When operating on 128-bit operands, the destination operand must be an XMM register and the source operand can be either an XMM register or a 128-bit memory location.

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. When a result is too large to be represented in the 8/16/32 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.

In 64-bit mode, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

128-bit Legacy SSE version: Bits (VLMAX-1:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: Bits (VLMAX-1:128) of the destination YMM register are zeroed. VEX.L must be 0, otherwise the instruction will #UD.

Operation

PADDB (with 64-bit operands)

```
DEST[7:0] ← DEST[7:0] + SRC[7:0];
(* Repeat add operation for 2nd through 7th byte *)
DEST[63:56] ← DEST[63:56] + SRC[63:56];
```

PADDB (with 128-bit operands)

```
DEST[7:0] ← DEST[7:0] + SRC[7:0];
(* Repeat add operation for 2nd through 14th byte *)
DEST[127:120] ← DEST[111:120] + SRC[127:120];
```

PADDW (with 64-bit operands)

```
DEST[15:0] ← DEST[15:0] + SRC[15:0];
(* Repeat add operation for 2nd and 3th word *)
DEST[63:48] ← DEST[63:48] + SRC[63:48];
```

PADDW (with 128-bit operands)

```
DEST[15:0] ← DEST[15:0] + SRC[15:0];
(* Repeat add operation for 2nd through 7th word *)
DEST[127:112] ← DEST[127:112] + SRC[127:112];
```

PADDD (with 64-bit operands)

```
DEST[31:0] ← DEST[31:0] + SRC[31:0];
DEST[63:32] ← DEST[63:32] + SRC[63:32];
```

PADDD (with 128-bit operands)

```
DEST[31:0] ← DEST[31:0] + SRC[31:0];
(* Repeat add operation for 2nd and 3th doubleword *)
DEST[127:96] ← DEST[127:96] + SRC[127:96];
```

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[VLMAX-1:128] ← 0
```

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[VLMAX-1:128] ← 0
```

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[VLMAX-1:128] ← 0
```

Intel C/C++ Compiler Intrinsic Equivalents

PADDB __m64 _mm_add_pi8(__m64 m1, __m64 m2)

PADDB __m128i _mm_add_epi8 (__m128ia,__m128ib)

PADDW __m64 _mm_add_pi16(__m64 m1, __m64 m2)

PADDW __m128i _mm_add_epi16 (__m128i a, __m128i b)

PADDD __m64 _mm_add_pi32(__m64 m1, __m64 m2)

PADDD __m128i _mm_add_epi32 (__m128i a, __m128i b)

Flags Affected

None.

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Exceptions Type 4; additionally

#UD	If VEX.L = 1.

...

PADDQ—Add Packed Quadword Integers

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
0F D4 /r[1] PADDQ *mm1, mm2/m64*	A	V/V	SSE2	Add quadword integer *mm2/m64* to *mm1.*
66 0F D4 /r PADDQ *xmm1, xmm2/m128*	A	V/V	SSE2	Add packed quadword integers *xmm2/m128* to *xmm1.*
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.

NOTES:

1. See note in Section 2.4, "Instruction Exception Specification".

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
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 first operand (destination operand) to the second operand (source operand) and stores the result in the destination operand. The source operand can be a quadword integer stored in an MMX technology register or a 64-bit memory location, or it can be

two packed quadword integers stored in an XMM register or an 128-bit memory location. The destination operand can be a quadword integer stored in an MMX technology register or two packed quadword integers stored in an XMM register. When packed quadword operands are used, a SIMD add is performed. When a quadword result is too large to be represented in 64 bits (overflow), the result is wrapped around and the low 64 bits are written to the destination element (that is, the carry is ignored).

Note that the PADDQ instruction can operate on either unsigned or signed (two's complement notation) integers; however, it does not set bits in the EFLAGS register to indicate overflow and/or a carry. To prevent undetected overflow conditions, software must control the ranges of the values operated on.

In 64-bit mode, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

128-bit Legacy SSE version: Bits (VLMAX-1:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: Bits (VLMAX-1:128) of the destination YMM register are zeroed. VEX.L must be 0, otherwise the instruction will #UD.

Operation

PADDQ (with 64-Bit operands)

```
DEST[63:0] ← DEST[63:0] + SRC[63:0];
```

PADDQ (with 128-Bit operands)

```
DEST[63:0] ← DEST[63:0] + SRC[63:0];
DEST[127:64] ← DEST[127:64] + SRC[127:64];
```

VPADDQ (VEX.128 encoded version)

```
DEST[63:0] ← SRC1[63:0]+SRC2[63:0]
DEST[127:64] ← SRC1[127:64]+SRC2[127:64]
DEST[VLMAX-1:128] ← 0
```

Intel C/C++ Compiler Intrinsic Equivalents

PADDQ __m64 _mm_add_si64 (__m64 a, __m64 b)

PADDQ __m128i _mm_add_epi64 (__m128i a, __m128i b)

Flags Affected

None.

Numeric 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
0F EC /r[1] PADDSB *mm, mm/m64*	A	V/V	MMX	Add packed signed byte integers from *mm/m64 and mm* and saturate the results.
66 0F EC /r PADDSB *xmm1, xmm2/m128*	A	V/V	SSE2	Add packed signed byte integers from *xmm2/m128* and *xmm1* saturate the results.
0F ED /r[1] PADDSW *mm, mm/m64*	A	V/V	MMX	Add packed signed word integers from *mm/m64 and mm* and saturate the results.
66 0F ED /r PADDSW *xmm1, xmm2/m128*	A	V/V	SSE2	Add packed signed word integers from *xmm2/m128* and *xmm1* and saturate the results.
VEX.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.

NOTES:

1. See note in Section 2.4, "Instruction Exception Specification".

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
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 source operand (second operand) and the destination operand (first operand), and stores the packed integer results in the destination operand. See Figure 9-4 in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*, for an illustration of a SIMD operation. Overflow is handled with signed saturation, as described in the following paragraphs.

These instructions can operate on either 64-bit or 128-bit operands. When operating on 64-bit operands, the destination operand must be an MMX technology register and the source operand can be either an MMX technology register or a 64-bit memory location. When operating on 128-bit operands, the destination operand must be an XMM register and the source operand can be either an XMM register or a 128-bit memory location.

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.

In 64-bit mode, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

128-bit Legacy SSE version: Bits (VLMAX-1:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: Bits (VLMAX-1:128) of the destination YMM register are zeroed. VEX.L must be 0, otherwise the instruction will #UD.

Operation

PADDSB (with 64-bit operands)

```
DEST[7:0] ← SaturateToSignedByte(DEST[7:0] + SRC (7:0]);
(* Repeat add operation for 2nd through 7th bytes *)
DEST[63:56] ← SaturateToSignedByte(DEST[63:56] + SRC[63:56] );
```

PADDSB (with 128-bit operands)

```
DEST[7:0] ←SaturateToSignedByte (DEST[7:0] + SRC[7:0]);
(* Repeat add operation for 2nd through 14th bytes *)
DEST[127:120] ← SaturateToSignedByte (DEST[111:120] + SRC[127:120]);
```

VPADDSB

```
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[VLMAX-1:128] ← 0
```

PADDSW (with 64-bit operands)

```
DEST[15:0] ← SaturateToSignedWord(DEST[15:0] + SRC[15:0] );
(* Repeat add operation for 2nd and 7th words *)
DEST[63:48] ← SaturateToSignedWord(DEST[63:48] + SRC[63:48] );
```

PADDSW (with 128-bit operands)

```
DEST[15:0]  ← SaturateToSignedWord (DEST[15:0] + SRC[15:0]);
(* Repeat add operation for 2nd through 7th words *)
DEST[127:112] ← SaturateToSignedWord (DEST[127:112] + SRC[127:112]);
```

VPADDSW

```
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[VLMAX-1:128] ← 0
```

Intel C/C++ Compiler Intrinsic Equivalents

PADDSB __m64 _mm_adds_pi8(__m64 m1, __m64 m2)

PADDSB __m128i _mm_adds_epi8 (__m128i a, __m128i b)

PADDSW __m64 _mm_adds_pi16(__m64 m1, __m64 m2)

PADDSW __m128i _mm_adds_epi16 (__m128i a, __m128i b)

Flags Affected

None.

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
0F DC /r[1] PADDUSB *mm, mm/m64*		A	V/V	MMX	Add packed unsigned byte integers from *mm/m64 and mm* and saturate the results.
66 0F DC /r PADDUSB xmm1, xmm2/m128		A	V/V	SSE2	Add packed unsigned byte integers from *xmm2/m128* and *xmm1* saturate the results.
0F DD /r[1] PADDUSW *mm, mm/m64*		A	V/V	MMX	Add packed unsigned word integers from *mm/m64 and mm* and saturate the results.
66 0F DD /r PADDUSW xmm1, xmm2/m128		A	V/V	SSE2	Add packed unsigned word integers from *xmm2/m128* to *xmm1* and saturate the results.
VEX.NDS.128.66.0F.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.

NOTES:

1. See note in Section 2.4, "Instruction Exception Specification".

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
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 source operand (second operand) and the destination operand (first operand), and stores the packed integer results in the destination operand. See Figure 9-4 in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*, for an illustration of a SIMD operation. Overflow is handled with unsigned saturation, as described in the following paragraphs.

These instructions can operate on either 64-bit or 128-bit operands. When operating on 64-bit operands, the destination operand must be an MMX technology register and the source operand can be either an MMX technology register or a 64-bit memory location.

When operating on 128-bit operands, the destination operand must be an XMM register and the source operand can be 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.

In 64-bit mode, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

128-bit Legacy SSE version: Bits (VLMAX-1:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: Bits (VLMAX-1:128) of the destination YMM register are zeroed. VEX.L must be 0, otherwise the instruction will #UD.

Operation

PADDUSB (with 64-bit operands)

```
DEST[7:0] ← SaturateToUnsignedByte(DEST[7:0] + SRC (7:0] );
(* Repeat add operation for 2nd through 7th bytes *)
DEST[63:56] ← SaturateToUnsignedByte(DEST[63:56] + SRC[63:56]
```

PADDUSB (with 128-bit operands)

```
DEST[7:0] ← SaturateToUnsignedByte (DEST[7:0] + SRC[7:0]);
(* Repeat add operation for 2nd through 14th bytes *)
DEST[127:120] ← SaturateToUnSignedByte (DEST[127:120] + SRC[127:120]);
```

VPADDUSB

```
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[VLMAX-1:128] ← 0
```

PADDUSW (with 64-bit operands)

```
DEST[15:0] ← SaturateToUnsignedWord(DEST[15:0] + SRC[15:0] );
(* Repeat add operation for 2nd and 3rd words *)
DEST[63:48] ← SaturateToUnsignedWord(DEST[63:48] + SRC[63:48] );
```

PADDUSW (with 128-bit operands)

```
DEST[15:0] ← SaturateToUnsignedWord (DEST[15:0] + SRC[15:0]);
(* Repeat add operation for 2nd through 7th words *)
DEST[127:112] ← SaturateToUnSignedWord (DEST[127:112] + SRC[127:112]);
```

VPADDUSW

```
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[VLMAX-1:128] ← 0
```

Intel C/C++ Compiler Intrinsic Equivalents

PADDUSB __m64 _mm_adds_pu8(__m64 m1, __m64 m2)

PADDUSW __m64 _mm_adds_pu16(__m64 m1, __m64 m2)

PADDUSB __m128i _mm_adds_epu8 (__m128i a, __m128i b)

PADDUSW __m128i _mm_adds_epu16 (__m128i a, __m128i b)

Flags Affected

None.

Numeric Exceptions

None.

Other Exceptions

See Exceptions Type 4; additionally

#UD	If VEX.L = 1.

...

PALIGNR — Packed Align Right

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
0F 3A 0F[1] PALIGNR mm1, mm2/m64, imm8	A	V/V	SSSE3	Concatenate destination and source operands, extract byte-aligned result shifted to the right by constant value in imm8 into mm1.
66 0F 3A 0F PALIGNR xmm1, xmm2/m128, imm8	A	V/V	SSSE3	Concatenate destination and source operands, extract byte-aligned result shifted to the right by constant value in imm8 into xmm1
VEX.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.

NOTES:

1. See note in Section 2.4, “Instruction Exception Specification”.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
A	ModRM:reg (r, w)	ModRM:r/m (r)	imm8	NA
B	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA

Description

PALIGNR concatenates the destination operand (the first operand) and the source operand (the second operand) into an intermediate composite, shifts the composite at byte granularity to the right by a constant immediate, and extracts the right-aligned result into the destination. The first and the second operands can be an MMX or an XMM register. The immediate value is considered unsigned. Immediate shift counts larger than the 2L (i.e. 32 for 128-bit operands, or 16 for 64-bit operands) produce a zero result. Both operands can be MMX register or XMM registers. When the source operand is a 128-bit memory operand, the operand must be aligned on a 16-byte boundary or a general-protection exception (#GP) will be generated.

In 64-bit mode, use the REX prefix to access additional registers.

128-bit Legacy SSE version: Bits (VLMAX-1:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: Bits (VLMAX-1:128) of the destination YMM register are zeroed. VEX.L must be 0, otherwise the instruction will #UD.

Operation

PALIGNR (with 64-bit operands)

```
temp1[127:0] = CONCATENATE(DEST,SRC)>>(imm8*8)
DEST[63:0] = temp1[63:0]
```

PALIGNR (with 128-bit operands)

```
temp1[255:0] = CONCATENATE(DEST,SRC)>>(imm8*8)
DEST[127:0] = temp1[127:0]
```

VPALIGNR

```
temp1[255:0] ← CONCATENATE(SRC1,SRC2)>>(imm8*8)
DEST[127:0] ← temp1[127:0]
DEST[VLMAX-1:128] ← 0
```

Intel C/C++ Compiler Intrinsic Equivalents

PALIGNR __m64 _mm_alignr_pi8 (__m64 a, __m64 b, int n)

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
0F DB /r[1] PAND *mm, mm/m64*	A	V/V	MMX	Bitwise AND *mm/m64* and *mm*.
66 0F DB /r PAND *xmm1, xmm2/m128*	A	V/V	SSE2	Bitwise AND of *xmm2/m128* and *xmm1*.
VEX.NDS.128.66.0F.WIG DB /r VPAND xmm1, xmm2, xmm3/m128	B	V/V	AVX	Bitwise AND of xmm3/m128 and xmm.

NOTES:

1. See note in Section 2.4, “Instruction Exception Specification”.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
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 source operand (second operand) and the destination operand (first operand) and stores the result in the destination operand. The source operand can be an MMX technology register or a 64-bit memory location or it can be an XMM register or a 128-bit memory location. The destination operand can be an MMX technology register or an XMM register. Each bit of the result is set to 1 if the corresponding bits of the first and second operands are 1; otherwise, it is set to 0.

In 64-bit mode, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

128-bit Legacy SSE version: Bits (VLMAX-1:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: Bits (VLMAX-1:128) of the destination YMM register are zeroed. VEX.L must be 0, otherwise the instruction will #UD.

Operation

PAND (128-bit Legacy SSE version)

DEST ← DEST AND SRC

DEST[VLMAX-1:1288] (Unmodified)

VPAND (VEX.128 encoded version)

DEST ← SRC1 AND SRC2

DEST[VLMAX-1:128] ← 0

Intel C/C++ Compiler Intrinsic Equivalent

PAND __m64 _mm_and_si64 (__m64 m1, __m64 m2)

PAND __m128i _mm_and_si128 (__m128i a, __m128i b)

Flags Affected

None.

Numeric 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
0F DF /r[1] PANDN *mm, mm/m64*	A	V/V	MMX	Bitwise AND NOT of *mm/m64* and *mm*.
66 0F DF /r PANDN *xmm1, xmm2/m128*	A	V/V	SSE2	Bitwise AND NOT of *xmm2/m128* and *xmm1*.
VEX.NDS.128.66.0F.WIG DF /r VPANDN xmm1, xmm2, xmm3/m128	B	V/V	AVX	Bitwise AND NOT of xmm3/m128 and xmm2.

NOTES:

1. See note in Section 2.4, "Instruction Exception Specification".

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
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 of the destination operand (first operand), then performs a bitwise logical AND of the source operand (second operand) and the inverted destination operand. The result is stored in the destination operand. The source operand can be an MMX technology register or a 64-bit memory location or it can be an XMM register or a 128-bit memory location. The destination operand can be an MMX technology register or an XMM register. Each bit of the result is set to 1 if the corresponding bit in the first operand is 0 and the corresponding bit in the second operand is 1; otherwise, it is set to 0.

In 64-bit mode, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

128-bit Legacy SSE version: Bits (VLMAX-1:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: Bits (VLMAX-1:1288) of the destination YMM register are zeroed. VEX.L must be 0, otherwise the instruction will #UD.

Operation

PANDN(128-bit Legacy SSE version)
DEST ← NOT(DEST) AND SRC
DEST[VLMAX-1:128] (Unmodified)

VPANDN (VEX.128 encoded version)
DEST ← NOT(SRC1) AND SRC2
DEST[VLMAX-1:128] ← 0

Intel C/C++ Compiler Intrinsic Equivalent

PANDN __m64 _mm_andnot_si64 (__m64 m1, __m64 m2)

PANDN _m128i _mm_andnot_si128 (__m128i a, __m128i b)

Flags Affected

None.

Numeric 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
0F E0 /r[1] PAVGB *mm1, mm2/m64*	A	V/V	SSE	Average packed unsigned byte integers from mm2/m64 and mm1 with rounding.
66 0F E0, /r PAVGB *xmm1, xmm2/m128*	A	V/V	SSE2	Average packed unsigned byte integers from *xmm2/m128* and *xmm1* with rounding.
0F E3 /r[1] PAVGW *mm1, mm2/m64*	A	V/V	SSE	Average packed unsigned word integers from mm2/m64 and mm1 with rounding.
66 0F E3 /r PAVGW *xmm1, xmm2/m128*	A	V/V	SSE2	Average packed unsigned word integers from *xmm2/m128* and *xmm1* with rounding.
VEX.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.

NOTES:

1. See note in Section 2.4, "Instruction Exception Specification".

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
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 source operand (second operand) and the destination operand (first operand), and stores the results in the destination operand. For each corresponding pair of data elements in the first and second operands, the elements are added together, a 1 is added to the temporary sum, and that result is shifted right one bit position. The source operand can be an MMX technology register or a 64-bit memory location or it can be an XMM register or a 128-bit memory location. The destination operand can be an MMX technology register or an XMM register.

The PAVGB instruction operates on packed unsigned bytes and the PAVGW instruction operates on packed unsigned words.

In 64-bit mode, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

128-bit Legacy SSE version: Bits (VLMAX-1:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: Bits (VLMAX-1:128) of the destination YMM register are zeroed. VEX.L must be 0, otherwise the instruction will #UD.

Operation

PAVGB (with 64-bit operands)

```
DEST[7:0] ← (SRC[7:0] + DEST[7:0] + 1) >> 1; (* Temp sum before shifting is 9 bits *)
(* Repeat operation performed for bytes 2 through 6 *)
DEST[63:56] ← (SRC[63:56] + DEST[63:56] + 1) >> 1;
```

PAVGW (with 64-bit operands)

```
DEST[15:0] ← (SRC[15:0] + DEST[15:0] + 1) >> 1; (* Temp sum before shifting is 17 bits *)
(* Repeat operation performed for words 2 and 3 *)
DEST[63:48] ← (SRC[63:48] + DEST[63:48] + 1) >> 1;
```

PAVGB (with 128-bit operands)

```
DEST[7:0] ← (SRC[7:0] + DEST[7:0] + 1) >> 1; (* Temp sum before shifting is 9 bits *)
(* Repeat operation performed for bytes 2 through 14 *)
DEST[127:120] ← (SRC[127:120] + DEST[127:120] + 1) >> 1;
```

PAVGW (with 128-bit operands)

```
DEST[15:0] ← (SRC[15:0] + DEST[15:0] + 1) >> 1; (* Temp sum before shifting is 17 bits *)
(* Repeat operation performed for words 2 through 6 *)
DEST[127:112] ← (SRC[127:112] + DEST[127:112] + 1) >> 1;
```

VPAVGB (VEX.128 encoded version)

```
DEST[7:0] ← (SRC1[7:0] + SRC2[7:0] + 1) >> 1;
(* Repeat operation performed for bytes 2 through 15 *)
DEST[127:120] ← (SRC1[127:120] + SRC2[127:120] + 1) >> 1
DEST[VLMAX-1:128] ← 0
```

VPAVGW (VEX.128 encoded version)

```
DEST[15:0] ← (SRC1[15:0] + SRC2[15:0] + 1) >> 1;
(* Repeat operation performed for 16-bit words 2 through 7 *)
DEST[127:112] ← (SRC1[127:112] + SRC2[127:112] + 1) >> 1
DEST[VLMAX-1:128] ← 0
```

Intel C/C++ Compiler Intrinsic Equivalent

PAVGB __m64 _mm_avg_pu8 (__m64 a, __m64 b)

PAVGW __m64 _mm_avg_pu16 (__m64 a, __m64 b)

PAVGB __m128i _mm_avg_epu8 (__m128i a, __m128i b)

PAVGW __m128i _mm_avg_epu16 (__m128i a, __m128i b)

Flags Affected

None.

Numeric 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* from mask specified in the high bit of each byte in *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

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
A	ModRM:reg (r, w)	ModRM:r/m (r)	<XMM0>	NA
B	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA

Description

Conditionally copies byte elements from the source operand (second operand) to the destination operand (first operand) depending on mask bits defined in the implicit third register argument, XMM0. The mask bits are the most significant bit in each byte element of the XMM0 register.

If a mask bit is "1", then the corresponding byte element in the source operand is copied to the destination, else the byte element in the destination operand is left unchanged.

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 (VLMAX-1:128) 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 (VLMAX-1:128) 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

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[VLMAX-1:128] (Unmodified)
```

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[VLMAX-1:128] ← 0
```

Intel C/C++ Compiler Intrinsic Equivalent

PBLENDVB __m128i _mm_blendv_epi8 (__m128i v1, __m128i v2, __m128i mask);

Flags Affected

None.

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

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
A	ModRM:reg (r, w)	ModRM:r/m (r)	imm8	NA
B	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA

Description

Conditionally copies word elements from the source operand (second operand) to the destination operand (first operand) depending on the immediate byte (third operand). Each bit of Imm8 correspond to a word element.

If a bit is "1", then the corresponding word element in the source operand is copied to the destination, else the word element in the destination operand is left unchanged.

128-bit Legacy SSE version: Bits (VLMAX-1:1288) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: Bits (VLMAX-1:128) of the destination YMM register are zeroed. VEX.L must be 0, otherwise the instruction will #UD.

Operation

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]

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[VLMAX-1:128] ← 0

Intel C/C++ Compiler Intrinsic Equivalent

PBLENDW __m128i _mm_blend_epi16 (__m128i v1, __m128i v2, const int mask);

Flags Affected

None.

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

Op/En	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 4-10, other bits of the immediate byte are ignored.

Table 4-10 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 loca-

tion. Bits (VLMAX-1:128) 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 4-11 Pseudo-Op and PCLMULQDQ Implementation

Pseudo-Op	Imm8 Encoding
PCLMULLQLQDQ *xmm1, xmm2*	**0000_0000B**
PCLMULHQLQDQ *xmm1, xmm2*	**0000_0001B**
PCLMULLQHDQ *xmm1, xmm2*	**0001_0000B**
PCLMULHQHDQ *xmm1, xmm2*	**0001_0001B**

Operation

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[VLMAX-1:128] (Unmodified)
```

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[VLMAX-1:127] ← 0;
```

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— Compare Packed Data for Equal

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
0F 74 /r[1] PCMPEQB *mm, mm/m64*	A	V/V	MMX	Compare packed bytes in *mm/m64* and *mm* for equality.
66 0F 74 /r PCMPEQB *xmm1, xmm2/m128*	A	V/V	SSE2	Compare packed bytes in *xmm2/m128* and xmm1 for equality.
0F 75 /r[1] PCMPEQW *mm, mm/m64*	A	V/V	MMX	Compare packed words in *mm/m64* and *mm* for equality.
66 0F 75 /r PCMPEQW *xmm1, xmm2/m128*	A	V/V	SSE2	Compare packed words in *xmm2/m128* and xmm1 for equality.
0F 76 /r[1] PCMPEQD *mm, mm/m64*	A	V/V	MMX	Compare packed doublewords in *mm/m64* and *mm* for equality.
66 0F 76 /r PCMPEQD *xmm1, xmm2/m128*	A	V/V	SSE2	Compare packed doublewords in *xmm2/m128* and xmm1 for equality.
VEX.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.

NOTES:

1. See note in Section 2.4, "Instruction Exception Specification".

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
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, or doublewords in the destination operand (first operand) and the source operand (second operand). If a pair of data elements is equal, the corresponding data element in the destination operand is set to all 1s; otherwise, it is set to all 0s. The source operand can be an MMX technology

register or a 64-bit memory location, or it can be an XMM register or a 128-bit memory location. The destination operand can be an MMX technology register or an XMM register.

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; and the PCMPEQD instruction compares the corresponding doublewords in the destination and source operands.

In 64-bit mode, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

128-bit Legacy SSE version: Bits (VLMAX-1:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: Bits (VLMAX-1:128) of the destination YMM register are zeroed. VEX.L must be 0, otherwise the instruction will #UD.

Operation

PCMPEQB (with 64-bit operands)

```
IF DEST[7:0] = SRC[7:0]
    THEN DEST[7:0) ← FFH;
    ELSE DEST[7:0] ← 0; FI;
(* Continue comparison of 2nd through 7th bytes in DEST and SRC *)
IF DEST[63:56] = SRC[63:56]
    THEN DEST[63:56] ← FFH;
    ELSE DEST[63:56] ← 0; FI;
```

PCMPEQB (with 128-bit operands)

```
IF DEST[7:0] = SRC[7:0]
    THEN DEST[7:0) ← FFH;
    ELSE DEST[7:0] ← 0; FI;
(* Continue comparison of 2nd through 15th bytes in DEST and SRC *)
IF DEST[127:120] = SRC[127:120]
    THEN DEST[127:120] ← FFH;
    ELSE DEST[127:120] ← 0; FI;
```

PCMPEQW (with 64-bit operands)

```
IF DEST[15:0] = SRC[15:0]
    THEN DEST[15:0] ← FFFFH;
    ELSE DEST[15:0] ← 0; FI;
(* Continue comparison of 2nd and 3rd words in DEST and SRC *)
IF DEST[63:48] = SRC[63:48]
    THEN DEST[63:48] ← FFFFH;
    ELSE DEST[63:48] ← 0; FI;
```

PCMPEQW (with 128-bit operands)

```
IF DEST[15:0] = SRC[15:0]
    THEN DEST[15:0] ← FFFFH;
    ELSE DEST[15:0] ← 0; FI;
(* Continue comparison of 2nd through 7th words in DEST and SRC *)
IF DEST[127:112] = SRC[127:112]
    THEN DEST[127:112] ← FFFFH;
    ELSE DEST[127:112] ← 0; FI;
```

PCMPEQD (with 64-bit operands)

```
IF DEST[31:0] = SRC[31:0]
    THEN DEST[31:0] ← FFFFFFFFH;
    ELSE DEST[31:0] ← 0; FI;
IF DEST[63:32] = SRC[63:32]
    THEN DEST[63:32] ← FFFFFFFFH;
    ELSE DEST[63:32] ← 0; FI;
```

PCMPEQD (with 128-bit operands)

```
IF DEST[31:0] = SRC[31:0]
    THEN DEST[31:0] ← FFFFFFFFH;
    ELSE DEST[31:0] ← 0; FI;
(* Continue comparison of 2nd and 3rd doublewords in DEST and SRC *)
IF DEST[127:96] = SRC[127:96]
    THEN DEST[127:96] ← FFFFFFFFH;
    ELSE DEST[127:96] ← 0; FI;
```

VPCMPEQB (VEX.128 encoded version)

```
DEST[127:0] ←COMPARE_BYTES_EQUAL(SRC1,SRC2)
DEST[VLMAX-1:128] ← 0
```

VPCMPEQW (VEX.128 encoded version)

```
DEST[127:0] ←COMPARE_WORDS_EQUAL(SRC1,SRC2)
DEST[VLMAX-1:128] ← 0
```

VPCMPEQD (VEX.128 encoded version)

```
DEST[127:0] ←COMPARE_DWORDS_EQUAL(SRC1,SRC2)
DEST[VLMAX-1:128] ← 0
```

Intel C/C++ Compiler Intrinsic Equivalents

PCMPEQB __m64 _mm_cmpeq_pi8 (__m64 m1, __m64 m2)

PCMPEQW __m64 _mm_cmpeq_pi16 (__m64 m1, __m64 m2)

PCMPEQD __m64 _mm_cmpeq_pi32 (__m64 m1, __m64 m2)

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)

Flags Affected

None.

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Exceptions Type 4; additionally

#UD	If VEX.L = 1.

...

PCMPEQQ — Compare Packed Qword Data for Equal

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 38 29 /r PCMPEQQ *xmm1, xmm2/m128*	A	V/V	SSE4_1	Compare packed qwords in *xmm2/m128* and *xmm1* for equality.
VEX.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

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
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 for equality of the packed quadwords in the destination operand (first operand) and the source operand (second operand). If a pair of data elements is equal, the corresponding data element in the destination is set to all 1s; otherwise, it is set to 0s.

128-bit Legacy SSE version: Bits (VLMAX-1:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: Bits (VLMAX-1:128) of the destination YMM register are zeroed. VEX.L must be 0, otherwise the instruction will #UD.

Operation

```
IF (DEST[63:0] = SRC[63:0])
    THEN DEST[63:0] ← FFFFFFFFFFFFFFFFH;
    ELSE DEST[63:0] ← 0; FI;
IF (DEST[127:64] = SRC[127:64])
    THEN DEST[127:64] ← FFFFFFFFFFFFFFFFH;
    ELSE DEST[127:64] ← 0; FI;
```

VPCMPEQQ (VEX.128 encoded version)

```
DEST[127:0] ←COMPARE_QWORDS_EQUAL(SRC1,SRC2)
DEST[VLMAX-1:128] ← 0
```

Intel C/C++ Compiler Intrinsic Equivalent

PCMPEQQ __m128i _mm_cmpeq_epi64(__m128i a, __m128i b);

Flags Affected

None.

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 imm8* 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

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
A	ModRM:reg (r)	ModRM:r/m (r)	imm8	NA

Description

The instruction compares and processes data from two string fragments based on the encoded value in the Imm8 Control Byte (see Section 4.1, "Imm8 Control Byte Operation for PCMPESTRI / PCMPESTRM / PCMPISTRI / PCMPISTRM"), and generates an index stored to the count register (ECX/RCX).

Each string fragment is represented by two values. The first value is an xmm (or possibly m128 for the second operand) which contains the data elements of the string (byte or word data). The second value is stored in an input length register. The input length register is EAX/RAX (for xmm1) or EDX/RDX (for xmm2/m128). The length 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 the length register. The absolute-value computation saturates to 16 (for bytes) and 8 (for words), based on the value of imm8[bit3] when the value in the length register is greater than 16 (8) or less than -16 (-8).

The comparison and aggregation operations are performed according to the encoded value of Imm8 bit fields (see Section 4.1). The index of the first (or last, according to imm8[6]) set bit of IntRes2 (see Section 4.1.4) 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 in order 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

Effective Operand Size

Operating mode/size	Operand 1	Operand 2	Length 1	Length 2	Result
16 bit	xmm	xmm/m128	EAX	EDX	ECX
32 bit	xmm	xmm/m128	EAX	EDX	ECX
64 bit	xmm	xmm/m128	EAX	EDX	ECX
64 bit + REX.W	xmm	xmm/m128	RAX	RDX	RCX

Intel C/C++ Compiler Intrinsic Equivalent For Returning Index

int _mm_cmpestri (__m128i a, int la, __m128i b, int lb, const int mode);

Intel C/C++ Compiler Intrinsics For Reading EFlag Results

int _mm_cmpestra (__m128i a, int la, __m128i b, int lb, const int mode);
int _mm_cmpestrc (__m128i a, int la, __m128i b, int lb, const int mode);
int _mm_cmpestro (__m128i a, int la, __m128i b, int lb, const int mode);
int _mm_cmpestrs (__m128i a, int la, __m128i b, int lb, const int mode);
int _mm_cmpestrz (__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 imm8 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

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
A	ModRM:reg (r)	ModRM:r/m (r)	imm8	NA

Description

The instruction compares data from two string fragments based on the encoded value in the imm8 control byte (see Section 4.1, "Imm8 Control Byte Operation for PCMPESTRI / PCMPESTRM / PCMPISTRI / PCMPISTRM"), and generates a mask stored to XMM0.

Each string fragment is represented by two values. The first value is an xmm (or possibly m128 for the second operand) which contains the data elements of the string (byte or word data). The second value is stored in an input length register. The input length register is EAX/RAX (for xmm1) or EDX/RDX (for xmm2/m128). The length 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 the length register. The absolute-value computation saturates to 16 (for bytes) and 8 (for words), based on the value of imm8[bit3] when the value in the length register is greater than 16 (8) or less than -16 (-8).

The comparison and aggregation operations are performed according to the encoded value of Imm8 bit fields (see Section 4.1). 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 in order 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.

Effective Operand Size

Operating mode/size	Operand1	Operand 2	Length1	Length2	Result
16 bit	xmm	xmm/m128	EAX	EDX	XMM0
32 bit	xmm	xmm/m128	EAX	EDX	XMM0
64 bit	xmm	xmm/m128	EAX	EDX	XMM0
64 bit + REX.W	xmm	xmm/m128	RAX	RDX	XMM0

Intel C/C++ Compiler Intrinsic Equivalent For Returning Mask

__m128i _mm_cmpestrm (__m128i a, int la, __m128i b, int lb, const int mode);

Intel C/C++ Compiler Intrinsics For Reading EFlag Results

int _mm_cmpestra (__m128i a, int la, __m128i b, int lb, const int mode);
int _mm_cmpestrc (__m128i a, int la, __m128i b, int lb, const int mode);
int _mm_cmpestro (__m128i a, int la, __m128i b, int lb, const int mode);
int _mm_cmpestrs (__m128i a, int la, __m128i b, int lb, const int mode);
int _mm_cmpestrz (__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—Compare Packed Signed Integers for Greater Than

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
0F 64 /r[1] PCMPGTB *mm, mm/m64*	A	V/V	MMX	Compare packed signed byte integers in *mm* and *mm/m64* for greater than.
66 0F 64 /r PCMPGTB *xmm1, xmm2/m128*	A	V/V	SSE2	Compare packed signed byte integers in *xmm1* and *xmm2/m128* for greater than.
0F 65 /r[1] PCMPGTW *mm, mm/m64*	A	V/V	MMX	Compare packed signed word integers in *mm* and *mm/m64* for greater than.
66 0F 65 /r PCMPGTW *xmm1, xmm2/m128*	A	V/V	SSE2	Compare packed signed word integers in *xmm1* and *xmm2/m128* for greater than.
0F 66 /r[1] PCMPGTD *mm, mm/m64*	A	V/V	MMX	Compare packed signed doubleword integers in *mm* and *mm/m64* for greater than.
66 0F 66 /r PCMPGTD *xmm1, xmm2/m128*	A	V/V	SSE2	Compare packed signed doubleword integers in *xmm1* and *xmm2/m128* for greater than.
VEX.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.

NOTES:

1. See note in Section 2.4, "Instruction Exception Specification".

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
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, or doubleword integers in the destination operand (first operand) and the source operand (second operand). If a data element in the destination operand is greater than the corresponding date element in the source operand, the corresponding data element in the destination operand is set to all 1s; otherwise, it is set to all 0s. The source operand can be an MMX technology register or a 64-bit memory location, or it can be an XMM register or a 128-bit memory location. The destination operand can be an MMX technology register or an XMM register.

The PCMPGTB instruction compares the corresponding signed byte integers in the destination and source operands; the PCMPGTW instruction compares the corresponding signed word integers in the destination and source operands; and the PCMPGTD instruction compares the corresponding signed doubleword integers in the destination and source operands.

In 64-bit mode, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

128-bit Legacy SSE version: Bits (VLMAX-1:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: Bits (VLMAX-1:128) of the destination YMM register are zeroed. VEX.L must be 0, otherwise the instruction will #UD.

Operation

PCMPGTB (with 64-bit operands)

```
IF DEST[7:0] > SRC[7:0]
    THEN DEST[7:0) ← FFH;
    ELSE DEST[7:0] ← 0; FI;
(* Continue comparison of 2nd through 7th bytes in DEST and SRC *)
IF DEST[63:56] > SRC[63:56]
    THEN DEST[63:56] ← FFH;
    ELSE DEST[63:56] ← 0; FI;
```

PCMPGTB (with 128-bit operands)

```
IF DEST[7:0] > SRC[7:0]
    THEN DEST[7:0) ← FFH;
    ELSE DEST[7:0] ← 0; FI;
(* Continue comparison of 2nd through 15th bytes in DEST and SRC *)
IF DEST[127:120] > SRC[127:120]
    THEN DEST[127:120] ← FFH;
    ELSE DEST[127:120] ← 0; FI;
```

PCMPGTW (with 64-bit operands)

```
IF DEST[15:0] > SRC[15:0]
    THEN DEST[15:0] ← FFFFH;
```

```
        ELSE DEST[15:0] ← 0; FI;
    (* Continue comparison of 2nd and 3rd words in DEST and SRC *)
    IF DEST[63:48] > SRC[63:48]
        THEN DEST[63:48] ← FFFFH;
        ELSE DEST[63:48] ← 0; FI;
```

PCMPGTW (with 128-bit operands)

```
    IF DEST[15:0] > SRC[15:0]
        THEN DEST[15:0] ← FFFFH;
        ELSE DEST[15:0] ← 0; FI;
    (* Continue comparison of 2nd through 7th words in DEST and SRC *)
    IF DEST[63:48] > SRC[127:112]
        THEN DEST[127:112] ← FFFFH;
        ELSE DEST[127:112] ← 0; FI;
```

PCMPGTD (with 64-bit operands)

```
    IF DEST[31:0] > SRC[31:0]
        THEN DEST[31:0] ← FFFFFFFFH;
        ELSE DEST[31:0] ← 0; FI;
    IF DEST[63:32] > SRC[63:32]
        THEN DEST[63:32] ← FFFFFFFFH;
        ELSE DEST[63:32] ← 0; FI;
```

PCMPGTD (with 128-bit operands)

```
    IF DEST[31:0] > SRC[31:0]
        THEN DEST[31:0] ← FFFFFFFFH;
        ELSE DEST[31:0] ← 0; FI;
    (* Continue comparison of 2nd and 3rd doublewords in DEST and SRC *)
    IF DEST[127:96] > SRC[127:96]
        THEN DEST[127:96] ← FFFFFFFFH;
        ELSE DEST[127:96] ← 0; FI;
```

VPCMPGTB (VEX.128 encoded version)

```
DEST[127:0] ←COMPARE_BYTES_GREATER(SRC1,SRC2)
DEST[VLMAX-1:128] ← 0
```

VPCMPGTW (VEX.128 encoded version)

```
DEST[127:0] ←COMPARE_WORDS_GREATER(SRC1,SRC2)
DEST[VLMAX-1:128] ← 0
```

VPCMPGTD (VEX.128 encoded version)

```
DEST[127:0] ←COMPARE_DWORDS_GREATER(SRC1,SRC2)
DEST[VLMAX-1:128] ← 0
```

Intel C/C++ Compiler Intrinsic Equivalents

PCMPGTB __m64 _mm_cmpgt_pi8 (__m64 m1, __m64 m2)

PCMPGTW __m64 _mm_pcmpgt_pi16 (__m64 m1, __m64 m2)

DCMPGTD __m64 _mm_pcmpgt_pi32 (__m64 m1, __m64 m2)

PCMPGTB __m128i _mm_cmpgt_epi8 (__m128i a, __m128i b)

PCMPGTW __m128i _mm_cmpgt_epi16 (__m128i a, __m128i b)

DCMPGTD __m128i _mm_cmpgt_epi32 (__m128i a, __m128i b)

Flags Affected

None.

Numeric Exceptions

None.

Other Exceptions

See Exceptions Type 4; additionally

#UD	If VEX.L = 1.

...

PCMPGTQ — Compare Packed Data for Greater Than

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 38 37 */r* PCMPGTQ *xmm1,xmm2/m128*	A	V/V	SSE4_2	Compare packed qwords in *xmm2/m128* and *xmm1* 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

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
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 for the packed quadwords in the destination operand (first operand) and the source operand (second operand). If the data element in the first (destination) operand is greater than the corresponding element in the second (source) operand, the corresponding data element in the destination is set to all 1s; otherwise, it is set to 0s.

128-bit Legacy SSE version: Bits (VLMAX-1:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: Bits (VLMAX-1:128) of the destination YMM register are zeroed. VEX.L must be 0, otherwise the instruction will #UD.

Operation

```
IF (DEST[63-0] > SRC[63-0])
    THEN DEST[63-0] ← FFFFFFFFFFFFFFFFH;
```

ELSE DEST[63-0] ← 0; FI
IF (DEST[127-64] > SRC[127-64])
THEN DEST[127-64] ← FFFFFFFFFFFFFFFFH;
ELSE DEST[127-64] ← 0; FI

VPCMPGTQ (VEX.128 encoded version)
DEST[127:0] ←COMPARE_QWORDS_GREATER(SRC1,SRC2)
DEST[VLMAX-1:128] ← 0

Intel C/C++ Compiler Intrinsic Equivalent

PCMPGTQ __m128i _mm_cmpgt_epi64(__m128i a, __m128i b)

Flags Affected

None.

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 imm8* 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

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
A	ModRM:reg (r)	ModRM:r/m (r)	imm8	NA

Description

The instruction compares data from two strings based on the encoded value in the Imm8 Control Byte (see Section 4.1, "Imm8 Control Byte Operation for PCMPESTRI / PCMPESTRM / PCMPISTRI / PCMPISTRM"), and generates 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 data elements of the string (byte or word 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.)

The comparison and aggregation operations are performed according to the encoded value of Imm8 bit fields (see Section 4.1). 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 in order 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.

Effective Operand Size

Operating mode/size	Operand1	Operand 2	Result
16 bit	xmm	xmm/m128	ECX
32 bit	xmm	xmm/m128	ECX
64 bit	xmm	xmm/m128	ECX
64 bit + REX.W	xmm	xmm/m128	RCX

Intel C/C++ Compiler Intrinsic Equivalent For Returning Index

int _mm_cmpistri (__m128i a, __m128i b, const int mode);

Intel C/C++ Compiler Intrinsics For Reading EFlag Results

int _mm_cmpistra (__m128i a, __m128i b, const int mode);
int _mm_cmpistrc (__m128i a, __m128i b, const int mode);
int _mm_cmpistro (__m128i a, __m128i b, const int mode);
int _mm_cmpistrs (__m128i a, __m128i b, const int mode);
int _mm_cmpistrz (__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 imm8* 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

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
A	ModRM:reg (r)	ModRM:r/m (r)	imm8	NA

Description

The instruction compares data from two strings based on the encoded value in the imm8 byte (see Section 4.1, "Imm8 Control Byte Operation for PCMPESTRI / PCMPESTRM / PCMPISTRI / PCMPISTRM") 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 data elements of the string (byte or word 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.)

The comparison and aggregation operation are performed according to the encoded value of Imm8 bit fields (see Section 4.1). 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 in order 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 versions, VEX.vvvv is reserved and must be 1111b, VEX.L must be 1, otherwise the instruction will #UD.

Effective Operand Size

Operating mode/size	Operand1	Operand 2	Result
16 bit	xmm	xmm/m128	XMM0
32 bit	xmm	xmm/m128	XMM0
64 bit	xmm	xmm/m128	XMM0
64 bit + REX.W	xmm	xmm/m128	XMM0

Intel C/C++ Compiler Intrinsic Equivalent For Returning Mask

__m128i _mm_cmpistrm (__m128i a, __m128i b, const int mode);

Intel C/C++ Compiler Intrinsics For Reading EFlag Results

int _mm_cmpistra (__m128i a, __m128i b, const int mode);
int _mm_cmpistrc (__m128i a, __m128i b, const int mode);
int _mm_cmpistro (__m128i a, __m128i b, const int mode);
int _mm_cmpistrs (__m128i a, __m128i b, const int mode);
int _mm_cmpistrz (__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

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
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.

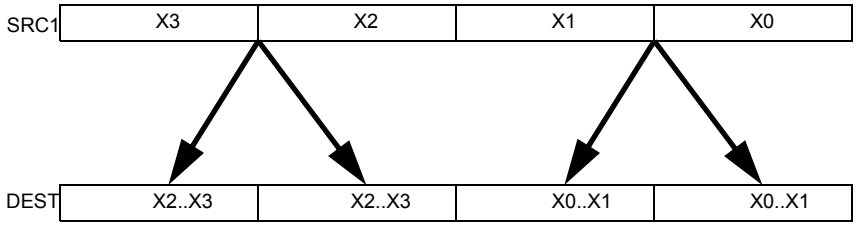


Figure 4-3 VPERMILPD operation

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 4-4) 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.

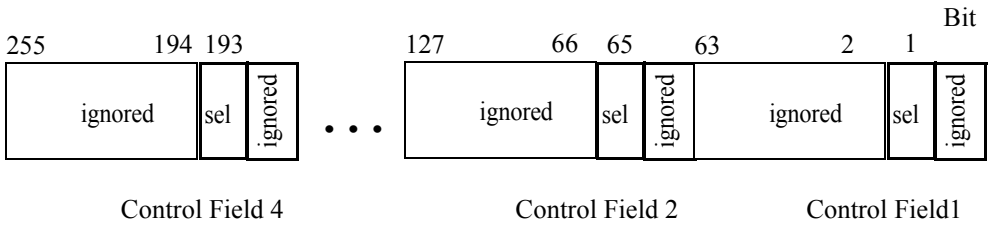


Figure 4-4 VPERMILPD Shuffle Control

(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[VLMAX-1:128] ← 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[VLMAX-1:128] ← 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

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
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.

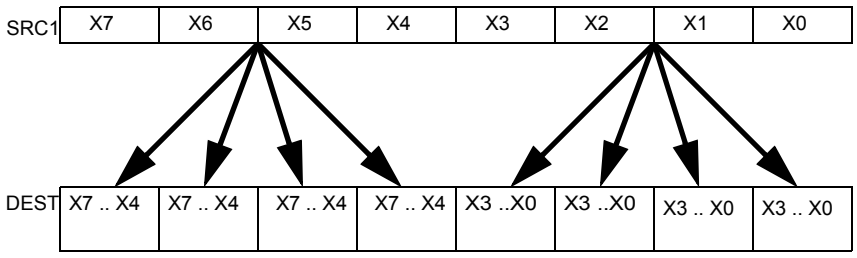


Figure 4-5 VPERMILPS Operation

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 4-6) 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.

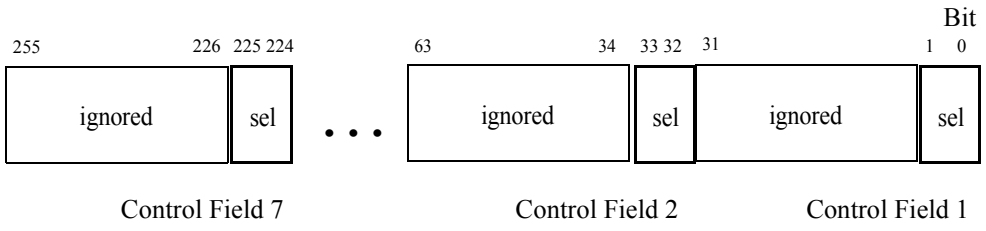


Figure 4-6 VPERMILPS Shuffle Control

(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[VLMAX-1:128] ← 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[VLMAX-1:128] ← 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

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
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.

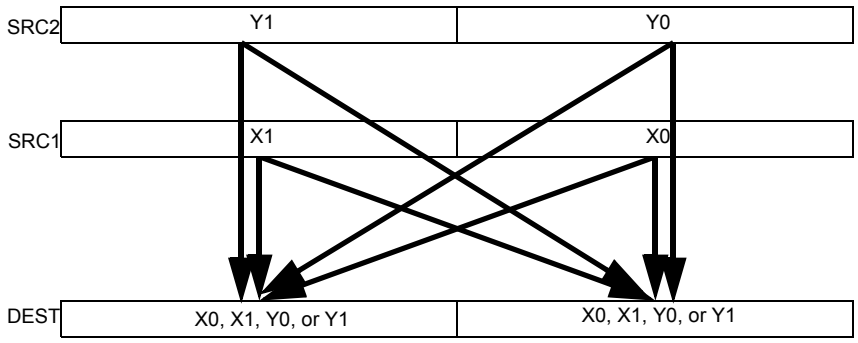


Figure 4-7 VPERM2F128 Operation

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[VLMAX-1:128] ← 0
FI
```

Intel C/C++ Compiler Intrinsic Equivalent

VPERM2F128 __m256 _mm256_permute2f128_ps (__m256 a, __m256 b, int control)

VPERM2F128 __m256d _mm256_permute2f128_pd (__m256d a, __m256d b, int control)

VPERM2F128 __m256i _mm256_permute2f128_si256 (__m256i a, __m256i b, int control)

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Exceptions Type 6; additionally

#UD	If VEX.L = 0
	If VEX.W = 1.

...

PEXTRB/PEXTRD/PEXTRQ — Extract Byte/Dword/Qword

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 3A 14 /r ib PEXTRB *reg/m8, xmm2, imm8*	A	V/V	SSE4_1	Extract a byte integer value from *xmm2* at the source byte offset specified by *imm8* into *rreg or m8. The upper bits of r32 or r64 are zeroed.*
66 0F 3A 16 /r ib PEXTRD *r/m32, xmm2, imm8*	A	V/V	SSE4_1	Extract a dword integer value from *xmm2* at the source dword offset specified by *imm8* into *r/m32*.
66 REX.W 0F 3A 16 /r ib PEXTRQ *r/m64, xmm2, imm8*	A	V/N.E.	SSE4_1	Extract a qword integer value from *xmm2* at the source qword offset specified by *imm8* into r/m64.
VEX.128.66.0F3A.W0 14 /r ib VPEXTRB reg/m8, xmm2, imm8	A	V[1]/V	AVX	Extract a byte integer value from xmm2 at the source byte offset specified by imm8 into reg or m8. The upper bits of r64/r32 is filled with zeros.
VEX.128.66.0F3A.W0 16 /r ib VPEXTRD r32/m32, xmm2, imm8	A	V/V	AVX	Extract a dword integer value from xmm2 at the source dword offset specified by imm8 into r32/m32.
VEX.128.66.0F3A.W1 16 /r ib VPEXTRQ r64/m64, xmm2, imm8	A	V/i	AVX	Extract a qword integer value from xmm2 at the source dword offset specified by imm8 into r64/m64.

NOTES:

1. In 64-bit mode, VEX.W1 is ignored for VPEXTRB (similar to legacy REX.W=1 prefix in PEXTRB).

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
A	ModRM:r/m (w)	ModRM:reg (r)	imm8	NA

Description

Extract a byte/dword/qword integer value from the source XMM register at a byte/ dword/qword offset determined from imm8[3:0]. The destination can be a register or byte/dword/qword memory location. If the destination is a register, the upper bits of the register are zero extended.

In legacy non-VEX encoded version and if the destination operand is a register, the default operand size in 64-bit mode for PEXTRB/PEXTRD is 64 bits, the bits above the least significant byte/dword data are filled with zeros. PEXTRQ is not encodable in non-64-bit modes and requires REX.W in 64-bit mode.

Note: In VEX.128 encoded versions, VEX.vvvv is reserved and must be 1111b, VEX.L must be 0, otherwise the instruction will #UD. If the destination operand is a register, the default operand size in 64-bit mode for VPEXTRB/VPEXTRD is 64 bits, the bits above the least significant byte/word/dword data are filled with zeros. Attempt to execute VPEXTRQ in non-64-bit mode will cause #UD.

Operation

```
CASE of
    PEXTRB: SEL ← COUNT[3:0];
            TEMP ← (Src >> SEL*8) AND FFH;
            IF (DEST = Mem8)
                THEN
                Mem8 ← TEMP[7:0];
            ELSE IF (64-Bit Mode and 64-bit register selected)
                THEN
                    R64[7:0] ← TEMP[7:0];
                    r64[63:8] ← ZERO_FILL; };
            ELSE
                    R32[7:0] ← TEMP[7:0];
                    r32[31:8] ← ZERO_FILL; };
            FI;
    PEXTRD:SEL ← COUNT[1:0];
            TEMP ← (Src >> SEL*32) AND FFFF_FFFFH;
            DEST ← TEMP;
    PEXTRQ: SEL ← COUNT[0];
            TEMP ← (Src >> SEL*64);
            DEST ← TEMP;
EASC:

(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)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);

PEXTRD int _mm_extract_epi32 (__m128i src, const int ndx);
PEXTRQ __int64 _mm_extract_epi64 (__m128i src, const int ndx);

Flags Affected

None.

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Exceptions Type 5; additionally

#UD	If VEX.L = 1.
	If VEX.vvvv != 1111B.
	If VPEXTRQ in non-64-bit mode, VEX.W=1.

...

PEXTRW—Extract Word

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
0F C5 /*r* ib[1] PEXTRW *reg, mm, imm8*	A	V/V	SSE	Extract the word specified by *imm8* from *mm* and move it to *reg*, bits 15-0. The upper bits of r32 or r64 is zeroed.
66 0F C5 /*r* ib PEXTRW *reg, xmm, imm8*	A	V/V	SSE4_1	Extract the word specified by *imm8* from *xmm* and move it to *reg*, bits 15-0. The upper bits of r32 or r64 is zeroed.
66 0F 3A 15 /r ib PEXTRW *reg/m16, xmm, imm8*	B	V/V	SSE4_1	Extract the word specified by *imm8* from *xmm* and copy it to lowest 16 bits of *reg or m16*. Zero-extend the result in the destination, r32 or r64.
VEX.128.66.0F.W0 C5 /r ib VPEXTRW reg, xmm1, imm8	A	V[2]/V	AVX	Extract the word specified by imm8 from xmm1 and move it to reg, bits 15:0. Zero-extend the result. The upper bits of r64/r32 is filled with zeros.
VEX.128.66.0F3A.W0 15 /r ib VPEXTRW reg/m16, xmm2, imm8	B	V/V	AVX	Extract a word integer value from xmm2 at the source word offset specified by imm8 into reg or m16. The upper bits of r64/r32 is filled with zeros.

NOTES:

1. See note in Section 2.4, "Instruction Exception Specification".
2. In 64-bit mode, VEX.W1 is ignored for VPEXTRW (similar to legacy REX.W=1 prefix in PEXTRW).

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
A	ModRM:reg (w)	ModRM:r/m (r)	imm8	NA
B	ModRM:r/m (w)	ModRM:reg (r)	imm8	NA

Description

Copies the word in the source operand (second operand) specified by the count operand (third operand) to the destination operand (first operand). The source operand can be an MMX technology register or an XMM register. The destination operand can be the low word of a general-purpose register or a 16-bit memory address. The count operand is an 8-bit immediate. When specifying a word location in an MMX technology register, the 2 least-significant bits of the count operand specify the location; for an XMM

register, the 3 least-significant bits specify the location. The content of the destination register above bit 16 is cleared (set to all 0s).

In 64-bit mode, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15, R8-15). If the destination operand is a general-purpose register, the default operand size is 64-bits in 64-bit mode.

Note: In VEX.128 encoded versions, VEX.vvvv is reserved and must be 1111b, VEX.L must be 0, otherwise the instruction will #UD. If the destination operand is a register, the default operand size in 64-bit mode for VPEXTRW is 64 bits, the bits above the least significant byte/word/dword data are filled with zeros.

Operation

```
IF (DEST = Mem16)
THEN
    SEL ← COUNT[2:0];
    TEMP ← (Src >> SEL*16) AND FFFFH;
    Mem16 ← TEMP[15:0];
ELSE IF (64-Bit Mode and destination is a general-purpose register)
    THEN
        FOR (PEXTRW instruction with 64-bit source operand)
          {  SEL ← COUNT[1:0];
             TEMP ← (SRC >> (SEL * 16)) AND FFFFH;
             r64[15:0] ← TEMP[15:0];
             r64[63:16] ← ZERO_FILL; };
        FOR (PEXTRW instruction with 128-bit source operand)
          {  SEL ← COUNT[2:0];
             TEMP ← (SRC >> (SEL * 16)) AND FFFFH;
             r64[15:0] ← TEMP[15:0];
             r64[63:16] ← ZERO_FILL; }
    ELSE
        FOR (PEXTRW instruction with 64-bit source operand)
          {  SEL ← COUNT[1:0];
             TEMP ← (SRC >> (SEL * 16)) AND FFFFH;
             r32[15:0] ← TEMP[15:0];
             r32[31:16] ← ZERO_FILL; };
        FOR (PEXTRW instruction with 128-bit source operand)
          {  SEL ← COUNT[2:0];
             TEMP ← (SRC >> (SEL * 16)) AND FFFFH;
             r32[15:0] ← TEMP[15:0];
             r32[31:16] ← ZERO_FILL; };
    FI;
FI;
```

(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
```

Intel C/C++ Compiler Intrinsic Equivalent

PEXTRW int _mm_extract_pi16 (__m64 a, int n)

PEXTRW int _mm_extract_epi16 (__m128i a, int imm)

Flags Affected

None.

Numeric Exceptions

None.

Other Exceptions

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
0F 38 01 /r[1] PHADDW mm1, mm2/m64	A	V/V	SSSE3	Add 16-bit signed integers horizontally, pack to MM1.
66 0F 38 01 /r PHADDW xmm1, xmm2/m128	A	V/V	SSSE3	Add 16-bit signed integers horizontally, pack to XMM1.
0F 38 02 /r PHADDD mm1, mm2/m64	A	V/V	SSSE3	Add 32-bit signed integers horizontally, pack to MM1.
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.

NOTES:

1. See note in Section 2.4, "Instruction Exception Specification".

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
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 source and destination operands and packs the 16-bit signed results to the destination operand (first operand). PHADDD adds two adjacent 32-bit signed integers horizontally from the source and destination operands and packs the 32-bit signed results to the destination operand (first operand). Both operands can be MMX or XMM registers. When the source operand is a 128-bit memory operand, the operand must be aligned on a 16-byte boundary or a general-protection exception (#GP) will be generated.

In 64-bit mode, use the REX prefix to access additional registers.

128-bit Legacy SSE version: Bits (VLMAX-1:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: Bits (VLMAX-1:128) of the destination YMM register are zeroed. VEX.L must be 0, otherwise the instruction will #UD.

Operation

PHADDW (with 64-bit operands)

```
mm1[15-0] = mm1[31-16] + mm1[15-0];
```

mm1[31-16] = mm1[63-48] + mm1[47-32];
mm1[47-32] = mm2/m64[31-16] + mm2/m64[15-0];
mm1[63-48] = mm2/m64[63-48] + mm2/m64[47-32];

PHADDW (with 128-bit operands)

xmm1[15-0] = xmm1[31-16] + xmm1[15-0];
xmm1[31-16] = xmm1[63-48] + xmm1[47-32];
xmm1[47-32] = xmm1[95-80] + xmm1[79-64];
xmm1[63-48] = xmm1[127-112] + xmm1[111-96];
xmm1[79-64] = xmm2/m128[31-16] + xmm2/m128[15-0];
xmm1[95-80] = xmm2/m128[63-48] + xmm2/m128[47-32];
xmm1[111-96] = xmm2/m128[95-80] + xmm2/m128[79-64];
xmm1[127-112] = xmm2/m128[127-112] + xmm2/m128[111-96];

PHADDD (with 64-bit operands)

mm1[31-0] = mm1[63-32] + mm1[31-0];
mm1[63-32] = mm2/m64[63-32] + mm2/m64[31-0];

PHADDD (with 128-bit operands)

xmm1[31-0] = xmm1[63-32] + xmm1[31-0];
xmm1[63-32] = xmm1[127-96] + xmm1[95-64];
xmm1[95-64] = xmm2/m128[63-32] + xmm2/m128[31-0];
xmm1[127-96] = xmm2/m128[127-96] + xmm2/m128[95-64];

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[VLMAX-1:128] ← 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[VLMAX-1:128] ← 0

Intel C/C++ Compiler Intrinsic Equivalents

PHADDW __m64 _mm_hadd_pi16 (__m64 a, __m64 b)

PHADDW __m128i _mm_hadd_epi16 (__m128i a, __m128i b)

PHADDD __m64 _mm_hadd_pi32 (__m64 a, __m64 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 and Saturate

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
0F 38 03 /r[1] PHADDSW mm1, mm2/m64	A	V/V	SSSE3	Add 16-bit signed integers horizontally, pack saturated integers to MM1.
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.

NOTES:

1. See note in Section 2.4, “Instruction Exception Specification”.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
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 source and destination operands and saturates the signed results; packs the signed, saturated 16-bit results to the destination operand (first operand) Both operands can be MMX or XMM registers. When the source operand is a 128-bit memory operand, the operand must be aligned on a 16-byte boundary or a general-protection exception (#GP) will be generated.

In 64-bit mode, use the REX prefix to access additional registers.

128-bit Legacy SSE version: Bits (VLMAX-1:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: Bits (VLMAX-1:128) of the destination YMM register are zeroed. VEX.L must be 0, otherwise the instruction will #UD.

Operation

PHADDSW (with 64-bit operands)

```
mm1[15-0]  = SaturateToSignedWord((mm1[31-16] + mm1[15-0]);
mm1[31-16] = SaturateToSignedWord(mm1[63-48] + mm1[47-32]);
mm1[47-32] = SaturateToSignedWord(mm2/m64[31-16] + mm2/m64[15-0]);
mm1[63-48] = SaturateToSignedWord(mm2/m64[63-48] + mm2/m64[47-32]);
```

PHADDSW (with 128-bit operands)

```
xmm1[15-0]= SaturateToSignedWord(xmm1[31-16] + xmm1[15-0]);
xmm1[31-16] = SaturateToSignedWord(xmm1[63-48] + xmm1[47-32]);
xmm1[47-32] = SaturateToSignedWord(xmm1[95-80] + xmm1[79-64]);
xmm1[63-48] = SaturateToSignedWord(xmm1[127-112] + xmm1[111-96]);
xmm1[79-64] = SaturateToSignedWord(xmm2/m128[31-16] + xmm2/m128[15-0]);
xmm1[95-80] = SaturateToSignedWord(xmm2/m128[63-48] + xmm2/m128[47-32]);
xmm1[111-96] = SaturateToSignedWord(xmm2/m128[95-80] + xmm2/m128[79-64]);
xmm1[127-112] = SaturateToSignedWord(xmm2/m128[127-112] + xmm2/m128[111-96]);
```

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[VLMAX-1:128] ← 0
```

Intel C/C++ Compiler Intrinsic Equivalent

PHADDSW __m64 _mm_hadds_pi16 (__m64 a, __m64 b)

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 — Packed Horizontal Word Minimum

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

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

Description

Determine the minimum unsigned word value in the source operand (second operand) and place the unsigned word in the low word (bits 0-15) of the destination operand (first 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 (VLMAX-1:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: Bits (VLMAX-1:128) of the destination YMM register are zeroed. VEX.vvvv is reserved and must be 1111b, VEX.L must be 0, otherwise the instruction will #UD.

Operation

PHMINPOSUW (128-bit Legacy SSE version)

```
INDEX ← 0;
MIN ← SRC[15:0]
IF (SRC[31:16] < MIN)
    THEN INDEX ← 1; MIN ← SRC[31:16]; FI;
IF (SRC[47:32] < MIN)
    THEN INDEX ← 2; MIN ← SRC[47:32]; FI;
* Repeat operation for words 3 through 6
IF (SRC[127:112] < MIN)
    THEN INDEX ← 7; MIN ← SRC[127:112]; FI;
DEST[15:0] ← MIN;
DEST[18:16] ← INDEX;
DEST[127:19] ← 00000000000000000000000000000H;
```

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[VLMAX-1:128] ← 0

Intel C/C++ Compiler Intrinsic Equivalent

PHMINPOSUW __m128i _mm_minpos_epu16(__m128i packed_words);

Flags Affected

None.

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
0F 38 05 /r[1] PHSUBW mm1, mm2/m64	A	V/V	SSSE3	Subtract 16-bit signed integers horizontally, pack to MM1.
66 0F 38 05 /r PHSUBW xmm1, xmm2/m128	A	V/V	SSSE3	Subtract 16-bit signed integers horizontally, pack to XMM1.
0F 38 06 /r PHSUBD mm1, mm2/m64	A	V/V	SSSE3	Subtract 32-bit signed integers horizontally, pack to MM1.
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.

NOTES:

1. See note in Section 2.4, “Instruction Exception Specification”.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
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 source and destination operands, and packs the signed 16-bit results to the destination operand (first 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. Both operands can be MMX or XMM registers. When the source operand is a 128-bit memory operand, the operand must be aligned on a 16-byte boundary or a general-protection exception (#GP) will be generated.

In 64-bit mode, use the REX prefix to access additional registers.

128-bit Legacy SSE version: Bits (VLMAX-1:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: Bits (VLMAX-1:128) of the destination YMM register are zeroed. VEX.L must be 0, otherwise the instruction will #UD.

Operation

PHSUBW (with 64-bit operands)

```
mm1[15-0] = mm1[15-0] - mm1[31-16];
mm1[31-16] = mm1[47-32] - mm1[63-48];
mm1[47-32] = mm2/m64[15-0] - mm2/m64[31-16];
mm1[63-48] = mm2/m64[47-32] - mm2/m64[63-48];
```

PHSUBW (with 128-bit operands)

```
xmm1[15-0] = xmm1[15-0] - xmm1[31-16];
xmm1[31-16] = xmm1[47-32] - xmm1[63-48];
xmm1[47-32] = xmm1[79-64] - xmm1[95-80];
xmm1[63-48] = xmm1[111-96] - xmm1[127-112];
xmm1[79-64] = xmm2/m128[15-0] - xmm2/m128[31-16];
xmm1[95-80] = xmm2/m128[47-32] - xmm2/m128[63-48];
xmm1[111-96] = xmm2/m128[79-64] - xmm2/m128[95-80];
xmm1[127-112] = xmm2/m128[111-96] - xmm2/m128[127-112];
```

PHSUBD (with 64-bit operands)

```
mm1[31-0] = mm1[31-0] - mm1[63-32];
mm1[63-32] = mm2/m64[31-0] - mm2/m64[63-32];
```

PHSUBD (with 128-bit operands)

```
xmm1[31-0] = xmm1[31-0] - xmm1[63-32];
xmm1[63-32] = xmm1[95-64] - xmm1[127-96];
xmm1[95-64] = xmm2/m128[31-0] - xmm2/m128[63-32];
xmm1[127-96] = xmm2/m128[95-64] - xmm2/m128[127-96];
```

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[VLMAX-1:128] ← 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[VLMAX-1:128] ← 0
```

Intel C/C++ Compiler Intrinsic Equivalents

PHSUBW __m64 _mm_hsub_pi16 (__m64 a, __m64 b)

PHSUBW __m128i _mm_hsub_epi16 (__m128i a, __m128i b)

PHSUBD __m64 _mm_hsub_pi32 (__m64 a, __m64 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 and Saturate

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
0F 38 07 /r[1] PHSUBSW mm1, mm2/m64	A	V/V	SSSE3	Subtract 16-bit signed integer horizontally, pack saturated integers to MM1.
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.

NOTES:

1. See note in Section 2.4, “Instruction Exception Specification”.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
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 source and destination operands. The signed, saturated 16-bit results are packed to the destination operand (first operand). Both operands can be MMX or XMM register. When the source operand is a 128-bit memory operand, the operand must be aligned on a 16-byte boundary or a general-protection exception (#GP) will be generated.

In 64-bit mode, use the REX prefix to access additional registers.

128-bit Legacy SSE version: Bits (VLMAX-1:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: Bits (VLMAX-1:128) of the destination YMM register are zeroed. VEX.L must be 0, otherwise the instruction will #UD.

Operation

PHSUBSW (with 64-bit operands)

```
mm1[15-0] = SaturateToSignedWord(mm1[15-0] - mm1[31-16]);
mm1[31-16] = SaturateToSignedWord(mm1[47-32] - mm1[63-48]);
mm1[47-32] = SaturateToSignedWord(mm2/m64[15-0] - mm2/m64[31-16]);
mm1[63-48] = SaturateToSignedWord(mm2/m64[47-32] - mm2/m64[63-48]);
```

PHSUBSW (with 128-bit operands)

```
xmm1[15-0] = SaturateToSignedWord(xmm1[15-0] - xmm1[31-16]);
xmm1[31-16] = SaturateToSignedWord(xmm1[47-32] - xmm1[63-48]);
xmm1[47-32] = SaturateToSignedWord(xmm1[79-64] - xmm1[95-80]);
xmm1[63-48] = SaturateToSignedWord(xmm1[111-96] - xmm1[127-112]);
xmm1[79-64] = SaturateToSignedWord(xmm2/m128[15-0] - xmm2/m128[31-16]);
xmm1[95-80] =SaturateToSignedWord(xmm2/m128[47-32] - xmm2/m128[63-48]);
xmm1[111-96] =SaturateToSignedWord(xmm2/m128[79-64] - xmm2/m128[95-80]);
xmm1[127-112]= SaturateToSignedWord(xmm2/m128[111-96] - xmm2/m128[127-112]);
```

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[VLMAX-1:128] ← 0
```

Intel C/C++ Compiler Intrinsic Equivalent

PHSUBSW __m64 _mm_hsubs_pi16 (__m64 a, __m64 b)

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/PINSRD/PINSRQ — Insert Byte/Dword/Qword

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 3A 20 /r ib PINSRB *xmm1, r32/m8, imm8*	A	V/V	SSE4_1	Insert a byte integer value from *r32/m8* into *xmm1* at the destination element in *xmm1* specified by *imm8.*
66 0F 3A 22 /r ib PINSRD *xmm1, r/m32, imm8*	A	V/V	SSE4_1	Insert a dword integer value from *r/m32* into the *xmm1* at the destination element specified by *imm8.*
66 REX.W 0F 3A 22 /r ib PINSRQ *xmm1, r/m64, imm8*	A	N. E./V	SSE4_1	Insert a qword integer value from *r/m32 i*nto the *xmm1* at the destination element specified by *imm8.*
VEX.NDS.128.66.0F3A.W0 20 /r ib VPINSRB xmm1, xmm2, r32/m8, imm8	B	V[1]/V	AVX	Merge a byte integer value from r32/m8 and rest from xmm2 into xmm1 at the byte offset in imm8.
VEX.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/I	AVX	Insert a qword integer value from r64/m64 and rest from xmm2 into xmm1 at the qword offset in imm8.

NOTES:

1. In 64-bit mode, VEX.W1 is ignored for VPINSRB (similar to legacy REX.W=1 prefix with PINSRB).

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
A	ModRM:reg (w)	ModRM:r/m (r)	imm8	NA
B	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA

Description

Copies a byte/dword/qword from the source operand (second operand) and inserts it in the destination operand (first operand) at the location specified with the count operand (third operand). (The other elements in the destination register are left untouched.) The source operand can be a general-purpose register or a memory location. (When the source operand is a general-purpose register, PINSRB copies the low byte of the register.) The destination operand is an XMM register. The count operand is an 8-bit immediate. When specifying a qword[dword, byte] location in an XMM register, the [2, 4] least-significant bit(s) of the count operand specify the location.

In 64-bit mode, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15, R8-15). Use of REX.W permits the use of 64 bit general purpose registers.

128-bit Legacy SSE version: Bits (VLMAX-1:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: Bits (VLMAX-1:128) 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

```
CASE OF
    PINSRB:  SEL ← COUNT[3:0];
             MASK ← (0FFH << (SEL * 8));
             TEMP ← (((SRC[7:0] << (SEL *8)) AND MASK);
    PINSRD:  SEL ← COUNT[1:0];
             MASK ← (0FFFFFFFFH << (SEL * 32));
             TEMP ← (((SRC << (SEL *32)) AND MASK)  ;
    PINSRQ:  SEL ← COUNT[0]
             MASK ← (0FFFFFFFFFFFFFFFFH << (SEL * 64));
             TEMP ← (((SRC << (SEL *32)) AND MASK)  ;
ESAC;
        DEST ← ((DEST AND NOT MASK) OR TEMP);
```

VPINSRB (VEX.128 encoded version)

```
SEL ← imm8[3:0]
DEST[127:0] ← write_b_element(SEL, SRC2, SRC1)
DEST[VLMAX-1:128] ← 0
```

VPINSRD (VEX.128 encoded version)

```
SEL ← imm8[1:0]
DEST[127:0] ← write_d_element(SEL, SRC2, SRC1)
DEST[VLMAX-1:128] ← 0
```

VPINSRQ (VEX.128 encoded version)

```
SEL ← imm8[0]
DEST[127:0] ← write_q_element(SEL, SRC2, SRC1)
DEST[VLMAX-1:128] ← 0
```

Intel C/C++ Compiler Intrinsic Equivalent

PINSRB __m128i _mm_insert_epi8 (__m128i s1, int s2, const int ndx);

PINSRD __m128i _mm_insert_epi32 (__m128i s2, int s, const int ndx);
PINSRQ __m128i _mm_insert_epi64(__m128i s2, __int64 s, const int ndx);

Flags Affected

None.

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Exceptions Type 5; additionally

#UD	If VEX.L = 1.
	If VINSRQ in non-64-bit mode with VEX.W=1.

...

PINSRW—Insert Word

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
0F C4 /*r* ib[1] PINSRW *mm, r32/m16, imm8*	A	V/V	SSE	Insert the low word from *r32* or from *m16* into *mm* at the word position specified by *imm8*
66 0F C4 /*r* ib PINSRW xmm, *r32/m16*, imm8	A	V/V	SSE2	Move the low word of *r32* or from *m16* into xmm at the word position specified by *imm8*.
VEX.NDS.128.66.0F.W0 C4 /r ib VPINSRW xmm1, xmm2, r32/m16, imm8	B	V[2]/V	AVX	Insert a word integer value from r32/m16 and rest from xmm2 into xmm1 at the word offset in imm8.

NOTES:

1. See note in Section 2.4, “Instruction Exception Specification”.
2. In 64-bit mode, VEX.W1 is ignored for VPINSRW (similar to legacy REX.W=1 prefix in PINSRW).

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
A	ModRM:reg (w)	ModRM:r/m (r)	imm8	NA
B	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA

Description

Copies a word from the source operand (second operand) and inserts it in the destination operand (first operand) at the location specified with the count operand (third operand). (The other words in the destination register are left untouched.) The source operand can be a general-purpose register or a 16-bit memory location. (When the source operand is a general-purpose register, the low word of the register is copied.) The destination operand can be an MMX technology register or an XMM register. The count operand is an 8-bit immediate. When specifying a word location in an MMX technology register, the 2 least-significant bits of the count operand specify the location; for an XMM register, the 3 least-significant bits specify the location.

In 64-bit mode, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15, R8-15).

128-bit Legacy SSE version: Bits (VLMAX-1:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: Bits (VLMAX-1:128) of the destination YMM register are zeroed. VEX.L must be 0, otherwise the instruction will #UD.

Operation

PINSRW (with 64-bit source operand)

```
SEL ← COUNT AND 3H;
    CASE (Determine word position) OF
        SEL ← 0:    MASK ← 000000000000FFFFH;
        SEL ← 1:    MASK ← 00000000FFFF0000H;
        SEL ← 2:    MASK ← 0000FFFF00000000H;
        SEL ← 3:    MASK ← FFFF000000000000H;
DEST ← (DEST AND NOT MASK) OR (((SRC << (SEL * 16)) AND MASK);
```

PINSRW (with 128-bit source operand)

```
SEL ← COUNT AND 7H;
    CASE (Determine word position) OF
        SEL ← 0:    MASK ← 00000000000000000000000000000FFFFH;
        SEL ← 1:    MASK ← 00000000000000000000000000FFFF0000H;
        SEL ← 2:    MASK ← 000000000000000000000000FFFF00000000H;
        SEL ← 3:    MASK ← 0000000000000000000000FFFF000000000000H;
        SEL ← 4:    MASK ← 00000000000000000000FFFF0000000000000000H;
        SEL ← 5:    MASK ← 000000000000000000FFFF000000000000000000H;
        SEL ← 6:    MASK ← 0000000000000000FFFF00000000000000000000H;
        SEL ← 7:    MASK ← FFFF0000000000000000000000000000000000H;
DEST ← (DEST AND NOT MASK) OR (((SRC << (SEL * 16)) AND MASK);
```

VPINSRW (VEX.128 encoded version)

```
SEL ← imm8[2:0]
DEST[127:0] ← write_w_element(SEL, SRC2, SRC1)
DEST[VLMAX-1:128] ← 0
```

Intel C/C++ Compiler Intrinsic Equivalent

PINSRW __m64 _mm_insert_pi16 (__m64 a, int d, int n)

PINSRW __m128i _mm_insert_epi16 (__m128i a, int b, int imm)

Flags Affected

None.

Numeric Exceptions

None.

Other Exceptions

See Exceptions Type 5; additionally

#UD	If VEX.L = 1.
	If VINSRQ in non-64-bit mode with VEX.W=1.

...

PMADDUBSW — Multiply and Add Packed Signed and Unsigned Bytes

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
0F 38 04 /r[1] PMADDUBSW mm1, mm2/m64	A	V/V	MMX	Multiply signed and unsigned bytes, add horizontal pair of signed words, pack saturated signed-words to MM1.
66 0F 38 04 /r PMADDUBSW xmm1, xmm2/m128	A	V/V	SSSE3	Multiply signed and unsigned bytes, add horizontal pair of signed words, pack saturated signed-words to XMM1.
VEX.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.

NOTES:

1. See note in Section 2.4, "Instruction Exception Specification".

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
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 destination operand (first operand) with the corresponding signed byte of the source operand (second operand), producing intermediate signed 16-bit integers. Each adjacent pair of signed words is added and the saturated result is packed to the destination operand. For example, the lowest-order bytes (bits 7-0) in the source and destination operands are multiplied and the intermediate signed word result is added with the corresponding intermediate result from the 2nd lowest-order bytes (bits 15-8) of the operands; the sign-saturated result is stored in the lowest word of the destination register (15-0). The same operation is performed on the other pairs of adjacent bytes. Both operands can be MMX register or XMM registers. When the source operand is a 128-bit memory operand, the operand must be aligned on a 16-byte boundary or a general-protection exception (#GP) will be generated.

In 64-bit mode, use the REX prefix to access additional registers.

128-bit Legacy SSE version: Bits (VLMAX-1:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: Bits (VLMAX-1:128) of the destination YMM register are zeroed. VEX.L must be 0, otherwise the instruction will #UD.

Operation

PMADDUBSW (with 64 bit operands)

```
DEST[15-0] = SaturateToSignedWord(SRC[15-8]*DEST[15-8]+SRC[7-0]*DEST[7-0]);
DEST[31-16] = SaturateToSignedWord(SRC[31-24]*DEST[31-24]+SRC[23-16]*DEST[23-16]);
DEST[47-32] = SaturateToSignedWord(SRC[47-40]*DEST[47-40]+SRC[39-32]*DEST[39-32]);
DEST[63-48] = SaturateToSignedWord(SRC[63-56]*DEST[63-56]+SRC[55-48]*DEST[55-48]);
```

PMADDUBSW (with 128 bit operands)

```
DEST[15-0] = SaturateToSignedWord(SRC[15-8]* DEST[15-8]+SRC[7-0]*DEST[7-0]);
// Repeat operation for 2nd through 7th word
SRC1/DEST[127-112] = SaturateToSignedWord(SRC[127-120]*DEST[127-120]+ SRC[119-112]*
DEST[119-112]);
```

VPMADDUBSW (VEX.128 encoded version)

```
DEST[15:0] ← SaturateToSignedWord(SRC2[15:8]* SRC1[15:8]+SRC2[7:0]*SRC1[7:0])
// Repeat operation for 2nd through 7th word
DEST[127:112] ← SaturateToSignedWord(SRC2[127:120]*SRC1[127:120]+ SRC2[119:112]*
SRC1[119:112])
DEST[VLMAX-1:128] ← 0
```

Intel C/C++ Compiler Intrinsic Equivalents

PMADDUBSW __m64 _mm_maddubs_pi16 (__m64 a, __m64 b)

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.

...

PMADDWD—Multiply and Add Packed Integers

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
0F F5 /r[1] PMADDWD *mm, mm/m64*	A	V/V	MMX	Multiply the packed words in *mm* by the packed words in *mm/m64*, add adjacent doubleword results, and store in *mm*.
66 0F F5 /r PMADDWD *xmm1, xmm2/m128*	A	V/V	SSE2	Multiply the packed word integers in *xmm1* by the packed word integers in *xmm2/m128*, add adjacent doubleword results, and store in *xmm1*.
VEX.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.

NOTES:

1. See note in Section 2.4, "Instruction Exception Specification".

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
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 destination operand (first operand) by the corresponding signed words of the source operand (second operand), producing temporary signed, doubleword results. The adjacent doubleword results are then summed and stored in the destination operand. For example, the corresponding low-order words (15-0) and (31-16) in the source and destination operands are multiplied by one another and the doubleword results are added together and stored in the low doubleword of the destination register (31-0). The same operation is performed on the other pairs of adjacent words. (Figure 4-8 shows this operation when using 64-bit operands.) The source operand can be an MMX technology register or a 64-bit memory location, or it can be an XMM register or a 128-bit memory location. The destination operand can be an MMX technology register or an XMM register.

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.

In 64-bit mode, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

128-bit Legacy SSE version: Bits (VLMAX-1:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: Bits (VLMAX-1:128) of the destination YMM register are zeroed. VEX.L must be 0, otherwise the instruction will #UD.

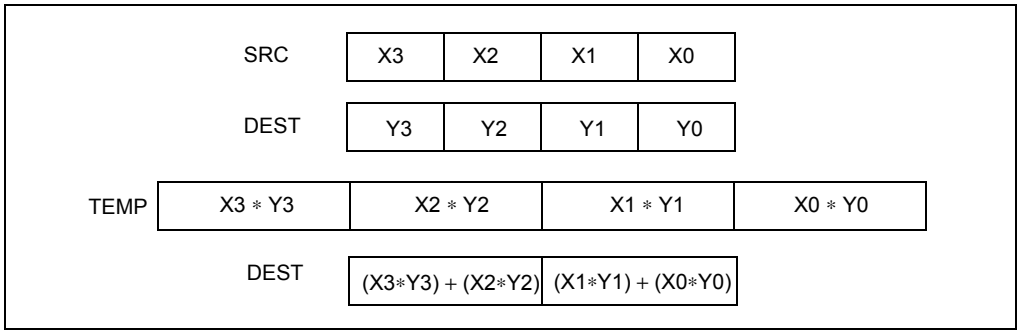


Figure 4-8 PMADDWD Execution Model Using 64-bit Operands

Operation

PMADDWD (with 64-bit operands)

```
DEST[31:0] ← (DEST[15:0] * SRC[15:0]) + (DEST[31:16] * SRC[31:16]);
DEST[63:32] ← (DEST[47:32] * SRC[47:32]) + (DEST[63:48] * SRC[63:48]);
```

PMADDWD (with 128-bit operands)

```
DEST[31:0] ← (DEST[15:0] * SRC[15:0]) + (DEST[31:16] * SRC[31:16]);
DEST[63:32] ← (DEST[47:32] * SRC[47:32]) + (DEST[63:48] * SRC[63:48]);
DEST[95:64] ← (DEST[79:64] * SRC[79:64]) + (DEST[95:80] * SRC[95:80]);
DEST[127:96] ← (DEST[111:96] * SRC[111:96]) + (DEST[127:112] * SRC[127:112]);
```

VPMADDWD (VEX.128 encoded version)

```
DEST[31:0] ← (SRC1[15:0] * SRC2[15:0]) + (SRC1[31:16] * SRC2[31:16])
DEST[63:32] ← (SRC1[47:32] * SRC2[47:32]) + (SRC1[63:48] * SRC2[63:48])
DEST[95:64] ← (SRC1[79:64] * SRC2[79:64]) + (SRC1[95:80] * SRC2[95:80])
DEST[127:96] ← (SRC1[111:96] * SRC2[111:96]) + (SRC1[127:112] * SRC2[127:112])
DEST[VLMAX-1:128] ← 0
```

Intel C/C++ Compiler Intrinsic Equivalent

PMADDWD __m64 _mm_madd_pi16(__m64 m1, __m64 m2)

PMADDWD __m128i _mm_madd_epi16 (__m128i a, __m128i b)

Flags Affected

None.

Numeric Exceptions

None.

Other Exceptions

See Exceptions Type 4; additionally

#UD	If VEX.L = 1.

...

PMAXSB — Maximum of Packed Signed Byte 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*.
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.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
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 packed signed byte integers in the destination operand (first operand) and the source operand (second operand), and returns the maximum for each packed value in the destination operand.

128-bit Legacy SSE version: Bits (VLMAX-1:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: Bits (VLMAX-1:1288) of the destination YMM register are zeroed. VEX.L must be 0, otherwise the instruction will #UD.

Operation

```
IF (DEST[7:0] > SRC[7:0])
    THEN DEST[7:0] ← DEST[7:0];
    ELSE DEST[7:0] ← SRC[7:0]; FI;
IF (DEST[15:8] > SRC[15:8])
    THEN DEST[15:8] ← DEST[15:8];
    ELSE DEST[15:8] ← SRC[15:8]; FI;
IF (DEST[23:16] > SRC[23:16])
    THEN DEST[23:16] ← DEST[23:16];
    ELSE DEST[23:16] ← SRC[23:16]; FI;
IF (DEST[31:24] > SRC[31:24])
    THEN DEST[31:24] ← DEST[31:24];
    ELSE DEST[31:24] ← SRC[31:24]; FI;
IF (DEST[39:32] > SRC[39:32])
    THEN DEST[39:32] ← DEST[39:32];
```

```
    ELSE DEST[39:32] ← SRC[39:32]; FI;
IF (DEST[47:40] > SRC[47:40])
    THEN DEST[47:40] ← DEST[47:40];
    ELSE DEST[47:40] ← SRC[47:40]; FI;
IF (DEST[55:48] > SRC[55:48])
    THEN DEST[55:48] ← DEST[55:48];
    ELSE DEST[55:48] ← SRC[55:48]; FI;
IF (DEST[63:56] > SRC[63:56])
    THEN DEST[63:56] ← DEST[63:56];
    ELSE DEST[63:56] ← SRC[63:56]; FI;
IF (DEST[71:64] > SRC[71:64])
    THEN DEST[71:64] ← DEST[71:64];
    ELSE DEST[71:64] ← SRC[71:64]; FI;
IF (DEST[79:72] > SRC[79:72])
    THEN DEST[79:72] ← DEST[79:72];
    ELSE DEST[79:72] ← SRC[79:72]; FI;
IF (DEST[87:80] > SRC[87:80])
    THEN DEST[87:80] ← DEST[87:80];
    ELSE DEST[87:80] ← SRC[87:80]; FI;
IF (DEST[95:88] > SRC[95:88])
    THEN DEST[95:88] ← DEST[95:88];
    ELSE DEST[95:88] ← SRC[95:88]; FI;
IF (DEST[103:96] > SRC[103:96])
    THEN DEST[103:96] ← DEST[103:96];
    ELSE DEST[103:96] ← SRC[103:96]; FI;
IF (DEST[111:104] > SRC[111:104])
    THEN DEST[111:104] ← DEST[111:104];
    ELSE DEST[111:104] ← SRC[111:104]; FI;
IF (DEST[119:112] > SRC[119:112])
    THEN DEST[119:112] ← DEST[119:112];
    ELSE DEST[119:112] ← SRC[119:112]; FI;
IF (DEST[127:120] > SRC[127:120])
    THEN DEST[127:120] ← DEST[127:120];
    ELSE DEST[127:120] ← SRC[127:120]; FI;
```

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[VLMAX-1:128] ← 0
```

Intel C/C++ Compiler Intrinsic Equivalent

PMAXSB __m128i _mm_max_epi8 (__m128i a, __m128i b);

Flags Affected

None.

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Exceptions Type 4; additionally

#UD	If VEX.L = 1.

...

PMAXSD — Maximum of Packed Signed Dword Integers

Opcode/ Instruction		Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
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 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

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
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 packed signed dword integers in the destination operand (first operand) and the source operand (second operand), and returns the maximum for each packed value in the destination operand.

128-bit Legacy SSE version: Bits (VLMAX-1:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: Bits (VLMAX-1:1288) of the destination YMM register are zeroed. VEX.L must be 0, otherwise the instruction will #UD.

Operation

```
IF (DEST[31:0] > SRC[31:0])
    THEN DEST[31:0] ← DEST[31:0];
    ELSE DEST[31:0] ← SRC[31:0]; FI;
IF (DEST[63:32] > SRC[63:32])
```

```
    THEN DEST[63:32] ← DEST[63:32];
    ELSE DEST[63:32] ← SRC[63:32]; FI;
IF (DEST[95:64] > SRC[95:64])
    THEN DEST[95:64] ← DEST[95:64];
    ELSE DEST[95:64] ← SRC[95:64]; FI;
IF (DEST[127:96] > SRC[127:96])
    THEN DEST[127:96] ← DEST[127:96];
    ELSE DEST[127:96] ← SRC[127:96]; FI;
```

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[VLMAX-1:128] ← 0
```

Intel C/C++ Compiler Intrinsic Equivalent

PMAXSD __m128i _mm_max_epi32 (__m128i a, __m128i b);

Flags Affected

None.

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Exceptions Type 4; additionally

#UD	If VEX.L = 1.

...

PMAXSW—Maximum of Packed Signed Word Integers

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
0F EE /r[1] PMAXSW *mm1, mm2/m64*	A	V/V	SSE	Compare signed word integers in *mm2/m64* and *mm1* and return maximum values.
66 0F EE /r PMAXSW *xmm1, xmm2/m128*	A	V/V	SSE2	Compare signed word integers in *xmm2/m128* and *xmm1* and return maximum values.
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.

NOTES:

1. See note in Section 2.4, "Instruction Exception Specification".

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
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 word integers in the destination operand (first operand) and the source operand (second operand), and returns the maximum value for each pair of word integers to the destination operand. The source operand can be an MMX technology register or a 64-bit memory location, or it can be an XMM register or a 128-bit memory location. The destination operand can be an MMX technology register or an XMM register.

In 64-bit mode, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

128-bit Legacy SSE version: Bits (VLMAX-1:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: Bits (VLMAX-1:128) of the destination YMM register are zeroed. VEX.L must be 0, otherwise the instruction will #UD.

Operation

PMAXSW (64-bit operands)

```
IF DEST[15:0] > SRC[15:0]) THEN
    DEST[15:0] ← DEST[15:0];
ELSE
    DEST[15:0] ← SRC[15:0]; FI;
(* Repeat operation for 2nd and 3rd words in source and destination operands *)
```

```
    IF DEST[63:48] > SRC[63:48]) THEN
        DEST[63:48] ← DEST[63:48];
    ELSE
        DEST[63:48] ← SRC[63:48]; FI;
```

PMAXSW (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;
```

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[VLMAX-1:128] ← 0
```

Intel C/C++ Compiler Intrinsic Equivalent

PMAXSW __m64 _mm_max_pi16(__m64 a, __m64 b)

PMAXSW __m128i _mm_max_epi16 (__m128i a, __m128i b)

Flags Affected

None.

Numeric Exceptions

None.

Other Exceptions

See Exceptions Type 4; additionally

#UD	If VEX.L = 1.

...

PMAXUB—Maximum of Packed Unsigned Byte Integers

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
0F DE /r[1] PMAXUB *mm1, mm2/m64*	A	V/V	SSE	Compare unsigned byte integers in *mm2/m64* and *mm1* and returns maximum values.
66 0F DE /r PMAXUB *xmm1, xmm2/m128*	A	V/V	SSE2	Compare unsigned byte integers in *xmm2/m128* and *xmm1* and returns maximum values.
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.

NOTES:

1. See note in Section 2.4, "Instruction Exception Specification".

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
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 integers in the destination operand (first operand) and the source operand (second operand), and returns the maximum value for each pair of byte integers to the destination operand. The source operand can be an MMX technology register or a 64-bit memory location, or it can be an XMM register or a 128-bit memory location. The destination operand can be an MMX technology register or an XMM register.

In 64-bit mode, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

128-bit Legacy SSE version: Bits (VLMAX-1:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: Bits (VLMAX-1:128) of the destination YMM register are zeroed. VEX.L must be 0, otherwise the instruction will #UD.

Operation

PMAXUB (64-bit operands)

```
IF DEST[7:0] > SRC[17:0]) THEN
    DEST[7:0] ← DEST[7:0];
ELSE
    DEST[7:0] ← SRC[7:0]; FI;
(* Repeat operation for 2nd through 7th bytes in source and destination operands *)
```

```
    IF DEST[63:56] > SRC[63:56]) THEN
        DEST[63:56] ← DEST[63:56];
    ELSE
        DEST[63:56] ← SRC[63:56]; FI;
```

PMAXUB (128-bit operands)

```
    IF DEST[7:0] > SRC[17:0]) THEN
        DEST[7:0] ← DEST[7:0];
    ELSE
        DEST[7:0] ← SRC[7:0]; FI;
    (* Repeat operation for 2nd through 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;
```

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[VLMAX-1:128] ← 0
```

Intel C/C++ Compiler Intrinsic Equivalent

PMAXUB __m64 _mm_max_pu8(__m64 a, __m64 b)

PMAXUB __m128i _mm_max_epu8 (__m128i a, __m128i b)

Flags Affected

None.

Numeric Exceptions

None.

Other Exceptions

See Exceptions Type 4; additionally

#UD	If VEX.L = 1.

...

PMAXUD — Maximum of Packed Unsigned Dword Integers

Opcode/ Instruction		Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 38 3F /r PMAXUD *xmm1, xmm2/m128*		A	V/V	SSE4_1	Compare packed unsigned dword integers in *xmm1* and *xmm2/m128* and store packed maximum values in *xmm1*.
VEX.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

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
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 packed unsigned dword integers in the destination operand (first operand) and the source operand (second operand), and returns the maximum for each packed value in the destination operand.

128-bit Legacy SSE version: Bits (VLMAX-1:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: Bits (VLMAX-1:128) of the destination YMM register are zeroed. VEX.L must be 0, otherwise the instruction will #UD.

Operation

```
IF (DEST[31:0] > SRC[31:0])
    THEN DEST[31:0] ← DEST[31:0];
    ELSE DEST[31:0] ← SRC[31:0]; FI;
IF (DEST[63:32] > SRC[63:32])
    THEN DEST[63:32] ← DEST[63:32];
    ELSE DEST[63:32] ← SRC[63:32]; FI;
IF (DEST[95:64] > SRC[95:64])
    THEN DEST[95:64] ← DEST[95:64];
    ELSE DEST[95:64] ← SRC[95:64]; FI;
IF (DEST[127:96] > SRC[127:96])
    THEN DEST[127:96] ← DEST[127:96];
    ELSE DEST[127:96] ← SRC[127:96]; FI;
```

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[VLMAX-1:128] ← 0
```

Intel C/C++ Compiler Intrinsic Equivalent

PMAXUD __m128i _mm_max_epu32 (__m128i a, __m128i b);

Flags Affected

None.

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Exceptions Type 4; additionally

#UD	If VEX.L = 1.

...

PMAXUW — Maximum of Packed Word Integers

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 38 3E /r PMAXUW *xmm1, xmm2/m128*	A	V/V	SSE4_1	Compare packed unsigned word integers in *xmm1* and *xmm2/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.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
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 packed unsigned word integers in the destination operand (first operand) and the source operand (second operand), and returns the maximum for each packed value in the destination operand.

128-bit Legacy SSE version: Bits (VLMAX-1:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: Bits (VLMAX-1:128) of the destination YMM register are zeroed. VEX.L must be 0, otherwise the instruction will #UD.

Operation

```
IF (DEST[15:0] > SRC[15:0])
    THEN DEST[15:0] ← DEST[15:0];
    ELSE DEST[15:0] ← SRC[15:0]; FI;
IF (DEST[31:16] > SRC[31:16])
    THEN DEST[31:16] ← DEST[31:16];
    ELSE DEST[31:16] ← SRC[31:16]; FI;
IF (DEST[47:32] > SRC[47:32])
    THEN DEST[47:32] ← DEST[47:32];
    ELSE DEST[47:32] ← SRC[47:32]; FI;
IF (DEST[63:48] > SRC[63:48])
    THEN DEST[63:48] ← DEST[63:48];
    ELSE DEST[63:48] ← SRC[63:48]; FI;
IF (DEST[79:64] > SRC[79:64])
    THEN DEST[79:64] ← DEST[79:64];
    ELSE DEST[79:64] ← SRC[79:64]; FI;
IF (DEST[95:80] > SRC[95:80])
    THEN DEST[95:80] ← DEST[95:80];
    ELSE DEST[95:80] ← SRC[95:80]; FI;
IF (DEST[111:96] > SRC[111:96])
    THEN DEST[111:96] ← DEST[111:96];
    ELSE DEST[111:96] ← SRC[111:96]; FI;
IF (DEST[127:112] > SRC[127:112])
    THEN DEST[127:112] ← DEST[127:112];
    ELSE DEST[127:112] ← SRC[127:112]; FI;
```

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[VLMAX-1:128] ← 0
```

Intel C/C++ Compiler Intrinsic Equivalent

PMAXUW__m128i _mm_max_epu16 (__m128i a, __m128i b);

Flags Affected

None.

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Exceptions Type 4; additionally

#UD	If VEX.L = 1.

...

PMINSB — Minimum of Packed Signed Byte 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*.
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.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
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 packed signed byte integers in the destination operand (first operand) and the source operand (second operand), and returns the minimum for each packed value in the destination operand.

128-bit Legacy SSE version: Bits (VLMAX-1:1288) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: Bits (VLMAX-1:128) of the destination YMM register are zeroed. VEX.L must be 0, otherwise the instruction will #UD.

Operation

```
IF (DEST[7:0] < SRC[7:0])
    THEN DEST[7:0] ← DEST[7:0];
    ELSE DEST[7:0] ← SRC[7:0]; FI;
IF (DEST[15:8] < SRC[15:8])
    THEN DEST[15:8] ← DEST[15:8];
    ELSE DEST[15:8] ← SRC[15:8]; FI;
IF (DEST[23:16] < SRC[23:16])
    THEN DEST[23:16] ← DEST[23:16];
    ELSE DEST[23:16] ← SRC[23:16]; FI;
IF (DEST[31:24] < SRC[31:24])
    THEN DEST[31:24] ← DEST[31:24];
    ELSE DEST[31:24] ← SRC[31:24]; FI;
IF (DEST[39:32] < SRC[39:32])
    THEN DEST[39:32] ← DEST[39:32];
    ELSE DEST[39:32] ← SRC[39:32]; FI;
IF (DEST[47:40] < SRC[47:40])
```

```
    THEN DEST[47:40] ← DEST[47:40];
    ELSE DEST[47:40] ← SRC[47:40]; FI;
IF (DEST[55:48] < SRC[55:48])
    THEN DEST[55:48] ← DEST[55:48];
    ELSE DEST[55:48] ← SRC[55:48]; FI;
IF (DEST[63:56] < SRC[63:56])
    THEN DEST[63:56] ← DEST[63:56];
    ELSE DEST[63:56] ← SRC[63:56]; FI;
IF (DEST[71:64] < SRC[71:64])
    THEN DEST[71:64] ← DEST[71:64];
    ELSE DEST[71:64] ← SRC[71:64]; FI;
IF (DEST[79:72] < SRC[79:72])
    THEN DEST[79:72] ← DEST[79:72];
    ELSE DEST[79:72] ← SRC[79:72]; FI;
IF (DEST[87:80] < SRC[87:80])
    THEN DEST[87:80] ← DEST[87:80];
    ELSE DEST[87:80] ← SRC[87:80]; FI;
IF (DEST[95:88] < SRC[95:88])
    THEN DEST[95:88] ← DEST[95:88];
    ELSE DEST[95:88] ← SRC[95:88]; FI;
IF (DEST[103:96] < SRC[103:96])
    THEN DEST[103:96] ← DEST[103:96];
    ELSE DEST[103:96] ← SRC[103:96]; FI;
IF (DEST[111:104] < SRC[111:104])
    THEN DEST[111:104] ← DEST[111:104];
    ELSE DEST[111:104] ← SRC[111:104]; FI;
IF (DEST[119:112] < SRC[119:112])
    THEN DEST[119:112] ← DEST[119:112];
    ELSE DEST[119:112] ← SRC[119:112]; FI;
IF (DEST[127:120] < SRC[127:120])
    THEN DEST[127:120] ← DEST[127:120];
    ELSE DEST[127:120] ← SRC[127:120]; FI;
```

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[VLMAX-1:128] ← 0
```

Intel C/C++ Compiler Intrinsic Equivalent

PMINSB __m128i _mm_min_epi8 (__m128i a, __m128i b);

Flags Affected

None.

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Exceptions Type 4; additionally

#UD	If VEX.L = 1.

...

PMINSD — Minimum of Packed Dword Integers

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 38 39 /r PMINSD *xmm1, xmm2/m128*	A	V/V	SSE4_1	Compare packed signed dword integers in *xmm1* and *xmm2/m128* and store packed minimum values in *xmm1*.
VEX.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

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
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 packed signed dword integers in the destination operand (first operand) and the source operand (second operand), and returns the minimum for each packed value in the destination operand.

128-bit Legacy SSE version: Bits (VLMAX-1:1288) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: Bits (VLMAX-1:128) of the destination YMM register are zeroed. VEX.L must be 0, otherwise the instruction will #UD.

Operation

```
IF (DEST[31:0] < SRC[31:0])
    THEN DEST[31:0] ← DEST[31:0];
    ELSE DEST[31:0] ← SRC[31:0]; FI;
IF (DEST[63:32] < SRC[63:32])
```

```
    THEN DEST[63:32] ← DEST[63:32];
    ELSE DEST[63:32] ← SRC[63:32]; FI;
IF (DEST[95:64] < SRC[95:64])
    THEN DEST[95:64] ← DEST[95:64];
    ELSE DEST[95:64] ← SRC[95:64]; FI;
IF (DEST[127:96] < SRC[127:96])
    THEN DEST[127:96] ← DEST[127:96];
    ELSE DEST[127:96] ← SRC[127:96]; FI;
```

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[VLMAX-1:128] ← 0
```

Intel C/C++ Compiler Intrinsic Equivalent

PMINSD __m128i _mm_min_epi32 (__m128i a, __m128i b);

Flags Affected

None.

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Exceptions Type 4; additionally

#UD	If VEX.L = 1.

...

PMINSW—Minimum of Packed Signed Word Integers

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
0F EA /r[1] PMINSW *mm1, mm2/m64*	A	V/V	SSE	Compare signed word integers in *mm2/m64* and *mm1* and return minimum values.
66 0F EA /r PMINSW *xmm1, xmm2/m128*	A	V/V	SSE2	Compare signed word integers in *xmm2/m128* and *xmm1* and return minimum values.
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.

NOTES:

1. See note in Section 2.4, "Instruction Exception Specification".

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
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 word integers in the destination operand (first operand) and the source operand (second operand), and returns the minimum value for each pair of word integers to the destination operand. The source operand can be an MMX technology register or a 64-bit memory location, or it can be an XMM register or a 128-bit memory location. The destination operand can be an MMX technology register or an XMM register.

In 64-bit mode, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

128-bit Legacy SSE version: Bits (VLMAX-1:1288) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: Bits (VLMAX-1:128) of the destination YMM register are zeroed. VEX.L must be 0, otherwise the instruction will #UD.

Operation

PMINSW (64-bit operands)

```
IF DEST[15:0] < SRC[15:0] THEN
    DEST[15:0] ← DEST[15:0];
ELSE
    DEST[15:0] ← SRC[15:0]; FI;
(* Repeat operation for 2nd and 3rd words in source and destination operands *)
```

```
    IF DEST[63:48] < SRC[63:48] THEN
        DEST[63:48] ← DEST[63:48];
    ELSE
        DEST[63:48] ← SRC[63:48]; FI;
```

PMINSW (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/m64[127:112] THEN
        DEST[127:112] ← DEST[127:112];
    ELSE
        DEST[127:112] ← SRC[127:112]; FI;
```

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[VLMAX-1:128] ← 0
```

Intel C/C++ Compiler Intrinsic Equivalent

PMINSW __m64 _mm_min_pi16 (__m64 a, __m64 b)

PMINSW __m128i _mm_min_epi16 (__m128i a, __m128i b)

Flags Affected

None.

Numeric Exceptions

None.

Other Exceptions

See Exceptions Type 4; additionally

#UD	If VEX.L = 1.
#MF	(64-bit operations only) If there is a pending x87 FPU exception.

...

PMINUB—Minimum of Packed Unsigned Byte Integers

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
0F DA /r[1] PMINUB *mm1, mm2/m64*	A	V/V	SSE	Compare unsigned byte integers in *mm2/m64* and *mm1* and returns minimum values.
66 0F DA /r PMINUB *xmm1, xmm2/m128*	A	V/V	SSE2	Compare unsigned byte integers in *xmm2/m128* and *xmm1* and returns minimum values.
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.

NOTES:

1. See note in Section 2.4, "Instruction Exception Specification".

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
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 integers in the destination operand (first operand) and the source operand (second operand), and returns the minimum value for each pair of byte integers to the destination operand. The source operand can be an MMX technology register or a 64-bit memory location, or it can be an XMM register or a 128-bit memory location. The destination operand can be an MMX technology register or an XMM register.

In 64-bit mode, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

128-bit Legacy SSE version: Bits (VLMAX-1:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: Bits (VLMAX-1:128) of the destination YMM register are zeroed. VEX.L must be 0, otherwise the instruction will #UD.

Operation

PMINUB (for 64-bit operands)

```
IF DEST[7:0] < SRC[17:0] THEN
    DEST[7:0] ← DEST[7:0];
ELSE
    DEST[7:0] ← SRC[7:0]; FI;
(* Repeat operation for 2nd through 7th bytes in source and destination operands *)
```

```
    IF DEST[63:56] < SRC[63:56] THEN
        DEST[63:56] ← DEST[63:56];
    ELSE
        DEST[63:56] ← SRC[63:56]; FI;
```

PMINUB (for 128-bit operands)

```
    IF DEST[7:0] < SRC[17:0] THEN
        DEST[7:0] ← DEST[7:0];
    ELSE
        DEST[7:0] ← SRC[7:0]; FI;
    (* Repeat operation for 2nd through 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;
```

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[VLMAX-1:128] ← 0
```

Intel C/C++ Compiler Intrinsic Equivalent

PMINUB __m64 _m_min_pu8 (__m64 a, __m64 b)

PMINUB __m128i _mm_min_epu8 (__m128i a, __m128i b)

Flags Affected

None.

Numeric Exceptions

None.

Other Exceptions

See Exceptions Type 4; additionally

#UD	If VEX.L = 1.

...

PMINUD — Minimum of Packed Dword Integers

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 38 3B /r PMINUD *xmm1, xmm2/m128*	A	V/V	SSE4_1	Compare packed unsigned dword integers in *xmm1* and *xmm2/m128* and store packed minimum values in *xmm1*.
VEX.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

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
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 packed unsigned dword integers in the destination operand (first operand) and the source operand (second operand), and returns the minimum for each packed value in the destination operand.

128-bit Legacy SSE version: Bits (VLMAX-1:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: Bits (VLMAX-1:128) of the destination YMM register are zeroed. VEX.L must be 0, otherwise the instruction will #UD.

Operation

```
IF (DEST[31:0] < SRC[31:0])
    THEN DEST[31:0] ← DEST[31:0];
    ELSE DEST[31:0] ← SRC[31:0]; FI;
IF (DEST[63:32] < SRC[63:32])
    THEN DEST[63:32] ← DEST[63:32];
    ELSE DEST[63:32] ← SRC[63:32]; FI;
IF (DEST[95:64] < SRC[95:64])
    THEN DEST[95:64] ← DEST[95:64];
    ELSE DEST[95:64] ← SRC[95:64]; FI;
IF (DEST[127:96] < SRC[127:96])
    THEN DEST[127:96] ← DEST[127:96];
    ELSE DEST[127:96] ← SRC[127:96]; FI;
```

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[VLMAX-1:128] ← 0
```

Intel C/C++ Compiler Intrinsic Equivalent

PMINUD __m128i _mm_min_epu32 (__m128i a, __m128i b);

Flags Affected

None.

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Exceptions Type 4; additionally

#UD	If VEX.L = 1.

...

PMINUW — Minimum of Packed Word Integers

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 38 3A /r PMINUW *xmm1, xmm2/m128*	A	V/V	SSE4_1	Compare packed unsigned word integers in *xmm1* and *xmm2/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.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
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 packed unsigned word integers in the destination operand (first operand) and the source operand (second operand), and returns the minimum for each packed value in the destination operand.

128-bit Legacy SSE version: Bits (VLMAX-1:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: Bits (VLMAX-1:128) of the destination YMM register are zeroed. VEX.L must be 0, otherwise the instruction will #UD.

Operation

```
IF (DEST[15:0] < SRC[15:0])
    THEN DEST[15:0] ← DEST[15:0];
    ELSE DEST[15:0] ← SRC[15:0]; FI;
IF (DEST[31:16] < SRC[31:16])
    THEN DEST[31:16] ← DEST[31:16];
    ELSE DEST[31:16] ← SRC[31:16]; FI;
IF (DEST[47:32] < SRC[47:32])
    THEN DEST[47:32] ← DEST[47:32];
    ELSE DEST[47:32] ← SRC[47:32]; FI;
IF (DEST[63:48] < SRC[63:48])
    THEN DEST[63:48] ← DEST[63:48];
    ELSE DEST[63:48] ← SRC[63:48]; FI;
IF (DEST[79:64] < SRC[79:64])
    THEN DEST[79:64] ← DEST[79:64];
    ELSE DEST[79:64] ← SRC[79:64]; FI;
IF (DEST[95:80] < SRC[95:80])
    THEN DEST[95:80] ← DEST[95:80];
    ELSE DEST[95:80] ← SRC[95:80]; FI;
IF (DEST[111:96] < SRC[111:96])
    THEN DEST[111:96] ← DEST[111:96];
    ELSE DEST[111:96] ← SRC[111:96]; FI;
IF (DEST[127:112] < SRC[127:112])
    THEN DEST[127:112] ← DEST[127:112];
    ELSE DEST[127:112] ← SRC[127:112]; FI;
```

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[VLMAX-1:128] ← 0
```

Intel C/C++ Compiler Intrinsic Equivalent

PMINUW __m128i _mm_min_epu16 (__m128i a, __m128i b);

Flags Affected

None.

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
0F D7 /r[1]	PMOVMSKB *reg, mm*	A	V/V	SSE	Move a byte mask of *mm* to *reg*. The upper bits of r32 or r64 are zeroed
66 0F D7 /r	PMOVMSKB *reg, xmm*	A	V/V	SSE2	Move a byte mask of *xmm* to *reg*. The upper bits of r32 or r64 are zeroed
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.

NOTES:

1. See note in Section 2.4, “Instruction Exception Specification”.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
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 MMX technology register or an XMM register; the destination operand is a general-purpose register. When operating on 64-

bit operands, the byte mask is 8 bits; when operating on 128-bit operands, the byte mask is 16-bits.

In 64-bit mode, the instruction can access additional registers (XMM8-XMM15, R8-R15) when used with a REX.R prefix. The default operand size is 64-bit in 64-bit mode.

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

PMOVMSKB (with 64-bit source operand and r32)

```
r32[0] ← SRC[7];
r32[1] ← SRC[15];
(* Repeat operation for bytes 2 through 6 *)
r32[7] ← SRC[63];
r32[31:8] ← ZERO_FILL;
```

(V)PMOVMSKB (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;
```

PMOVMSKB (with 64-bit source operand and r64)

```
r64[0] ← SRC[7];
r64[1] ← SRC[15];
(* Repeat operation for bytes 2 through 6 *)
r64[7] ← SRC[63];
r64[63:8] ← ZERO_FILL;
```

(V)PMOVMSKB (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_pi8(__m64 a)
PMOVMSKB	int _mm_movemask_epi8 (__m128i a)

Flags Affected

None.

Numeric 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 signed 8-bit integers in the low 8 bytes of *xmm2/m64* to 8 packed signed 16-bit integers in *xmm1.*
66 0f 38 21 /r PMOVSXBD *xmm1, xmm2/m32*	A	V/V	SSE4_1	Sign extend 4 packed signed 8-bit integers in the low 4 bytes of *xmm2/m32* to 4 packed signed 32-bit integers in *xmm1.*
66 0f 38 22 /r PMOVSXBQ *xmm1, xmm2/m16*	A	V/V	SSE4_1	Sign extend 2 packed signed 8-bit integers in the low 2 bytes of *xmm2/m16* to 2 packed signed 64-bit integers in *xmm1.*
66 0f 38 23 /r PMOVSXWD *xmm1, xmm2/m64*	A	V/V	SSE4_1	Sign extend 4 packed signed 16-bit integers in the low 8 bytes of *xmm2/m64* to 4 packed signed 32-bit integers in *xmm1.*
66 0f 38 24 /r PMOVSXWQ *xmm1, xmm2/m32*	A	V/V	SSE4_1	Sign extend 2 packed signed 16-bit integers in the low 4 bytes of *xmm2/m32* to 2 packed signed 64-bit integers in *xmm1.*
66 0f 38 25 /r PMOVSXDQ *xmm1, xmm2/m64*	A	V/V	SSE4_1	Sign extend 2 packed signed 32-bit integers in the low 8 bytes of *xmm2/m64* to 2 packed signed 64-bit integers in *xmm1.*
VEX.128.66.0F38.WIG 20 /r VPMOVSXBW xmm1, xmm2/m64	A	V/V	AVX	Sign extend 8 packed 8-bit integers in the low 8 bytes of xmm2/m64 to 8 packed 16-bit integers in xmm1.
VEX.128.66.0F38.WIG 21 /r VPMOVSXBD xmm1, xmm2/m32	A	V/V	AVX	Sign extend 4 packed 8-bit integers in the low 4 bytes of xmm2/m32 to 4 packed 32-bit integers in xmm1.
VEX.128.66.0F38.WIG 22 /r VPMOVSXBQ xmm1, xmm2/m16	A	V/V	AVX	Sign extend 2 packed 8-bit integers in the low 2 bytes of xmm2/m16 to 2 packed 64-bit integers in xmm1.

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
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

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

Description

Sign-extend the low byte/word/dword values in each word/dword/qword element of the source operand (second operand) to word/dword/qword integers and stored as packed data in the destination operand (first operand).

128-bit Legacy SSE version: Bits (VLMAX-1:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: Bits (VLMAX-1:128) of the destination YMM register are zeroed. VEX.vvvv is reserved and must be 1111b, VEX.L must be 0, otherwise the instruction will #UD.

Operation

PMOVSXBW

```
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]);
```

PMOVSXBD

```
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]);
```

PMOVSXBQ
DEST[63:0] ← SignExtend(SRC[7:0]);
DEST[127:64] ← SignExtend(SRC[15:8]);

PMOVSXWD
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]);

PMOVSXWQ
DEST[63:0] ← SignExtend(SRC[15:0]);
DEST[127:64] ← SignExtend(SRC[31:16]);

PMOVSXDQ
DEST[63:0] ← SignExtend(SRC[31:0]);
DEST[127:64] ← SignExtend(SRC[63:32]);

VPMOVSXBW
Packed_Sign_Extend_BYTE_to_WORD()
DEST[VLMAX-1:128] ← 0

VPMOVSXBD
Packed_Sign_Extend_BYTE_to_DWORD()
DEST[VLMAX-1:128] ← 0

VPMOVSXBQ
Packed_Sign_Extend_BYTE_to_QWORD()
DEST[VLMAX-1:128] ← 0

VPMOVSXWD
Packed_Sign_Extend_WORD_to_DWORD()
DEST[VLMAX-1:128] ← 0

VPMOVSXWQ
Packed_Sign_Extend_WORD_to_QWORD()
DEST[VLMAX-1:128] ← 0

VPMOVSXDQ
Packed_Sign_Extend_DWORD_to_QWORD()
DEST[VLMAX-1:128] ← 0

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);

Flags Affected

None.

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

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

Description

Zero-extend the low byte/word/dword values in each word/dword/qword element of the source operand (second operand) to word/dword/qword integers and stored as packed data in the destination operand (first operand).

128-bit Legacy SSE version: Bits (VLMAX-1:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: Bits (VLMAX-1:128) of the destination YMM register are zeroed. VEX.vvvv is reserved and must be 1111b, VEX.L must be 0, otherwise the instruction will #UD.

Operation

PMOVZXBW

```
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]);
```

PMOVZXBD

```
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]);

PMOVZXQB
DEST[63:0] ← ZeroExtend(SRC[7:0]);
DEST[127:64] ← ZeroExtend(SRC[15:8]);

PMOVZXWD
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]);

PMOVZXWQ
DEST[63:0] ← ZeroExtend(SRC[15:0]);
DEST[127:64] ← ZeroExtend(SRC[31:16]);

PMOVZXDQ
DEST[63:0] ← ZeroExtend(SRC[31:0]);
DEST[127:64] ← ZeroExtend(SRC[63:32]);

VPMOVZXBW
Packed_Zero_Extend_BYTE_to_WORD()
DEST[VLMAX-1:128] ← 0

VPMOVZXBD
Packed_Zero_Extend_BYTE_to_DWORD()
DEST[VLMAX-1:128] ← 0

VPMOVZXBQ
Packed_Zero_Extend_BYTE_to_QWORD()
DEST[VLMAX-1:128] ← 0

VPMOVZXWD
Packed_Zero_Extend_WORD_to_DWORD()
DEST[VLMAX-1:128] ← 0

VPMOVZXWQ
Packed_Zero_Extend_WORD_to_QWORD()
DEST[VLMAX-1:128] ← 0

VPMOVZXDQ
Packed_Zero_Extend_DWORD_to_QWORD()
DEST[VLMAX-1:128] ← 0

Flags Affected

None

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);

Flags Affected

None.

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Exceptions Type 5; additionally

#UD	If VEX.L = 1.
	If VEX.vvvv != 1111B.

...

PMULDQ — Multiply Packed Signed Dword 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 the packed signed dword integers in *xmm1* and *xmm2/m128* and store the quadword product 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

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
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 two signed multiplications from two pairs of signed dword integers and stores two 64-bit products in the destination operand (first operand). The 64-bit product from the first/third dword element in the destination operand and the first/third dword

element of the source operand (second operand) is stored to the low/high qword element of the destination.

If the source is a memory operand then all 128 bits will be fetched from memory but the second and fourth dwords will not be used in the computation.

128-bit Legacy SSE version: Bits (VLMAX-1:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: Bits (VLMAX-1:128) of the destination YMM register are zeroed. VEX.L must be 0, otherwise the instruction will #UD.

Operation

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[VLMAX-1:128] (Unmodified)

VPMULDQ (VEX.128 encoded version)
DEST[63:0] ← SRC1[31:0] * SRC2[31:0]
DEST[127:64] ← SRC1[95:64] * SRC2[95:64]
DEST[VLMAX-1:128] ← 0

Intel C/C++ Compiler Intrinsic Equivalent

PMULDQ __m128i _mm_mul_epi32(__m128i a, __m128i b);

Flags Affected

None.

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Exceptions Type 5; additionally

#UD	If VEX.L = 1.
	If VEX.vvvv != 1111B.

...

PMULHRSW — Packed Multiply High with Round and Scale

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
0F 38 0B /r[1] PMULHRSW mm1, mm2/m64	A	V/V	SSSE3	Multiply 16-bit signed words, scale and round signed doublewords, pack high 16 bits to MM1.
66 0F 38 0B /r PMULHRSW xmm1, xmm2/m128	A	V/V	SSSE3	Multiply 16-bit signed words, scale and round signed doublewords, pack high 16 bits to XMM1.
VEX.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.

NOTES:

1. See note in Section 2.4, "Instruction Exception Specification".

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
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 destination operand (first operand) with the corresponding signed 16-bit integer of the source operand (second operand), producing intermediate, signed 32-bit integers. Each intermediate 32-bit integer is truncated to the 18 most significant bits. Rounding is always performed by adding 1 to the least significant bit of the 18-bit intermediate result. The final result is obtained by selecting the 16 bits immediately to the right of the most significant bit of each 18-bit intermediate result and packed to the destination operand. Both operands can be MMX register or XMM registers.

When the source operand is a 128-bit memory operand, the operand must be aligned on a 16-byte boundary or a general-protection exception (#GP) will be generated.

In 64-bit mode, use the REX prefix to access additional registers.

128-bit Legacy SSE version: Bits (VLMAX-1:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: Bits (VLMAX-1:128) of the destination YMM register are zeroed. VEX.L must be 0, otherwise the instruction will #UD.

Operation

PMULHRSW (with 64-bit operands)

```
temp0[31:0] = INT32 ((DEST[15:0] * SRC[15:0]) >>14) + 1;
temp1[31:0] = INT32 ((DEST[31:16] * SRC[31:16]) >>14) + 1;
temp2[31:0] = INT32 ((DEST[47:32] * SRC[47:32]) >> 14) + 1;
```

```
temp3[31:0] = INT32 ((DEST[63:48] * SRc[63:48]) >> 14) + 1;
DEST[15:0] = temp0[16:1];
DEST[31:16] = temp1[16:1];
DEST[47:32] = temp2[16:1];
DEST[63:48] = temp3[16:1];
```

PMULHRSW (with 128-bit operand)

```
temp0[31:0] = INT32 ((DEST[15:0] * SRC[15:0]) >>14) + 1;
temp1[31:0] = INT32 ((DEST[31:16] * SRC[31:16]) >>14) + 1;
temp2[31:0] = INT32 ((DEST[47:32] * SRC[47:32]) >>14) + 1;
temp3[31:0] = INT32 ((DEST[63:48] * SRC[63:48]) >>14) + 1;
temp4[31:0] = INT32 ((DEST[79:64] * SRC[79:64]) >>14) + 1;
temp5[31:0] = INT32 ((DEST[95:80] * SRC[95:80]) >>14) + 1;
temp6[31:0] = INT32 ((DEST[111:96] * SRC[111:96]) >>14) + 1;
temp7[31:0] = INT32 ((DEST[127:112] * SRC[127:112) >>14) + 1;
DEST[15:0] = temp0[16:1];
DEST[31:16] = temp1[16:1];
DEST[47:32] = temp2[16:1];
DEST[63:48] = temp3[16:1];
DEST[79:64] = temp4[16:1];
DEST[95:80] = temp5[16:1];
DEST[111:96] = temp6[16:1];
DEST[127:112] = temp7[16:1];
```

VPMULHRSW (VEX.128 encoded version)

```
temp0[31:0] ← INT32 ((SRC1[15:0] * SRC2[15:0]) >>14) + 1
temp1[31:0] ← INT32 ((SRC1[31:16] * SRC2[31:16]) >>14) + 1
temp2[31:0] ← INT32 ((SRC1[47:32] * SRC2[47:32]) >>14) + 1
temp3[31:0] ← INT32 ((SRC1[63:48] * SRC2[63:48]) >>14) + 1
temp4[31:0] ← INT32 ((SRC1[79:64] * SRC2[79:64]) >>14) + 1
temp5[31:0] ← INT32 ((SRC1[95:80] * SRC2[95:80]) >>14) + 1
temp6[31:0] ← INT32 ((SRC1[111:96] * SRC2[111:96]) >>14) + 1
temp7[31:0] ← INT32 ((SRC1[127:112] * SRC2[127:112) >>14) + 1
DEST[15:0] ← temp0[16:1]
DEST[31:16] ← temp1[16:1]
DEST[47:32] ← temp2[16:1]
DEST[63:48] ← temp3[16:1]
DEST[79:64] ← temp4[16:1]
DEST[95:80] ← temp5[16:1]
DEST[111:96] ← temp6[16:1]
DEST[127:112] ← temp7[16:1]
DEST[VLMAX-1:128] ← 0
```

Intel C/C++ Compiler Intrinsic Equivalents

PMULHRSW __m64 _mm_mulhrs_pi16 (__m64 a, __m64 b)

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.

...

PMULHUW—Multiply Packed Unsigned Integers and Store High Result

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
0F E4 /r[1] PMULHUW *mm1*, *mm2/m64*	A	V/V	SSE	Multiply the packed unsigned word integers in *mm1* register and *mm2/m64*, and store the high 16 bits of the results in *mm1*.
66 0F E4 /r PMULHUW *xmm1*, *xmm2/m128*	A	V/V	SSE2	Multiply the packed unsigned word integers in *xmm1* and *xmm2/m128*, and store the high 16 bits of the results in *xmm1*.
VEX.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.

NOTES:

1. See note in Section 2.4, “Instruction Exception Specification”.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
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 destination operand (first operand) and the source operand (second operand), and stores the high 16 bits of each 32-bit intermediate results in the destination operand. (Figure 4-9 shows this operation when using 64-bit operands.) The source operand can be an MMX technology register or a 64-bit memory location, or it can be an XMM register or a 128-bit memory location. The destination operand can be an MMX technology register or an XMM register.

In 64-bit mode, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

128-bit Legacy SSE version: Bits (VLMAX-1:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: Bits (VLMAX-1:128) of the destination YMM register are zeroed. VEX.L must be 0, otherwise the instruction will #UD.

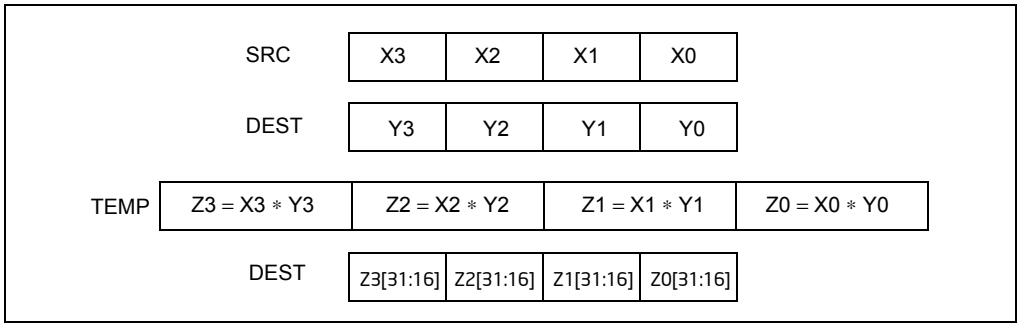


Figure 4-9 PMULHUW and PMULHW Instruction Operation Using 64-bit Operands

Operation

PMULHUW (with 64-bit operands)

```
TEMP0[31:0] ← DEST[15:0] * SRC[15:0]; (* Unsigned multiplication *)
TEMP1[31:0] ← DEST[31:16] * SRC[31:16];
TEMP2[31:0] ← DEST[47:32] * SRC[47:32];
TEMP3[31:0] ← DEST[63:48] * SRC[63:48];
DEST[15:0] ← TEMP0[31:16];
DEST[31:16] ← TEMP1[31:16];
DEST[47:32] ← TEMP2[31:16];
DEST[63:48] ← TEMP3[31:16];
```

PMULHUW (with 128-bit operands)

```
TEMP0[31:0] ← DEST[15:0] * SRC[15:0]; (* Unsigned multiplication *)
TEMP1[31:0] ← DEST[31:16] * SRC[31:16];
TEMP2[31:0] ← DEST[47:32] * SRC[47:32];
TEMP3[31:0] ← DEST[63:48] * SRC[63:48];
TEMP4[31:0] ← DEST[79:64] * SRC[79:64];
TEMP5[31:0] ← DEST[95:80] * SRC[95:80];
TEMP6[31:0] ← DEST[111:96] * SRC[111:96];
TEMP7[31:0] ← DEST[127:112] * SRC[127:112];
DEST[15:0] ← TEMP0[31:16];
DEST[31:16] ← TEMP1[31:16];
DEST[47:32] ← TEMP2[31:16];
DEST[63:48] ← TEMP3[31:16];
DEST[79:64] ← TEMP4[31:16];
DEST[95:80] ← TEMP5[31:16];
DEST[111:96] ← TEMP6[31:16];
DEST[127:112] ← TEMP7[31:16];
```

VPMULHUW (VEX.128 encoded version)

```
TEMP0[31:0] ← SRC1[15:0] * SRC2[15:0]
TEMP1[31:0] ← SRC1[31:16] * SRC2[31:16]
TEMP2[31:0] ← SRC1[47:32] * SRC2[47:32]
TEMP3[31:0] ← SRC1[63:48] * SRC2[63:48]
```

```
TEMP4[31:0] ← SRC1[79:64] * SRC2[79:64]
TEMP5[31:0] ← SRC1[95:80] * SRC2[95:80]
TEMP6[31:0] ← SRC1[111:96] * SRC2[111:96]
TEMP7[31:0] ← SRC1[127:112] * SRC2[127:112]
DEST[15:0] ← TEMP0[31:16]
DEST[31:16] ← TEMP1[31:16]
DEST[47:32] ← TEMP2[31:16]
DEST[63:48] ← TEMP3[31:16]
DEST[79:64] ← TEMP4[31:16]
DEST[95:80] ← TEMP5[31:16]
DEST[111:96] ← TEMP6[31:16]
DEST[127:112] ← TEMP7[31:16]
DEST[VLMAX-1:128] ← 0
```

Intel C/C++ Compiler Intrinsic Equivalent

PMULHUW __m64 _mm_mulhi_pu16(__m64 a, __m64 b)

PMULHUW __m128i _mm_mulhi_epu16 (__m128i a, __m128i b)

Flags Affected

None.

Numeric Exceptions

None.

Other Exceptions

See Exceptions Type 4; additionally

#UD	If VEX.L = 1.

...

PMULHW—Multiply Packed Signed Integers and Store High Result

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
0F E5 /r[1] PMULHW *mm, mm/m64*	A	V/V	MMX	Multiply the packed signed word integers in *mm1* register and *mm2/m64*, and store the high 16 bits of the results in *mm1*.
66 0F E5 /r PMULHW *xmm1, xmm2/m128*	A	V/V	SSE2	Multiply the packed signed word integers in *xmm1* and *xmm2/m128*, and store the high 16 bits of the results in *xmm1*.
VEX.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.

NOTES:

1. See note in Section 2.4, "Instruction Exception Specification".

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
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 destination operand (first operand) and the source operand (second operand), and stores the high 16 bits of each intermediate 32-bit result in the destination operand. (Figure 4-9 shows this operation when using 64-bit operands.) The source operand can be an MMX technology register or a 64-bit memory location, or it can be an XMM register or a 128-bit memory location. The destination operand can be an MMX technology register or an XMM register.

n 64-bit mode, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

128-bit Legacy SSE version: Bits (VLMAX-1:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: Bits (VLMAX-1:128) of the destination YMM register are zeroed. VEX.L must be 0, otherwise the instruction will #UD.

Operation

PMULHW (with 64-bit operands)

```
TEMP0[31:0] ← DEST[15:0] * SRC[15:0]; (* Signed multiplication *)
TEMP1[31:0] ← DEST[31:16] * SRC[31:16];
```

```
    TEMP2[31:0] ←    DEST[47:32] * SRC[47:32];
    TEMP3[31:0] ←    DEST[63:48] * SRC[63:48];
    DEST[15:0] ←     TEMP0[31:16];
    DEST[31:16] ←    TEMP1[31:16];
    DEST[47:32] ←    TEMP2[31:16];
    DEST[63:48] ←    TEMP3[31:16];
```

PMULHW (with 128-bit operands)

```
    TEMP0[31:0] ←    DEST[15:0] * SRC[15:0]; (* Signed multiplication *)
    TEMP1[31:0] ←    DEST[31:16] * SRC[31:16];
    TEMP2[31:0] ←    DEST[47:32] * SRC[47:32];
    TEMP3[31:0] ←    DEST[63:48] * SRC[63:48];
    TEMP4[31:0] ←    DEST[79:64] * SRC[79:64];
    TEMP5[31:0] ←    DEST[95:80] * SRC[95:80];
    TEMP6[31:0] ←    DEST[111:96] * SRC[111:96];
    TEMP7[31:0] ←    DEST[127:112] * SRC[127:112];
    DEST[15:0] ←     TEMP0[31:16];
    DEST[31:16] ←    TEMP1[31:16];
    DEST[47:32] ←    TEMP2[31:16];
    DEST[63:48] ←    TEMP3[31:16];
    DEST[79:64] ←    TEMP4[31:16];
    DEST[95:80] ←    TEMP5[31:16];
    DEST[111:96] ←   TEMP6[31:16];
    DEST[127:112] ←  TEMP7[31:16];
```

VPMULHW (VEX.128 encoded version)

```
TEMP0[31:0] ← SRC1[15:0] * SRC2[15:0] (*Signed Multiplication*)
TEMP1[31:0] ← SRC1[31:16] * SRC2[31:16]
TEMP2[31:0] ← SRC1[47:32] * SRC2[47:32]
TEMP3[31:0] ← SRC1[63:48] * SRC2[63:48]
TEMP4[31:0] ← SRC1[79:64] * SRC2[79:64]
TEMP5[31:0] ← SRC1[95:80] * SRC2[95:80]
TEMP6[31:0] ← SRC1[111:96] * SRC2[111:96]
TEMP7[31:0] ← SRC1[127:112] * SRC2[127:112]
DEST[15:0] ← TEMP0[31:16]
DEST[31:16] ← TEMP1[31:16]
DEST[47:32] ← TEMP2[31:16]
DEST[63:48] ← TEMP3[31:16]
DEST[79:64] ← TEMP4[31:16]
DEST[95:80] ← TEMP5[31:16]
DEST[111:96] ← TEMP6[31:16]
DEST[127:112] ← TEMP7[31:16]
DEST[VLMAX-1:128] ← 0
```

Intel C/C++ Compiler Intrinsic Equivalent

PMULHW __m64 _mm_mulhi_pi16 (__m64 m1, __m64 m2)

PMULHW __m128i _mm_mulhi_epi16 (__m128i a, __m128i b)

Flags Affected

None.

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Exceptions Type 4; additionally

#UD	If VEX.L = 1.

...

PMULLD — Multiply Packed Signed Dword Integers and Store Low Result

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 38 40 /r PMULLD *xmm1, xmm2/m128*	A	V/V	SSE4_1	Multiply the packed dword signed integers in *xmm1* and *xmm2/m128* and store the low 32 bits of each product in *xmm1*.
VEX.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

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
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 four signed multiplications from four pairs of signed dword integers and stores the lower 32 bits of the four 64-bit products in the destination operand (first operand). Each dword element in the destination operand is multiplied with the corresponding dword element of the source operand (second operand) to obtain a 64-bit intermediate product.

128-bit Legacy SSE version: Bits (VLMAX-1:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: Bits (VLMAX-1:128) of the destination YMM register are zeroed. VEX.L must be 0, otherwise the instruction will #UD.

Operation

```
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];
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[VLMAX-1:128] ← 0

Intel C/C++ Compiler Intrinsic Equivalent

PMULLUD __m128i _mm_mullo_epi32(__m128i a, __m128i b);

Flags Affected

None.

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Exceptions Type 4; additionally

#UD	If VEX.L = 1.

...

PMULLW—Multiply Packed Signed Integers and Store Low Result

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
0F D5 /r[1] PMULLW *mm, mm/m64*	A	V/V	MMX	Multiply the packed signed word integers in *mm1* register and *mm2/m64*, and store the low 16 bits of the results in *mm1*.
66 0F D5 /r PMULLW *xmm1, xmm2/m128*	A	V/V	SSE2	Multiply the packed signed word integers in *xmm1* and *xmm2/m128*, and store the low 16 bits of the results in *xmm1*.
VEX.NDS.128.66.0F.WIG D5 /r VPMULLW xmm1, xmm2, xmm3/m128	B	V/V	AVX	Multiply the packed dword signed integers in xmm2 and xmm3/m128 and store the low 32 bits of each product in xmm1.

NOTES:

1. See note in Section 2.4, “Instruction Exception Specification”.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
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 destination operand (first operand) and the source operand (second operand), and stores the low 16 bits of each intermediate 32-bit result in the destination operand. (Figure 4-9 shows this operation when using 64-bit operands.) The source operand can be an MMX technology register or a 64-bit memory location, or it can be an XMM register or a 128-bit memory location. The destination operand can be an MMX technology register or an XMM register.

In 64-bit mode, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

128-bit Legacy SSE version: Bits (VLMAX-1:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: Bits (VLMAX-1:128) of the destination YMM register are zeroed. VEX.L must be 0, otherwise the instruction will #UD.

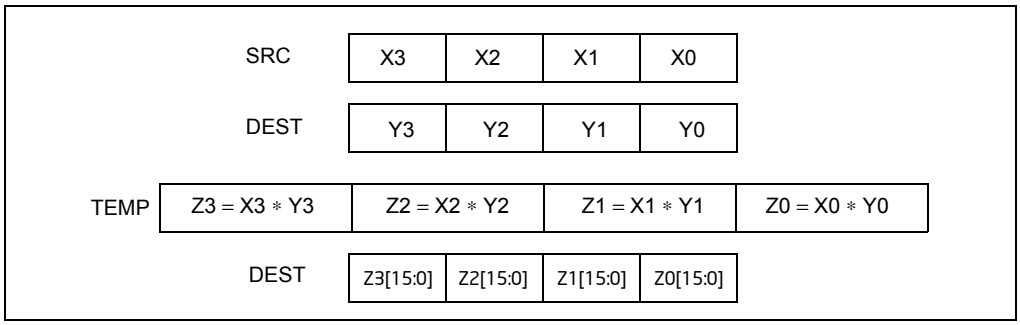


Figure 4-10 PMULLU Instruction Operation Using 64-bit Operands

Operation

PMULLW (with 64-bit operands)

```
TEMP0[31:0] ← DEST[15:0] * SRC[15:0]; (* Signed multiplication *)
TEMP1[31:0] ← DEST[31:16] * SRC[31:16];
TEMP2[31:0] ← DEST[47:32] * SRC[47:32];
TEMP3[31:0] ← DEST[63:48] * SRC[63:48];
DEST[15:0] ← TEMP0[15:0];
DEST[31:16] ← TEMP1[15:0];
DEST[47:32] ← TEMP2[15:0];
DEST[63:48] ← TEMP3[15:0];
```

PMULLW (with 128-bit operands)

```
TEMP0[31:0] ← DEST[15:0] * SRC[15:0]; (* Signed multiplication *)
TEMP1[31:0] ← DEST[31:16] * SRC[31:16];
TEMP2[31:0] ← DEST[47:32] * SRC[47:32];
TEMP3[31:0] ← DEST[63:48] * SRC[63:48];
TEMP4[31:0] ← DEST[79:64] * SRC[79:64];
TEMP5[31:0] ← DEST[95:80] * SRC[95:80];
TEMP6[31:0] ← DEST[111:96] * SRC[111:96];
TEMP7[31:0] ← DEST[127:112] * SRC[127:112];
DEST[15:0] ← TEMP0[15:0];
DEST[31:16] ← TEMP1[15:0];
DEST[47:32] ← TEMP2[15:0];
DEST[63:48] ← TEMP3[15:0];
DEST[79:64] ← TEMP4[15:0];
DEST[95:80] ← TEMP5[15:0];
DEST[111:96] ← TEMP6[15:0];
DEST[127:112] ← TEMP7[15:0];
```

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[VLMAX-1:128] ← 0
```

Intel C/C++ Compiler Intrinsic Equivalent

PMULLW __m64 _mm_mullo_pi16(__m64 m1, __m64 m2)

PMULLW __m128i _mm_mullo_epi16 (__m128i a, __m128i b)

Flags Affected

None.

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
0F F4 /r[1] PMULUDQ *mm1*, *mm2/m64*	A	V/V	SSE2	Multiply unsigned doubleword integer in *mm1* by unsigned doubleword integer in *mm2/m64*, and store the quadword result in *mm1*.
66 0F F4 /r PMULUDQ *xmm1*, *xmm2/m128*	A	V/V	SSE2	Multiply packed unsigned doubleword integers in *xmm1* by packed unsigned doubleword integers in *xmm2/m128*, and store the quadword results in *xmm1*.
VEX.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.

NOTES:

1. See note in Section 2.4, “Instruction Exception Specification”.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
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 operand (destination operand) by the second operand (source operand) and stores the result in the destination operand. The source operand can be an unsigned doubleword integer stored in the low doubleword of an MMX technology register or a 64-bit memory location, or it can be two packed unsigned doubleword integers stored in the first (low) and third doublewords of an XMM register or an 128-bit memory location. The destination operand can be an unsigned doubleword integer stored in the low doubleword an MMX technology register or two packed doubleword integers stored in the first and third doublewords of an XMM register. The result is an unsigned quadword integer stored in the destination an MMX technology register or 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 64-bit memory operands, 64 bits are fetched from memory, but only the low doubleword is used in the computation; for 128-bit memory operands, 128 bits are fetched from memory, but only the first and third doublewords are used in the computation.

In 64-bit mode, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

128-bit Legacy SSE version: Bits (VLMAX-1:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: Bits (VLMAX-1:128) of the destination YMM register are zeroed. VEX.L must be 0, otherwise the instruction will #UD.

Operation

PMULUDQ (with 64-Bit operands)

```
DEST[63:0] ← DEST[31:0] * SRC[31:0];
```

PMULUDQ (with 128-Bit operands)

```
DEST[63:0] ← DEST[31:0] * SRC[31:0];
DEST[127:64] ← DEST[95:64] * SRC[95:64];
```

VPMULUDQ (VEX.128 encoded version)

```
DEST[63:0] ← SRC1[31:0] * SRC2[31:0]
DEST[127:64] ← SRC1[95:64] * SRC2[95:64]
DEST[VLMAX-1:128] ← 0
```

Intel C/C++ Compiler Intrinsic Equivalent

PMULUDQ __m64 _mm_mul_su32 (__m64 a, __m64 b)

PMULUDQ __m128i _mm_mul_epu32 (__m128i a, __m128i b)

Flags Affected

None.

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
0F EB /r[1]	POR *mm, mm/m64*	A	V/V	MMX	Bitwise OR of *mm/m64* and *mm*.
66 0F EB /r	POR *xmm1, xmm2/m128*	A	V/V	SSE2	Bitwise OR of *xmm2/m128* and *xmm1*.
VEX.NDS.128.66.0F.WIG EB /r	VPOR xmm1, xmm2, xmm3/m128	B	V/V	AVX	Bitwise OR of xmm2/m128 and xmm3.

NOTES:

1. See note in Section 2.4, “Instruction Exception Specification”.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
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 source operand (second operand) and the destination operand (first operand) and stores the result in the destination operand. The source operand can be an MMX technology register or a 64-bit memory location or it can be an XMM register or a 128-bit memory location. The destination operand can be an MMX technology register or an XMM register. Each bit of the result is set to 1 if either or both of the corresponding bits of the first and second operands are 1; otherwise, it is set to 0.

In 64-bit mode, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

128-bit Legacy SSE version: Bits (VLMAX-1:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: Bits (VLMAX-1:128) of the destination YMM register are zeroed. VEX.L must be 0, otherwise the instruction will #UD.

Operation

POR (128-bit Legacy SSE version)

```
DEST ← DEST OR SRC
DEST[VLMAX-1:128] (Unmodified)
```

VPOR (VEX.128 encoded version)

```
DEST ← SRC1 OR SRC2
DEST[VLMAX-1:128] ← 0
```

Intel C/C++ Compiler Intrinsic Equivalent

POR __m64 _mm_or_si64(__m64 m1, __m64 m2)

POR __m128i _mm_or_si128(__m128i m1, __m128i m2)

Flags Affected

None.

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
0F F6 /r[1] PSADBW *mm1, mm2/m64*	A	V/V	SSE	Computes the absolute differences of the packed unsigned byte integers from *mm2 /m64* and *mm1*; differences are then summed to produce an unsigned word integer result.
66 0F F6 /r PSADBW *xmm1, xmm2/m128*	A	V/V	SSE2	Computes the absolute differences of the packed unsigned byte integers from *xmm2 /m128* and *xmm1*; the 8 low differences and 8 high differences are then summed separately to produce two unsigned word integer results.
VEX.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.

NOTES:

1. See note in Section 2.4, "Instruction Exception Specification".

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
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 8 unsigned byte integers from the source operand (second operand) and from the destination operand (first operand). These 8 differences are then summed to produce an unsigned word integer result that is stored in the destination operand. The source operand can be an MMX technology register or a 64-bit memory location or it can be an XMM register or a 128-bit memory location. The destination operand can be an MMX technology register or an XMM register. Figure 4-11 shows the operation of the PSADBW instruction when using 64-bit operands.

When operating on 64-bit operands, the word integer result is stored in the low word of the destination operand, and the remaining bytes in the destination operand are cleared to all 0s.

When operating on 128-bit operands, two packed results are computed. Here, the 8 low-order bytes of the source and destination operands are operated on to produce a word result that is stored in the low word of the destination operand, and the 8 high-order bytes are operated on to produce a word result that is stored in bits 64 through 79 of the destination operand. The remaining bytes of the destination operand are cleared.

In 64-bit mode, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

128-bit Legacy SSE version: Bits (VLMAX-1:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: Bits (VLMAX-1:128) of the destination YMM register are zeroed. VEX.L must be 0, otherwise the instruction will #UD.

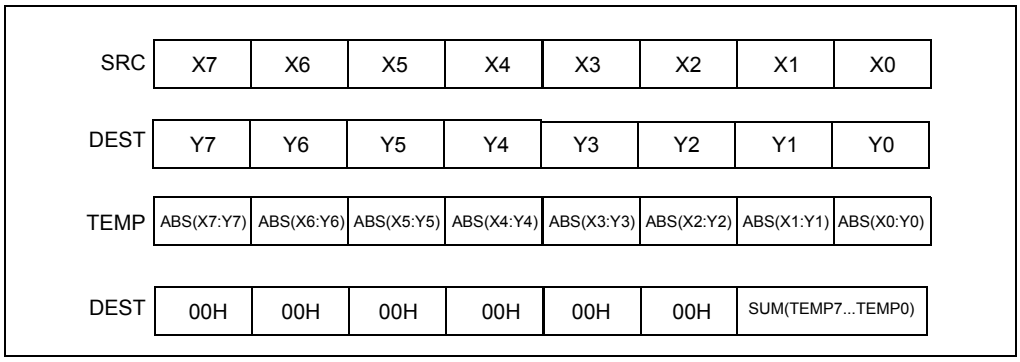


Figure 4-11 PSADBW Instruction Operation Using 64-bit Operands

Operation

PSADBW (when using 64-bit operands)

```
TEMP0 ← ABS(DEST[7:0] – SRC[7:0]);
(* Repeat operation for bytes 2 through 6 *)
TEMP7 ← ABS(DEST[63:56] – SRC[63:56]);
DEST[15:0] ← SUM(TEMP0:TEMP7);
DEST[63:16] ← 000000000000H;
```

PSADBW (when using 128-bit operands)

```
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] ← 000000000000H;

DEST[VLMAX-1:128] (Unmodified)
```

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[VLMAX-1:128] ← 0

Intel C/C++ Compiler Intrinsic Equivalent

PSADBW __m64 _mm_sad_pu8(__m64 a,__m64 b)

PSADBW __m128i _mm_sad_epu8(__m128i a, __m128i b)

Flags Affected

None.

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
0F 38 00 /r[1] PSHUFB mm1, mm2/m64	A	V/V	SSSE3	Shuffle bytes in mm1 according to contents of mm2/m64.
66 0F 38 00 /r PSHUFB xmm1, xmm2/m128	A	V/V	SSSE3	Shuffle bytes in xmm1 according to contents of xmm2/m128.
VEX.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.

NOTES:

1. See note in Section 2.4, “Instruction Exception Specification”.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
A	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
B	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA

Description

PSHUFB performs in-place shuffles of bytes in the destination operand (the first operand) according to the shuffle control mask in the source operand (the second

operand). The instruction permutes the data in the destination operand, leaving the shuffle mask unaffected. If the most significant bit (bit[7]) of each byte of the shuffle control mask is set, then constant zero is written in the result byte. Each byte in the shuffle control mask forms an index to permute the corresponding byte in the destination operand. The value of each index is the least significant 4 bits (128-bit operation) or 3 bits (64-bit operation) of the shuffle control byte. Both operands can be MMX register or XMM registers. When the source operand is a 128-bit memory operand, the operand must be aligned on a 16-byte boundary or a general-protection exception (#GP) will be generated.

In 64-bit mode, use the REX prefix to access additional registers.

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

VEX.128 encoded version: The destination operand is the first operand, the first source operand is the second operand, the second source operand is the third operand. Bits (VLMAX-1:128) of the destination YMM register are zeroed. VEX.L must be 0, otherwise the instruction will #UD.

Operation

PSHUFB (with 64 bit operands)

```
for i = 0 to 7 {
    if (SRC[(i * 8)+7] = 1 ) then
        DEST[(i*8)+7...(i*8)+0] ← 0;
    else
        index[2..0] ← SRC[(i*8)+2 .. (i*8)+0];
        DEST[(i*8)+7...(i*8)+0] ← DEST[(index*8+7)..(index*8+0)];
    endif;
}
```

PSHUFB (with 128 bit operands)

```
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[VLMAX-1:128] ← 0
```

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[VLMAX-1:128] ← 0

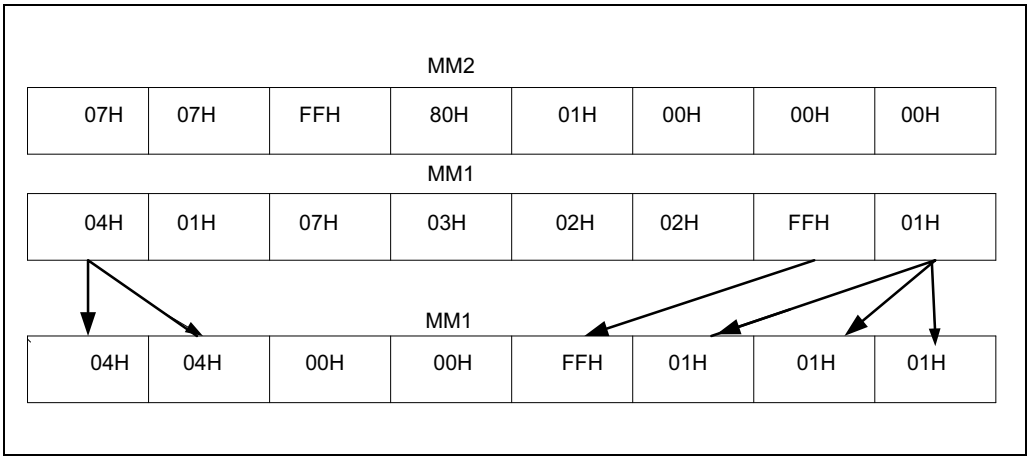


Figure 4-12 PSHUB with 64-Bit Operands

Intel C/C++ Compiler Intrinsic Equivalent

PSHUFB __m64 _mm_shuffle_pi8 (__m64 a, __m64 b)

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

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
A	ModRM:reg (w)	ModRM:r/m (r)	imm8	NA

Description

Copies doublewords from source operand (second operand) and inserts them in the destination operand (first operand) at the locations selected with the order operand (third operand). Figure 4-13 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 4-13) determines which doubleword from the source operand will be copied to doubleword 0 of the destination operand.

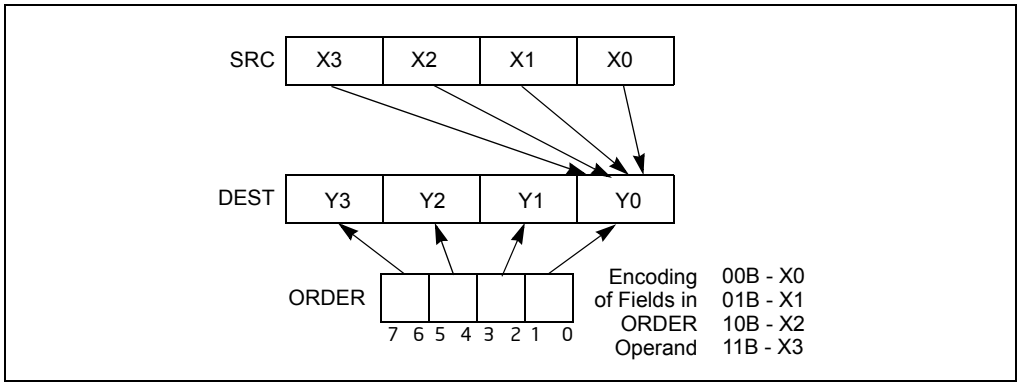


Figure 4-13 PSHUFD Instruction Operation

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.

In 64-bit mode, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

128-bit Legacy SSE version: Bits (VLMAX-1:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: Bits (VLMAX-1:1288) 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

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[VLMAX-1:128] (Unmodified)
```

VPSHUFD (VEX.128 encoded version)
DEST[31:0] ← (SRC >> (ORDER[1:0] * 32))[31:0];
DEST[63:32] ← (SRC >> (ORDER[3:2] * 32))[31:0];
DEST[95:64] ← (SRC >> (ORDER[5:4] * 32))[31:0];
DEST[127:96] ← (SRC >> (ORDER[7:6] * 32))[31:0];
DEST[VLMAX-1:128] ← 0

Intel C/C++ Compiler Intrinsic Equivalent

PSHUFD __m128i _mm_shuffle_epi32(__m128i a, int n)

Flags Affected

None.

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

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
A	ModRM:reg (w)	ModRM:r/m (r)	imm8	NA

Description

Copies words from the high quadword of the source operand (second operand) and inserts them in the high quadword of the destination operand (first operand) at word locations selected with the order operand (third operand). This operation is similar to the operation used by the PSHUFD instruction, which is illustrated in Figure 4-13. For the

PSHUFHW instruction, each 2-bit field in the order operand selects the contents of one word location in the high quadword of the destination operand. The binary encodings of the order operand fields select words (0, 1, 2 or 3, 4) from the high quadword of the source operand to be copied to the destination operand. The low quadword of the source operand is copied to the low quadword of the destination operand.

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 word in the high quadword of the source operand to be copied to more than one word location in the high quadword of the destination operand.

In 64-bit mode, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

128-bit Legacy SSE version: Bits (VLMAX-1:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: Bits (VLMAX-1:128) of the destination YMM register are zeroed. VEX.vvvv is reserved and must be 1111b, VEX.L must be 0, otherwise the instruction will #UD.

Operation

PSHUFHW (128-bit Legacy SSE version)

```
DEST[63:0] ← SRC[63:0]
DEST[79:64] ← (SRC >> (imm[1:0] *16))[79:64]
DEST[95:80] ← (SRC >> (imm[3:2] * 16))[79:64]
DEST[111:96] ← (SRC >> (imm[5:4] * 16))[79:64]
DEST[127:112] ← (SRC >> (imm[7:6] * 16))[79:64]
DEST[VLMAX-1:128] (Unmodified)
```

VPSHUFHW (VEX.128 encoded version)

```
DEST[63:0] ← SRC1[63:0]
DEST[79:64] ← (SRC1 >> (imm[1:0] *16))[79:64]
DEST[95:80] ← (SRC1 >> (imm[3:2] * 16))[79:64]
DEST[111:96] ← (SRC1 >> (imm[5:4] * 16))[79:64]
DEST[127:112] ← (SRC1 >> (imm[7:6] * 16))[79:64]
DEST[VLMAX-1:128] ← 0
```

Intel C/C++ Compiler Intrinsic Equivalent

PSHUFHW __m128i _mm_shufflehi_epi16(__m128i a, int n)

Flags Affected

None.

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

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
A	ModRM:reg (w)	ModRM:r/m (r)	imm8	NA

Description

Copies words from the low quadword of the source operand (second operand) and inserts them in the low quadword of the destination operand (first operand) at word locations selected with the order operand (third operand). This operation is similar to the operation used by the PSHUFD instruction, which is illustrated in Figure 4-13. For the PSHUFLW instruction, each 2-bit field in the order operand selects the contents of one word location in the low quadword of the destination operand. The binary encodings of the order 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 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 word in the low quadword of the source operand to be copied to more than one word location in the low quadword of the destination operand.

In 64-bit mode, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

128-bit Legacy SSE version: Bits (VLMAX-1:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: Bits (VLMAX-1:128) 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

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[VLMAX-1:128] (Unmodified)

VPSHUFLW (VEX.128 encoded version)
DEST[15:0] ← (SRC1 >> (imm[1:0] *16))[15:0]
DEST[31:16] ← (SRC1 >> (imm[3:2] * 16))[15:0]
DEST[47:32] ← (SRC1 >> (imm[5:4] * 16))[15:0]
DEST[63:48] ← (SRC1 >> (imm[7:6] * 16))[15:0]
DEST[127:64] ← SRC[127:64]
DEST[VLMAX-1:128] ← 0

Intel C/C++ Compiler Intrinsic Equivalent

PSHUFLW __m128i _mm_shufflelo_epi16(__m128i a, int n)

Flags Affected

None.

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
0F 38 08 /r[1]	PSIGNB mm1, mm2/m64	A	V/V	SSSE3	Negate/zero/preserve packed byte integers in mm1 depending on the corresponding sign in mm2/m64
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.
0F 38 09 /r[1]	PSIGNW mm1, mm2/m64	A	V/V	SSSE3	Negate/zero/preserve packed word integers in mm1 depending on the corresponding sign in mm2/m128.

Opcode Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
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.
0F 38 0A /r[1] PSIGND mm1, mm2/m64	A	V/V	SSSE3	Negate/zero/preserve packed doubleword integers in mm1 depending on the corresponding sign in mm2/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.

NOTES:

1. See note in Section 2.4, “Instruction Exception Specification”.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
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 destination operand (the first operand) if the signed integer value of the corresponding data element in the source operand (the second operand) is less than zero. If the signed integer value of a data element in the source operand is positive, the corresponding data element in the destination operand is unchanged. If a data element in the source operand is zero, the corresponding data element in the destination operand is set to zero.

PSIGNB operates on signed bytes. PSIGNW operates on 16-bit signed words. PSIGND operates on signed 32-bit integers. Both operands can be MMX register or XMM registers. When the source operand is a 128bit memory operand, the operand must be aligned on a 16-byte boundary or a general-protection exception (#GP) will be generated.

In 64-bit mode, use the REX prefix to access additional registers.

128-bit Legacy SSE version: Bits (VLMAX-1:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: Bits (VLMAX-1:128) of the destination YMM register are zeroed. VEX.L must be 0, otherwise instructions will #UD.

Operation

PSIGNB (with 64 bit operands)

```
IF (SRC[7:0] < 0 )
    DEST[7:0] ← Neg(DEST[7:0])
ELSEIF (SRC[7:0] = 0 )
    DEST[7:0] ← 0
ELSEIF (SRC[7:0] > 0 )
    DEST[7:0] ← DEST[7:0]
Repeat operation for 2nd through 7th bytes

IF (SRC[63:56] < 0 )
    DEST[63:56] ← Neg(DEST[63:56])
ELSEIF (SRC[63:56] = 0 )
    DEST[63:56] ← 0
ELSEIF (SRC[63:56] > 0 )
    DEST[63:56] ← DEST[63:56]
```

PSIGNB (with 128 bit operands)

```
IF (SRC[7:0] < 0 )
    DEST[7:0] ← Neg(DEST[7:0])
ELSEIF (SRC[7:0] = 0 )
    DEST[7:0] ← 0
ELSEIF (SRC[7:0] > 0 )
    DEST[7:0] ← DEST[7:0]
Repeat operation for 2nd through 15th bytes
IF (SRC[127:120] < 0 )
    DEST[127:120] ← Neg(DEST[127:120])
ELSEIF (SRC[127:120] = 0 )
    DEST[127:120] ← 0
ELSEIF (SRC[127:120] > 0 )
    DEST[127:120] ← DEST[127:120]
```

PSIGNW (with 64 bit operands)

```
IF (SRC[15:0] < 0 )
    DEST[15:0] ← Neg(DEST[15:0])
ELSEIF (SRC[15:0] = 0 )
    DEST[15:0] ← 0
```

```
    ELSEIF (SRC[15:0] > 0 )
        DEST[15:0] ← DEST[15:0]
Repeat operation for 2nd through 3rd words
    IF (SRC[63:48] < 0 )
        DEST[63:48] ← Neg(DEST[63:48])
    ELSEIF (SRC[63:48] = 0 )
        DEST[63:48] ← 0
    ELSEIF (SRC[63:48] > 0 )
        DEST[63:48] ← DEST[63:48]
```

PSIGNW (with 128 bit operands)

```
    IF (SRC[15:0] < 0 )
        DEST[15:0] ← Neg(DEST[15:0])
    ELSEIF (SRC[15:0] = 0 )
        DEST[15:0] ← 0
    ELSEIF (SRC[15:0] > 0 )
        DEST[15:0] ← DEST[15:0]
    Repeat operation for 2nd through 7th words
    IF (SRC[127:112] < 0 )
        DEST[127:112] ← Neg(DEST[127:112])
    ELSEIF (SRC[127:112] = 0 )
        DEST[127:112] ← 0
    ELSEIF (SRC[127:112] > 0 )
        DEST[127:112] ← DEST[127:112]
```

PSIGND (with 64 bit operands)

```
    IF (SRC[31:0] < 0 )
        DEST[31:0] ← Neg(DEST[31:0])
    ELSEIF (SRC[31:0] = 0 )
        DEST[31:0] ← 0
    ELSEIF (SRC[31:0] > 0 )
        DEST[31:0] ← DEST[31:0]
    IF (SRC[63:32] < 0 )
        DEST[63:32] ← Neg(DEST[63:32])
    ELSEIF (SRC[63:32] = 0 )
        DEST[63:32] ← 0
    ELSEIF (SRC[63:32] > 0 )
        DEST[63:32] ← DEST[63:32]
```

PSIGND (with 128 bit operands)

```
    IF (SRC[31:0] < 0 )
        DEST[31:0] ← Neg(DEST[31:0])
    ELSEIF (SRC[31:0] = 0 )
        DEST[31:0] ← 0
    ELSEIF (SRC[31:0] > 0 )
        DEST[31:0] ← DEST[31:0]
    Repeat operation for 2nd through 3rd double words
    IF (SRC[127:96] < 0 )
        DEST[127:96] ← Neg(DEST[127:96])
```

```
ELSEIF (SRC[127:96] = 0 )
    DEST[127:96] ← 0
ELSEIF (SRC[127:96] > 0 )
    DEST[127:96] ← DEST[127:96]
```

VPSIGNB (VEX.128 encoded version)
DEST[127:0] ←BYTE_SIGN(SRC1, SRC2)
DEST[VLMAX-1:128] ← 0

VPSIGNW (VEX.128 encoded version)
DEST[127:0] ←WORD_SIGN(SRC1, SRC2)
DEST[VLMAX-1:128] ← 0

VPSIGND (VEX.128 encoded version)
DEST[127:0] ←DWORD_SIGN(SRC1, SRC2)
DEST[VLMAX-1:128] ← 0

Intel C/C++ Compiler Intrinsic Equivalent

PSIGNB __m64 _mm_sign_pi8 (__m64 a, __m64 b)

PSIGNB __m128i _mm_sign_epi8 (__m128i a, __m128i b)

PSIGNW __m64 _mm_sign_pi16 (__m64 a, __m64 b)

PSIGNW __m128i _mm_sign_epi16 (__m128i a, __m128i b)

PSIGND __m64 _mm_sign_pi32 (__m64 a, __m64 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—Shift Double Quadword Left Logical

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 73 /7 ib PSLLDQ *xmm1*, *imm8*	A	V/V	SSE2	Shift *xmm1* left by *imm8* bytes while shifting in 0s.
VEX.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

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
A	ModRM:r/m (r, w)	imm8	NA	NA
B	VEX.vvvv (w)	ModRM:r/m (r)	NA	NA

Description

Shifts the destination operand (first operand) to the left by the number of bytes specified in the count operand (second operand). The empty low-order bytes are cleared (set to all 0s). If the value specified by the count operand is greater than 15, the destination operand is set to all 0s. The destination operand is an XMM register. The count operand is an 8-bit immediate.

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

VEX.128 encoded version: Bits (VLMAX-1:128) 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

PSLLDQ(128-bit Legacy SSE version)

```
TEMP ← COUNT
IF (TEMP > 15) THEN TEMP ← 16; FI
DEST ← DEST << (TEMP * 8)
DEST[VLMAX-1:128] (Unmodified)
```

VPSLLDQ (VEX.128 encoded version)

```
TEMP ← COUNT
IF (TEMP > 15) THEN TEMP ← 16; FI
DEST ← SRC << (TEMP * 8)
DEST[VLMAX-1:128] ← 0
```

Intel C/C++ Compiler Intrinsic Equivalent

PSLLDQ __m128i _mm_slli_si128 (__m128i a, int imm)

Flags Affected

None.

Numeric Exceptions

None.

Other Exceptions

See Exceptions Type 7; additionally

#UD	If VEX.L = 1.

...

PSLLW/PSLLD/PSLLQ—Shift Packed Data Left Logical

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
0F F1 /r[1] PSLLW *mm, mm/m64*	A	V/V	MMX	Shift words in *mm* left *mm/m64* while shifting in 0s.
66 0F F1 /r PSLLW *xmm1, xmm2/m128*	A	V/V	SSE2	Shift words in *xmm1* left by *xmm2/m128* while shifting in 0s.
0F 71 /6 ib PSLLW *xmm1, imm8*	B	V/V	MMX	Shift words in *mm* left by *imm8* while shifting in 0s.
66 0F 71 /6 ib PSLLW *xmm1, imm8*	B	V/V	SSE2	Shift words in *xmm1* left by *imm8* while shifting in 0s.
0F F2 /r[1] PSLLD *mm, mm/m64*	A	V/V	MMX	Shift doublewords in *mm* left by *mm/m64* while shifting in 0s.
66 0F F2 /r PSLLD *xmm1, xmm2/m128*	A	V/V	SSE2	Shift doublewords in *xmm1* left by *xmm2/m128* while shifting in 0s.
0F 72 /6 ib[1] PSLLD *mm, imm8*	B	V/V	MMX	Shift doublewords in *mm* left by *imm8* while shifting in 0s.
66 0F 72 /6 ib PSLLD *xmm1, imm8*	B	V/V	SSE2	Shift doublewords in *xmm1* left by *imm8* while shifting in 0s.
0F F3 /r[1] PSLLQ *mm, mm/m64*	A	V/V	MMX	Shift quadword in *mm* left by *mm/m64* while shifting in 0s.
66 0F F3 /r PSLLQ *xmm1, xmm2/m128*	A	V/V	SSE2	Shift quadwords in *xmm1* left by *xmm2/m128* while shifting in 0s.
0F 73 /6 ib[1] PSLLQ *mm, imm8*	B	V/V	MMX	Shift quadword in *mm* left by *imm8* while shifting in 0s.
66 0F 73 /6 ib PSLLQ *xmm1, imm8*	B	V/V	SSE2	Shift quadwords in *xmm1* left by *imm8* while shifting in 0s.
VEX.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.

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
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.
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.

NOTES:

1. See note in Section 2.4, “Instruction Exception Specification”.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
A	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
B	ModRM:r/m (r, w)	imm8	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 destination operand (first operand) to the left by the number of bits specified in the count operand (second operand). As the bits in the data elements are shifted left, the empty low-order bits are cleared (set to 0). If the value specified by the count operand is greater than 15 (for words), 31 (for doublewords), or 63 (for a quadword), then the destination operand is set to all 0s. Figure 4-14 gives an example of shifting words in a 64-bit operand.

The destination operand may be an MMX technology register or an XMM register; the count operand can be either an MMX technology register or an 64-bit memory location, 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.

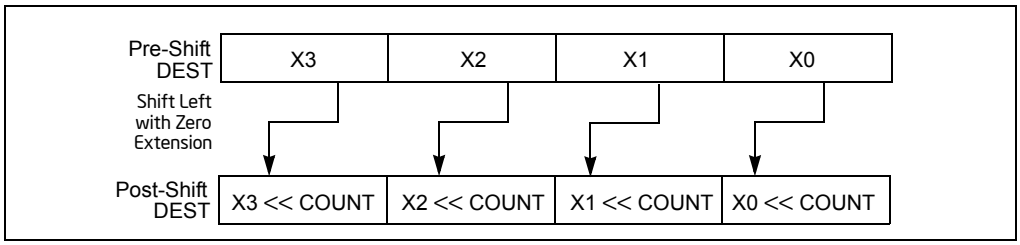


Figure 4-14 PSLLW, PSLLD, and PSLLQ Instruction Operation Using 64-bit Operand

The PSLLW instruction shifts each of the words in the destination operand to the left by the number of bits specified in the count operand; the PSLLD instruction shifts each of the doublewords in the destination operand; and the PSLLQ instruction shifts the quad-word (or quadwords) in the destination operand.

In 64-bit mode, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

128-bit Legacy SSE version: Bits (VLMAX-1:128) 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 (VLMAX-1:128) 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

PSLLW (with 64-bit operand)

```
IF (COUNT > 15)
THEN
    DEST[64:0] ← 0000000000000000H;
ELSE
    DEST[15:0] ← ZeroExtend(DEST[15:0] << COUNT);
    (* Repeat shift operation for 2nd and 3rd words *)
    DEST[63:48] ← ZeroExtend(DEST[63:48] << COUNT);
FI;
```

PSLLD (with 64-bit operand)

```
IF (COUNT > 31)
THEN
    DEST[64:0] ← 0000000000000000H;
ELSE
    DEST[31:0] ← ZeroExtend(DEST[31:0] << COUNT);
    DEST[63:32] ← ZeroExtend(DEST[63:32] << COUNT);
FI;
```

PSLLQ (with 64-bit operand)

```
IF (COUNT > 63)
THEN
    DEST[64:0] ← 0000000000000000H;
```

```
    ELSE
        DEST ← ZeroExtend(DEST << COUNT);
    FI;
```

PSLLW (with 128-bit operand)

```
    COUNT ← COUNT_SOURCE[63:0];
    IF (COUNT > 15)
    THEN
        DEST[128:0] ← 00000000000000000000000000000000H;
    ELSE
        DEST[15:0]  ← ZeroExtend(DEST[15:0] << COUNT);
        (* Repeat shift operation for 2nd through 7th words *)
        DEST[127:112] ← ZeroExtend(DEST[127:112] << COUNT);
    FI;
```

PSLLD (with 128-bit operand)

```
    COUNT ← COUNT_SOURCE[63:0];
    IF (COUNT > 31)
    THEN
        DEST[128:0] ← 00000000000000000000000000000000H;
    ELSE
        DEST[31:0]  ← ZeroExtend(DEST[31:0] << COUNT);
        (* Repeat shift operation for 2nd and 3rd doublewords *)
        DEST[127:96] ← ZeroExtend(DEST[127:96] << COUNT);
    FI;
```

PSLLQ (with 128-bit operand)

```
    COUNT ← COUNT_SOURCE[63:0];
    IF (COUNT > 63)
    THEN
        DEST[128:0] ← 00000000000000000000000000000000H;
    ELSE
        DEST[63:0]  ← ZeroExtend(DEST[63:0] << COUNT);
        DEST[127:64] ← ZeroExtend(DEST[127:64] << COUNT);
    FI;
```

PSLLW (xmm, xmm, xmm/m128)

```
DEST[127:0] ← LOGICAL_LEFT_SHIFT_WORDS(DEST, SRC)
DEST[VLMAX-1:128] (Unmodified)
```

PSLLW (xmm, imm8)

```
DEST[127:0] ← LOGICAL_LEFT_SHIFT_WORDS(DEST, imm8)
DEST[VLMAX-1:128] (Unmodified)
```

VPSLLD (xmm, xmm, xmm/m128)

```
DEST[127:0] ← LOGICAL_LEFT_SHIFT_DWORDS(SRC1, SRC2)
DEST[VLMAX-1:128] ← 0
```

VPSLLD (xmm, imm8)

```
DEST[127:0] ← LOGICAL_LEFT_SHIFT_DWORDS(SRC1, imm8)
DEST[VLMAX-1:128] ← 0
```

PSLLD (xmm, xmm, xmm/m128)
DEST[127:0] ← LOGICAL_LEFT_SHIFT_DWORDS(DEST, SRC)
DEST[VLMAX-1:128] (Unmodified)

PSLLD (xmm, imm8)
DEST[127:0] ← LOGICAL_LEFT_SHIFT_DWORDS(DEST, imm8)
DEST[VLMAX-1:128] (Unmodified)

VPSLLQ (xmm, xmm, xmm/m128)
DEST[127:0] ← LOGICAL_LEFT_SHIFT_QWORDS(SRC1, SRC2)
DEST[VLMAX-1:128] ← 0

VPSLLQ (xmm, imm8)
DEST[127:0] ← LOGICAL_LEFT_SHIFT_QWORDS(SRC1, imm8)
DEST[VLMAX-1:128] ← 0

PSLLQ (xmm, xmm, xmm/m128)
DEST[127:0] ← LOGICAL_LEFT_SHIFT_QWORDS(DEST, SRC)
DEST[VLMAX-1:128] (Unmodified)

PSLLQ (xmm, imm8)
DEST[127:0] ← LOGICAL_LEFT_SHIFT_QWORDS(DEST, imm8)
DEST[VLMAX-1:128] (Unmodified)

VPSLLW (xmm, xmm, xmm/m128)
DEST[127:0] ← LOGICAL_LEFT_SHIFT_WORDS(SRC1, SRC2)
DEST[VLMAX-1:128] ← 0

VPSLLW (xmm, imm8)
DEST[127:0] ← LOGICAL_LEFT_SHIFT_WORDS(SRC1, imm8)
DEST[VLMAX-1:128] ← 0

PSLLW (xmm, xmm, xmm/m128)
DEST[127:0] ← LOGICAL_LEFT_SHIFT_WORDS(DEST, SRC)
DEST[VLMAX-1:128] (Unmodified)

PSLLW (xmm, imm8)
DEST[127:0] ← LOGICAL_LEFT_SHIFT_WORDS(DEST, imm8)
DEST[VLMAX-1:128] (Unmodified)

VPSLLD (xmm, xmm, xmm/m128)
DEST[127:0] ← LOGICAL_LEFT_SHIFT_DWORDS(SRC1, SRC2)
DEST[VLMAX-1:128] ← 0

VPSLLD (xmm, imm8)
DEST[127:0] ← LOGICAL_LEFT_SHIFT_DWORDS(SRC1, imm8)
DEST[VLMAX-1:128] ← 0

Intel C/C++ Compiler Intrinsic Equivalents

PSLLW __m64 _mm_slli_pi16 (__m64 m, int count)
PSLLW __m64 _mm_sll_pi16(__m64 m, __m64 count)
PSLLW __m128i _mm_slli_pi16(__m64 m, int count)
PSLLW __m128i _mm_slli_pi16(__m128i m, __m128i count)
PSLLD __m64 _mm_slli_pi32(__m64 m, int count)
PSLLD __m64 _mm_sll_pi32(__m64 m, __m64 count)
PSLLD __m128i _mm_slli_epi32(__m128i m, int count)
PSLLD __m128i _mm_sll_epi32(__m128i m, __m128i count)
PSLLQ __m64 _mm_slli_si64(__m64 m, int count)
PSLLQ __m64 _mm_sll_si64(__m64 m, __m64 count)
PSLLQ __m128i _mm_slli_epi64(__m128i m, int count)
PSLLQ __m128i _mm_sll_epi64(__m128i m, __m128i count)

Flags Affected

None.

Numeric Exceptions

None.

Other Exceptions

See Exceptions Type 4 and 7 for non-VEX-encoded instructions.
#UD If VEX.L = 1.

...

PSRAW/PSRAD—Shift Packed Data Right Arithmetic

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
0F E1 /r[1] PSRAW *mm, mm/m64*	A	V/V	MMX	Shift words in *mm* right by *mm/m64* while shifting in sign bits.
66 0F E1 /r PSRAW *xmm1*, *xmm2/m128*	A	V/V	SSE2	Shift words in *xmm1* right by *xmm2/m128* while shifting in sign bits.
0F 71 /4 ib[1] PSRAW *mm, imm8*	B	V/V	MMX	Shift words in *mm* right by *imm8* while shifting in sign bits
66 0F 71 /4 ib PSRAW *xmm1*, imm8	B	V/V	SSE2	Shift words in *xmm1* right by imm8 while shifting in sign bits
0F E2 /r[1] PSRAD *mm, mm/m64*	A	V/V	MMX	Shift doublewords in *mm* right by *mm/m64* while shifting in sign bits.
66 0F E2 /r PSRAD *xmm1*, *xmm2/m128*	A	V/V	SSE2	Shift doubleword in *xmm1* right by *xmm2 /m128* while shifting in sign bits.
0F 72 /4 ib[1] PSRAD *mm, imm8*	B	V/V	MMX	Shift doublewords in *mm* right by *imm8* while shifting in sign bits.
66 0F 72 /4 ib PSRAD *xmm1*, imm8	B	V/V	SSE2	Shift doublewords in *xmm1* right by *imm8* while shifting in sign bits.
VEX.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.

NOTES:

1. See note in Section 2.4, “Instruction Exception Specification”.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
A	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
B	ModRM:r/m (r, w)	imm8	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 or doublewords) in the destination operand (first operand) to the right by the number of bits specified in the count operand (second operand). As the bits in the data elements are shifted right, the empty high-order bits are filled with the initial value of the sign bit of the data element. If the value specified by the count operand is greater than 15 (for words) or 31 (for doublewords), each destination data element is filled with the initial value of the sign bit of the element. (Figure 4-15 gives an example of shifting words in a 64-bit operand.)

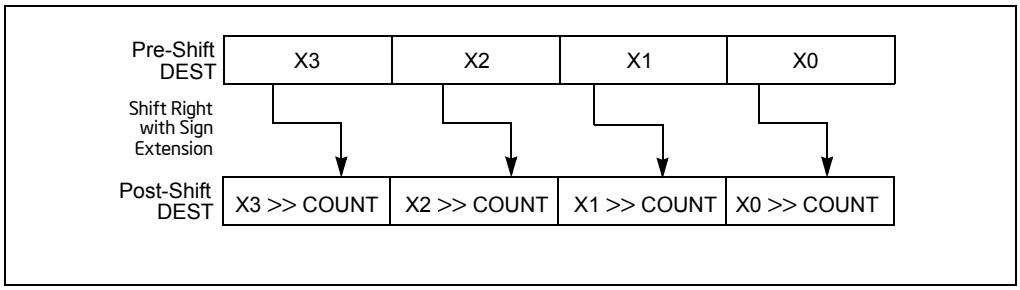


Figure 4-15 PSRAW and PSRAD Instruction Operation Using a 64-bit Operand

The destination operand may be an MMX technology register or an XMM register; the count operand can be either an MMX technology register or an 64-bit memory location, 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 destination operand to the right by the number of bits specified in the count operand, and the PSRAD instruction shifts each of the doublewords in the destination operand.

In 64-bit mode, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

128-bit Legacy SSE version: Bits (VLMAX-1:128) 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 (VLMAX-1:128) 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

PSRAW (with 64-bit operand)

```
IF (COUNT > 15)
    THEN COUNT ← 16;
FI;
DEST[15:0] ← SignExtend(DEST[15:0] >> COUNT);
(* Repeat shift operation for 2nd and 3rd words *)
DEST[63:48] ← SignExtend(DEST[63:48] >> COUNT);
```

PSRAD (with 64-bit operand)

```
IF (COUNT > 31)
    THEN COUNT ← 32;
FI;
DEST[31:0] ← SignExtend(DEST[31:0] >> COUNT);
DEST[63:32] ← SignExtend(DEST[63:32] >> COUNT);
```

PSRAW (with 128-bit operand)

```
COUNT ← COUNT_SOURCE[63:0];
IF (COUNT > 15)
    THEN COUNT ← 16;
FI;
DEST[15:0]  ← SignExtend(DEST[15:0] >> COUNT);
(* Repeat shift operation for 2nd through 7th words *)
DEST[127:112] ← SignExtend(DEST[127:112] >> COUNT);
```

PSRAD (with 128-bit operand)

```
COUNT ← COUNT_SOURCE[63:0];
IF (COUNT > 31)
    THEN COUNT ← 32;
FI;
DEST[31:0]  ← SignExtend(DEST[31:0] >> COUNT);
(* Repeat shift operation for 2nd and 3rd doublewords *)
DEST[127:96] ← SignExtend(DEST[127:96] >>COUNT);
```

PSRAW (xmm, xmm, xmm/m128)

```
DEST[127:0] ← ARITHMETIC_RIGHT_SHIFT_WORDS(DEST, SRC)
DEST[VLMAX-1:128] (Unmodified)
```

PSRAW (xmm, imm8)

```
DEST[127:0] ← ARITHMETIC_RIGHT_SHIFT_WORDS(DEST, imm8)
DEST[VLMAX-1:128] (Unmodified)
```

VPSRAW (xmm, xmm, xmm/m128)

```
DEST[127:0] ← ARITHMETIC_RIGHT_SHIFT_WORDS(SRC1, SRC2)
DEST[VLMAX-1:128] ← 0
```

VPSRAW (xmm, imm8)

DEST[127:0] ← ARITHMETIC_RIGHT_SHIFT_WORDS(SRC1, imm8)
DEST[VLMAX-1:128] ← 0

PSRAD (xmm, xmm, xmm/m128)
DEST[127:0] ← ARITHMETIC_RIGHT_SHIFT_DWORDS(DEST, SRC)
DEST[VLMAX-1:128] (Unmodified)

PSRAD (xmm, imm8)
DEST[127:0] ← ARITHMETIC_RIGHT_SHIFT_DWORDS(DEST, imm8)
DEST[VLMAX-1:128] (Unmodified)

VPSRAD (xmm, xmm, xmm/m128)
DEST[127:0] ← ARITHMETIC_RIGHT_SHIFT_DWORDS(SRC1, SRC2)
DEST[VLMAX-1:128] ← 0

VPSRAD (xmm, imm8)
DEST[127:0] ← ARITHMETIC_RIGHT_SHIFT_DWORDS(SRC1, imm8)
DEST[VLMAX-1:128] ← 0

Intel C/C++ Compiler Intrinsic Equivalents

PSRAW __m64 _mm_srai_pi16 (__m64 m, int count)

PSRAW __m64 _mm_sra_pi16 (__m64 m, __m64 count)

PSRAD __m64 _mm_srai_pi32 (__m64 m, int count)

PSRAD __m64 _mm_sra_pi32 (__m64 m, __m64 count)

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)

Flags Affected

None.

Numeric Exceptions

None.

Other Exceptions

See Exceptions Type 4 and 7 for non-VEX-encoded instructions.

#UD	If VEX.L = 1.

...

PSRLDQ—Shift Double Quadword Right Logical

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 73 /3 ib PSRLDQ *xmm1, imm8*	A	V/V	SSE2	Shift *xmm1* right by *imm8* while shifting in 0s.
VEX.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

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
A	ModRM:r/m (r, w)	imm8	NA	NA
B	VEX.vvvv (w)	ModRM:r/m (r)	NA	NA

Description

Shifts the destination operand (first operand) to the right by the number of bytes specified in the count operand (second operand). The empty high-order bytes are cleared (set to all 0s). If the value specified by the count operand is greater than 15, the destination operand is set to all 0s. The destination operand is an XMM register. The count operand is an 8-bit immediate.

In 64-bit mode, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

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

VEX.128 encoded version: Bits (VLMAX-1:128) 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

PSRLDQ(128-bit Legacy SSE version)

```
TEMP ← COUNT
IF (TEMP > 15) THEN TEMP ← 16; FI
DEST ← DEST >> (TEMP * 8)
DEST[VLMAX-1:128] (Unmodified)
```

VPSRLDQ (VEX.128 encoded version)

```
TEMP ← COUNT
IF (TEMP > 15) THEN TEMP ← 16; FI
DEST ← SRC >> (TEMP * 8)
DEST[VLMAX-1:128] ← 0
```

Intel C/C++ Compiler Intrinsic Equivalents

PSRLDQ __m128i _mm_srli_si128 (__m128i a, int imm)

Flags Affected

None.

Numeric Exceptions

None.

Other Exceptions

See Exceptions Type 7; additionally

#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
0F D1 /r[1] PSRLW *mm, mm/m64*	A	V/V	MMX	Shift words in *mm* right by amount specified in *mm/m64* while shifting in 0s.
66 0F D1 /r PSRLW *xmm1, xmm2/m128*	A	V/V	SSE2	Shift words in *xmm1* right by amount specified in *xmm2/m128* while shifting in 0s.
0F 71 /2 ib[1] PSRLW *mm, imm8*	B	V/V	MMX	Shift words in *mm* right by *imm8* while shifting in 0s.
66 0F 71 /2 ib PSRLW *xmm1, imm8*	B	V/V	SSE2	Shift words in *xmm1* right by *imm8* while shifting in 0s.
0F D2 /r[1] PSRLD *mm, mm/m64*	A	V/V	MMX	Shift doublewords in *mm* right by amount specified in *mm/m64* while shifting in 0s.
66 0F D2 /r PSRLD *xmm1, xmm2/m128*	A	V/V	SSE2	Shift doublewords in *xmm1* right by amount specified in *xmm2 /m128* while shifting in 0s.
0F 72 /2 ib[1] PSRLD *mm, imm8*	B	V/V	MMX	Shift doublewords in *mm* right by *imm8* while shifting in 0s.
66 0F 72 /2 ib PSRLD *xmm1*, imm8	B	V/V	SSE2	Shift doublewords in *xmm1* right by *imm8* while shifting in 0s.
0F D3 /r[1] PSRLQ *mm, mm/m64*	A	V/V	MMX	Shift *mm* right by amount specified in *mm/m64* while shifting in 0s.

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
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.
0F 73 /2 ib[1] PSRLQ *mm, imm8*	B	V/V	MMX	Shift *mm* right by *imm8* while shifting in 0s.
66 0F 73 /2 ib PSRLQ *xmm1, imm8*	B	V/V	SSE2	Shift quadwords in *xmm1* right by *imm8* while shifting in 0s.
VEX.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.
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.

NOTES:

1. See note in Section 2.4, "Instruction Exception Specification".

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
A	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
B	ModRM:r/m (r, w)	imm8	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 destination operand (first operand) to the right by the number of bits specified in the count operand (second operand). As the bits in the data elements are shifted right, the

empty high-order bits are cleared (set to 0). If the value specified by the count operand is greater than 15 (for words), 31 (for doublewords), or 63 (for a quadword), then the destination operand is set to all 0s. Figure 4-16 gives an example of shifting words in a 64-bit operand.

The destination operand may be an MMX technology register or an XMM register; the count operand can be either an MMX technology register or an 64-bit memory location, 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.

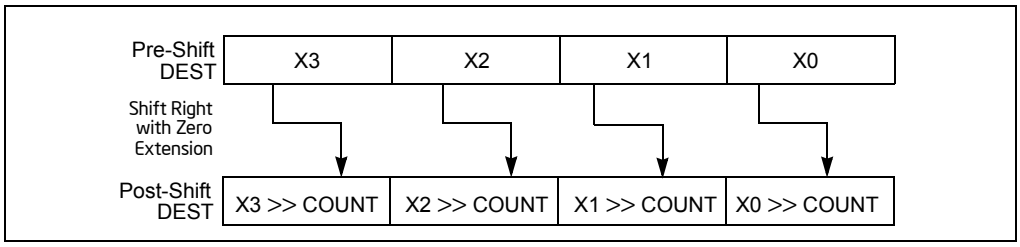


Figure 4-16 PSRLW, PSRLD, and PSRLQ Instruction Operation Using 64-bit Operand

The PSRLW instruction shifts each of the words in the destination operand to the right by the number of bits specified in the count operand; the PSRLD instruction shifts each of the doublewords in the destination operand; and the PSRLQ instruction shifts the quad-word (or quadwords) in the destination operand.

In 64-bit mode, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

128-bit Legacy SSE version: Bits (VLMAX-1:128) 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 (VLMAX-1:128) 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

PSRLW (with 64-bit operand)

```
IF (COUNT > 15)
THEN
    DEST[64:0] ← 0000000000000000H
ELSE
    DEST[15:0] ← ZeroExtend(DEST[15:0] >> COUNT);
    (* Repeat shift operation for 2nd and 3rd words *)
    DEST[63:48] ← ZeroExtend(DEST[63:48] >> COUNT);
FI;
```

PSRLD (with 64-bit operand)

```
IF (COUNT > 31)
THEN
    DEST[64:0] ← 0000000000000000H
ELSE
```

```
        DEST[31:0] ← ZeroExtend(DEST[31:0] >> COUNT);
        DEST[63:32] ← ZeroExtend(DEST[63:32] >> COUNT);
    FI;
```

PSRLQ (with 64-bit operand)

```
    IF (COUNT > 63)
    THEN
        DEST[64:0] ← 0000000000000000H
    ELSE
        DEST ← ZeroExtend(DEST >> COUNT);
    FI;
```

PSRLW (with 128-bit operand)

```
    COUNT ← COUNT_SOURCE[63:0];
    IF (COUNT > 15)
    THEN
        DEST[128:0] ← 00000000000000000000000000000000H
    ELSE
        DEST[15:0]  ← ZeroExtend(DEST[15:0] >> COUNT);
        (* Repeat shift operation for 2nd through 7th words *)
        DEST[127:112] ← ZeroExtend(DEST[127:112] >> COUNT);
    FI;
```

PSRLD (with 128-bit operand)

```
    COUNT ← COUNT_SOURCE[63:0];
    IF (COUNT > 31)
    THEN
        DEST[128:0] ← 00000000000000000000000000000000H
    ELSE
        DEST[31:0]  ← ZeroExtend(DEST[31:0] >> COUNT);
        (* Repeat shift operation for 2nd and 3rd doublewords *)
        DEST[127:96] ← ZeroExtend(DEST[127:96] >> COUNT);
    FI;
```

PSRLQ (with 128-bit operand)

```
    COUNT ← COUNT_SOURCE[63:0];
    IF (COUNT > 15)
    THEN
        DEST[128:0] ← 00000000000000000000000000000000H
    ELSE
        DEST[63:0]  ← ZeroExtend(DEST[63:0] >> COUNT);
        DEST[127:64] ← ZeroExtend(DEST[127:64] >> COUNT);
    FI;
```

PSRLW (xmm, xmm, xmm/m128)

```
DEST[127:0] ← LOGICAL_RIGHT_SHIFT_WORDS(DEST, SRC)
DEST[VLMAX-1:128] (Unmodified)
```

PSRLW (xmm, imm8)

```
DEST[127:0] ← LOGICAL_RIGHT_SHIFT_WORDS(DEST, imm8)
DEST[VLMAX-1:128] (Unmodified)
```

VPSRLW (xmm, xmm, xmm/m128)
DEST[127:0] ← LOGICAL_RIGHT_SHIFT_WORDS(SRC1, SRC2)
DEST[VLMAX-1:128] ← 0

VPSRLW (xmm, imm8)
DEST[127:0] ← LOGICAL_RIGHT_SHIFT_WORDS(SRC1, imm8)
DEST[VLMAX-1:128] ← 0

PSRLD (xmm, xmm, xmm/m128)
DEST[127:0] ← LOGICAL_RIGHT_SHIFT_DWORDS(DEST, SRC)
DEST[VLMAX-1:128] (Unmodified)

PSRLD (xmm, imm8)
DEST[127:0] ← LOGICAL_RIGHT_SHIFT_DWORDS(DEST, imm8)
DEST[VLMAX-1:128] (Unmodified)

VPSRLD (xmm, xmm, xmm/m128)
DEST[127:0] ← LOGICAL_RIGHT_SHIFT_DWORDS(SRC1, SRC2)
DEST[VLMAX-1:128] ← 0

VPSRLD (xmm, imm8)
DEST[127:0] ← LOGICAL_RIGHT_SHIFT_DWORDS(SRC1, imm8)
DEST[VLMAX-1:128] ← 0

PSRLQ (xmm, xmm, xmm/m128)
DEST[127:0] ← LOGICAL_RIGHT_SHIFT_QWORDS(DEST, SRC)
DEST[VLMAX-1:128] (Unmodified)

PSRLQ (xmm, imm8)
DEST[127:0] ← LOGICAL_RIGHT_SHIFT_QWORDS(DEST, imm8)
DEST[VLMAX-1:128] (Unmodified)

VPSRLQ (xmm, xmm, xmm/m128)
DEST[127:0] ← LOGICAL_RIGHT_SHIFT_QWORDS(SRC1, SRC2)
DEST[VLMAX-1:128] ← 0

VPSRLQ (xmm, imm8)
DEST[127:0] ← LOGICAL_RIGHT_SHIFT_QWORDS(SRC1, imm8)
DEST[VLMAX-1:128] ← 0

Intel C/C++ Compiler Intrinsic Equivalents

PSRLW __m64 _mm_srli_pi16(__m64 m, int count)

PSRLW __m64 _mm_srl_pi16 (__m64 m, __m64 count)

PSRLW __m128i _mm_srli_epi16 (__m128i m, int count)

PSRLW __m128i _mm_srl_epi16 (__m128i m, __m128i count)

PSRLD __m64 _mm_srli_pi32 (__m64 m, int count)

PSRLD __m64 _mm_srl_pi32 (__m64 m, __m64 count)

PSRLD __m128i _mm_srli_epi32 (__m128i m, int count)
PSRLD __m128i _mm_srl_epi32 (__m128i m, __m128i count)
PSRLQ __m64 _mm_srli_si64 (__m64 m, int count)
PSRLQ __m64 _mm_srl_si64 (__m64 m, __m64 count)
PSRLQ __m128i _mm_srli_epi64 (__m128i m, int count)
PSRLQ __m128i _mm_srl_epi64 (__m128i m, __m128i count)

Flags Affected

None.

Numeric Exceptions

None.

Other Exceptions

See Exceptions Type 4 and 7 for non-VEX-encoded instructions.
#UD If VEX.L = 1.

...

PSUBB/PSUBW/PSUBD—Subtract Packed Integers

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
0F F8 /r[1] PSUBB *mm, mm/m64*	A	V/V	MMX	Subtract packed byte integers in *mm/m64* from packed byte integers in *mm*.
66 0F F8 /r PSUBB *xmm1, xmm2/m128*	A	V/V	SSE2	Subtract packed byte integers in *xmm2/m128* from packed byte integers in *xmm1*.
0F F9 /r[1] PSUBW *mm, mm/m64*	A	V/V	MMX	Subtract packed word integers in *mm/m64* from packed word integers in *mm*.
66 0F F9 /r PSUBW *xmm1, xmm2/m128*	A	V/V	SSE2	Subtract packed word integers in *xmm2/m128* from packed word integers in *xmm1*.
0F FA /r[1] PSUBD *mm, mm/m64*	A	V/V	MMX	Subtract packed doubleword integers in *mm/m64* from packed doubleword integers in *mm*.
66 0F FA /r PSUBD *xmm1, xmm2/m128*	A	V/V	SSE2	Subtract packed doubleword integers in *xmm2/mem128* from packed doubleword integers in *xmm1*.

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
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.

NOTES:

1. See note in Section 2.4, "Instruction Exception Specification".

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
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 integers of the source operand (second operand) from the packed integers of the destination operand (first operand), and stores the packed integer results in the destination operand. See Figure 9-4 in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*, for an illustration of a SIMD operation. Overflow is handled with wraparound, as described in the following paragraphs.

These instructions can operate on either 64-bit or 128-bit operands. When operating on 64-bit operands, the destination operand must be an MMX technology register and the source operand can be either an MMX technology register or a 64-bit memory location. When operating on 128-bit operands, the destination operand must be an XMM register and the source operand can be either an XMM register or a 128-bit memory location.

The PSUBB instruction subtracts packed byte integers. When an individual result is too large or too small to be represented in a byte, the result is wrapped around and the low 8 bits are written to the destination element.

The PSUBW instruction subtracts packed word integers. When an individual result is too large or too small to be represented in a word, the result is wrapped around and the low 16 bits are written to the destination element.

The PSUBD instruction subtracts packed doubleword integers. When an individual result is too large or too small to be represented in a doubleword, the result is wrapped around and the low 32 bits are written to the destination element.

Note that the PSUBB, PSUBW, and PSUBD instructions can operate on either unsigned or signed (two's complement notation) packed integers; however, it does not set bits in the EFLAGS register to indicate overflow and/or a carry. To prevent undetected overflow conditions, software must control the ranges of values upon which it operates.

In 64-bit mode, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

128-bit Legacy SSE version: Bits (VLMAX-1:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: Bits (VLMAX-1:128) of the destination YMM register are zeroed. VEX.L must be 0, otherwise instructions will #UD.

Operation

PSUBB (with 64-bit operands)

```
DEST[7:0] ← DEST[7:0] – SRC[7:0];
(* Repeat subtract operation for 2nd through 7th byte *)
DEST[63:56] ← DEST[63:56] – SRC[63:56];
```

PSUBB (with 128-bit operands)

```
DEST[7:0] ← DEST[7:0] – SRC[7:0];
(* Repeat subtract operation for 2nd through 14th byte *)
DEST[127:120] ← DEST[111:120] – SRC[127:120];
```

PSUBW (with 64-bit operands)

```
DEST[15:0] ← DEST[15:0] – SRC[15:0];
(* Repeat subtract operation for 2nd and 3rd word *)
DEST[63:48] ← DEST[63:48] – SRC[63:48];
```

PSUBW (with 128-bit operands)

```
DEST[15:0]  ← DEST[15:0] – SRC[15:0];
(* Repeat subtract operation for 2nd through 7th word *)
DEST[127:112] ← DEST[127:112] – SRC[127:112];
```

PSUBD (with 64-bit operands)

```
DEST[31:0] ← DEST[31:0] – SRC[31:0];
DEST[63:32] ← DEST[63:32] – SRC[63:32];
```

PSUBD (with 128-bit operands)

```
DEST[31:0]  ← DEST[31:0] – SRC[31:0];
(* Repeat subtract operation for 2nd and 3rd doubleword *)
DEST[127:96] ← DEST[127:96] – SRC[127:96];
```

VPSUBB (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[VLMAX-1:128] ← 00

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[VLMAX-1:128] ← 0

VPSUBD (VEX.128 encoded version)
DEST[31:0] ← SRC1[31:0]-SRC2[31:0]
DEST[63:32] ← SRC1[63:32]-SRC2[63:32]
DEST[95:64] ← SRC1[95:64]-SRC2[95:64]
DEST[127:96] ← SRC1[127:96]-SRC2[127:96]
DEST[VLMAX-1:128] ← 0

Intel C/C++ Compiler Intrinsic Equivalents

PSUBB __m64 _mm_sub_pi8(__m64 m1, __m64 m2)

PSUBW __m64 _mm_sub_pi16(__m64 m1, __m64 m2)

PSUBD __m64 _mm_sub_pi32(__m64 m1, __m64 m2)

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)

Flags Affected

None.

Numeric Exceptions

None.

Other Exceptions

See Exceptions Type 4; additionally

#UD	If VEX.L = 1.

...

PSUBQ—Subtract Packed Quadword Integers

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
0F FB /r[1] PSUBQ *mm1, mm2/m64*	A	V/V	SSE2	Subtract quadword integer in *mm1* from *mm2 /m64*.
66 0F FB /r PSUBQ *xmm1, xmm2/m128*	A	V/V	SSE2	Subtract packed quadword integers in *xmm1* from *xmm2 /m128*.
VEX.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.

NOTES:

1. See note in Section 2.4, “Instruction Exception Specification”.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
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 second operand (source operand) from the first operand (destination operand) and stores the result in the destination operand. The source operand can be a quadword integer stored in an MMX technology register or a 64-bit memory location, or it can be two packed quadword integers stored in an XMM register or an 128-bit memory location. The destination operand can be a quadword integer stored in an MMX technology register or two packed quadword integers stored in an XMM register. When packed quadword operands are used, a SIMD subtract is performed. When a quadword result is too large to be represented in 64 bits (overflow), the result is wrapped around and the low 64 bits are written to the destination element (that is, the carry is ignored).

Note that the PSUBQ instruction can operate on either unsigned or signed (two’s complement notation) integers; however, it does not set bits in the EFLAGS register to indicate overflow and/or a carry. To prevent undetected overflow conditions, software must control the ranges of the values upon which it operates.

In 64-bit mode, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

128-bit Legacy SSE version: Bits (VLMAX-1:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: Bits (VLMAX-1:128) of the destination YMM register are zeroed. VEX.L must be 0, otherwise instructions will #UD.

Operation

PSUBQ (with 64-Bit operands)

```
DEST[63:0] ← DEST[63:0] – SRC[63:0];
```

PSUBQ (with 128-Bit operands)

DEST[63:0] ← DEST[63:0] – SRC[63:0];
DEST[127:64] ← DEST[127:64] – SRC[127:64];

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

Intel C/C++ Compiler Intrinsic Equivalents

PSUBQ __m64 _mm_sub_si64(__m64 m1, __m64 m2)

PSUBQ __m128i _mm_sub_epi64(__m128i m1, __m128i m2)

Flags Affected

None.

Numeric 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
0F E8 /r[1] PSUBSB *mm, mm/m64*	A	V/V	MMX	Subtract signed packed bytes in *mm/m64* from signed packed bytes in *mm* and saturate results.
66 0F E8 /r PSUBSB *xmm1, xmm2/m128*	A	V/V	SSE2	Subtract packed signed byte integers in *xmm2/m128* from packed signed byte integers in *xmm1* and saturate results.
0F E9 /r[1] PSUBSW *mm, mm/m64*	A	V/V	MMX	Subtract signed packed words in *mm/m64* from signed packed words in *mm* and saturate results.
66 0F E9 /r PSUBSW *xmm1, xmm2/m128*	A	V/V	SSE2	Subtract packed signed word integers in *xmm2/m128* from packed signed word integers in *xmm1* and saturate results.
VEX.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.

NOTES:

1. See note in Section 2.4, "Instruction Exception Specification".

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
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 source operand (second operand) from the packed signed integers of the destination operand (first operand), and stores the packed integer results in the destination operand. See Figure 9-4 in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*, for an illustration of a SIMD operation. Overflow is handled with signed saturation, as described in the following paragraphs.

These instructions can operate on either 64-bit or 128-bit operands. When operating on 64-bit operands, the destination operand must be an MMX technology register and the source operand can be either an MMX technology register or a 64-bit memory location. When operating on 128-bit operands, the destination operand must be an XMM register and the source operand can be 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.

In 64-bit mode, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

128-bit Legacy SSE version: Bits (VLMAX-1:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: Bits (VLMAX-1:128) of the destination YMM register are zeroed. VEX.L must be 0, otherwise instructions will #UD.

Operation

PSUBSB (with 64-bit operands)

```
DEST[7:0] ← SaturateToSignedByte (DEST[7:0] – SRC (7:0]);
(* Repeat subtract operation for 2nd through 7th bytes *)
DEST[63:56] ← SaturateToSignedByte (DEST[63:56] – SRC[63:56] );
```

PSUBSB (with 128-bit operands)

```
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]);
```

PSUBSW (with 64-bit operands)

```
DEST[15:0] ← SaturateToSignedWord (DEST[15:0] – SRC[15:0] );
(* Repeat subtract operation for 2nd and 7th words *)
DEST[63:48] ← SaturateToSignedWord (DEST[63:48] – SRC[63:48] );
```

PSUBSW (with 128-bit operands)

```
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]);
```

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[VLMAX-1:128] ← 0
```

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[VLMAX-1:128] ← 0

Intel C/C++ Compiler Intrinsic Equivalents

PSUBSB __m64 _mm_subs_pi8(__m64 m1, __m64 m2)

PSUBSB __m128i _mm_subs_epi8(__m128i m1, __m128i m2)

PSUBSW __m64 _mm_subs_pi16(__m64 m1, __m64 m2)

PSUBSW __m128i _mm_subs_epi16(__m128i m1, __m128i m2)

Flags Affected

None.

Numeric 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
0F D8 /r[1] PSUBUSB *mm, mm/m64*	A	V/V	MMX	Subtract unsigned packed bytes in *mm/m64* from unsigned packed bytes in *mm* and saturate result.
66 0F D8 /r PSUBUSB *xmm1, xmm2/m128*	A	V/V	SSE2	Subtract packed unsigned byte integers in *xmm2/m128* from packed unsigned byte integers in xmm1 and saturate result.
0F D9 /r[1] PSUBUSW *mm, mm/m64*	A	V/V	MMX	Subtract unsigned packed words in *mm/m64* from unsigned packed words in *mm* and saturate result.
66 0F D9 /r PSUBUSW *xmm1, xmm2/m128*	A	V/V	SSE2	Subtract packed unsigned word integers in *xmm2/m128* from packed unsigned word integers in xmm1 and saturate result.
VEX.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.

NOTES:

1. See note in Section 2.4, "Instruction Exception Specification".

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
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 source operand (second operand) from the packed unsigned integers of the destination operand (first operand), and stores the packed unsigned integer results in the destination operand. See Figure 9-4 in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*, for an illustration of a SIMD operation. Overflow is handled with unsigned saturation, as described in the following paragraphs.

These instructions can operate on either 64-bit or 128-bit operands. When operating on 64-bit operands, the destination operand must be an MMX technology register and the source operand can be either an MMX technology register or a 64-bit memory location. When operating on 128-bit operands, the destination operand must be an XMM register and the 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.

In 64-bit mode, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

128-bit Legacy SSE version: Bits (VLMAX-1:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: Bits (VLMAX-1:128) of the destination YMM register are zeroed. VEX.L must be 0, otherwise instructions will #UD.

Operation

PSUBUSB (with 64-bit operands)

```
DEST[7:0] ← SaturateToUnsignedByte (DEST[7:0] – SRC (7:0] );
(* Repeat add operation for 2nd through 7th bytes *)
DEST[63:56] ← SaturateToUnsignedByte (DEST[63:56] – SRC[63:56];
```

PSUBUSB (with 128-bit operands)

```
DEST[7:0] ← SaturateToUnsignedByte (DEST[7:0] – SRC[7:0]);
(* Repeat add operation for 2nd through 14th bytes *)
DEST[127:120] ← SaturateToUnSignedByte (DEST[127:120] – SRC[127:120]);
```

PSUBUSW (with 64-bit operands)

```
DEST[15:0] ← SaturateToUnsignedWord (DEST[15:0] – SRC[15:0] );
(* Repeat add operation for 2nd and 3rd words *)
DEST[63:48] ← SaturateToUnsignedWord (DEST[63:48] – SRC[63:48] );
```

PSUBUSW (with 128-bit operands)

```
DEST[15:0]  ← SaturateToUnsignedWord (DEST[15:0] – SRC[15:0]);
(* Repeat add operation for 2nd through 7th words *)
DEST[127:112] ← SaturateToUnSignedWord (DEST[127:112] – SRC[127:112]);
```

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[VLMAX-1:128] ← 0
```

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[VLMAX-1:128] ← 0
```

Intel C/C++ Compiler Intrinsic Equivalents

PSUBUSB __m64 _mm_subs_pu8(__m64 m1, __m64 m2)

PSUBUSB __m128i _mm_subs_epu8(__m128i m1, __m128i m2)

PSUBUSW __m64 _mm_subs_pu16(__m64 m1, __m64 m2)

PSUBUSW __m128i _mm_subs_epu16(__m128i m1, __m128i m2)

Flags Affected

None.

Numeric Exceptions

None.

Other Exceptions

See Exceptions Type 4; additionally

#UD	If VEX.L = 1.

...

PTEST- Logical Compare

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 if *xmm2/m128* AND *xmm1* result is all 0s. Set CF if *xmm2/m128* AND NOT *xmm1* result is all 0s.
VEX.128.66.0F38.WIG 17 /r VPTEST xmm1, xmm2/m128	A	V/V	AVX	Set ZF and CF depending on bitwise 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 bitwise 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

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
A	ModRM:reg (r)	ModRM:r/m (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 source sign bits with the inverted dest sign bits produces all zeros the CF is set else the CF is clear. An attempt to execute VTESTPS with VEX.W=1 will cause #UD.

VTESTPD performs a bitwise comparison of all the sign bits of the double-precision elements in the first source operation and corresponding sign bits in the second source operand. If the AND of the source sign bits with the dest sign bits produces all zeros, the ZF is set else the ZF is clear. If the AND the source sign bits with the inverted dest sign bits produces all zeros the CF is set else the CF is clear. An attempt to execute VTESTPS with VEX.W=1 will cause #UD.

The first source register is specified by the ModR/M *reg* field.

128-bit version: The first source register is an XMM register. The second source register can be an XMM register or a 256-bit memory location. The destination register is not modified.

VEX.256 encoded version: The first source register is a YMM register. The second source register can be a YMM register or a 256-bit memory location. The destination register is not modified.

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

Operation

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;
```

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;
```

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

PTEST int _mm_testz_si128 (__m128i s1, __m128i s2);
int _mm_testc_si128 (__m128i s1, __m128i s2);
int _mm_testnzc_si128 (__m128i s1, __m128i s2);

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);

Flags Affected

The 0F, AF, PF, SF flags are cleared and the ZF, CF flags are set according to the operation.

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Exceptions Type 4; additionally

#UD	If VEX.vvvv != 1111B.
	If VEX.W = 1 for VTESTPS or VTESTPD.

...

PUNPCKHBW/PUNPCKHWD/PUNPCKHDQ/PUNPCKHQDQ— Unpack High Data

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
0F 68 /r[1] PUNPCKHBW *mm, mm/m64*	A	V/V	MMX	Unpack and interleave high-order bytes from *mm* and *mm/m64* into *mm*.
66 0F 68 /r PUNPCKHBW *xmm1, xmm2/m128*	A	V/V	SSE2	Unpack and interleave high-order bytes from *xmm1* and *xmm2/m128* into *xmm1*.
0F 69 /r[1] PUNPCKHWD *mm, mm/m64*	A	V/V	MMX	Unpack and interleave high-order words from *mm* and *mm/m64* into *mm*.
66 0F 69 /r PUNPCKHWD *xmm1, xmm2/m128*	A	V/V	SSE2	Unpack and interleave high-order words from *xmm1* and *xmm2/m128* into *xmm1*.
0F 6A /r[1] PUNPCKHDQ *mm, mm/m64*	A	V/V	MMX	Unpack and interleave high-order doublewords from *mm* and *mm/m64* into *mm*.
66 0F 6A /r PUNPCKHDQ *xmm1, xmm2/m128*	A	V/V	SSE2	Unpack and interleave high-order doublewords from *xmm1* and *xmm2/m128* into *xmm1*.
66 0F 6D /r PUNPCKHQDQ *xmm1, xmm2/m128*	A	V/V	SSE2	Unpack and interleave high-order quadwords from *xmm1* and *xmm2/m128* into *xmm1*.
VEX.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.

NOTES:

1. See note in Section 2.4, “Instruction Exception Specification”.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
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, or quadwords) of the destination operand (first operand) and source operand (second operand) into the destination operand. Figure 4-17 shows the unpack operation for bytes in 64-bit operands. The low-order data elements are ignored.

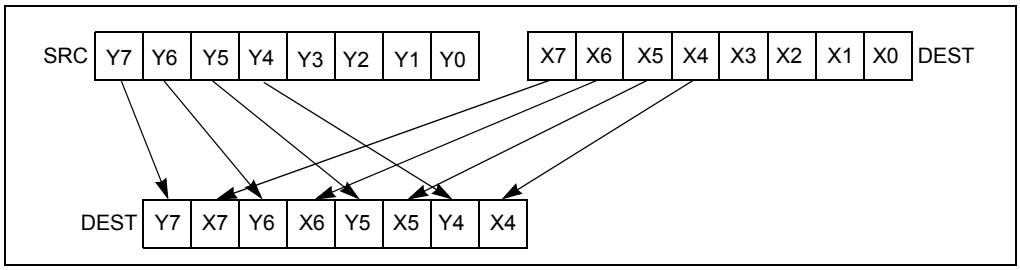


Figure 4-17 PUNPCKHBW Instruction Operation Using 64-bit Operands

The source operand can be an MMX technology register or a 64-bit memory location, or it can be an XMM register or a 128-bit memory location. The destination operand can be an MMX technology register or an XMM register. When the source data comes from a 64-bit memory operand, the full 64-bit operand is accessed from memory, but the instruction uses only the high-order 32 bits. When the source data comes from a 128-bit memory operand, an implementation may fetch only the appropriate 64 bits; however, alignment to a 16-byte boundary and normal segment checking will still be enforced.

The 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.

These instructions can be used to convert bytes to words, words to doublewords, doublewords to quadwords, and quadwords to double quadwords, respectively, by placing all 0s in the source operand. Here, if the source operand contains all 0s, the result (stored in the destination operand) contains zero extensions of the high-order data elements from the original value in the destination operand. For example, with the PUNPCKHBW instruction the high-order bytes are zero extended (that is, unpacked into unsigned word integers), and with the PUNPCKHWD instruction, the high-order words are zero extended (unpacked into unsigned doubleword integers).

In 64-bit mode, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

128-bit Legacy SSE versions: Bits (VLMAX-1:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded versions: Bits (VLMAX-1:128) of the destination YMM register are zeroed. VEX.L must be 0, otherwise instructions will #UD.

Operation

PUNPCKHBW instruction with 64-bit operands:

```
DEST[7:0] ← DEST[39:32];
DEST[15:8] ← SRC[39:32];
DEST[23:16] ← DEST[47:40];
DEST[31:24] ← SRC[47:40];
DEST[39:32] ← DEST[55:48];
DEST[47:40] ← SRC[55:48];
DEST[55:48] ← DEST[63:56];
DEST[63:56] ← SRC[63:56];
```

PUNPCKHW instruction with 64-bit operands:

```
DEST[15:0] ← DEST[47:32];
DEST[31:16] ← SRC[47:32];
DEST[47:32] ← DEST[63:48];
DEST[63:48] ← SRC[63:48];
```

PUNPCKHDQ instruction with 64-bit operands:

```
DEST[31:0] ← DEST[63:32];
DEST[63:32] ← SRC[63:32];
```

PUNPCKHBW instruction with 128-bit operands:

```
DEST[7:0]← DEST[71:64];
DEST[15:8]  ← SRC[71:64];
DEST[23:16] ← DEST[79:72];
DEST[31:24] ← SRC[79:72];
DEST[39:32] ← DEST[87:80];
DEST[47:40] ← SRC[87:80];
DEST[55:48] ← DEST[95:88];
DEST[63:56] ← SRC[95:88];
DEST[71:64] ← DEST[103:96];
DEST[79:72] ← SRC[103:96];
DEST[87:80] ← DEST[111:104];
DEST[95:88] ← SRC[111:104];
DEST[103:96]  ← DEST[119:112];
DEST[111:104] ← SRC[119:112];
DEST[119:112] ← DEST[127:120];
DEST[127:120] ← SRC[127:120];
```

PUNPCKHWD instruction with 128-bit operands:

```
DEST[15:0]  ← DEST[79:64];
DEST[31:16] ← SRC[79:64];
DEST[47:32] ← DEST[95:80];
DEST[63:48] ← SRC[95:80];
DEST[79:64] ← DEST[111:96];
DEST[95:80] ← SRC[111:96];
DEST[111:96]  ← DEST[127:112];
DEST[127:112] ← SRC[127:112];
```

PUNPCKHDQ instruction with 128-bit operands:
DEST[31:0] ← DEST[95:64];
DEST[63:32] ← SRC[95:64];
DEST[95:64] ← DEST[127:96];
DEST[127:96] ← SRC[127:96];

PUNPCKHQDQ instruction:
DEST[63:0] ← DEST[127:64];
DEST[127:64] ← SRC[127:64];

PUNPCKHBW
DEST[127:0] ← INTERLEAVE_HIGH_BYTES(DEST, SRC)
DEST[VLMAX-1:128] (Unmodified)

VPUNPCKHBW
DEST[127:0] ← INTERLEAVE_HIGH_BYTES(SRC1, SRC2)
DEST[VLMAX-1:128] ← 0

PUNPCKHWD
DEST[127:0] ← INTERLEAVE_HIGH_WORDS(DEST, SRC)
DEST[VLMAX-1:128] (Unmodified)

VPUNPCKHWD
DEST[127:0] ← INTERLEAVE_HIGH_WORDS(SRC1, SRC2)
DEST[VLMAX-1:128] ← 0

PUNPCKHDQ
DEST[127:0] ← INTERLEAVE_HIGH_DWORDS(DEST, SRC)
DEST[VLMAX-1:128] (Unmodified)

VPUNPCKHDQ
DEST[127:0] ← INTERLEAVE_HIGH_DWORDS(SRC1, SRC2)
DEST[VLMAX-1:128] ← 0

PUNPCKHQDQ
DEST[127:0] ← INTERLEAVE_HIGH_QWORDS(DEST, SRC)
DEST[VLMAX-1:128] (Unmodified)

VPUNPCKHQDQ
DEST[127:0] ← INTERLEAVE_HIGH_QWORDS(SRC1, SRC2)

DEST[VLMAX-1:128] ← 0

Intel C/C++ Compiler Intrinsic Equivalents

PUNPCKHBW __m64 _mm_unpackhi_pi8(__m64 m1, __m64 m2)
PUNPCKHBW __m128i _mm_unpackhi_epi8(__m128i m1, __m128i m2)
PUNPCKHWD __m64 _mm_unpackhi_pi16(__m64 m1,__m64 m2)
PUNPCKHWD __m128i _mm_unpackhi_epi16(__m128i m1,__m128i m2)
PUNPCKHDQ __m64 _mm_unpackhi_pi32(__m64 m1, __m64 m2)

PUNPCKHDQ __m128i _mm_unpackhi_epi32(__m128i m1, __m128i m2)
PUNPCKHQDQ __m128i _mm_unpackhi_epi64 (__m128i a, __m128i b)

Flags Affected

None.

Numeric 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
0F 60 /r[1] PUNPCKLBW *mm, mm/m32*	A	V/V	MMX	Interleave low-order bytes from *mm* and *mm/m32* into *mm*.
66 0F 60 /r PUNPCKLBW *xmm1*, *xmm2/m128*	A	V/V	SSE2	Interleave low-order bytes from *xmm1* and *xmm2/m128* into *xmm1*.
0F 61 /r[1] PUNPCKLWD *mm, mm/m32*	A	V/V	MMX	Interleave low-order words from *mm* and *mm/m32* into *mm*.
66 0F 61 /r PUNPCKLWD *xmm1*, *xmm2/m128*	A	V/V	SSE2	Interleave low-order words from *xmm1* and *xmm2/m128* into *xmm1*.
0F 62 /r[1] PUNPCKLDQ *mm, mm/m32*	A	V/V	MMX	Interleave low-order doublewords from *mm* and *mm/m32* into *mm*.
66 0F 62 /r PUNPCKLDQ *xmm1*, *xmm2/m128*	A	V/V	SSE2	Interleave low-order doublewords from *xmm1* and *xmm2/m128* into xmm1.
66 0F 6C /r PUNPCKLQDQ *xmm1*, *xmm2/m128*	A	V/V	SSE2	Interleave low-order quadword from *xmm1* and *xmm2/m128* into *xmm1* register.
VEX.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.

NOTES:

1. See note in Section 2.4, “Instruction Exception Specification”.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
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 destination operand (first operand) and source operand (second operand) into the destination operand. (Figure 4-18 shows the unpack operation for bytes in 64-bit operands.). The high-order data elements are ignored.

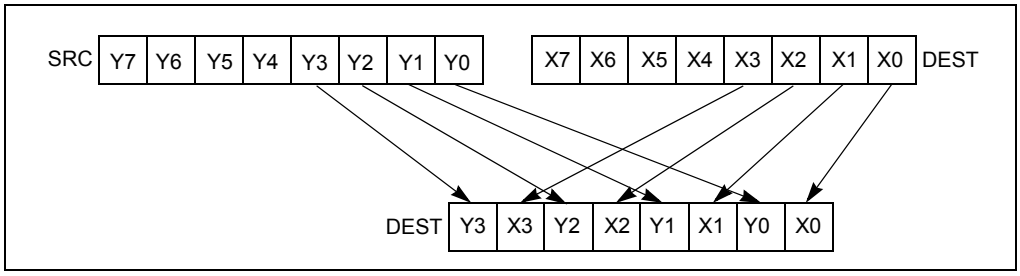


Figure 4-18 PUNPCKLBW Instruction Operation Using 64-bit Operands

The source operand can be an MMX technology register or a 32-bit memory location, or it can be an XMM register or a 128-bit memory location. The destination operand can be an MMX technology register or an XMM register. 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.

These instructions can be used to convert bytes to words, words to doublewords, doublewords to quadwords, and quadwords to double quadwords, respectively, by placing all 0s in the source operand. Here, if the source operand contains all 0s, the result (stored in the destination operand) contains zero extensions of the high-order data elements from the original value in the destination operand. For example, with the PUNPCKLBW instruction the high-order bytes are zero extended (that is, unpacked into unsigned word integers), and with the PUNPCKLWD instruction, the high-order words are zero extended (unpacked into unsigned doubleword integers).

In 64-bit mode, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

128-bit Legacy SSE versions: Bits (VLMAX-1:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded versions: Bits (VLMAX-1:128) of the destination YMM register are zeroed. VEX.L must be 0, otherwise instructions will #UD.

Operation

PUNPCKLBW instruction with 64-bit operands:

```
DEST[63:56] ← SRC[31:24];
DEST[55:48] ← DEST[31:24];
DEST[47:40] ← SRC[23:16];
DEST[39:32] ← DEST[23:16];
DEST[31:24] ← SRC[15:8];
DEST[23:16] ← DEST[15:8];
DEST[15:8] ← SRC[7:0];
DEST[7:0] ← DEST[7:0];
```

PUNPCKLWD instruction with 64-bit operands:

```
DEST[63:48] ← SRC[31:16];
DEST[47:32] ← DEST[31:16];
DEST[31:16] ← SRC[15:0];
DEST[15:0] ← DEST[15:0];
```

PUNPCKLDQ instruction with 64-bit operands:

```
DEST[63:32] ← SRC[31:0];
DEST[31:0] ← DEST[31:0];
```

PUNPCKLBW instruction with 128-bit operands:

```
DEST[7:0]← DEST[7:0];
DEST[15:8] ← SRC[7:0];
DEST[23:16] ← DEST[15:8];
DEST[31:24] ← SRC[15:8];
DEST[39:32] ← DEST[23:16];
DEST[47:40] ← SRC[23:16];
DEST[55:48] ← DEST[31:24];
DEST[63:56] ← SRC[31:24];
DEST[71:64] ← DEST[39:32];
DEST[79:72] ← SRC[39:32];
DEST[87:80] ← DEST[47:40];
DEST[95:88] ← SRC[47:40];
DEST[103:96] ← DEST[55:48];
DEST[111:104] ← SRC[55:48];
DEST[119:112] ← DEST[63:56];
DEST[127:120] ← SRC[63:56];
```

PUNPCKLWD instruction with 128-bit operands:

```
DEST[15:0] ← DEST[15:0];
DEST[31:16] ← SRC[15:0];
DEST[47:32] ← DEST[31:16];
DEST[63:48] ← SRC[31:16];
DEST[79:64] ← DEST[47:32];
DEST[95:80] ← SRC[47:32];
DEST[111:96] ← DEST[63:48];
DEST[127:112] ← SRC[63:48];
```

PUNPCKLDQ instruction with 128-bit operands:

```
DEST[31:0] ← DEST[31:0];
```

```
    DEST[63:32]  ← SRC[31:0];
    DEST[95:64]  ← DEST[63:32];
    DEST[127:96] ← SRC[63:32];

PUNPCKLQDQ
    DEST[63:0] ← DEST[63:0];
    DEST[127:64] ← SRC[63:0];
```

VPUNPCKLBW

```
DEST[127:0] ← INTERLEAVE_BYTES(SRC1, SRC2)
DEST[VLMAX-1:128] ← 0
```

VPUNPCKLWD

```
DEST[127:0] ← INTERLEAVE_WORDS(SRC1, SRC2)
DEST[VLMAX-1:128] ← 0
```

VPUNPCKLDQ

```
DEST[127:0] ← INTERLEAVE_DWORDS(SRC1, SRC2)
DEST[VLMAX-1:128] ← 0
```

VPUNPCKLQDQ

```
DEST[127:0] ← INTERLEAVE_QWORDS(SRC1, SRC2)
DEST[VLMAX-1:128] ← 0
```

Intel C/C++ Compiler Intrinsic Equivalents

PUNPCKLBW	__m64 _mm_unpacklo_pi8 (__m64 m1, __m64 m2)
PUNPCKLBW	__m128i _mm_unpacklo_epi8 (__m128i m1, __m128i m2)
PUNPCKLWD	__m64 _mm_unpacklo_pi16 (__m64 m1, __m64 m2)
PUNPCKLWD	__m128i _mm_unpacklo_epi16 (__m128i m1, __m128i m2)
PUNPCKLDQ	__m64 _mm_unpacklo_pi32 (__m64 m1, __m64 m2)
PUNPCKLDQ	__m128i _mm_unpacklo_epi32 (__m128i m1, __m128i m2)
PUNPCKLQDQ	__m128i _mm_unpacklo_epi64 (__m128i m1, __m128i m2)

Flags Affected

None.

Numeric Exceptions

None.

Other Exceptions

See Exceptions Type 4; additionally

#UD	If VEX.L = 1.

...

PXOR—Logical Exclusive OR

Opcode*/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
0F EF /r[1] PXOR *mm, mm/m64*	A	V/V	MMX	Bitwise XOR of *mm/m64* and *mm*.
66 0F EF /r PXOR *xmm1, xmm2/m128*	A	V/V	SSE2	Bitwise XOR of *xmm2/m128* and *xmm1*.
VEX.NDS.128.66.0F.WIG EF /r VPXOR xmm1, xmm2, xmm3/m128	B	V/V	AVX	Bitwise XOR of xmm3/m128 and xmm2.

NOTES:

1. See note in Section 2.4, "Instruction Exception Specification".

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
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 exclusive-OR (XOR) operation on the source operand (second operand) and the destination operand (first operand) and stores the result in the destination operand. The source operand can be an MMX technology register or a 64-bit memory location or it can be an XMM register or a 128-bit memory location. The destination operand can be an MMX technology register or an XMM register. Each bit of the result is 1 if the corresponding bits of the two operands are different; each bit is 0 if the corresponding bits of the operands are the same.

In 64-bit mode, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

128-bit Legacy SSE version: Bits (VLMAX-1:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: Bits (VLMAX-1:128) of the destination YMM register are zeroed. VEX.L must be 0, otherwise instructions will #UD.

Operation

PXOR (128-bit Legacy SSE version)
DEST ← DEST XOR SRC
DEST[VLMAX-1:128] (Unmodified)

VPXOR (VEX.128 encoded version)
DEST ← SRC1 XOR SRC2

Intel C/C++ Compiler Intrinsic Equivalent

PXOR __m64 _mm_xor_si64 (__m64 m1, __m64 m2)

PXOR __m128i _mm_xor_si128 (__m128i a, __m128i b)

Flags Affected

None.

Numeric Exceptions

None.

Other Exceptions

See Exceptions Type 4; additionally

#UD	If VEX.L = 1.

...

RCPPS—Compute 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 the packed single-precision floating-point values in *xmm2/m128* 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

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

Description

Performs a SIMD computation of the approximate reciprocals of the four packed single-precision floating-point values in the source operand (second operand) stores the packed single-precision floating-point results in the destination operand. The source operand can be an XMM register or a 128-bit memory location. The destination operand is an XMM register. See Figure 10-5 in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*, for an illustration of a SIMD single-precision floating-point operation.

The relative error for this approximation is:

$$|\text{Relative Error}| \leq 1.5 * 2^{-12}$$

The RCPPS instruction is not affected by the rounding control bits in the MXCSR register. When a source value is a 0.0, an ∞ 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. (Input values greater than or equal to $|1.11111111110100000000000B*2^{125}|$ are guaranteed to not produce tiny results; input values less than or equal to $|1.00000000000110000000001B*2^{126}|$ are guaranteed to produce tiny results, which are in turn flushed to 0.0; and 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.

In 64-bit mode, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

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 (VLMAX-1:128) 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 (VLMAX-1:128) 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

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[VLMAX-1:128] (Unmodified)
```

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[VLMAX-1:128] ← 0
```

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])

Intel C/C++ Compiler Intrinsic Equivalent

RCCPS __m128 _mm_rcp_ps(__m128 a)

RCPPS __m256 _mm256_rcp_ps (__m256 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 Values

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

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
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 source operand (second operand) and stores the single-precision floating-point result in the destination operand. The source operand can be an XMM register or a 32-bit

memory location. The destination operand 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.

The relative error for this approximation is:

$$|\text{Relative Error}| \leq 1.5 * 2^{-12}$$

The RCPSS instruction is not affected by the rounding control bits in the MXCSR register. When a source value is a 0.0, an ∞ 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. (Input values greater than or equal to $|1.11111111110100000000000B*2^{125}|$ are guaranteed to not produce tiny results; input values less than or equal to $|1.00000000000110000000001B*2^{126}|$ are guaranteed to produce tiny results, which are in turn flushed to 0.0; and 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.

In 64-bit mode, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

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

VEX.128 encoded version: Bits (VLMAX-1:128) of the destination YMM register are zeroed.

Operation

RCPSS (128-bit Legacy SSE version)

```
DEST[31:0] ← APPROXIMATE(1/SRC[31:0])
DEST[VLMAX-1:32] (Unmodified)
```

VRCPSS (VEX.128 encoded version)

```
DEST[31:0] ← APPROXIMATE(1/SRC2[31:0])
DEST[127:32] ← SRC1[127:32]
DEST[VLMAX-1:128] ← 0
```

Intel C/C++ Compiler Intrinsic Equivalent

RCPSS __m128 _mm_rcp_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

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
A	ModRM:reg (w)	ModRM:r/m (r)	imm8	NA

Description

Round the 2 double-precision floating-point values in the source operand (second operand) using the rounding mode specified in the immediate operand (third operand) and place the results in the destination operand (first operand). The rounding process rounds each input floating-point value to an integer value and returns the integer 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 4-19. 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 (Table 4-14 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 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 (VLMAX-1:128) of the corresponding YMM register destination are unmodified.

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 (VLMAX-1:128) of the corresponding YMM register destination are zeroed.

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

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

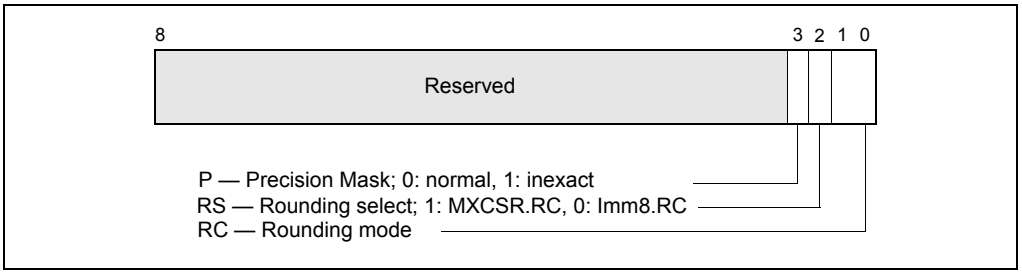


Figure 4-19 Bit Control Fields of Immediate Byte for ROUNDxx Instruction

Table 4-14 Rounding Modes and Encoding of Rounding Control (RC) Field

Rounding Mode	RC Field Setting	Description
Round to nearest (even)	00B	Rounded result is the closest to the infinitely precise result. If two values are equally close, the result is the even value (i.e., the integer value with the least-significant bit of zero).
Round down (toward −∞)	01B	Rounded result is closest to but no greater than the infinitely precise result.
Round up (toward +∞)	10B	Rounded result is closest to but no less than the infinitely precise result.
Round toward zero (Truncate)	11B	Rounded result is closest to but no greater in absolute value than the infinitely precise result.

Operation

```
IF (imm[2] = '1)
    THEN    // rounding mode is determined by MXCSR.RC
        DEST[63:0] ← ConvertDPFPToInteger_M(SRC[63:0]);
        DEST[127:64] ← ConvertDPFPToInteger_M(SRC[127:64]);
    ELSE    // rounding mode is determined by IMM8.RC
        DEST[63:0] ← ConvertDPFPToInteger_Imm(SRC[63:0]);
        DEST[127:64] ← ConvertDPFPToInteger_Imm(SRC[127:64]);
FI
```

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[VLMAX-1:128] (Unmodified)
```

VROUNDPD (VEX.128 encoded version)

```
DEST[63:0] ← RoundToInteger(SRC[63:0]], ROUND_CONTROL)
DEST[127:64] ← RoundToInteger(SRC[127:64]], ROUND_CONTROL)
DEST[VLMAX-1:128] ← 0
```

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)

Intel C/C++ Compiler Intrinsic Equivalent

__m128 _mm_round_pd(__m128d s1, int iRoundMode);

__m128 _mm_floor_pd(__m128d s1);

__m128 _mm_ceil_pd(__m128d s1)

__m256 _mm256_round_pd(__m256d s1, int iRoundMode);

__m256 _mm256_floor_pd(__m256d s1);

__m256 _mm256_ceil_pd(__m256d 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 and precision exception is not signaled.)

Note that Denormal is not signaled by ROUNDPD.

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

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
A	ModRM:reg (w)	ModRM:r/m (r)	imm8	NA

Description

Round the 4 single-precision floating-point values in the source operand (second operand) using the rounding mode specified in the immediate operand (third operand) and place the results in the destination operand (first operand). The rounding process rounds each input floating-point value to an integer value and returns the integer 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 4-19. 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 (Table 4-14 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 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 (VLMAX-1:128) of the corresponding YMM register destination are unmodified.

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 (VLMAX-1:128) of the corresponding YMM register destination are zeroed.

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

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

Operation

```
IF (imm[2] = '1)
    THEN     // rounding mode is determined by MXCSR.RC
         DEST[31:0] ← ConvertSPFPToInteger_M(SRC[31:0]);
         DEST[63:32] ← ConvertSPFPToInteger_M(SRC[63:32]);
         DEST[95:64] ← ConvertSPFPToInteger_M(SRC[95:64]);
         DEST[127:96] ← ConvertSPFPToInteger_M(SRC[127:96]);
    ELSE     // rounding mode is determined by IMM8.RC
         DEST[31:0] ← ConvertSPFPToInteger_Imm(SRC[31:0]);
         DEST[63:32] ← ConvertSPFPToInteger_Imm(SRC[63:32]);
         DEST[95:64] ← ConvertSPFPToInteger_Imm(SRC[95:64]);
         DEST[127:96] ← ConvertSPFPToInteger_Imm(SRC[127:96]);
FI;
```

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[VLMAX-1:128] (Unmodified)
```

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[VLMAX-1:128] ← 0
```

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)
```

Intel C/C++ Compiler Intrinsic Equivalent

```
__m128 _mm_round_ps(__m128 s1, int iRoundMode);

__m128 _mm_floor_ps(__m128 s1);
```

__m128 _mm_ceil_ps(__m128 s1)

__m256 _mm256_round_ps(__m256 s1, int iRoundMode);

__m256 _mm256_floor_ps(__m256 s1);

__m256 _mm256_ceil_ps(__m256 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 and precision exception is not signaled.)

Note that Denormal is not signaled by ROUNDPS.

Other Exceptions

See Exceptions Type 2; additionally

#UD	If VEX.vvvv != 1111B.

...

ROUNDSD — Round Scalar Double Precision Floating-Point Values

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

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
A	ModRM:reg (w)	ModRM:r/m (r)	imm8	NA
B	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA

Description

Round the DP FP value in the lower qword of the source operand (second operand) using the rounding mode specified in the immediate operand (third operand) and place the result in the destination operand (first operand). The rounding process rounds a double-precision floating-point input to an integer value and returns the integer 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 4-19. 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 (Table 4-14 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 (VLMAX-1:64) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: Bits (VLMAX-1:128) of the destination YMM register are zeroed.

Operation

```
IF (imm[2] = '1)
    THEN    // rounding mode is determined by MXCSR.RC
        DEST[63:0] ← ConvertDPFPToInteger_M(SRC[63:0]);
    ELSE    // rounding mode is determined by IMM8.RC
        DEST[63:0] ← ConvertDPFPToInteger_Imm(SRC[63:0]);
FI;
DEST[127:63] remains unchanged ;
```

ROUNDSD (128-bit Legacy SSE version)

```
DEST[63:0] ← RoundToInteger(SRC[63:0], ROUND_CONTROL)
DEST[VLMAX-1:64] (Unmodified)
```

VROUNDSD (VEX.128 encoded version)

```
DEST[63:0] ← RoundToInteger(SRC2[63:0], ROUND_CONTROL)
DEST[127:64] ← SRC1[127:64]
DEST[VLMAX-1:128] ← 0
```

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 and precision exception is not signaled.)

Note that Denormal is not signaled by ROUNDSD.

Other Exceptions

See Exceptions Type 3.

...

ROUNDSS — Round Scalar Single Precision Floating-Point Values

Opcode*/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 3A 0A /r 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

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
A	ModRM:reg (w)	ModRM:r/m (r)	imm8	NA
B	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA

Description

Round the single-precision floating-point value in the lowest dword of the source operand (second operand) using the rounding mode specified in the immediate operand (third operand) and place the result in the destination operand (first operand). The rounding process rounds a single-precision floating-point input to an integer 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 retained.

The immediate operand specifies control fields for the rounding operation, three bit fields are defined and shown in Figure 4-19. 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 (Table 4-14 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 (VLMAX-1:32) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: Bits (VLMAX-1:128) of the destination YMM register are zeroed.

Operation

```
IF (imm[2] = '1)
    THEN    // rounding mode is determined by MXCSR.RC
        DEST[31:0] ← ConvertSPFPToInteger_M(SRC[31:0]);
    ELSE    // rounding mode is determined by IMM8.RC
        DEST[31:0] ← ConvertSPFPToInteger_Imm(SRC[31:0]);
FI;
DEST[127:32] remains unchanged ;
```

ROUNDSS (128-bit Legacy SSE version)

```
DEST[31:0] ← RoundToInteger(SRC[31:0], ROUND_CONTROL)
DEST[VLMAX-1:32] (Unmodified)
```

VROUNDSS (VEX.128 encoded version)

```
DEST[31:0] ← RoundToInteger(SRC2[31:0], ROUND_CONTROL)
DEST[127:32] ← SRC1[127:32]
DEST[VLMAX-1:128] ← 0
```

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 and precision exception is not signaled.)

Note that Denormal is not signaled by ROUNDSS.

Other Exceptions

See Exceptions Type 3.

...

RSM—Resume from System Management Mode

…

Operation

```
ReturnFromSMM;
IF (IA-32e mode supported) or (CPUID DisplayFamily_DisplayModel = 06H_0CH )
    THEN
        ProcessorState ← Restore(SMMDump(IA-32e SMM STATE MAP));
    Else
        ProcessorState ← Restore(SMMDump(Non-32-Bit-Mode SMM STATE MAP));
FI
```

…

RSQRTPS—Compute 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 the packed single-precision floating-point values in *xmm2/m128* 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

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

Description

Performs a SIMD computation of the approximate reciprocals of the square roots of the four 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. The source operand can be an XMM register or a 128-bit memory location. The

destination operand is an XMM register. See Figure 10-5 in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*, for an illustration of a SIMD single-precision floating-point operation.

The relative error for this approximation is:

$$|\text{Relative Error}| \leq 1.5 * 2^{-12}$$

The RSQRTPS instruction is not affected by the rounding control bits in the MXCSR register. When a source value is a 0.0, an ∞ 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.

In 64-bit mode, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

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 (VLMAX-1:128) 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 (VLMAX-1:128) 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

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[VLMAX-1:128] (Unmodified)
```

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[VLMAX-1:128] ← 0
```

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]))

Intel C/C++ Compiler Intrinsic Equivalent

RSQRTPS __m128 _mm_rsqrt_ps(__m128 a)
RSQRTPS __m256 _mm256_rsqrt_ps (__m256 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

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
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 source operand (second operand) stores the single-precision floating-point result in the destination operand. The source operand can be an XMM

register or a 32-bit memory location. The destination operand 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.

The relative error for this approximation is:

$$|\text{Relative Error}| \leq 1.5 * 2^{-12}$$

The RSQRTSS instruction is not affected by the rounding control bits in the MXCSR register. When a source value is a 0.0, an ∞ 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.

In 64-bit mode, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

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

VEX.128 encoded version: Bits (VLMAX-1:128) of the destination YMM register are zeroed.

Operation

RSQRTSS (128-bit Legacy SSE version)

```
DEST[31:0] ← APPROXIMATE(1/SQRT(SRC2[31:0]))
DEST[VLMAX-1:32] (Unmodified)
```

VRSQRTSS (VEX.128 encoded version)

```
DEST[31:0] ← APPROXIMATE(1/SQRT(SRC2[31:0]))
DEST[127:32] ← SRC1[31:0]
DEST[VLMAX-1:128] ← 0
```

Intel C/C++ Compiler Intrinsic Equivalent

RSQRTSS __m128 _mm_rsqrt_ss(__m128 a)

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Exceptions Type 5.

...

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/m128* to *xmm1*.
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

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
A	ModRM:reg (r, w)	ModRM:r/m (r)	imm8	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 destination operand (first operand) into the low quadword of the destination operand; moves either of the two packed double-precision floating-point values from the source operand into to the high quadword of the destination operand (see Figure 4-20). The select operand (third operand) determines which values are moved to the destination operand.

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 (VLMAX-1:128) of the corresponding YMM register destination are unmodified.

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 (VLMAX-1:128) of the corresponding YMM register destination are zeroed.

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

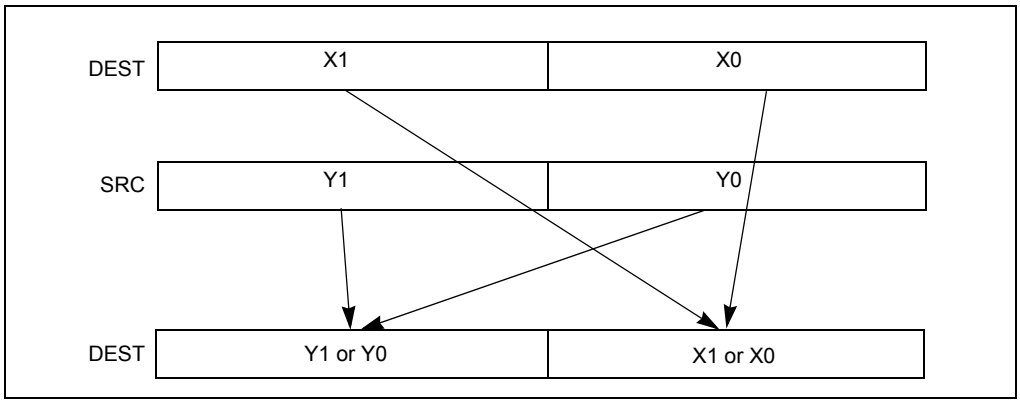


Figure 4-20 SHUFPD Shuffle Operation

The source operand can be an XMM register or a 128-bit memory location. The destination operand is an XMM register. The select operand is an 8-bit immediate: bit 0 selects which value is moved from the destination operand to the result (where 0 selects the low quadword and 1 selects the high quadword) and bit 1 selects which value is moved from the source operand to the result. Bits 2 through 7 of the select operand are reserved and must be set to 0.

In 64-bit mode, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

Operation

```
IF SELECT[0] = 0
    THEN DEST[63:0] ← DEST[63:0];
    ELSE DEST[63:0] ← DEST[127:64]; FI;

IF SELECT[1] = 0
    THEN DEST[127:64] ← SRC[63:0];
    ELSE DEST[127:64] ← SRC[127:64]; FI;
```

SHUFPD (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[VLMAX-1:128] (Unmodified)
```

VSHUFPD (VEX.128 encoded version)

```
IF IMM0[0] = 0
    THEN DEST[63:0] ← SRC1[63:0]
    ELSE DEST[63:0] ← SRC1[127:64] FI;
IF IMM0[1] = 0
    THEN DEST[127:64] ← SRC2[63:0]
    ELSE DEST[127:64] ← SRC2[127:64] FI;
```

DEST[VLMAX-1:128] ← 0

VSHUFPD (VEX.256 encoded version)

```
IF IMM0[0] = 0
    THEN DEST[63:0] ← SRC1[63:0]
    ELSE DEST[63:0] ← SRC1[127:64] FI;
IF IMM0[1] = 0
    THEN DEST[127:64] ← SRC2[63:0]
    ELSE DEST[127:64] ← SRC2[127:64] FI;
IF IMM0[2] = 0
    THEN DEST[191:128] ← SRC1[191:128]
    ELSE DEST[191:128] ← SRC1[255:192] FI;
IF IMM0[3] = 0
    THEN DEST[255:192] ← SRC2[191:128]
    ELSE DEST[255:192] ← SRC2[255:192] FI;
```

Intel C/C++ Compiler Intrinsic Equivalent

SHUFPD __m128d _mm_shuffle_pd(__m128d a, __m128d b, unsigned int imm8)
VSHUFPD __m256d _mm256_shuffle_pd (__m256d a, __m256d 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, xmm2/m128, imm8*	A	V/V	SSE	Shuffle packed single-precision floating-point values selected by *imm8* from *xmm1* and *xmm1/m128* to *xmm1*.
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

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
A	ModRM:reg (r, w)	ModRM:r/m (r)	imm8	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 the destination operand (first operand) into the low quadword of the destination operand; moves two of the four packed single-precision floating-point values from the source operand (second operand) into to the high quadword of the destination operand (see Figure 4-21). The select operand (third operand) determines which values are moved to the destination operand.

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 (VLMAX-1:128) of the corresponding YMM register destination are unmodified.

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 (VLMAX-1:128) of the corresponding YMM register destination are zeroed.

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.

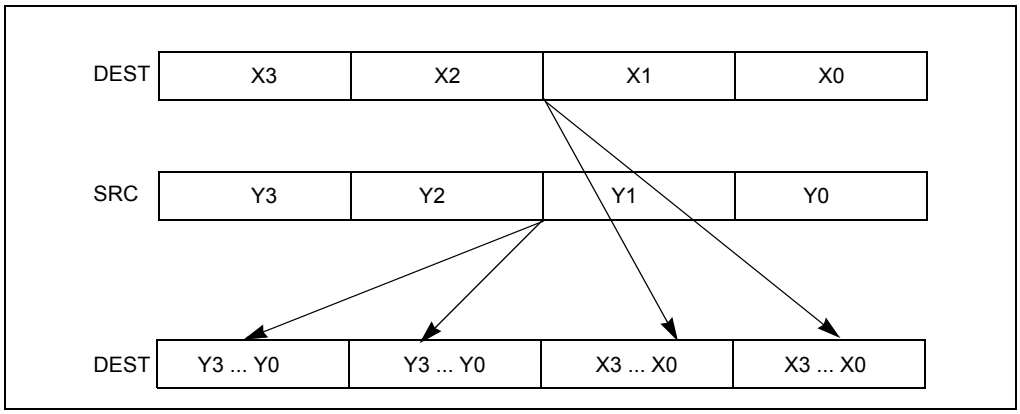


Figure 4-21 SHUFPS Shuffle Operation

The source operand can be an XMM register or a 128-bit memory location. The destination operand is an XMM register. The select operand is an 8-bit immediate: bits 0 and 1 select the value to be moved from the destination operand to the low doubleword of the result, bits 2 and 3 select the value to be moved from the destination operand to the second doubleword of the result, bits 4 and 5 select the value to be moved from the source operand to the third doubleword of the result, and bits 6 and 7 select the value to be moved from the source operand to the high doubleword of the result.

In 64-bit mode, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

Operation

```
CASE (SELECT[1:0]) OF
    0:  DEST[31:0] ← DEST[31:0];
    1:  DEST[31:0] ← DEST[63:32];
    2:  DEST[31:0] ← DEST[95:64];
    3:  DEST[31:0] ← DEST[127:96];
ESAC;

CASE (SELECT[3:2]) OF
    0:  DEST[63:32] ← DEST[31:0];
    1:  DEST[63:32] ← DEST[63:32];
    2:  DEST[63:32] ← DEST[95:64];
    3:  DEST[63:32] ← DEST[127:96];
ESAC;

CASE (SELECT[5:4]) OF
    0:  DEST[95:64] ← SRC[31:0];
    1:  DEST[95:64] ← SRC[63:32];
    2:  DEST[95:64] ← SRC[95:64];
    3:  DEST[95:64] ← SRC[127:96];
ESAC;

CASE (SELECT[7:6]) OF
    0:  DEST[127:96] ← SRC[31:0];
    1:  DEST[127:96] ← SRC[63:32];
    2:  DEST[127:96] ← SRC[95:64];
    3:  DEST[127:96] ← SRC[127:96];
ESAC;
```

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[VLMAX-1:128] (Unmodified)
```

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[VLMAX-1:128] ← 0
```

VSHUFPS (VEX.256 encoded version)

```
DEST[31:0] ← Select4(SRC1[127:0], imm8[1:0]);
DEST[63:32] ← Select4(SRC1[127:0], imm8[3:2]);
DEST[95:64] ← Select4(SRC2[127:0], imm8[5:4]);
DEST[127:96] ← Select4(SRC2[127:0], imm8[7:6]);
DEST[159:128] ← Select4(SRC1[255:128], imm8[1:0]);
DEST[191:160] ← Select4(SRC1[255:128], imm8[3:2]);
```

DEST[223:192] ← Select4(SRC2[255:128], imm8[5:4]);
DEST[255:224] ← Select4(SRC2[255:128], imm8[7:6]);

Intel C/C++ Compiler Intrinsic Equivalent

SHUFPS __m128 _mm_shuffle_ps(__m128 a, __m128 b, unsigned int imm8)
VSHUFPS __m256 _mm256_shuffle_ps (__m256 a, __m256 b, const int select);

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Exceptions Type 4.

...

SQRTPD—Compute Square Roots of Packed 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 results 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

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

Description

Performs a SIMD computation of the square roots of the two packed double-precision floating-point values in the source operand (second operand) stores the packed double-precision floating-point results in the destination operand. The source operand can be an XMM register or a 128-bit memory location. The destination operand is an XMM register. See Figure 11-3 in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*, for an illustration of a SIMD double-precision floating-point operation.

In 64-bit mode, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

128-bit Legacy SSE version: The second source 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 (VLMAX-1:128) of the corresponding YMM register destination are unmodified.

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 (VLMAX-1:128) of the corresponding YMM register destination are zeroed.

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

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

Operation

SQRTPD (128-bit Legacy SSE version)
DEST[63:0] ← SQRT(SRC[63:0])
DEST[127:64] ← SQRT(SRC[127:64])
DEST[VLMAX-1:128] (Unmodified)

VSQRTPD (VEX.128 encoded version)
DEST[63:0] ← SQRT(SRC[63:0])
DEST[127:64] ← SQRT(SRC[127:64])
DEST[VLMAX-1:128] ← 0

VSQRTPD (VEX.256 encoded version)
DEST[63:0] ← SQRT(SRC[63:0])
DEST[127:64] ← SQRT(SRC[127:64])
DEST[191:128] ← SQRT(SRC[191:128])
DEST[255:192] ← SQRT(SRC[255:192])

Intel C/C++ Compiler Intrinsic Equivalent

SQRTPD __m128d _mm_sqrt_pd (m128d a)

SQRTPD __m256d _mm256_sqrt_pd (__m256d a);

SIMD Floating-Point Exceptions

Invalid, Precision, Denormal.

Other Exceptions

See Exceptions Type 2; additionally

#UD	If VEX.vvvv != 1111B.

...

SQRTPS—Compute Square Roots of Packed 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 results 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

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

Description

Performs a SIMD computation of the square roots of the four packed single-precision floating-point values in the source operand (second operand) stores the packed single-precision floating-point results in the destination operand. The source operand can be an XMM register or a 128-bit memory location. The destination operand is an XMM register. See Figure 10-5 in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*, for an illustration of a SIMD single-precision floating-point operation.

In 64-bit mode, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

128-bit Legacy SSE version: The second source 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 (VLMAX-1:128) of the corresponding YMM register destination are unmodified.

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 (VLMAX-1:128) of the corresponding YMM register destination are zeroed.

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

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

Operation

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[VLMAX-1:128] (Unmodified)

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[VLMAX-1:128] ← 0

VSQRTPS (VEX.256 encoded version)
DEST[31:0] ← SQRT(SRC[31:0])
DEST[63:32] ← SQRT(SRC[63:32])
DEST[95:64] ← SQRT(SRC[95:64])
DEST[127:96] ← SQRT(SRC[127:96])
DEST[159:128] ← SQRT(SRC[159:128])
DEST[191:160] ← SQRT(SRC[191:160])
DEST[223:192] ← SQRT(SRC[223:192])
DEST[255:224] ← SQRT(SRC[255:224])

Intel C/C++ Compiler Intrinsic Equivalent

SQRTPS __m128 _mm_sqrt_ps(__m128 a)

SQRTPS __m256 _mm256_sqrt_ps (__m256 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 /*r* SQRTSD *xmm1, xmm2/m64*	A	V/V	SSE2	Computes square root of the low double-precision floating-point value in *xmm2/m64* and stores the results in *xmm1*.
VEX.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

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
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 source operand (second operand) and stores the double-precision floating-point result in the destination operand. The source operand can be an XMM register or a 64-bit memory location. The destination operand is an XMM register. 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.

In 64-bit mode, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

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

VEX.128 encoded version: Bits (VLMAX-1:128) of the destination YMM register are zeroed.

Operation

SQRTSD (128-bit Legacy SSE version)
DEST[63:0] ← SQRT(SRC[63:0])
DEST[VLMAX-1:64] (Unmodified)

VSQRTSD (VEX.128 encoded version)

DEST[63:0] ← SQRT(SRC2[63:0])
DEST[127:64] ← SRC1[127:64]
DEST[VLMAX-1:128] ← 0

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 Floating-Point Value

Opcode*/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
F3 0F 51 /r SQRTSS *xmm1, xmm2/m32*	A	V/V	SSE	Computes square root of the low single-precision floating-point value in *xmm2/m32* and stores the results in *xmm1.*
VEX.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

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
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 source operand (second operand) and stores the single-precision floating-point result in the destination operand. The source operand can be an XMM register or a 32-bit memory location. The destination operand 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.

In 64-bit mode, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

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

VEX.128 encoded version: Bits (VLMAX-1:128) of the destination YMM register are zeroed.

Operation

SQRTSS (128-bit Legacy SSE version)
DEST[31:0] ← SQRT(SRC2[31:0])
DEST[VLMAX-1:32] (Unmodified)

VSQRTSS (VEX.128 encoded version)
DEST[31:0] ← SQRT(SRC2[31:0])
DEST[127:32] ← SRC1[127:32]
DEST[VLMAX-1:128] ← 0

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

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
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/m128* from *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

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
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 two packed double-precision floating-point values in the source operand (second operand) from the two packed double-precision floating-point values in the destination operand (first operand), and stores the packed double-precision floating-point results in the destination operand. The source operand can be an XMM register or a 128-bit memory location. The destination operand is an XMM register. See Figure 11-3 in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*, for an illustration of a SIMD double-precision floating-point operation.

In 64-bit mode, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

128-bit Legacy SSE version: 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 (VLMAX-1:128) of the corresponding YMM register destination are unmodified.

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 (VLMAX-1:128) of the corresponding YMM register destination are zeroed.

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

Operation

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[VLMAX-1:128] (Unmodified)

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

VSUBPD (VEX.256 encoded version)
DEST[63:0] ← SRC1[63:0] - SRC2[63:0]
DEST[127:64] ← SRC1[127:64] - SRC2[127:64]
DEST[191:128] ← SRC1[191:128] - SRC2[191:128]
DEST[255:192] ← SRC1[255:192] - SRC2[255:192]

Intel C/C++ Compiler Intrinsic Equivalent

SUBPD __m128d _mm_sub_pd (m128d a, m128d b)

VSUBPD __m256d _mm256_sub_pd (__m256d a, __m256d 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*.
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

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
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 four packed single-precision floating-point values in the source operand (second operand) from the four packed single-precision floating-point values in the destination operand (first operand), and stores the packed single-precision floating-point results in the destination operand. The source operand can be an XMM register or a 128-bit memory location. The destination operand is an XMM register. See Figure 10-5 in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*, for an illustration of a SIMD double-precision floating-point operation.

In 64-bit mode, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

128-bit Legacy SSE version: The second source 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 (VLMAX-1:128) of the corresponding YMM register destination are unmodified.

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 (VLMAX-1:128) of the corresponding YMM register destination are zeroed.

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

Operation

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[VLMAX-1:128] (Unmodified)

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[VLMAX-1:128] ← 0

VSUBPS (VEX.256 encoded version)
DEST[31:0] ← SRC1[31:0] - SRC2[31:0]
DEST[63:32] ← SRC1[63:32] - SRC2[63:32]
DEST[95:64] ← SRC1[95:64] - SRC2[95:64]
DEST[127:96] ← SRC1[127:96] - SRC2[127:96]
DEST[159:128] ← SRC1[159:128] - SRC2[159:128]
DEST[191:160]← SRC1[191:160] - SRC2[191:160]
DEST[223:192] ← SRC1[223:192] - SRC2[223:192]
DEST[255:224] ← SRC1[255:224] - SRC2[255:224].

Intel C/C++ Compiler Intrinsic Equivalent

SUBPS __m128 _mm_sub_ps(__m128 a, __m128 b)

VSUBPS __m256 _mm256_sub_ps (__m256 a, __m256 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	Subtracts the low double-precision floating-point values in *xmm2/mem64* from *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

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
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 low double-precision floating-point value in the source operand (second operand) from the low double-precision floating-point value in the destination operand (first operand), and stores the double-precision floating-point result in the destination operand. The source operand can be an XMM register or a 64-bit memory location. The destination operand is an XMM register. 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.

In 64-bit mode, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

128-bit Legacy SSE version: The destination and first source operand are the same. Bits (VLMAX-1:64) 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 (VLMAX-1:128) of the destination YMM register are zeroed.

Operation

SUBSD (128-bit Legacy SSE version)
DEST[63:0] ← DEST[63:0] - SRC[63:0]
DEST[VLMAX-1:64] (Unmodified)
VSUBSD (VEX.128 encoded version)
DEST[63:0] ← SRC1[63:0] - SRC2[63:0]
DEST[127:64] ← SRC1[127:64]
DEST[VLMAX-1:128] ← 0

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 lower single-precision floating-point values in *xmm2/m32* from *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

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
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 low single-precision floating-point value in the source operand (second operand) from the low single-precision floating-point value in the destination operand (first operand), and stores the single-precision floating-point result in the destination operand. The source operand can be an XMM register or a 32-bit memory location. The destination operand 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.

In 64-bit mode, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

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

Operation

SUBSS (128-bit Legacy SSE version)
DEST[31:0] ← DEST[31:0] - SRC[31:0]
DEST[VLMAX-1:32] (Unmodified)

VSUBSS (VEX.128 encoded version)
DEST[31:0] ← SRC1[31:0] - SRC2[31:0]
DEST[127:32] ← SRC1[127:32]
DEST[VLMAX-1:128] ← 0

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.

...

SYSENTER—Fast System Call

...

Operation

```
IF CR0.PE = 0 THEN #GP(0); FI;
IF SYSENTER_CS_MSR[15:2] = 0 THEN #GP(0); FI;
EFLAGS.VM ← 0;                          (* ensures protected mode execution *)
EFLAGS.IF ← 0;                          (* Mask interrupts *)
EFLAGS.RF ← 0;

CS.SEL ← SYSENTER_CS_MSR                (* Operating system provides CS *)
(* Set rest of CS to a fixed value *)
CS.SEL.RPL ← 0;
CS.BASE ← 0;                            (* Flat segment *)
CS.ARbyte.G ← 1;                        (* 4-KByte granularity *)
CS.ARbyte.S ← 1;
CS.ARbyte.TYPE ← 1011B;                 (* Execute + Read, Accessed *)
CS.ARbyte.D ← 1;                        (* 32-bit code segment*)
CS.ARbyte.DPL ← 0;
CS.ARbyte.P ← 1;
CS.LIMIT ← FFFFFH;                      (* with 4-KByte granularity, implies a 4-GByte limit *)
CPL ← 0;

SS.SEL ← CS.SEL + 8;
(* Set rest of SS to a fixed value *)
SS.SEL.RPL ← 0;
```

```
SS.BASE ← 0;                          (* Flat segment *)
SS.ARbyte.G ← 1;                      (* 4-KByte granularity *)
SS.ARbyte.S ← 1;
SS.ARbyte.TYPE ← 0011B;               (* Read/Write, Accessed *)
SS.ARbyte.D ← 1;                      (* 32-bit stack segment*)
SS.ARbyte.DPL ← 0;
SS.ARbyte.P ← 1;
SS.LIMIT ← FFFFFH;                    (* with 4-KByte granularity, implies a 4-GByte limit *)

ESP ← SYSENTER_ESP_MSR;
EIP ← SYSENTER_EIP_MSR;
```

…

SYSEXIT—Fast Return from Fast System Call

…

Operation

```
IF SYSENTER_CS_MSR[15:2] = 0 THEN #GP(0); FI;
IF CR0.PE = 0 THEN #GP(0); FI;
IF CPL ≠ 0 THEN #GP(0); FI;

CS.SEL ← (SYSENTER_CS_MSR + 16);      (* Segment selector for return CS *)
(* Set rest of CS to a fixed value *)
CS.SEL.RPL ← 3;
CS.BASE ← 0;                          (* Flat segment *)
CS.ARbyte.G ← 1;                      (* 4-KByte granularity *)
CS.ARbyte.S ← 1;
CS.ARbyte.TYPE ← 1011B;               (* Execute, Read, Non-Conforming Code *)
CS.ARbyte.D ← 1;                      (* 32-bit code segment*)
CS.ARbyte.DPL ← 3;
CS.ARbyte.P ← 1;
CS.LIMIT ← FFFFFH;                    (* with 4-KByte granularity, implies a 4-GByte limit *)
CPL ← 3;

SS.SEL ← (SYSENTER_CS_MSR + 24);      (* Segment selector for return SS *)
(* Set rest of SS to a fixed value *);
SS.SEL.RPL ← 3;
SS.BASE ← 0;                          (* Flat segment *)
SS.ARbyte.G ←1;                       (* 4-KByte granularity *)
SS.ARbyte.S ← 1;
SS.ARbyte.TYPE ← 0011B;               (* Expand Up, Read/Write, Data *)
SS.ARbyte.D ← 1;                      (* 32-bit stack segment*)
SS.ARbyte.DPL ← 3;
SS.ARbyte.P ← 1;
SS.LIMIT ← FFFFFH;                    (* with 4-KByte granularity, implies a 4-GByte limit *)

ESP ← ECX;
EIP ← EDX;
```

…

VTESTPD/VTESTPS—Packed Bit Test

See "PTEST- Logical Compare" on page 270.

...

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	Compares (unordered) the low double-precision floating-point values in *xmm1* and *xmm2/m64* and set the EFLAGS 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

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

Description

Performs and unordered compare of the double-precision floating-point values in the low quadwords of source operand 1 (first operand) and source 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).

Source operand 1 is an XMM register; source operand 2 can be an XMM register or a 64 bit memory location.

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

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

In 64-bit mode, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

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

Operation

```
RESULT ← UnorderedCompare(SRC1[63:0] < > SRC2[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 lower single-precision floating-point value in *xmm1* register with lower single-precision floating-point value in *xmm2/mem* and set the status 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

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

Description

Performs and unordered compare of the single-precision floating-point values in the low doublewords of the source operand 1 (first operand) and the source 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). In 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).

Source operand 1 is an XMM register; source 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 when a source operand is an SNaN. The COMISS instruction signals an invalid operation exception if a source operand is either a QNaN or an SNaN.

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

In 64-bit mode, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

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

Operation

```
RESULT ← UnorderedCompare(SRC1[31:0] <> SRC2[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

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
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 source operand (second operand) and the destination operand (first operand). See Figure 4-22.

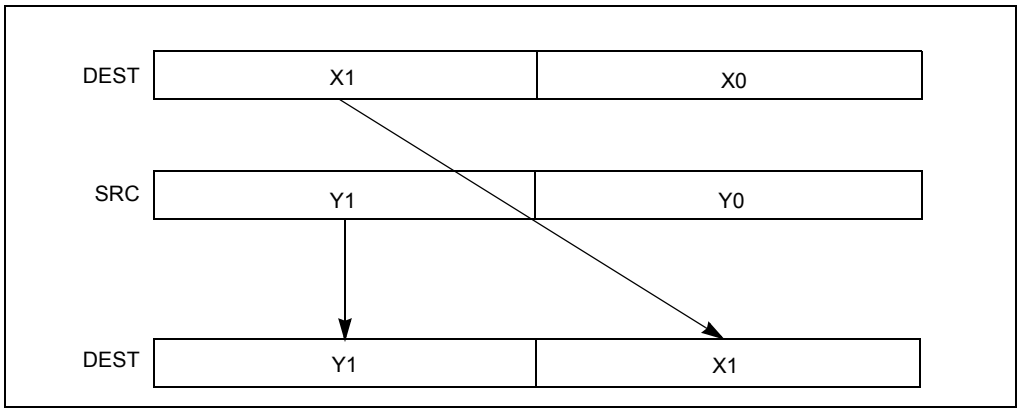


Figure 4-22 UNPCKHPD Instruction High Unpack and Interleave Operation

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.

In 64-bit mode, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

128-bit Legacy SSE version: The second source 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 (VLMAX-1:128) of the corresponding YMM register destination are unmodified.

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 (VLMAX-1:128) of the corresponding YMM register destination are zeroed.

Operation

UNPCKHPD (128-bit Legacy SSE version)
DEST[63:0] ← SRC1[127:64]
DEST[127:64] ← SRC2[127:64]
DEST[VLMAX-1:128] (Unmodified)

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

VUNPCKHPD (VEX.256 encoded version)
DEST[63:0] ← SRC1[127:64]
DEST[127:64] ← SRC2[127:64]
DEST[191:128]←SRC1[255:192]
DEST[255:192]←SRC2[255:192]

Intel C/C++ Compiler Intrinsic Equivalent

UNPCKHPD __m128d _mm_unpackhi_pd(__m128d a, __m128d b)

UNPCKHPD __m256d _mm256_unpackhi_pd(__m256d a, __m256d 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/mem* into *xmm1*.
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

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
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-order single-precision floating-point values from the source operand (second operand) and the destination operand (first operand). See Figure 4-23. The source operand can be an XMM register or a 128-bit memory location; the destination operand is an XMM register.

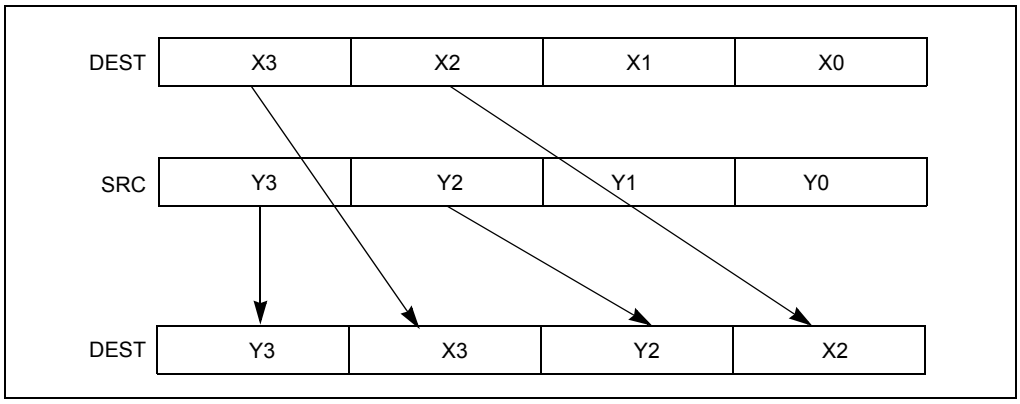


Figure 4-23 UNPCKHPS Instruction High Unpack and Interleave Operation

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.

In 64-bit mode, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

128-bit Legacy SSE version: 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 (VLMAX-1:128) of the corresponding YMM register destination are unmodified.

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 (VLMAX-1:128) of the corresponding YMM register destination are zeroed.

Operation

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[VLMAX-1:128] (Unmodified)

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[VLMAX-1:128] ← 0

VUNPCKHPS (VEX.256 encoded version)
DEST[31:0] ← SRC1[95:64]
DEST[63:32] ← SRC2[95:64]
DEST[95:64] ← SRC1[127:96]
DEST[127:96] ← SRC2[127:96]

DEST[159:128] ← SRC1[223:192]
DEST[191:160] ← SRC2[223:192]
DEST[223:192] ← SRC1[255:224]
DEST[255:224] ← SRC2[255:224]

Intel C/C++ Compiler Intrinsic Equivalent

UNPCKHPS __m128 _mm_unpackhi_ps(__m128 a, __m128 b)

UNPCKHPS __m256 _mm256_unpackhi_ps (__m256 a, __m256 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

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
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 source operand (second operand) and the destination operand (first operand). See

Figure 4-24. The source operand can be an XMM register or a 128-bit memory location; the destination operand is an XMM register.

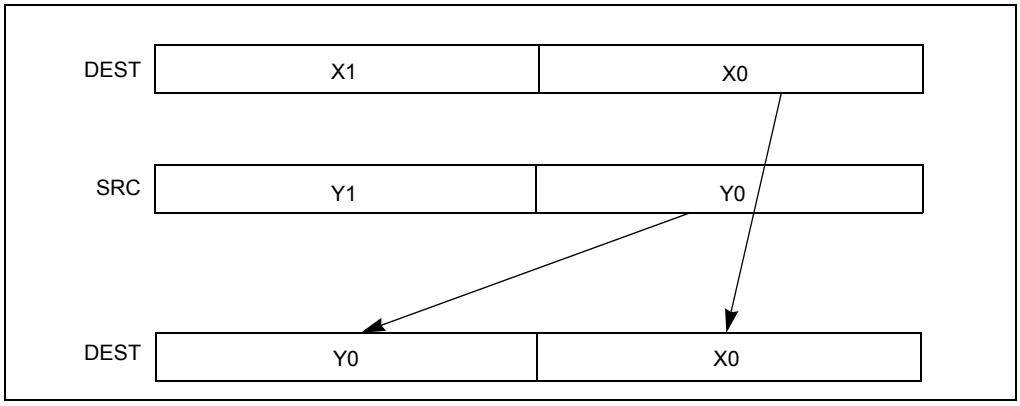


Figure 4-24 UNPCKLPD Instruction Low Unpack and Interleave Operation

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.

In 64-bit mode, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

128-bit Legacy SSE version: 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 (VLMAX-1:128) of the corresponding YMM register destination are unmodified.

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 (VLMAX-1:128) of the corresponding YMM register destination are zeroed.

Operation

UNPCKLPD (128-bit Legacy SSE version)
DEST[63:0] ← SRC1[63:0]
DEST[127:64] ← SRC2[63:0]
DEST[VLMAX-1:128] (Unmodified)

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

VUNPCKLPD (VEX.256 encoded version)
DEST[63:0] ← SRC1[63:0]
DEST[127:64] ← SRC2[63:0]
DEST[191:128] ← SRC1[191:128]
DEST[255:192] ← SRC2[191:128]

Intel C/C++ Compiler Intrinsic Equivalent

UNPCKHPD __m128d _mm_unpacklo_pd(__m128d a, __m128d b)

UNPCKLPD __m256d _mm256_unpacklo_pd(__m256d a, __m256d 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/mem* into *xmm1*.
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

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
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-order single-precision floating-point values from the source operand (second operand) and the destination operand (first operand). See Figure 4-25. The source operand can be an XMM register or a 128-bit memory location; the destination operand is an XMM register.

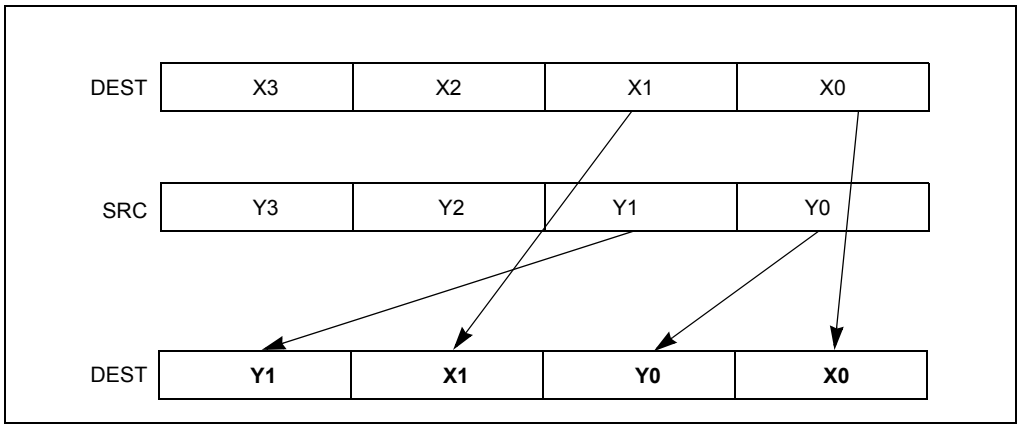


Figure 4-25 UNPCKLPS Instruction Low Unpack and Interleave Operation

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.

In 64-bit mode, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

128-bit Legacy SSE version: 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.

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.

Operation

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[VLMAX-1:128] (Unmodified)

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[VLMAX-1:128] ← 0
UNPCKLPS (VEX.256 encoded version)
DEST[31:0] ← SRC1[31:0]
DEST[63:32] ← SRC2[31:0]
DEST[95:64] ← SRC1[63:32]
DEST[127:96] ← SRC2[63:32]
DEST[159:128] ← SRC1[159:128]

DEST[191:160] ← SRC2[159:128]
DEST[223:192] ← SRC1[191:160]
DEST[255:224] ← SRC2[191:160]

Intel C/C++ Compiler Intrinsic Equivalent

UNPCKLPS __m128 _mm_unpacklo_ps(__m128 a, __m128 b)

UNPCKLPS __m256 _mm256_unpacklo_ps (__m256 a, __m256 b);

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Exceptions Type 4.

...

XORPD—Bitwise Logical XOR for 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	Bitwise exclusive-OR of *xmm2/m128* and *xmm1.*
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

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
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 exclusive-OR of the two packed double-precision floating-point values from the source operand (second operand) and the destination operand (first operand), and stores the result in the destination operand. The source operand can be an XMM register or a 128-bit memory location. The destination operand is an XMM register.

In 64-bit mode, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

128-bit Legacy SSE version: The second source 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 (VLMAX-1:128) of the corresponding YMM register destination are unmodified.

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 (VLMAX-1:128) of the corresponding YMM register destination are zeroed.

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

Operation

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[VLMAX-1:128] (Unmodified)

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[VLMAX-1:128] ← 0

VXORPD (VEX.256 encoded version)
DEST[63:0] ← SRC1[63:0] BITWISE XOR SRC2[63:0]
DEST[127:64] ← SRC1[127:64] BITWISE XOR SRC2[127:64]
DEST[191:128] ← SRC1[191:128] BITWISE XOR SRC2[191:128]
DEST[255:192] ← SRC1[255:192] BITWISE XOR SRC2[255:192]

Intel C/C++ Compiler Intrinsic Equivalent

XORPD __m128d _mm_xor_pd(__m128d a, __m128d b)

VXORPD __m256d _mm256_xor_pd (__m256d a, __m256d b);

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Exceptions Type 4.

...

XORPS—Bitwise Logical XOR for 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	Bitwise exclusive-OR of *xmm2/m128* and *xmm1*.
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

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
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 exclusive-OR of the four packed single-precision floating-point values from the source operand (second operand) and the destination operand (first operand), and stores the result in the destination operand. The source operand can be an XMM register or a 128-bit memory location. The destination operand is an XMM register.

In 64-bit mode, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

128-bit Legacy SSE version: The second source 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 (VLMAX-1:128) of the corresponding YMM register destination are unmodified.

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 (VLMAX-1:128) of the corresponding YMM register destination are zeroed.

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

Operation

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[VLMAX-1:128] (Unmodified)

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[VLMAX-1:128] ← 0

VXORPS (VEX.256 encoded version)
DEST[31:0] ← SRC1[31:0] BITWISE XOR SRC2[31:0]
DEST[63:32] ← SRC1[63:32] BITWISE XOR SRC2[63:32]
DEST[95:64] ← SRC1[95:64] BITWISE XOR SRC2[95:64]
DEST[127:96] ← SRC1[127:96] BITWISE XOR SRC2[127:96]
DEST[159:128] ← SRC1[159:128] BITWISE XOR SRC2[159:128]
DEST[191:160]← SRC1[191:160] BITWISE XOR SRC2[191:160]
DEST[223:192] ← SRC1[223:192] BITWISE XOR SRC2[223:192]
DEST[255:224] ← SRC1[255:224] BITWISE XOR SRC2[255:224].

Intel C/C++ Compiler Intrinsic Equivalent

XORPS __m128 _mm_xor_ps(__m128 a, __m128 b)

VXORPS __m256 _mm256_xor_ps (__m256 a, __m256 b);

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Exceptions Type 4.

…

XRSTOR—Restore Processor Extended States

…

XRSTOR operates on each subset of the processor state or a processor extended state in one of three ways (depending on the corresponding bit in XCR0 (XFEATURE_ENABLED_MASK register), the restore mask EDX:EAX, and the save mask XSAVE.HEADER.XSTATE_BV in memory):

- Updates the processor state component using the state information stored in the respective save area (see Table Table 13-8) of the source operand, if the corresponding bit in XCR0, EDX:EAX, and XSAVE.HEADER.XSTATE_BV are all 1.
- Writes certain registers in the processor state component using processor-supplied values (see Table Table 13-12) without using state information stored in respective save area of the memory region, if the corresponding bit in XCR0 and EDX:EAX are both 1, but the corresponding bit in XSAVE.HEADER.XSTATE_BV is 0.
- The processor state component is unchanged, if the corresponding bit in XCR0 or EDX:EAX is 0.

...

If a processor state component is not enabled in XCR0 but the corresponding save mask bit in XSAVE.HEADER.XSTATE_BV is 1, an attempt to execute XRSTOR will cause a #GP(0) exception. Software may specify all 1’s in the implicit restore mask EDX:EAX, so that all the enabled processors states in XCR0 are restored from state information stored in memory or from processor supplied values. When using all 1's as the restore mask, software is required to determine the total size of the XSAVE/XRSTOR save area (specified as source operand) to fit all enabled processor states by using the value enumerated in CPUID.(EAX=0D, ECX=0):EBX.

An attempt to restore processor states with writing 1s to reserved bits in certain registers (see Table 4-21) will cause a #GP(0) exception.

Because bit 63 of XCR0 is reserved for future bit vector expansion, it will not be used for any future processor state feature, and XRSTOR will ignore bit 63 of EDX:EAX (EDX[31].

...

XSAVE—Save Processor Extended States

...

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 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 Table 13-8. Each component save area is written if both the corresponding bits in the save mask operand and in XCR0 (the XFEATURE_ENABLED_MASK register) are 1. A processor state component save area is not updated if either one of the corresponding bits in the mask operand or in XCR0 is 0. If the mask operand (EDX:EAX) contains all 1's, all enabled processor state components in XCR0 are written to the respective component save area.

The bit assignment used for the EDX:EAX register pair matches XCR0 (see chapter 2 of Vol. 3B). For the XSAVE instruction, software can specify "1" in any bit position of EDX:EAX, irrespective of whether the corresponding bit position in XCR0 is valid for the processor. The bit vector in EDX:EAX is "anded" with XCR0 to determine which save area will be written. When specifying 1 in any bit position of EDX:EAX mask, software is required to determine the total size of the XSAVE/XRSTOR save area (specified as destination operand) to fit all enabled processor states by using the value enumerated in CPUID.(EAX=0D, ECX=0):EBX.

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 Table 13-8. 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 (refer to “FXSAVE—Save x87 FPU, MMX Technology, and SSE State” on page 468). 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.

Bytes 464:511 are available for software use. The processor does not write to bytes 464:511 when executing XSAVE.

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 (see the operation section of XRSTOR instruction). If a processor implementation discern that a processor state component is in its initialized state (according to Table Table 13-12) it may modify the corresponding bit in the HEADER.XSTATE_BV as '0'.

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.

Operation

```
TMP_MASK[62:0] ← ( (EDX[30:0] << 32 ) OR EAX[31:0] ) AND XCR0[62:0];
FOR i = 0, 62 STEP 1
    IF ( TMP_MASK[i] = 1) THEN
        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
                DEFAULT: // i corresponds to a valid sub-leaf index of CPUID leaf 0DH
                    DEST.Ext_Save_Area[ i ] ← processor state[i] ;
            ESAC:
            DEST.HEADER.XSTATE_BV[i] ← INIT_FUNCTION[i];
    FI;
NEXT;
```

…

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	Operand 2	Operand 3	Operand 4
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 XCR0 (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 XCR0 is valid for the processor. The bit vector in EDX:EAX is "anded" with XCR0 to determine which save area will be written. When specifying 1 in any bit position of EDX:EAX mask, software is required to determine the total size of the XSAVE/XRSTOR save area (specified as destination operand) to fit all enabled processor states by using the value enumerated in CPUID.(EAX=0D, ECX=0):EBX.

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 in XCR0 is 0. The processor state component that is updated to the save area is computed by bit-wise AND of the mask operand (EDX:EAX) with XCR0.

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 XCR0[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.

...

XSETBV—Set Extended Control Register

...

Description

Writes the contents of registers EDX:EAX into the 64-bit extended control register (XCR) specified in the ECX register. (On processors that support the Intel 64 architecture, the high-order 32 bits of RCX are ignored.) The contents of the EDX register are copied to high-order 32 bits of the selected XCR and the contents of the EAX register are copied to low-order 32 bits of the XCR. (On processors that support the Intel 64 architecture, the high-order 32 bits of each of RAX and RDX are ignored.) Undefined or reserved bits in an XCR should be set to values previously read.

This instruction must be executed at privilege level 0 or in real-address mode; otherwise, a general protection exception #GP(0) is generated. Specifying a reserved or unimplemented XCR in ECX will also cause a general protection exception. The processor will also generate a general protection exception if software attempts to write to reserved bits in an XCR.

Currently, only XCR0 (the XFEATURE_ENABLED_MASK register) is supported. Thus, all other values of ECX are reserved and will cause a #GP(0). Note that bit 0 of XCR0 (corre-

sponding to x87 state) must be set to 1; the instruction will cause a #GP(0) if an attempt is made to clear this bit.

...

Protected Mode Exceptions

#GP(0)	If the current privilege level is not 0.
	If an invalid XCR is specified in ECX.
	If the value in EDX:EAX sets bits that are reserved in the XCR specified by ECX.
	If an attempt is made to clear bit 0 of XCR0.
#UD	If CPUID.01H:ECX.XSAVE[bit 26] = 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 an invalid XCR is specified in ECX.
	If the value in EDX:EAX sets bits that are reserved in the XCR specified by ECX.
	If an attempt is made to clear bit 0 of XCR0.
#UD	If CPUID.01H:ECX.XSAVE[bit 26] = 0.
	If CR4.OSXSAVE[bit 18] = 0.
	If the LOCK prefix is used.
	If 66H, F3H or F2H prefix is used.

...

VZEROALL—Zero All YMM Registers

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
VEX.256.0F.WIG 77 VZEROALL	A	V/V	AVX	Zero all YMM registers.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
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. 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[VLMAX-1:0] ← 0
    YMM1[VLMAX-1:0] ← 0
    YMM2[VLMAX-1:0] ← 0
    YMM3[VLMAX-1:0] ← 0
    YMM4[VLMAX-1:0] ← 0
    YMM5[VLMAX-1:0] ← 0
    YMM6[VLMAX-1:0] ← 0
    YMM7[VLMAX-1:0] ← 0
    YMM8[VLMAX-1:0] ← 0
    YMM9[VLMAX-1:0] ← 0
    YMM10[VLMAX-1:0] ← 0
    YMM11[VLMAX-1:0] ← 0
    YMM12[VLMAX-1:0] ← 0
    YMM13[VLMAX-1:0] ← 0
    YMM14[VLMAX-1:0] ← 0
    YMM15[VLMAX-1:0] ← 0
ELSE
    YMM0[VLMAX-1:0] ← 0
    YMM1[VLMAX-1:0] ← 0
    YMM2[VLMAX-1:0] ← 0
    YMM3[VLMAX-1:0] ← 0
    YMM4[VLMAX-1:0] ← 0
    YMM5[VLMAX-1:0] ← 0
    YMM6[VLMAX-1:0] ← 0
    YMM7[VLMAX-1:0] ← 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.

...

VZEROUPPER—Zero Upper Bits of YMM Registers

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
VEX.128.0F.WIG 77 VZEROUPPER	A	V/V	AVX	Zero upper 128 bits of all YMM registers.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
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. In Compatibility and legacy 32-bit mode only the lower 8 registers are modified.

Operation

VZEROUPPER

```
IF (64-bit mode)
    YMM0[VLMAX-1:128] ← 0
    YMM1[VLMAX-1:128] ← 0
    YMM2[VLMAX-1:128] ← 0
    YMM3[VLMAX-1:128] ← 0
    YMM4[VLMAX-1:128] ← 0
    YMM5[VLMAX-1:128] ← 0
    YMM6[VLMAX-1:128] ← 0
    YMM7[VLMAX-1:128] ← 0
    YMM8[VLMAX-1:128] ← 0
    YMM9[VLMAX-1:128] ← 0
    YMM10[VLMAX-1:128] ← 0
    YMM11[VLMAX-1:128] ← 0
    YMM12[VLMAX-1:128] ← 0
    YMM13[VLMAX-1:128] ← 0
    YMM14[VLMAX-1:128] ← 0
    YMM15[VLMAX-1:128] ← 0
ELSE
    YMM0[VLMAX-1:128] ← 0
    YMM1[VLMAX-1:128] ← 0
    YMM2[VLMAX-1:128] ← 0
    YMM3[VLMAX-1:128] ← 0
    YMM4[VLMAX-1:128] ← 0
    YMM5[VLMAX-1:128] ← 0
    YMM6[VLMAX-1:128] ← 0
```

```
    YMM7[VLMAX-1:128] ← 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.

...

7. Updates to Chapter 5, Volume 2B

Change bars show changes to Chapter 5 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2B:* Instruction Set Reference, N-Z.

...

VMCLEAR—Clear Virtual-Machine Control Structure

...

Operation

```
IF (register operand) or (not in VMX operation) or (CR0.PE = 0) or (RFLAGS.VM = 1) or (IA32_EFER.LMA =
1 and CS.L = 0)
    THEN #UD;
ELSIF in VMX non-root operation
    THEN VM exit;
ELSIF CPL > 0
    THEN #GP(0);
    ELSE
        addr ← contents of 64-bit in-memory operand;
        IF addr is not 4KB-aligned OR
        addr sets any bits beyond the physical-address width[1]
            THEN VMfail(VMCLEAR with invalid physical address);
        ELSIF addr = VMXON pointer
            THEN VMfail(VMCLEAR with VMXON pointer);
            ELSE
                ensure that data for VMCS referenced by the operand is in memory;
                initialize implementation-specific data in VMCS region;
                launch state of VMCS referenced by the operand ← "clear"
```

1. If IA32_VMX_BASIC[48] is read as 1, VMfail occurs if addr sets any bits in the range 63:32; see Appendix G.1.

```
                IF operand addr = current-VMCS pointer
                    THEN current-VMCS pointer ← FFFFFFFF_FFFFFFFFH;
                FI;
                VMsucceed;
        FI;
FI;
```

…

VMPTRLD—Load Pointer to Virtual-Machine Control Structure

…

Operation

```
IF (register operand) or (not in VMX operation) or (CR0.PE = 0) or (RFLAGS.VM = 1) or (IA32_EFER.LMA =
1 and CS.L = 0)
    THEN #UD;
ELSIF in VMX non-root operation
    THEN VMexit;
ELSIF CPL > 0
    THEN #GP(0);
    ELSE
        addr ← contents of 64-bit in-memory source operand;
        IF addr is not 4KB-aligned OR
        addr sets any bits beyond the physical-address width[1]
            THEN VMfail(VMPTRLD with invalid physical address);
        ELSIF addr = VMXON pointer
            THEN VMfail(VMPTRLD with VMXON pointer);
            ELSE
                rev ← 32 bits located at physical address addr;
                IF rev ≠ VMCS revision identifier supported by processor
                    THEN VMfail(VMPTRLD with incorrect VMCS revision identifier);
                    ELSE
                        current-VMCS pointer ← addr;
                        VMsucceed;
                FI;
        FI;
FI;
```

…

VMXOFF—Leave VMX Operation

…

Operation

```
IF (not in VMX operation) or (CR0.PE = 0) or (RFLAGS.VM = 1) or (IA32_EFER.LMA = 1 and CS.L = 0)
```

1. If IA32_VMX_BASIC[48] is read as 1, VMfail occurs if addr sets any bits in the range 63:32; see Appendix G.1.

```
    THEN #UD;
ELSIF in VMX non-root operation
    THEN VMexit;
ELSIF CPL > 0
    THEN #GP(0);
ELSIF dual-monitor treatment of SMIs and SMM is active
    THEN VMfail(VMXOFF under dual-monitor treatment of SMIs and SMM);
    ELSE
        leave VMX operation;
        unblock INIT;
        IF IA32_SMM_MONITOR_CTL[2] = 0[1]
            THEN unblock SMIs;
        IF outside SMX operation[2]
            THEN unblock and enable A20M;
        FI;
        clear address-range monitoring;
        VMsucceed;
FI;
```

…

VMXON—Enter VMX Operation

…

Operation

```
IF (register operand) or (CR0.PE = 0) or (RFLAGS.VM = 1) or (IA32_EFER.LMA = 1 and CS.L = 0)
    THEN #UD;
ELSIF not in VMX operation
    THEN
        IF (CPL > 0) or (in A20M mode) or
        (the values of CR0 and CR4 are not supported in VMX operation[3]) or
        (bit 0 (lock bit) of IA32_FEATURE_CONTROL MSR is clear) or
        (in SMX operation[4] and bit 1 of IA32_FEATURE_CONTROL MSR is clear) or
        (outside SMX operation and bit 2 of IA32_FEATURE_CONTROL MSR is clear)
            THEN #GP(0);
            ELSE
```

---

1. Setting IA32_SMM_MONITOR_CTL[bit 2] to 1 prevents VMXOFF from unblocking SMIs regardless of the value of the register's value bit (bit 0). Not all processors allow this bit to be set to 1. Software should consult the VMX capability MSR IA32_VMX_MISC (see Appendix G.6) to determine whether this is allowed.
2. A logical processor is outside SMX operation if GETSEC[SENTER] has not been executed or if GETSEC[SEXIT] was executed after the last execution of GETSEC[SENTER]. See Chapter 6, "Safer Mode Extensions Reference."
3. See Section 19.8 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3B*.
4. A logical processor is in SMX operation if GETSEC[SEXIT] has not been executed since the last execution of GETSEC[SENTER]. A logical processor is outside SMX operation if GETSEC[SENTER] has not been executed or if GETSEC[SEXIT] was executed after the last execution of GETSEC[SENTER]. See Chapter 6, "Safer Mode Extensions Reference."

```
                    addr ← contents of 64-bit in-memory source operand;
                    IF addr is not 4KB-aligned or
                    addr sets any bits beyond the physical-address width[1]
                        THEN VMfailInvalid;
                        ELSE
                            rev ← 32 bits located at physical address addr;
                            IF rev ≠ VMCS revision identifier supported by processor
                                THEN VMfailInvalid;
                                ELSE
                                    current-VMCS pointer ← FFFFFFFF_FFFFFFFFH;
                                    enter VMX operation;
                                    block INIT signals;
                                    block and disable A20M;
                                    clear address-range monitoring;
                                    VMsucceed;
                            FI;
                    FI;
            FI;
ELSIF in VMX non-root operation
    THEN VMexit;
ELSIF CPL > 0
    THEN #GP(0);
    ELSE VMfail("VMXON executed in VMX root operation");
FI;

…
```

8. Updates to Chapter 6, Volume 2B

Change bars show changes to Chapter 5 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2B:* Instruction Set Reference, N-Z.

…

GETSEC[ENTERACCS] - Execute Authenticated Chipset Code

…

Table 6-4 Register State Initialization after GETSEC[ENTERACCS]

Register State	Initialization Status	Comment
CR0	PG←0, AM←0, WP←0: Others unchanged	Paging, Alignment Check, Write-protection are disabled
CR4	MCE←0: Others unchanged	Machine Check Exceptions Disabled
EFLAGS	00000002H	

1. If IA32_VMX_BASIC[48] is read as 1, VMfailInvalid occurs if addr sets any bits in the range 63:32; see Appendix G.1.

Table 6-4 Register State Initialization after GETSEC[ENTERACCS] (Continued)

Register State	Initialization Status	Comment
IA32_EFER	0H	IA-32e mode disabled
EIP	AC.base + EntryPoint	AC.base is in EBX as input to GETSEC[ENTERACCS]
[E|R]BX	Pre-ENTERACCS state: Next [E|R]IP prior to GETSEC[ENTERACCS]	Carry forward 64-bit processor state across GETSEC[ENTERACCS]
ECX	Pre-ENTERACCS state: [31:16]=GDTR.limit; [15:0]=CS.sel	Carry forward processor state across GETSEC[ENTERACCS]
[E|R]DX	Pre-ENTERACCS state: GDTR base	Carry forward 64-bit processor state across GETSEC[ENTERACCS]
EBP	AC.base	
CS	Sel=[SegSel], base=0, limit=FFFFFh, G=1, D=1, AR=9BH	
DS	Sel=[SegSel] +8, base=0, limit=FFFFFh, G=1, D=1, AR=93H	
GDTR	Base= AC.base (EBX) + [GDTBasePtr], Limit=[GDTLimit]	
DR7	00000400H	
IA32_DEBUGCTL	0H	
IA32_MISC_ENABLE	see Table 6-5 for example	The number of initialized fields may change due.to processor implementation

The segmentation related processor state that has not been initialized by GETSEC[ENTERACCS] requires appropriate initialization before use. Since a new GDT context has been established, the previous state of the segment selector values held in ES, SS, FS, GS, TR, and LDTR might not be valid.

The MSR IA32_EFER is also unconditionally cleared as part of the processor state initialized by ENTERACCS. Since paging is disabled upon entering authenticated code execution mode, a new paging environment will have to be reestablished in order to establish IA-32e mode while operating in authenticated code execution mode.

Debug exception and trap related signaling is also disabled as part of GETSEC[ENTERACCS]. This is achieved by resetting DR7, TF in EFLAGs, and the MSR IA32_DEBUGCTL. These debug functions are free to be re-enabled once supporting exception handler(s), descriptor tables, and debug registers have been properly initialized following entry into authenticated code execution mode. Also, any pending single-step trap condition will have been cleared upon entry into this mode.

The IA32_MISC_ENABLE MSR is initialized upon entry into authenticated execution mode. Certain bits of this MSR are preserved because preserving these bits may be important to maintain previously established platform settings (See the footnote for Table 6-5.). The remaining bits are cleared for the purpose of establishing a more consistent environment for the execution of authenticated code modules. One of the impacts of

initializing this MSR is any previous condition established by the MONITOR instruction will be cleared.

To support the possible return to the processor architectural state prior to execution of GETSEC[ENTERACCS], certain critical processor state is captured and stored in the general- purpose registers at instruction completion. [E|R]BX holds effective address ([E|R]IP) of the instruction that would execute next after GETSEC[ENTERACCS], ECX[15:0] holds the CS selector value, ECX[31:16] holds the GDTR limit field, and [E|R]DX holds the GDTR base field. The subsequent authenticated code can preserve the contents of these registers so that this state can be manually restored if needed, prior to exiting authenticated code execution mode with GETSEC[EXITAC]. For the processor state after exiting authenticated code execution mode, see the description of GETSEC[SEXIT].

Table 6-5 IA32_MISC_ENALBES MSR Initialization[1] by ENTERACCS and SENTER

Field	Bit position	Description
Fast strings enable	0	Clear to 0
FOPCODE compatibility mode enable	2	Clear to 0
Thermal monitor enable	3	Set to 1 if other thermal monitor capability is not enabled.[2]
Split-lock disable	4	Clear to 0
Bus lock on cache line splits disable	8	Clear to 0
Hardware prefetch disable	9	Clear to 0
GV1/2 legacy enable	15	Clear to 0
MONITOR/MWAIT s/m enable	18	Clear to 0
Adjacent sector prefetch disable	19	Clear to 0

NOTES:

1. The number of IA32_MISC_ENABLE fields that are initialized may vary due to processor implementations.
2. ENTERACCS (and SENTER) initialize the state of processor thermal throttling such that at least a minimum level is enabled. If thermal throttling is already enabled when executing one of these GETSEC leaves, then no change in the thermal throttling control settings will occur. If thermal throttling is disabled, then it will be enabled via setting of the thermal throttle control bit 3 as a result of executing these GETSEC leaves.

…

Operation in a Uni-Processor Platform

```
(* The state of the internal flag ACMODEFLAG persists across instruction boundary *)
IF (CR4.SMXE=0)
    THEN #UD;
```

```
ELSIF (in VMX non-root operation)
    THEN VM Exit (reason="GETSEC instruction");
ELSIF (GETSEC leaf unsupported)
    THEN #UD;
ELSIF ((in VMX operation) or
    (CR0.PE=0) or (CR0.CD=1) or (CR0.NW=1) or (CR0.NE=0) or
    (CPL>0) or (EFLAGS.VM=1) or
    (IA32_APIC_BASE.BSP=0) or
    (TXT chipset not present) or
    (ACMODEFLAG=1) or (IN_SMM=1))
        THEN #GP(0);
IF (GETSEC[PARAMETERS].Parameter_Type = 5, MCA_Handling (bit 6) = 0)
    FOR I = 0 to IA32_MCG_CAP.COUNT-1 DO
        IF (IA32_MC[I]_STATUS = uncorrectable error)
            THEN #GP(0);
    OD;
FI;
IF (IA32_MCG_STATUS.MCIP=1) or (IERR pin is asserted)
    THEN #GP(0);
ACBASE← EBX;
ACSIZE← ECX;
IF (((ACBASE MOD 4096) != 0) or ((ACSIZE MOD 64 )!= 0 ) or (ACSIZE < minimum module size) OR (ACSIZE
> authenticated RAM capacity)) or ((ACBASE+ACSIZE) > (2^32 -1)))
    THEN #GP(0);
IF (secondary thread(s) CR0.CD = 1) or ((secondary thread(s) NOT(wait-for-SIPI)) and
    (secondary thread(s) not in SENTER sleep state)
    THEN #GP(0);
Mask SMI, INIT, A20M, and NMI external pin events;
IA32_MISC_ENABLE← (IA32_MISC_ENABLE & MASK_CONST*)
(* The hexadecimal value of MASK_CONST may vary due to processor implementations *)
A20M← 0;
IA32_DEBUGCTL← 0;
Invalidate processor TLB(s);
Drain Outgoing Transactions;
ACMODEFLAG← 1;
SignalTXTMessage(ProcessorHold);
Load the internal ACRAM based on the AC module size;
(* Ensure that all ACRAM loads hit Write Back memory space *)
IF (ACRAM memory type != WB)
    THEN TXT-SHUTDOWN(#BadACMMType);
IF (AC module header version isnot supported) OR (ACRAM[ModuleType] <> 2)
    THEN TXT-SHUTDOWN(#UnsupportedACM);
 (* Authenticate the AC Module and shutdown with an error if it fails *)
KEY← GETKEY(ACRAM, ACBASE);
KEYHASH← HASH(KEY);
CSKEYHASH← READ(TXT.PUBLIC.KEY);
IF (KEYHASH <> CSKEYHASH)
    THEN TXT-SHUTDOWN(#AuthenticateFail);
SIGNATURE← DECRYPT(ACRAM, ACBASE, KEY);
(* The value of SIGNATURE_LEN_CONST is implementation-specific*)
```

```
FOR I=0 to SIGNATURE_LEN_CONST - 1 DO
    ACRAM[SCRATCH.I]← SIGNATURE[I];
COMPUTEDSIGNATURE← HASH(ACRAM, ACBASE, ACSIZE);
FOR I=0 to SIGNATURE_LEN_CONST - 1 DO
    ACRAM[SCRATCH.SIGNATURE_LEN_CONST+I]← COMPUTEDSIGNATURE[I];
IF (SIGNATURE<>COMPUTEDSIGNATURE)
    THEN TXT-SHUTDOWN(#AuthenticateFail);
ACMCONTROL← ACRAM[CodeControl];
IF ((ACMCONTROL.0 = 0) and (ACMCONTROL.1 = 1) and (snoop hit to modified line detected on ACRAM
load))
    THEN TXT-SHUTDOWN(#UnexpectedHITM);
IF (ACMCONTROL reserved bits are set)
    THEN TXT-SHUTDOWN(#BadACMFormat);
IF ((ACRAM[GDTBasePtr] < (ACRAM[HeaderLen] * 4 + Scratch_size)) OR
    ((ACRAM[GDTBasePtr] + ACRAM[GDTLimit]) >= ACSIZE))
    THEN TXT-SHUTDOWN(#BadACMFormat);
IF ((ACMCONTROL.0 = 1) and (ACMCONTROL.1 = 1) and (snoop hit to modified line detected on ACRAM
load))
    THEN ACEntryPoint← ACBASE+ACRAM[ErrorEntryPoint];
ELSE
    ACEntryPoint← ACBASE+ACRAM[EntryPoint];
IF ((ACEntryPoint >= ACSIZE) OR (ACEntryPoint < (ACRAM[HeaderLen] * 4 + Scratch_size)))THEN TXT-
SHUTDOWN(#BadACMFormat);
IF (ACRAM[GDTLimit] & FFFF0000h)
    THEN TXT-SHUTDOWN(#BadACMFormat);
IF ((ACRAM[SegSel] > (ACRAM[GDTLimit] - 15)) OR (ACRAM[SegSel] < 8))
    THEN TXT-SHUTDOWN(#BadACMFormat);
IF ((ACRAM[SegSel].TI=1) OR (ACRAM[SegSel].RPL!=0))
    THEN TXT-SHUTDOWN(#BadACMFormat);
CR0.[PG.AM.WP]← 0;
CR4.MCE← 0;
EFLAGS← 00000002h;
IA32_EFER← 0h;
[E|R]BX← [E|R]IP of the instruction after GETSEC[ENTERACCS];
ECX← Pre-GETSEC[ENTERACCS] GDT.limit:CS.sel;
[E|R]DX← Pre-GETSEC[ENTERACCS] GDT.base;
EBP← ACBASE;
GDTR.BASE← ACBASE+ACRAM[GDTBasePtr];
GDTR.LIMIT← ACRAM[GDTLimit];
CS.SEL← ACRAM[SegSel];
CS.BASE← 0;
CS.LIMIT← FFFFFh;
CS.G← 1;
CS.D← 1;
CS.AR← 9Bh;
DS.SEL← ACRAM[SegSel]+8;
DS.BASE← 0;
DS.LIMIT← FFFFFh;
DS.G← 1;
DS.D← 1;
```

DS.AR← 93h;
DR7← 00000400h;
IA32_DEBUGCTL← 0;
SignalTXTMsg(OpenPrivate);
SignalTXTMsg(OpenLocality3);
EIP← ACEntryPoint;
END;

…

GETSEC[SENTER]—Enter a Measured Environment

…

Table 6-6 Register State Initialization after GETSEC[SENTER] and GETSEC[WAKEUP]

Register State	ILP after GETSEC[SENTER]	RLP after GETSEC[WAKEUP]
CR0	PG←0, AM←0, WP←0; Others unchanged	PG←0, CD←0, NW←0, AM←0, WP←0; PE←1, NE←1
CR4	00004000H	00004000H
EFLAGS	00000002H	00000002H
IA32_EFER	0H	0
EIP	[EntryPoint from MLE header[1]]	[LT.MLE.JOIN + 12]
EBX	Unchanged [SINIT.BASE]	Unchanged
EDX	SENTER control flags	Unchanged
EBP	SINIT.BASE	Unchanged
CS	Sel=[SINIT SegSel], base=0, limit=FFFFFh, G=1, D=1, AR=9BH	Sel = [LT.MLE.JOIN + 8], base = 0, limit = FFFFFH, G = 1, D = 1, AR = 9BH
DS, ES, SS	Sel=[SINIT SegSel] +8, base=0, limit=FFFFFh, G=1, D=1, AR=93H	Sel = [LT.MLE.JOIN + 8] +8, base = 0, limit = FFFFFH, G = 1, D = 1, AR = 93H
GDTR	Base= SINIT.base (EBX) + [SINIT.GDTBasePtr], Limit=[SINIT.GDTLimit]	Base = [LT.MLE.JOIN + 4], Limit = [LT.MLE.JOIN]
DR7	00000400H	00000400H
IA32_DEBUGCTL	0H	0H

Table 6-6 Register State Initialization after GETSEC[SENTER] and GETSEC[WAKEUP]

Performance counters and counter control registers	0H	0H
IA32_MISC_ENABLE	See Table 6-5	See Table 6-5
IA32_SMM_MONITOR_CTL	Bit 2←0	Bit 2←0

NOTES:
1. See *Intel® Trusted Execution Technology Measured Launched Environment Programming Guide* for MLE header format.

Segmentation related processor state that has not been initialized by GETSEC[SENTER] requires appropriate initialization before use. Since a new GDT context has been established, the previous state of the segment selector values held in FS, GS, TR, and LDTR may no longer be valid. The IDTR will also require reloading with a new IDT context after launching the measured environment before exceptions or the external interrupts INTR and NMI can be handled. In the meantime, the programmer must take care in not executing an INT n instruction or any other condition that would result in an exception or trap signaling.

Debug exception and trap related signaling is also disabled as part of execution of GETSEC[SENTER]. This is achieved by clearing DR7, TF in EFLAGs, and the MSR IA32_DEBUGCTL as defined in Table 6-6. These can be re-enabled once supporting exception handler(s), descriptor tables, and debug registers have been properly re-initialized following SENTER. Also, any pending single-step trap condition will be cleared at the completion of SENTER for both the ILP and RLP(s).

Performance related counters and counter control registers are cleared as part of execution of SENTER on both the ILP and RLP. This implies any active performance counters at the time of SENTER execution will be disabled. To reactive the processor performance counters, this state must be re-initialized and re-enabled.

Since MCE along with all other state bits (with the exception of SMXE) are cleared in CR4 upon execution of SENTER processing, any enabled machine check error condition that occurs will result in the processor performing the TXT-shutdown action. This also applies to an RLP while in the SENTER sleep state. For each logical processor CR4.MCE must be reestablished with a valid machine check exception handler to otherwise avoid an TXT-shutdown under such conditions.

The MSR IA32_EFER is also unconditionally cleared as part of the processor state initialized by SENTER for both the ILP and RLP. Since paging is disabled upon entering authenticated code execution mode, a new paging environment will have to be re-established if it is desired to enable IA-32e mode while operating in authenticated code execution mode.

The miscellaneous feature control MSR, IA32_MISC_ENABLE, is initialized as part of the measured environment launch. Certain bits of this MSR are preserved because preserving these bits may be important to maintain previously established platform settings. See the footnote for Table 6-5 The remaining bits are cleared for the purpose of establishing a more consistent environment for the execution of authenticated code modules. Among the impact of initializing this MSR, any previous condition established by the MONITOR instruction will be cleared.

…

Operation in a Uni-Processor Platform

(* The state of the internal flag ACMODEFLAG and SENTERFLAG persist across instruction boundary *)

GETSEC[SENTER] (ILP only):

```
IF (CR4.SMXE=0)
    THEN #UD;
ELSE IF (in VMX non-root operation)
    THEN VM Exit (reason="GETSEC instruction");
ELSE IF (GETSEC leaf unsupported)
    THEN #UD;
ELSE IF ((in VMX root operation) or
    (CR0.PE=0) or (CR0.CD=1) or (CR0.NW=1) or (CR0.NE=0) or
    (CPL>0) or (EFLAGS.VM=1) or
    (IA32_APIC_BASE.BSP=0) or (TXT chipset not present) or
    (SENTERFLAG=1) or (ACMODEFLAG=1) or (IN_SMM=1) or
    (TPM interface is not present) or
    (EDX != (SENTER_EDX_support_mask & EDX)) or
    (IA32_CR_FEATURE_CONTROL[0]=0) or (IA32_CR_FEATURE_CONTROL[15]=0) or
    ((IA32_CR_FEATURE_CONTROL[14:8] & EDX[6:0]) != EDX[6:0]))
        THEN #GP(0);
IF (GETSEC[PARAMETERS].Parameter_Type = 5, MCA_Handling (bit 6) = 0)
    FOR I = 0 to IA32_MCG_CAP.COUNT-1 DO
        IF IA32_MC[I]_STATUS = uncorrectable error
            THEN #GP(0);
        FI;
    OD;
FI;
IF (IA32_MCG_STATUS.MCIP=1) or (IERR pin is asserted)
    THEN #GP(0);
ACBASE← EBX;
ACSIZE← ECX;
IF (((ACBASE MOD 4096) != 0) or ((ACSIZE MOD 64) != 0 ) or (ACSIZE < minimum
    module size) or (ACSIZE > AC RAM capacity) or ((ACBASE+ACSIZE) > (2^32 -1)))
        THEN #GP(0);
Mask SMI, INIT, A20M, and NMI external pin events;
SignalTXTMsg(SENTER);
DO
WHILE (no SignalSENTER message);
```

…

GETSEC[PARAMETERS]—Report the SMX Parameters

…

Operation

(* example of a processor supporting only a 0.0 HeaderVersion, 32K ACRAM size, memory types UC and WC *)

```
IF (CR4.SMXE=0)
    THEN #UD;
```

```
ELSE IF (in VMX non-root operation)
    THEN VM Exit (reason="GETSEC instruction");
ELSE IF (GETSEC leaf unsupported)
    THEN #UD;
    (* example of a processor supporting a 0.0 HeaderVersion *)
IF (EBX=0) THEN
    EAX← 00000001h;
    EBX← FFFFFFFFh;
    ECX← 00000000h;
ELSE IF (EBX=1)
    (* example of a processor supporting a 32K ACRAM size *)
    THEN EAX← 00008002h;
ESE IF (EBX= 2)
    (* example of a processor supporting external memory types of UC and WC *)
    THEN EAX← 00000303h;
ESE IF (EBX= other value(s) less than unsupported index value)
    (* EAX value varies. Consult Table 6-7 and Table 6-8*)
ELSE (* unsupported index*)
    EAX¨ 00000000h;
END;
```

…

GETSEC[WAKEUP]—Wake up sleeping processors in measured environment

…

Operation

```
(* The state of the internal flag ACMODEFLAG and SENTERFLAG persist across instruction boundary *)
IF (CR4.SMXE=0)
    THEN #UD;
ELSE IF (in VMX non-root operation)
    THEN VM Exit (reason="GETSEC instruction");
ELSE IF (GETSEC leaf unsupported)
    THEN #UD;
ELSE IF ((CR0.PE=0) or (CPL>0) or (EFLAGS.VM=1) or (SENTERFLAG=0) or (ACMODEFLAG=1) or
(IN_SMM=0) or (in VMX operation) or (IA32_APIC_BASE.BSP=0) or (TXT chipset not present))
    THEN #GP(0);
ELSE
    SignalTXTMsg(WAKEUP);
END;
```

RLP_SIPI_WAKEUP_FROM_SENTER_ROUTINE: (RLP only)

```
WHILE (no SignalWAKEUP event);
IF (IA32_SMM_MONITOR_CTL[0] != ILP.IA32_SMM_MONITOR_CTL[0])
    THEN TXT-SHUTDOWN(#IllegalEvent)
IF (IA32_SMM_MONITOR_CTL[0] = 0)
    THEN Unmask SMI pin event;
ELSE
```

```
    Mask SMI pin event;
Mask A20M, and NMI external pin events (unmask INIT);
Mask SignalWAKEUP event;
Invalidate processor TLB(s);
Drain outgoing transactions;
TempGDTRLIMIT← LOAD(LT.MLE.JOIN);
TempGDTRBASE← LOAD(LT.MLE.JOIN+4);
TempSegSel← LOAD(LT.MLE.JOIN+8);
TempEIP← LOAD(LT.MLE.JOIN+12);
IF (TempGDTLimit & FFFF0000h)
    THEN TXT-SHUTDOWN(#BadJOINFormat);
IF ((TempSegSel > TempGDTRLIMIT-15) or (TempSegSel < 8))
    THEN TXT-SHUTDOWN(#BadJOINFormat);
IF ((TempSegSel.TI=1) or (TempSegSel.RPL!=0))
    THEN TXT-SHUTDOWN(#BadJOINFormat);
CR0.[PG,CD,NW,AM,WP]← 0;
CR0.[NE,PE]← 1;
CR4← 00004000h;
EFLAGS← 00000002h;
IA32_EFER← 0;
GDTR.BASE← TempGDTRBASE;
GDTR.LIMIT← TempGDTRLIMIT;
CS.SEL← TempSegSel;
CS.BASE← 0;
CS.LIMIT← FFFFFh;
CS.G← 1;
CS.D← 1;
CS.AR← 9Bh;
DS.SEL← TempSegSel+8;
DS.BASE← 0;
DS.LIMIT← FFFFFh;
DS.G← 1;
DS.D← 1;
DS.AR← 93h;
SS← DS;
ES← DS;
DR7← 00000400h;
IA32_DEBUGCTL← 0;
EIP← TempEIP;
END;
```

…

9. Updates to Appendix A, Volume 2B

Change bars show changes to Chapter 5 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2B:* Instruction Set Reference, N-Z.

…

A.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).

K Mask registers: rK is reg field, mK is r/m field, vK is vvvv field.

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.

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).

A.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.

...

A.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), operand order (VEX.W), and operand size/opcode information (VEX.L). See section ### for details.

Opcode tables A2-A5 include instructions which do not include a VEX prefix. Opcode tables A6-A8 contain instructions which must include a VEX prefix. In the VEX opcode tables, where the VEX.pp, VEX.W, and VEX.L information differentiates instructions it is indicated by the general notation (vPPwL). Non-VEX-encoded instructions that require mandatory 66H, F2H, F3H prefix functionality before the opcode have used the form (PP), PP = 66, F2, F3, or white space. The parts of the VEX vPPwL notation should be interpreted as follows:

v - Indicates a VEX prefix encoding is required (vpsrlw in table A9). “v” is not shown but a VEX prefix is required for instructions in separate VEX tables (A6, A7 and A8).

PP - The value of none, 66, F2, or F3, indicates the VEX.pp field must be encoded accordingly. For non-VEX instructions this is with the legacy prefix. For VEX instructions this encoding is with the embedded “PP” field.

w - If the instruction uses VEX.W to determine operand order, the presence of the lower case 'w' indicates that the VEX.W=1 and the absence indicates VEX.W=0.

L - When VEX.L is used as an opcode extension rather than indicating operand size (VZEROALL, VZEROUPPER), presence of 'L' indicates VEX.L=1 and absence indicates VEX.L=0.

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'.

The entry "VMOVUPD Vx,Wx" indicates both VEX.L=0 and VEX.L=1 are supported, because VEX.L=0 and the operand notation Vx, indicate the destination operand can be 256-bit YMM register or 128-bit XMM register.

A.2.5 Superscripts Utilized in Opcode Tables

Table A-1 contains notes on particular encodings. These notes are indicated in the following opcode maps by superscripts. Gray cells indicate instruction groupings.

Table A-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 A.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).

Table A-1 Superscripts Utilized in Opcode Tables

Superscript Symbol	Meaning of Symbol
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 A-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.

…

Table A-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<OpMapSuper>i64	POP ES<OpMapSuper>i64
1	ADC Eb, Gb	Ev, Gv	Gb, Eb	Gv, Ev	AL, Ib	rAX, Iz	PUSH SS<OpMapSuper>i64	POP SS<OpMapSuper>i64
2	AND Eb, Gb	Ev, Gv	Gb, Eb	Gv, Ev	AL, Ib	rAX, Iz	SEG=ES (Prefix)	DAA<OpMapSuper>i64
3	XOR Eb, Gb	Ev, Gv	Gb, Eb	Gv, Ev	AL, Ib	rAX, Iz	SEG=SS (Prefix)	AAA<OpMapSuper>i64
4	INC<OpMapSuper>i64 general register / REX<OpMapSuper>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<OpMapSuper>d64 general register rAX/r8	rCX/r9	rDX/r10	rBX/r11	rSP/r12	rBP/r13	rSI/r14	rDI/r15
6	PUSHA<OpMapSuper>i64/ PUSHAD<OpMapSuper>i64	POPA<OpMapSuper>i64/ POPAD<OpMapSuper>i64	BOUND<OpMapSuper>i64 Gv, Ma	ARPL<OpMapSuper>i64 Ew, Gw MOVSXD<OpMapSuper>o64 Gv, Ev	SEG=FS (Prefix)	SEG=GS (Prefix)	Operand Size (Prefix)	Address Size (Prefix)
7	Jcc<OpMapSuper>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<OpMapSuper>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<OpMapSuper>f64 Iw	RETN<OpMapSuper>f64	LES<OpMapSuper>i64 Gz, MpVEX+2byte	LDS<OpMapSuper>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<OpMapSuper>i64 Ib	AAD<OpMapSuper>i64 Ib		XLAT/ XLATB
E	LOOPNE<OpMapSuper>f64 / LOOPNZ<OpMapSuper>f64 Jb	LOOPE<OpMapSuper>f64/ LOOPZ<OpMapSuper>f64 Jb	LOOP<OpMapSuper>f64 Jb	JrCXZ<OpMapSuper>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

...

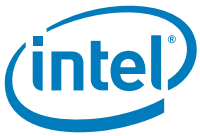
Table A-3 Two-byte Opcode Map: 00H — 77H (First Byte is 0FH) *

	pfx	0	1	2	3	4	5	6	7
0		Grp 6[1A]	Grp 7[1A]	LAR Gv, Ew	LSL Gv, Ew		SYSCALL<OpMapSuper>o64	CLTS	SYSRET<OpMapSuper>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<OpMapSuper>v1 Vdq, Hq, Mq vmovlhps Vdq, Hq, Uq	vmovhps<OpMapSuper>v1 Mq, Vq
	66	vmovupd	vmovupd Wpd,Vpd	vmovlpd Vq, Hq, Mq	vmovlpd Mq, Vq	vunpcklpd Vx,Hx,Wx	vunpckhpd Vx,Hx,Wx	vmovhpd<OpMapSuper>v1 Vdq, Hq, Mq	vmovhpd<OpMapSuper>v1 Mq, Vq
	F3	vmovss Vss, Wss Mss, Hss, Vss	vmovss Wss, Vss Mss, Hss, Vss	vmovsldup Vx, Wx				vmovshdup Vx, Wx	
	F2	vmovsd Vsd, Wsd Vsd, Hsd, Msd	vmovsd Vsd, Wsd Vsd, Hsd, Msd	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	emmsvzeroupper<OpMapSuper>v vzeroall(L)<OpMapSuper>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 A-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<OpMapSuper>v Ey, Vy	vmovdqa<OpMapSuper>v Wx,Vx
	F3							vmovq<OpMapSuper>v Vq, Wq	vmovdqu<OpMapSuper>v Wx,Vx
	F2					vhaddps Vps, Hps, Wps	vhsubps Vps, Hps, Wps		

Table A-3 Two-byte Opcode Map: 80H — F7H (First Byte is 0FH) *



	pfx	0	1	2	3	4	5	6	7
8		Jcc<OpMapSuper>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<OpMapSuper>d64 FS	POP<OpMapSuper>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 A-3 Two-byte Opcode Map: 88H — FFH (First Byte is 0FH) *

	pfx	8	9	A	B	C	D	E	F
8		Jcc<OpMapSuper>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<OpMapSuper>d64 GS	POP<OpMapSuper>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<OpMapSuper>v Vdq, Hdq, Wdq	vpsubsw<OpMapSuper>v Vdq, Hdq, Wdq	vpminsw<OpMapSuper>v Vdq, Hdq, Wdq	vpor<OpMapSuper>v Vdq, Hdq, Wdq	vpaddsb<OpMapSuper>v Vdq, Hdq, Wdq	vpaddsw<OpMapSuper>v Vdq, Hdq, Wdq	vpmaxsw<OpMapSuper>v Vdq, Hdq, Wdq	vpxor<OpMapSuper>v 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.

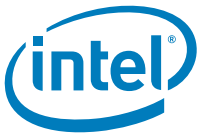


Table A-4 Three-byte Opcode Map: 00H — F7H (First Two Bytes are 0F 38H) *

	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				blendvps Vx, Wx	blendvpd Vx, 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 A-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<OpMapSuper>v Vx,Hx,Wx	vpermilpd<OpMapSuper>v Vx,Hx,Wx	vtestps<OpMapSuper>v Vx, Wx	vtestpd<OpMapSuper>v Vx, Wx
1						pabsb Pq, Qq	pabsw Pq, Qq	pabsd Pq, Qq	
	66	vbroadcastss<OpMapSuper>v Vx, Md	vbroadcastsd<OpMapSuper>v Vqq, Mq	vbroadcastf128<OpMapSuper>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<OpMapSuper>v Vx,Hx,Mx	vmaskmovpd<OpMapSuper>v Vx,Hx,Mx	vmaskmovps<OpMapSuper>v Mx,Vx,Hx	vmaskmovpd<OpMapSuper>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 A-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<OpMapSuper>v Vx, Wx, Ib	vpermilpd<OpMapSuper>v Vx, Wx, Ib	vperm2f128<OpMapSuper>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 A-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 Vss,Hss,Wss,Ib	vblendps Vx,Hx,Wx,Ib	vblendpd Vx,Hx,Wx,Ib	vpblendw Vdq,Hdq,Wdq,Ib	vpalignr Vdq,Hdq,Wdq,Ib
1	66	vinsertf128<OpMapSuper>v Vqq,Hqq,Wqq,Ib	vextractf128<OpMapSuper>v Wdq,Vqq,Ib						
2									
3									
4	66			vblendvps<OpMapSuper>vVx,Hx,Wx,Lx,Ib	vblendvpd<OpMapSuper>vVx,Hx,Wx,Lx,Ib	vpblendvb<OpMapSuper>vVdq,Hdq,Wdq,Ldq,Ib			
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.

A.4.2 Opcode Extension Tables

See Table A-6 below.

Table A-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<OpMapSuper>f64 Ev	CALLF Ep	JMPN<OpMapSuper>f64 Ev	JMPF Ep	PUSH<OpMapSuper>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<OpMapSuper>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									
0F B9	10	mem									
		11B									
C6	11	mem, 11B		MOV Eb, Ib							
C7		mem		MOV Ev, Iz							
		11B									

Table A-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
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.

...

10. Updates to Chapter 2, Volume 3A

Change bars show changes to Chapter 2 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A:* System Programming Guide, Part 1.

...

2.5 CONTROL REGISTERS

...

PCD **Page-level Cache Disable (bit 4 of CR3)** — Controls the memory type used to access the first paging structure of the current paging-structure hierarchy. See Section 4.9, "Paging and Memory Typing". This bit is not used if paging is disabled, with PAE paging, or with IA-32e paging if CR4.PCIDE=1.

PWT **Page-level Write-Through (bit 3 of CR3)** — Controls the memory type used to access the first paging structure of the current paging-structure hierarchy. See Section 4.9, "Paging and Memory Typing". This bit is not used if paging is disabled, with PAE paging, or with IA-32e paging if CR4.PCIDE=1.

...

OSXSAVE

XSAVE and Processor Extended States-Enable Bit (bit 18 of CR4) — When set, this flag: (1) indicates (via CPUID.01H:ECX.OSXSAVE[bit 27]) that the

operating system supports the use of the XGETBV, XSAVE and XRSTOR instructions by general software; (2) enables the XSAVE and XRSTOR instructions to save and restore the x87 FPU state (including MMX registers), the SSE state (XMM registers and MXCSR), along with other processor extended states enabled in XCR0; (3) enables the processor to execute XGETBV and XSETBV instructions in order to read and write XCR0. See Section 2.6 and Chapter 13, "System Programming for Instruction Set Extensions and Processor Extended States".

...

2.6 EXTENDED CONTROL REGISTERS (INCLUDING XCR0)

If CPUID.01H:ECX.XSAVE[bit 26] is 1, the processor supports one or more **extended control registers** (XCRs). Currently, the only such register defined is XCR0. This register specifies the set of processor states that the operating system enables on that processor, e.g. x87 FPU States, SSE states, and other processor extended states that Intel 64 architecture may introduce in the future. The OS programs XCR0 to reflect the features it supports.

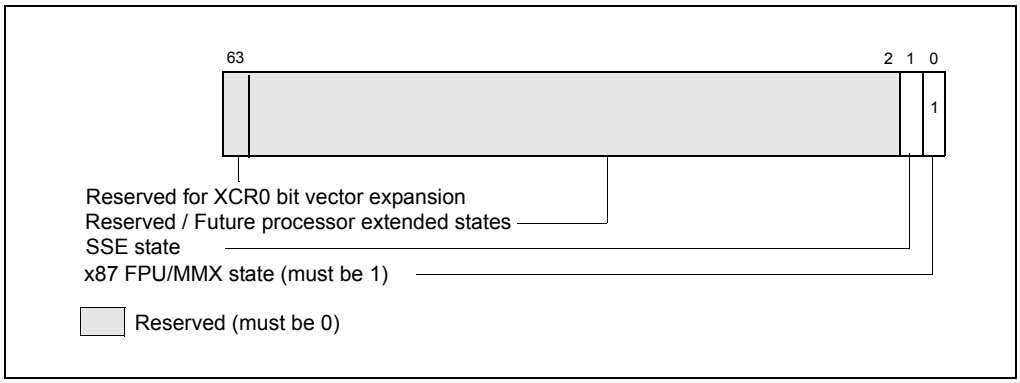


Figure 2-7 XCR0

Software can access XCR0 only if CR4.OSXSAVE[bit 18] = 1. (This bit is also readable as CPUID.01H:ECX.OSXSAVE[bit 27].) The layout of XCR0 is architected to allow software to use CPUID leaf function 0DH to enumerate the set of bits that the processor supports in XCR0 (see CPUID instruction in *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A*). Each processor state (X87 FPU state, SSE state, or a future processor extended state) is represented by a bit in XCR0. The OS can enable future processor extended states in a forward manner by specifying the appropriate bit mask value using the XSETBV instruction according to the results of the CPUID leaf 0DH.

With the exception of bit 63, each bit in XCR0 corresponds to a subset of the processor states. XCR0 thus provides space for up to 63 sets of processor state extensions. Bit 63 of XCR0 is reserved for future expansion and will not represent a processor extended state.

Currently, XCR0 has two processor states defined, with up to 61 bits reserved for future processor extended states:

- XCR0.X87 (bit 0): If 1, indicates x87 FPU state (including MMX register states) is supported in the processor. Bit 0 must be 1. An attempt to write 0 causes a #GP exception.
- XCR0.SSE (bit 1): If 1, indicates MXCSR and XMM registers (XMM0-XMM15 in 64-bit mode, otherwise XMM0-XMM7) are supported by XSAVE/XRESTOR in the processor.

Any attempt to set a reserved bit (as determined by the contents of EAX and EDX after executing CPUID with EAX=0DH, ECX= 0H) in XCR0 for a given processor will result in a #GP exception. An attempt to write 0 to XCR0.x87 (bit 0) will result in a #GP exception.

If a bit in XCR0 is 1, XSAVE instruction can selectively (in conjunction with a save mask) save a partial or full set of processor states to memory (See XSAVE instruction in *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2B*).

After reset, all bits (except bit 0) in XCR0 are cleared to zero, XCR0[0] is set to 1.

...

Table 2-2 Summary of System Instructions

Instruction	Description	Useful to Application?	Protected from Application?
LLDT	Load LDT Register	No	Yes
SLDT	Store LDT Register	No	No
LGDT	Load GDT Register	No	Yes
SGDT	Store GDT Register	No	No
LTR	Load Task Register	No	Yes
STR	Store Task Register	No	No
LIDT	Load IDT Register	No	Yes
SIDT	Store IDT Register	No	No
MOV CR*n*	Load and store control registers	No	Yes
SMSW	Store MSW	Yes	No
LMSW	Load MSW	No	Yes
CLTS	Clear TS flag in CR0	No	Yes
ARPL	Adjust RPL	Yes[1, 5]	No
LAR	Load Access Rights	Yes	No
LSL	Load Segment Limit	Yes	No
VERR	Verify for Reading	Yes	No
VERW	Verify for Writing	Yes	No
MOV DR*n*	Load and store debug registers	No	Yes
INVD	Invalidate cache, no writeback	No	Yes
WBINVD	Invalidate cache, with writeback	No	Yes
INVLPG	Invalidate TLB entry	No	Yes
HLT	Halt Processor	No	Yes
LOCK (Prefix)	Bus Lock	Yes	No

Table 2-2 Summary of System Instructions (Continued)

Instruction	Description	Useful to Application?	Protected from Application?
RSM	Return from system management mode	No	Yes
RDMSR[3]	Read Model-Specific Registers	No	Yes
WRMSR[3]	Write Model-Specific Registers	No	Yes
RDPMC[4]	Read Performance-Monitoring Counter	Yes	Yes[2]
RDTSC[3]	Read Time-Stamp Counter	Yes	Yes[2]
RDTSCP[7]	Read Serialized Time-Stamp Counter	Yes	Yes[2]
XGETBV	Return the state of XCR0	Yes	No
XSETBV	Enable one or more processor extended states	No[6]	Yes

NOTES:

1. Useful to application programs running at a CPL of 1 or 2.
2. The TSD and PCE flags in control register CR4 control access to these instructions by application programs running at a CPL of 3.
3. These instructions were introduced into the IA-32 Architecture with the Pentium processor.
4. This instruction was introduced into the IA-32 Architecture with the Pentium Pro processor and the Pentium processor with MMX technology.
5. This instruction is not supported in 64-bit mode.
6. Application uses XGETBV to query which set of processor extended states are enabled.
7. RDTSCP is introduced in Intel Core i7 processor.

...

2.7.8 Enabling Processor Extended States

The XSETBV instruction is required to enable OS support of individual processor extended states in XCR0 (see Section 2.6).

...

11. Updates to Chapter 4, Volume 3A

Change bars show changes to Chapter 4 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A:* System Programming Guide, Part 1.

...

4.4.1 PDPTE Registers

...

The logical processor loads these registers from the PDPTEs in memory as part of certain executions the MOV to CR instruction:

- If PAE paging would be in use following an execution of MOV to CR0 or MOV to CR4 (see Section 4.1.1) and the instruction is modifying any of CR0.CD, CR0.NW, CR0.PG, CR4.PAE, CR4.PGE, or CR4.PSE; then the PDPTEs are loaded from the address in CR3.
- If MOV to CR3 is executed while the logical processor is using PAE paging, the PDPTEs are loaded from the address being loaded into CR3.
- If PAE paging is in use and a task switch changes the value of CR3, the PDPTEs are loaded from the address in the new CR3 value.
- Certain VMX transitions load the PDPTE registers. See Section 4.11.1.

Table 4-8 gives the format of a PDPTE. If any of the PDPTEs sets both the P flag (bit 0) and any reserved bit, the MOV to CR instruction causes a general-protection exception (#GP(0)) and the PDPTEs are not loaded.[1] As shown in Table 4-8, bits 2:1, 8:5, and 63:MAXPHYADDR are reserved in the PDPTEs.

Table 4-8 Format of a PAE Page-Directory-Pointer-Table Entry (PDPTE)

Bit Position(s)	Contents
0 (P)	Present; must be 1 to reference a page directory
2:1	Reserved (must be 0)
3 (PWT)	Page-level write-through; indirectly determines the memory type used to access the page directory referenced by this entry (see Section 4.9)
4 (PCD)	Page-level cache disable; indirectly determines the memory type used to access the page directory referenced by this entry (see Section 4.9)
8:5	Reserved (must be 0)
11:9	Ignored
(M–1):12	Physical address of 4-KByte aligned page directory referenced by this entry[1]
63:M	Reserved (must be 0)

NOTES:
1. M is an abbreviation for MAXPHYADDR, which is at most 52; see Section 4.1.4.

...

4.9.1 Paging and Memory Typing When the PAT is Not Supported (Pentium Pro and Pentium II Processors)

NOTE

The PAT is supported on all processors that support IA-32e paging. Thus, this section applies only to 32-bit paging and PAE paging.

1. On some processors, reserved bits are checked even in PDPTEs in which the P flag (bit 0) is 0.

If the PAT is not supported, paging contributes to memory typing in conjunction with the memory-type range registers (MTRRs) as specified in Table 11-6 in Section 11.5.2.1.

For any access to a physical address, the table combines the memory type specified for that physical address by the MTRRs with a PCD value and a PWT value. The latter two values are determined as follows:

- For an access to a PDE with 32-bit paging, the PCD and PWT values come from CR3.
- For an access to a PDE with PAE paging, the PCD and PWT values come from the relevant PDPTE register.
- For an access to a PTE, the PCD and PWT values come from the relevant PDE.
- For an access to the physical address that is the translation of a linear address, the PCD and PWT values come from the relevant PTE (if the translation uses a 4-KByte page) or the relevant PDE (otherwise).
- With PAE paging, the UC memory type is used when loading the PDPTEs (see Section 4.4.1).

4.9.2 Paging and Memory Typing When the PAT is Supported (Pentium III and More Recent Processor Families)

If the PAT is supported, paging contributes to memory typing in conjunction with the PAT and the memory-type range registers (MTRRs) as specified in Table 11-7 in Section 11.5.2.2.

The PAT is a 64-bit MSR (IA32_PAT; MSR index 277H) comprising eight (8) 8-bit entries (entry *i* comprises bits 8*i*+7:8*i* of the MSR).

For any access to a physical address, the table combines the memory type specified for that physical address by the MTRRs with a memory type selected from the PAT. Table 11-11 in Section 11.12.3 specifies how a memory type is selected from the PAT. Specifically, it comes from entry *i* of the PAT, where *i* is defined as follows:

- For an access to an entry in a paging structure whose address is in CR3 (e.g., the PML4 table with IA-32e paging):
  - For IA-32e paging with CR4.PCIDE = 1, *i* = 0.
  - Otherwise, *i* = 2*PCD+PWT, where the PCD and PWT values come from CR3.
- For an access to a PDE with PAE paging, *i* = 2*PCD+PWT, where the PCD and PWT values come from the relevant PDPTE register.
- For an access to a paging-structure entry X whose address is in another paging-structure entry Y, *i* = 2*PCD+PWT, where the PCD and PWT values come from Y.
- For an access to the physical address that is the translation of a linear address, *i* = 4*PAT+2*PCD+PWT, where the PAT, PCD, and PWT values come from the relevant PTE (if the translation uses a 4-KByte page), the relevant PDE (if the translation uses a 2-MByte page or a 4-MByte page), or the relevant PDPTE (if the translation uses a 1-GByte page).
- With PAE paging, the WB memory type is used when loading the PDPTEs (see Section 4.4.1).[1]

1. Some older IA-32 processors used the UC memory type when loading the PDPTEs. Some processors may use the UC memory type if CR0.CD = 1 or if the MTRRs are disabled. These behaviors are model-specific and not architectural.

4.9.3 Caching Paging-Related Information about Memory Typing

A processor may cache information from the paging-structure entries in TLBs and paging-structure caches (see Section 4.10). These structures may include information about memory typing. The processor may use memory-typing information from the TLBs and paging-structure caches instead of from the paging structures in memory.

This fact implies that, if software modifies a paging-structure entry to change the memory-typing bits, the processor might not use that change for a subsequent translation using that entry or for access to an affected linear address. See Section 4.10.4.2 for how software can ensure that the processor uses the modified memory typing.

4.10 CACHING TRANSLATION INFORMATION

The Intel-64 and IA-32 architectures may accelerate the address-translation process by caching data from the paging structures on the processor. Because the processor does not ensure that the data that it caches are always consistent with the structures in memory, it is important for software developers to understand how and when the processor may cache such data. They should also understand what actions software can take to remove cached data that may be inconsistent and when it should do so. This section provides software developers information about the relevant processor operation.

...

4.10.4.2 Recommended Invalidation

The following items provide some recommendations regarding when software should perform invalidations:

- If software modifies a paging-structure entry that identifies the final page frame for a page number (either a PTE or a paging-structure entry in which the PS flag is 1), it should execute INVLPG for any linear address with a page number whose translation uses that PTE.[1] (If the paging-structure entry may be used in the translation of different page numbers — see Section 4.10.3.3 — software should execute INVLPG for linear addresses with each of those page numbers; alternatively, it could use MOV to CR3 or MOV to CR4.)
- If software modifies a paging-structure entry that references another paging structure, it may use one of the following approaches depending upon the types and number of translations controlled by the modified entry:
  - — Execute INVLPG for linear addresses with each of the page numbers with translations that would use the entry. However, if no page numbers that would use the entry have translations (e.g., because the P flags are 0 in all entries in the paging structure referenced by the modified entry), it remains necessary to execute INVLPG at least once.
  - — Execute MOV to CR3 if the modified entry controls no global pages.
  - — Execute MOV to CR4 to modify CR4.PGE.
- If CR4.PCIDE = 1 and software modifies a paging-structure entry that does not map a page or in which the G flag (bit 8) is 0, additional steps are required if the entry may be used for PCIDs other than the current one. Any one of the following suffices:

1. One execution of INVLPG is sufficient even for a page with size greater than 4 KBytes.

— Execute MOV to CR4 to modify CR4.PGE, either immediately or before again using any of the affected PCIDs. For example, software could use different (previously unused) PCIDs for the processes that used the affected PCIDs.

— For each affected PCID, execute MOV to CR3 to make that PCID current (and to load the address of the appropriate PML4 table). If the modified entry controls no global pages and bit 63 of the source operand to MOV to CR3 was 0, no further steps are required. Otherwise, execute INVLPG for linear addresses with each of the page numbers with translations that would use the entry; if no page numbers that would use the entry have translations, execute INVLPG at least once.

- If software using PAE paging modifies a PDPTE, it should reload CR3 with the register's current value to ensure that the modified PDPTE is loaded into the corresponding PDPTE register (see Section 4.4.1).
- If the nature of the paging structures is such that a single entry may be used for multiple purposes (see Section 4.10.3.3), software should perform invalidations for all of these purposes. For example, if a single entry might serve as both a PDE and PTE, it may be necessary to execute INVLPG with two (or more) linear addresses, one that uses the entry as a PDE and one that uses it as a PTE. (Alternatively, software could use MOV to CR3 or MOV to CR4.)
- As noted in Section 4.10.2, the TLBs may subsequently contain multiple translations for the address range if software modifies the paging structures so that the page size used for a 4-KByte range of linear addresses changes. A reference to a linear address in the address range may use any of these translations.

  Software wishing to prevent this uncertainty should not write to a paging-structure entry in a way that would change, for any linear address, both the page size and either the page frame, access rights, or other attributes. It can instead use the following algorithm: first clear the P flag in the relevant paging-structure entry (e.g., PDE); then invalidate any translations for the affected linear addresses (see above); and then modify the relevant paging-structure entry to set the P flag and establish modified translation(s) for the new page size.
- Software should clear bit 63 of the source operand to a MOV to CR3 instruction that establishes a PCID that had been used earlier for a different linear-address space (e.g., with a different value in bits 51:12 of CR3). This ensures invalidation of any information that may have been cached for the previous linear-address space.

  This assumes that both linear-address spaces use the same global pages and that it is thus not necessary to invalidate any global TLB entries. If that is not the case, software should invalidate those entries by executing MOV to CR4 to modify CR4.PGE.

...

12. Updates to Chapter 7, Volume 3A

Change bars show changes to Chapter 7 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A:* System Programming Guide, Part 1.

...

7.3 TASK SWITCHING

The processor transfers execution to another task in one of four cases:

- The current program, task, or procedure executes a JMP or CALL instruction to a TSS descriptor in the GDT.
- The current program, task, or procedure executes a JMP or CALL instruction to a task-gate descriptor in the GDT or the current LDT.
- An interrupt or exception vector points to a task-gate descriptor in the IDT.
- The current task executes an IRET when the NT flag in the EFLAGS register is set.

JMP, CALL, and IRET instructions, as well as interrupts and exceptions, are all mechanisms for redirecting a program. The referencing of a TSS descriptor or a task gate (when calling or jumping to a task) or the state of the NT flag (when executing an IRET instruction) determines whether a task switch occurs.

The processor performs the following operations when switching to a new task:

1. Obtains the TSS segment selector for the new task as the operand of the JMP or CALL instruction, from a task gate, or from the previous task link field (for a task switch initiated with an IRET instruction).
2. Checks that the current (old) task is allowed to switch to the new task. Data-access privilege rules apply to JMP and CALL instructions. The CPL of the current (old) task and the RPL of the segment selector for the new task must be less than or equal to the DPL of the TSS descriptor or task gate being referenced. Exceptions, interrupts (except for interrupts generated by the INT *n* instruction), and the IRET instruction are permitted to switch tasks regardless of the DPL of the destination task-gate or TSS descriptor. For interrupts generated by the INT *n* instruction, the DPL is checked.
3. Checks that the TSS descriptor of the new task is marked present and has a valid limit (greater than or equal to 67H).
4. Checks that the new task is available (call, jump, exception, or interrupt) or busy (IRET return).
5. Checks that the current (old) TSS, new TSS, and all segment descriptors used in the task switch are paged into system memory.
6. If the task switch was initiated with a JMP or IRET instruction, the processor clears the busy (B) flag in the current (old) task's TSS descriptor; if initiated with a CALL instruction, an exception, or an interrupt: the busy (B) flag is left set. (See Table 7-2.)
7. If the task switch was initiated with an IRET instruction, the processor clears the NT flag in a temporarily saved image of the EFLAGS register; if initiated with a CALL or JMP instruction, an exception, or an interrupt, the NT flag is left unchanged in the saved EFLAGS image.
8. Saves the state of the current (old) task in the current task's TSS. The processor finds the base address of the current TSS in the task register and then copies the states of the following registers into the current TSS: all the general-purpose registers, segment selectors from the segment registers, the temporarily saved image of the EFLAGS register, and the instruction pointer register (EIP).
9. If the task switch was initiated with a CALL instruction, an exception, or an interrupt, the processor will set the NT flag in the EFLAGS loaded from the new task. If initiated with an IRET instruction or JMP instruction, the NT flag will reflect the state of NT in the EFLAGS loaded from the new task (see Table 7-2).

10. If the task switch was initiated with a CALL instruction, JMP instruction, an exception, or an interrupt, the processor sets the busy (B) flag in the new task's TSS descriptor; if initiated with an IRET instruction, the busy (B) flag is left set.
11. Loads the task register with the segment selector and descriptor for the new task's TSS.
12. The TSS state is loaded into the processor. This includes the LDTR register, the PDBR (control register CR3), the EFLAGS register, the EIP register, the general-purpose registers, and the segment selectors. A fault during the load of this state may corrupt architectural state.
13. The descriptors associated with the segment selectors are loaded and qualified. Any errors associated with this loading and qualification occur in the context of the new task and may corrupt architectural state.

NOTES

If all checks and saves have been carried out successfully, the processor commits to the task switch. If an unrecoverable error occurs in steps 1 through 11, the processor does not complete the task switch and insures that the processor is returned to its state prior to the execution of the instruction that initiated the task switch.

If an unrecoverable error occurs in step 12, architectural state may be corrupted, but an attempt will be made to handle the error in the prior execution environment. If an unrecoverable error occurs after the commit point (in step 13), the processor completes the task switch (without performing additional access and segment availability checks) and generates the appropriate exception prior to beginning execution of the new task.

If exceptions occur after the commit point, the exception handler must finish the task switch itself before allowing the processor to begin executing the new task. See Chapter 6, "Interrupt 10—Invalid TSS Exception (#TS)," for more information about the affect of exceptions on a task when they occur after the commit point of a task switch.

...

13. Updates to Chapter 8, Volume 3A

Change bars show changes to Chapter 8 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A:* System Programming Guide, Part 1.

...

8.2.4 Out-of-Order Stores For String Operations

The Intel Core 2 Duo, Intel Core, Pentium 4, and P6 family processors modify the processors operation during the string store operations (initiated with the MOVS and STOS instructions) to maximize performance. Once the "fast string" operations initial conditions are met (as described below), the processor will essentially operate on, from an external perspective, the string in a cache line by cache line mode. This results in the processor looping on issuing a cache-line read for the source address and an invalidation

on the external bus for the destination address, knowing that all bytes in the destination cache line will be modified, for the length of the string. In this mode interrupts will only be accepted by the processor on cache line boundaries. It is possible in this mode that the destination line invalidations, and therefore stores, will be issued on the external bus out of order.

Code dependent upon sequential store ordering should not use the string operations for the entire data structure to be stored. Data and semaphores should be separated. Order dependent code should use a discrete semaphore uniquely stored to after any string operations to allow correctly ordered data to be seen by all processors.

"Fast string" operation can be disabled by clearing the fast-string-enable bit (bit 0) of IA32_MISC_ENABLE MSR.

Initial conditions for "fast string" operations are implementation specific. Example conditions include:

- EDI and ESI must be 8-byte aligned for the Pentium III processor. EDI must be 8-byte aligned for the Pentium 4 processor.
- String operation must be performed in ascending address order.
- The initial operation counter (ECX) must be equal to or greater than 64.
- Source and destination must not overlap by less than a cache line (64 bytes, for Intel Core 2 Duo, Intel Core, Pentium M, and Pentium 4 processors; 32 bytes P6 family and Pentium processors).
- The memory type for both source and destination addresses must be either WB or WC.

NOTE

Initial conditions for "fast string" operation in future Intel 64 or IA-32 processor families may differ from above.

...

8.7.8 IA32_MISC_ENABLE MSR

The IA32_MISC_ENABLE MSR (MSR address 1A0H) is generally shared between the logical processors in a processor core supporting Intel Hyper-Threading Technology. However, some bit fields within IA32_MISC_ENABLE MSR may be duplicated per logical processor. The partition of shared or duplicated bit fields within IA32_MISC_ENABLE is implementation dependent. Software should program duplicated fields carefully on all logical processors in the system to ensure consistent behavior.

...

14. Updates to Chapter 10, Volume 3A

Change bars show changes to Chapter 10 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A:* System Programming Guide, Part 1.

...

Table 10-1 Local APIC Register Address Map

Address	Register Name	Software Read/Write
FEE0 0000H	Reserved	
FEE0 0010H	Reserved	
FEE0 0020H	Local APIC ID Register	Read/Write.
FEE0 0030H	Local APIC Version Register	Read Only.
FEE0 0040H	Reserved	
FEE0 0050H	Reserved	
FEE0 0060H	Reserved	
FEE0 0070H	Reserved	
FEE0 0080H	Task Priority Register (TPR)	Read/Write.
FEE0 0090H	Arbitration Priority Register[1] (APR)	Read Only.
FEE0 00A0H	Processor Priority Register (PPR)	Read Only.
FEE0 00B0H	EOI Register	Write Only.
FEE0 00C0H	Remote Read Register[1] (RRD)	Read Only
FEE0 00D0H	Logical Destination Register	Read/Write.
FEE0 00E0H	Destination Format Register	Read/Write (see Section 10.6.2.2).
FEE0 00F0H	Spurious Interrupt Vector Register	Read/Write (see Section 10.9.
FEE0 0100H	In-Service Register (ISR); bits 31:0	Read Only.
FEE0 0110H	In-Service Register (ISR); bits 63:32	Read Only.
FEE0 0120H	In-Service Register (ISR); bits 95:64	Read Only.
FEE0 0130H	In-Service Register (ISR); bits 127:96	Read Only.
FEE0 0140H	In-Service Register (ISR); bits 159:128	Read Only.
FEE0 0150H	In-Service Register (ISR); bits 191:160	Read Only.
FEE0 0160H	In-Service Register (ISR); bits 223:192	Read Only.
FEE0 0170H	In-Service Register (ISR); bits 255:224	Read Only.
FEE0 0180H	Trigger Mode Register (TMR); bits 31:0	Read Only.
FEE0 0190H	Trigger Mode Register (TMR); bits 63:32	Read Only.
FEE0 01A0H	Trigger Mode Register (TMR); bits 95:64	Read Only.
FEE0 01B0H	Trigger Mode Register (TMR); bits 127:96	Read Only.
FEE0 01C0H	Trigger Mode Register (TMR); bits 159:128	Read Only.
FEE0 01D0H	Trigger Mode Register (TMR); bits 191:160	Read Only.
FEE0 01E0H	Trigger Mode Register (TMR); bits 223:192	Read Only.
FEE0 01F0H	Trigger Mode Register (TMR); bits 255:224	Read Only.
FEE0 0200H	Interrupt Request Register (IRR); bits 31:0	Read Only.

Table 10-1 Local APIC Register Address Map (Continued)

Address	Register Name	Software Read/Write
FEE0 0210H	Interrupt Request Register (IRR); bits 63:32	Read Only.
FEE0 0220H	Interrupt Request Register (IRR); bits 95:64	Read Only.
FEE0 0230H	Interrupt Request Register (IRR); bits 127:96	Read Only.
FEE0 0240H	Interrupt Request Register (IRR); bits 159:128	Read Only.
FEE0 0250H	Interrupt Request Register (IRR); bits 191:160	Read Only.
FEE0 0260H	Interrupt Request Register (IRR); bits 223:192	Read Only.
FEE0 0270H	Interrupt Request Register (IRR); bits 255:224	Read Only.
FEE0 0280H	Error Status Register	Read Only.
FEE0 0290H through FEE0 02E0H	Reserved	
FEE0 02F0H	LVT CMCI Register	Read/Write.
FEE0 0300H	Interrupt Command Register (ICR); bits 0-31	Read/Write.
FEE0 0310H	Interrupt Command Register (ICR); bits 32-63	Read/Write.
FEE0 0320H	LVT Timer Register	Read/Write.
FEE0 0330H	LVT Thermal Sensor Register[2]	Read/Write.
FEE0 0340H	LVT Performance Monitoring Counters Register[3]	Read/Write.
FEE0 0350H	LVT LINT0 Register	Read/Write.
FEE0 0360H	LVT LINT1 Register	Read/Write.
FEE0 0370H	LVT Error Register	Read/Write.
FEE0 0380H	Initial Count Register (for Timer)	Read/Write.
FEE0 0390H	Current Count Register (for Timer)	Read Only.
FEE0 03A0H through FEE0 03D0H	Reserved	
FEE0 03E0H	Divide Configuration Register (for Timer)	Read/Write.
FEE0 03F0H	Reserved	

NOTES:

1. Not supported in the Pentium 4 and Intel Xeon processors. The Illegal Register Access bit (7) of the ESR will not be set when writing to these registers.
2. Introduced in the Pentium 4 and Intel Xeon processors. This APIC register and its associated function are implementation dependent and may not be present in future IA-32 or Intel 64 processors.
3. Introduced in the Pentium Pro processor. This APIC register and its associated function are implementation dependent and may not be present in future IA-32 or Intel 64 processors.

…

10.4.3 Enabling or Disabling the Local APIC

The local APIC can be enabled or disabled in either of two ways:

1. Using the APIC global enable/disable flag in the IA32_APIC_BASE MSR (MSR address 1BH; see Figure 10-5):
   - When IA32_APIC_BASE[11] is 0, the processor is functionally equivalent to an IA-32 processor without an on-chip APIC. The CPUID feature flag for the APIC (see Section 10.4.2, "Presence of the Local APIC") is also set to 0.
   - When IA32_APIC_BASE[11] is set to 0, processor APICs based on the 3-wire APIC bus cannot be generally re-enabled until a system hardware reset. The 3-wire bus loses track of arbitration that would be necessary for complete re-enabling. Certain APIC functionality can be enabled (for example: performance and thermal monitoring interrupt generation).
   - For processors that use Front Side Bus (FSB) delivery of interrupts, software may disable or enable the APIC by setting and resetting IA32_APIC_BASE[11]. A hardware reset is not required to re-start APIC functionality, if software guarantees no interrupt will be sent to the APIC as IA32_APIC_BASE[11] is cleared.
   - When IA32_APIC_BASE[11] is set to 0, prior initialization to the APIC may be lost and the APIC may return to the state described in Section 10.4.7.1, "Local APIC State After Power-Up or Reset."
2. Using the APIC software enable/disable flag in the spurious-interrupt vector register (see Figure 10-23):
   - If IA32_APIC_BASE[11] is 1, software can temporarily disable a local APIC at any time by clearing the APIC software enable/disable flag in the spurious-interrupt vector register (see Figure 10-23). The state of the local APIC when in this software-disabled state is described in Section 10.4.7.2, "Local APIC State After It Has Been Software Disabled."
   - When the local APIC is in the software-disabled state, it can be re-enabled at any time by setting the APIC software enable/disable flag to 1.

For the Pentium processor, the APICEN pin (which is shared with the PICD1 pin) is used during power-up or reset to disable the local APIC.

Note that each entry in the LVT has a mask bit that can be used to inhibit interrupts from being delivered to the processor from selected local interrupt sources (the LINT0 and LINT1 pins, the APIC timer, the performance-monitoring counters, the thermal sensor, and/or the internal APIC error detector).

10.4.4 Local APIC Status and Location

The status and location of the local APIC are contained in the IA32_APIC_BASE MSR (see Figure 10-5). MSR bit functions are described below:

- **BSP flag, bit 8** — Indicates if the processor is the bootstrap processor (BSP). See Section 8.4, "Multiple-Processor (MP) Initialization." Following a power-up or reset, this flag is set to 1 for the processor selected as the BSP and set to 0 for the remaining processors (APs).
- **APIC Global Enable flag, bit 11** — Enables or disables the local APIC (see Section 10.4.3, "Enabling or Disabling the Local APIC"). This flag is available in the Pentium 4, Intel Xeon, and P6 family processors. It is not guaranteed to be available or available at the same location in future Intel 64 or IA-32 processors.

- **APIC Base field, bits 12 through 35** — Specifies the base address of the APIC registers. This 24-bit value is extended by 12 bits at the low end to form the base address. This automatically aligns the address on a 4-KByte boundary. Following a power-up or reset, the field is set to FEE0 0000H.
- Bits 0 through 7, bits 9 and 10, and bits MAXPHYADDR[1] through 63 in the IA32_APIC_BASE MSR are reserved.

...

10.4.7 Local APIC State

The following sections describe the state of the local APIC and its registers following a power-up or reset, after the local APIC has been software disabled, following an INIT reset, and following an INIT-deassert message.

x2APIC will introduce 32-bit ID; see Section 10.12.

10.4.7.1 Local APIC State After Power-Up or Reset

Following a power-up or reset of the processor, the state of local APIC and its registers are as follows:

- The following registers are reset to all 0s:
  - IRR, ISR, TMR, ICR, LDR, and TPR
  - Timer initial count and timer current count registers
  - Divide configuration register
- The DFR register is reset to all 1s.
- The LVT register is reset to 0s except for the mask bits; these are set to 1s.
- The local APIC version register is not affected.
- The local APIC ID register is set to a unique APIC ID. (Pentium and P6 family processors only). The Arb ID register is set to the value in the APIC ID register.
- The spurious-interrupt vector register is initialized to 000000FFH. By setting bit 8 to 0, software disables the local APIC.
- If the processor is the only processor in the system or it is the BSP in an MP system (see Section 8.4.1, "BSP and AP Processors"); the local APIC will respond normally to INIT and NMI messages, to INIT# signals and to STPCLK# signals. If the processor is in an MP system and has been designated as an AP; the local APIC will respond the same as for the BSP. In addition, it will respond to SIPI messages. For P6 family processors only, an AP will not respond to a STPCLK# signal.

10.4.7.2 Local APIC State After It Has Been Software Disabled

When the APIC software enable/disable flag in the spurious interrupt vector register has been explicitly cleared (as opposed to being cleared during a power up or reset), the local APIC is temporarily disabled (see Section 10.4.3, "Enabling or Disabling the Local APIC"). The operation and response of a local APIC while in this software-disabled state is as follows:

1. The MAXPHYADDR is 36 bits for processors that do not support CPUID leaf 80000008H, or indicated by CPUID.80000008H:EAX[bits 7:0] for processors that support CPUID leaf 80000008H.

- The local APIC will respond normally to INIT, NMI, SMI, and SIPI messages.
- Pending interrupts in the IRR and ISR registers are held and require masking or handling by the CPU.
- The local APIC can still issue IPIs. It is software's responsibility to avoid issuing IPIs through the IPI mechanism and the ICR register if sending interrupts through this mechanism is not desired.
- The reception or transmission of any IPIs that are in progress when the local APIC is disabled are completed before the local APIC enters the software-disabled state.
- The mask bits for all the LVT entries are set. Attempts to reset these bits will be ignored.
- (For Pentium and P6 family processors) The local APIC continues to listen to all bus messages in order to keep its arbitration ID synchronized with the rest of the system.

10.4.7.3 Local APIC State After an INIT Reset ("Wait-for-SIPI" State)

An INIT reset of the processor can be initiated in either of two ways:

- By asserting the processor's INIT# pin.
- By sending the processor an INIT IPI (an IPI with the delivery mode set to INIT).

Upon receiving an INIT through either of these mechanisms, the processor responds by beginning the initialization process of the processor core and the local APIC. The state of the local APIC following an INIT reset is the same as it is after a power-up or hardware reset, except that the APIC ID and arbitration ID registers are not affected. This state is also referred to at the "wait-for-SIPI" state (see also: Section 8.4.2, "MP Initialization Protocol Requirements and Restrictions").

...

10.5.1 Local Vector Table

The local vector table (LVT) allows software to specify the manner in which the local interrupts are delivered to the processor core. It consists of the following 32-bit APIC registers (see Figure 10-8), one for each local interrupt:

- **LVT CMCI Register (FEE0 02F0H)** — Specifies interrupt delivery when an overflow condition of corrected machine check error count reaching a threshold value occurred in a machine check bank supporting CMCI (see Section 15.5.1, "CMCI Local APIC Interface").
- **LVT Timer Register (FEE0 0320H)** — Specifies interrupt delivery when the APIC timer signals an interrupt (see Section 10.5.4, "APIC Timer").
- **LVT Thermal Monitor Register (FEE0 0330H)** — Specifies interrupt delivery when the thermal sensor generates an interrupt (see Section 14.5.2, "Thermal Monitor"). This LVT entry is implementation specific, not architectural. If implemented, it will always be at base address FEE0 0330H.
- **LVT Performance Counter Register (FEE0 0340H)** — Specifies interrupt delivery when a performance counter generates an interrupt on overflow (see Section 30.9.5.8, "Generating an Interrupt on Overflow"). This LVT entry is implementation specific, not architectural. If implemented, it is not guaranteed to be at base address FEE0 0340H.

- **LVT LINT0 Register (FEE0 0350H)** — Specifies interrupt delivery when an interrupt is signaled at the LINT0 pin.
- **LVT LINT1 Register (FEE0 0360H)** — Specifies interrupt delivery when an interrupt is signaled at the LINT1 pin.
- **LVT Error Register (FEE0 0370H)** — Specifies interrupt delivery when the APIC detects an internal error (see Section 10.5.3, "Error Handling").

The LVT performance counter register and its associated interrupt were introduced in the P6 processors and are also present in the Pentium 4 and Intel Xeon processors. The LVT thermal monitor register and its associated interrupt were introduced in the Pentium 4 and Intel Xeon processors. The LVT CMCI register and its associated interrupt were introduced in the Intel Xeon 5500 processors.

As shown in Figure 10-8, some of these fields and flags are not available (and reserved) for some entries.

The setup information that can be specified in the registers of the LVT table is as follows:

Vector Interrupt vector number.

Delivery Mode Specifies the type of interrupt to be sent to the processor. Some delivery modes will only operate as intended when used in conjunction with a specific trigger mode. The allowable delivery modes are as follows:

000 (Fixed)	Delivers the interrupt specified in the vector field.
010 (SMI)	Delivers an SMI interrupt to the processor core through the processor's local SMI signal path. When using this delivery mode, the vector field should be set to 00H for future compatibility.
100 (NMI)	Delivers an NMI interrupt to the processor. The vector information is ignored.
101 (INIT)	Delivers an INIT request to the processor core, which causes the processor to perform an INIT. When using this delivery mode, the vector field should be set to 00H for future compatibility. Not supported for the LVT CMCI register, the LVT thermal monitor register, or the LVT performance counter register.
110	Reserved; not supported for any LVT register.
111 (ExtINT)	Causes the processor to respond to the interrupt as if the interrupt originated in an externally connected (8259A-compatible) interrupt controller. A special INTA bus cycle corresponding to ExtINT, is routed to the external controller. The external controller is expected to supply the vector information. The APIC architecture supports only one ExtINT source in a system, usually contained in the compatibility bridge. Not supported for the LVT CMCI register, the LVT thermal monitor register, or the LVT performance counter register.

Delivery Status (Read Only)
Indicates the interrupt delivery status, as follows:

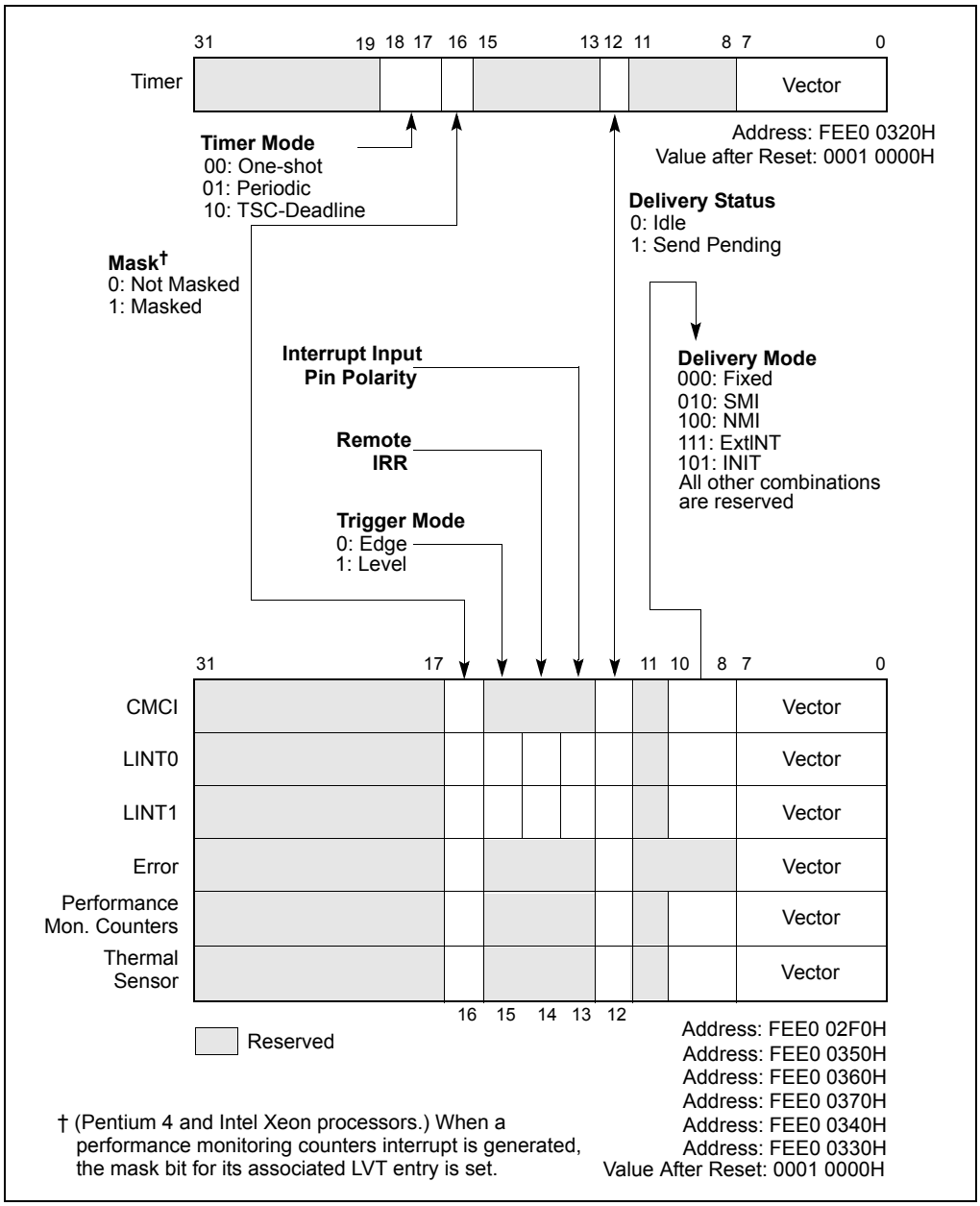


Figure 10-8 Local Vector Table (LVT)

0 (Idle) There is currently no activity for this interrupt source, or the previous interrupt from this source was delivered to the processor core and accepted.

1 (Send Pending)
Indicates that an interrupt from this source has been delivered to the processor core but has not

yet been accepted (see Section 10.5.5, "Local Interrupt Acceptance").

Interrupt Input Pin Polarity
Specifies the polarity of the corresponding interrupt pin: (0) active high or (1) active low.

Remote IRR Flag (Read Only)
For fixed mode, level-triggered interrupts; this flag is set when the local APIC accepts the interrupt for servicing and is reset when an EOI command is received from the processor. The meaning of this flag is undefined for edge-triggered interrupts and other delivery modes.

Trigger Mode
Selects the trigger mode for the local LINT0 and LINT1 pins: (0) edge sensitive and (1) level sensitive. This flag is only used when the delivery mode is Fixed. When the delivery mode is NMI, SMI, or INIT, the trigger mode is always edge sensitive. When the delivery mode is ExtINT, the trigger mode is always level sensitive. The timer and error interrupts are always treated as edge sensitive.

If the local APIC is not used in conjunction with an I/O APIC and fixed delivery mode is selected; the Pentium 4, Intel Xeon, and P6 family processors will always use level-sensitive triggering, regardless if edge-sensitive triggering is selected.

Mask
Interrupt mask: (0) enables reception of the interrupt and (1) inhibits reception of the interrupt. When the local APIC handles a performance-monitoring counters interrupt, it automatically sets the mask flag in the LVT performance counter register. This flag is set to 1 on reset. It can be cleared only by software.

Timer Mode
Bits 18:17 selects the timer mode (see Section 10.5.4):

(00b) one-shot mode using a count-down value,

(01b) periodic mode reloading a count-down value,

(10b) TSC-Deadline mode using absolute target value in IA32_TSC_DEADLINE MSR (see Section 10.5.4.1),

(11b) is reserved.

...

Table 10-6 Local APIC Register Address Map Supported by x2APIC

MSR Address (x2APIC mode)	MMIO Offset (xAPIC mode)	Register Name	MSR R/W Semantics	Comments
802H	020H	Local APIC ID register	Read-only[1]	See Section 10.12.5.1 for initial values.
803H	030H	Local APIC Version register	Read-only	Same version used in xAPIC mode and x2APIC mode.
808H	080H	Task Priority Register (TPR)	Read/write	Bits 31:8 are reserved.[2]
80AH	0A0H	Processor Priority Register (PPR)	Read-only	

Table 10-6 Local APIC Register Address Map Supported by x2APIC (Continued)

MSR Address (x2APIC mode)	MMIO Offset (xAPIC mode)	Register Name	MSR R/W Semantics	Comments
80BH	0B0H	EOI register	Write-only[3]	WRMSR of a non-zero value causes #GP(0).
80DH	0D0H	Logical Destination Register (LDR)	Read-only	Read/write in xAPIC mode.
80FH	0F0H	Spurious Interrupt Vector Register (SVR)	Read/write	See Section 10.9 for reserved bits.
810H	100H	In-Service Register (ISR); bits 31:0	Read-only	
811H	110H	ISR bits 63:32	Read-only	
812H	120H	ISR bits 95:64	Read-only	
813H	130H	ISR bits 127:96	Read-only	
814H	140H	ISR bits 159:128	Read-only	
815H	150H	ISR bits 191:160	Read-only	
816H	160H	ISR bits 223:192	Read-only	
817H	170H	ISR bits 255:224	Read-only	
818H	180H	Trigger Mode Register (TMR); bits 31:0	Read-only	
819H	190H	TMR bits 63:32	Read-only	
81AH	1A0H	TMR bits 95:64	Read-only	
81BH	1B0H	TMR bits 127:96	Read-only	
81CH	1C0H	TMR bits 159:128	Read-only	
81DH	1D0H	TMR bits 191:160	Read-only	
81EH	1E0H	TMR bits 223:192	Read-only	
81FH	1F0H	TMR bits 255:224	Read-only	
820H	200H	Interrupt Request Register (IRR); bits 31:0	Read-only	
821H	210H	IRR bits 63:32	Read-only	
822H	220H	IRR bits 95:64	Read-only	
823H	230H	IRR bits 127:96	Read-only	
824H	240H	IRR bits 159:128	Read-only	
825H	250H	IRR bits 191:160	Read-only	
826H	260H	IRR bits 223:192	Read-only	
827H	270H	IRR bits 255:224	Read-only	
828H	280H	Error Status Register (ESR)	Read/write	WRMSR of a non-zero value causes #GP(0). See Section 10.5.3.

Table 10-6 Local APIC Register Address Map Supported by x2APIC (Continued)

MSR Address (x2APIC mode)	MMIO Offset (xAPIC mode)	Register Name	MSR R/W Semantics	Comments
82FH	2F0H	LVT CMCI register	Read/write	See Figure 10-8 for reserved bits.
830H[4]	300H and 310H	Interrupt Command Register (ICR)	Read/write	See Figure 10-28 for reserved bits
832H	320H	LVT Timer register	Read/write	See Figure 10-8 for reserved bits.
833H	330H	LVT Thermal Sensor register	Read/write	See Figure 10-8 for reserved bits.
834H	340H	LVT Performance Monitoring register	Read/write	See Figure 10-8 for reserved bits.
835H	350H	LVT LINT0 register	Read/write	See Figure 10-8 for reserved bits.
836H	360H	LVT LINT1 register	Read/write	See Figure 10-8 for reserved bits.
837H	370H	LVT Error register	Read/write	See Figure 10-8 for reserved bits.
838H	380H	Initial Count register (for Timer)	Read/write	
839H	390H	Current Count register (for Timer)	Read-only	
83EH	3E0H	Divide Configuration Register (DCR; for Timer)	Read/write	See Figure 10-10 for reserved bits.
83FH	Not available	SELF IPI[5]	Write-only	Available only in x2APIC mode.

NOTES:

1. WRMSR causes #GP(0) for read-only registers.
2. WRMSR causes #GP(0) for attempts to set a reserved bit to 1 in a read/write register (including bits 63:32 of each register).
3. RDMSR causes #GP(0) for write-only registers.
4. MSR 831H is reserved; read/write operations cause general-protection exceptions. The contents of the APIC register at MMIO offset 310H are accessible in x2APIC mode through the MSR at address 830H.
5. SELF IPI register is supported only in x2APIC mode.

…

10.12.5 x2APIC State Transitions

This section provides a detailed description of the x2APIC states of a local x2APIC unit, transitions between these states as well as interactions of these states with INIT and reset.

10.12.5.1 x2APIC States

The valid states for a local x2APIC unit is listed in Table 10-5:

- APIC disabled: IA32_APIC_BASE[EN]=0 and IA32_APIC_BASE[EXTD]=0
- xAPIC mode: IA32_APIC_BASE[EN]=1 and IA32_APIC_BASE[EXTD]=0
- x2APIC mode: IA32_APIC_BASE[EN]=1 and IA32_APIC_BASE[EXTD]=1
- Invalid: IA32_APIC_BASE[EN]=0 and IA32_APIC_BASE[EXTD]=1

The state corresponding to EXTD=1 and EN=0 is not valid and it is not possible to get into this state. An execution of WRMSR to the IA32_APIC_BASE_MSR that attempts a transition from a valid state to this invalid state causes a general-protection exception. Figure 10-27 shows the comprehensive state transition diagram for a local x2APIC unit.

On coming out of reset, the local APIC unit is enabled and is in the xAPIC mode: IA32_APIC_BASE[EN]=1 and IA32_APIC_BASE[EXTD]=0. The APIC registers are initialized as:

- The local APIC ID is initialized by hardware with a 32 bit ID (x2APIC ID). The lowest 8 bits of the x2APIC ID is the legacy local xAPIC ID, and is stored in the upper 8 bits of the APIC register for access in xAPIC mode.
- The following APIC registers are reset to all zeros for those fields that are defined in the xAPIC mode:
  - IRR, ISR, TMR, ICR, LDR, TPR, Divide Configuration Register (See Chapter 8 of "Intel® 64 and IA-32 Architectures Software Developer's Manual", Vol. 3B for details of individual APIC registers),
  - Timer initial count and timer current count registers,
- The LVT registers are reset to 0s except for the mask bits; these are set to 1s.
- The local APIC version register is not affected.
- The Spurious Interrupt Vector Register is initialized to 000000FFH.
- The DFR (available only in xAPIC mode) is reset to all 1s.
- SELF IPI register is reset to zero.

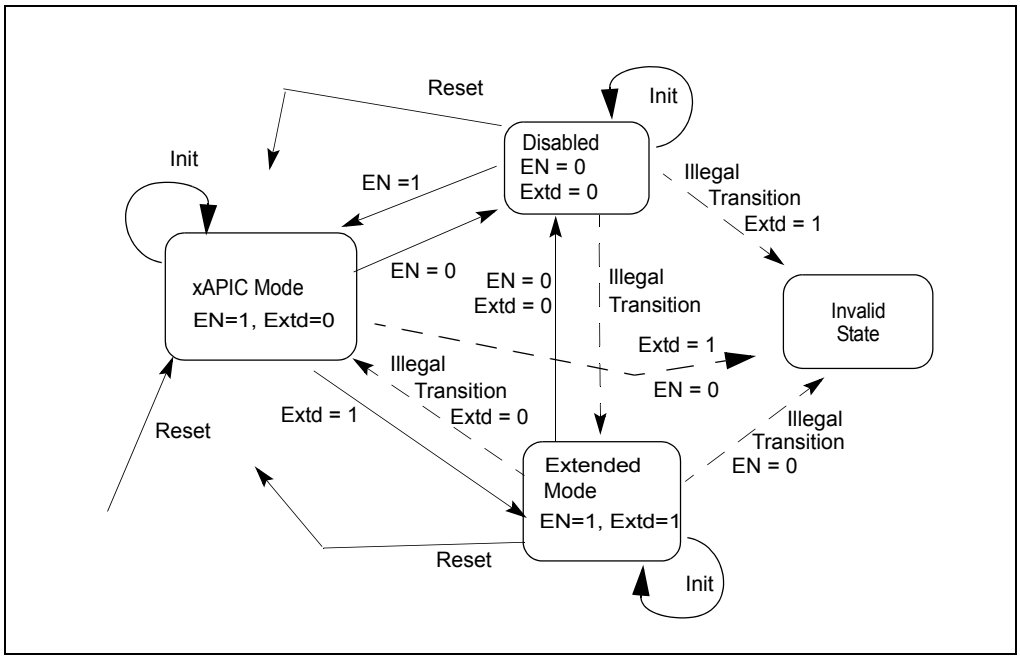


Figure 10-27 Local x2APIC State Transitions with IA32_APIC_BASE, INIT, and Reset

x2APIC After Reset

The valid transitions from the xAPIC mode state are:

- to the x2APIC mode by setting EXT to 1 (resulting EN=1, EXTD= 1). The physical x2APIC ID (see Figure 10-6) is preserved across this transition and the logical x2APIC ID (see Figure 10-29) is initialized by hardware during this transition as documented in Section 10.12.9.2. The state of the extended fields in other APIC registers, which was not initialized at reset, is not architecturally defined across this transition and system software should explicitly initialize those programmable APIC registers.
- to the disabled state by setting EN to 0 (resulting EN=0, EXTD= 0).

The result of an INIT in the xAPIC state places the APIC in the state with EN= 1, EXTD= 0. The state of the local APIC ID register is preserved (the 8-bit xAPIC ID is in the upper 8 bits of the APIC ID register). All the other APIC registers are initialized as a result of INIT.

A reset in this state places the APIC in the state with EN= 1, EXTD= 0. The state of the local APIC ID register is initialized as described in Section 10.12.5.1. All the other APIC registers are initialized described in Section 10.12.5.1.

x2APIC Transitions From x2APIC Mode

From the x2APIC mode, the only valid x2APIC transition using IA32_APIC_BASE is to the state where the x2APIC is disabled by setting EN to 0 and EXTD to 0. The x2APIC ID (32 bits) and the legacy local xAPIC ID (8 bits) are preserved across this transition. A transition from the x2APIC mode to xAPIC mode is not valid, and the corresponding WRMSR to the IA32_APIC_BASE MSR causes a general-protection exception.

A reset in this state places the x2APIC in xAPIC mode. All APIC registers (including the local APIC ID register) are initialized as described in Section 10.12.5.1.

An INIT in this state keeps the x2APIC in the x2APIC mode. The state of the local APIC ID register is preserved (all 32 bits). However, all the other APIC registers are initialized as a result of the INIT transition.

x2APIC Transitions From Disabled Mode

From the disabled state, the only valid x2APIC transition using IA32_APIC_BASE is to the xAPIC mode (EN= 1, EXTD = 0). Thus the only means to transition from x2APIC mode to xAPIC mode is a two-step process:

- first transition from x2APIC mode to local APIC disabled mode (EN= 0, EXTD = 0),
- followed by another transition from disabled mode to xAPIC mode (EN= 1, EXTD= 0).

Consequently, all the APIC register states in the x2APIC, except for the x2APIC ID (32 bits), are not preserved across mode transitions.

A reset in the disabled state places the x2APIC in the xAPIC mode. All APIC registers (including the local APIC ID register) are initialized as described in Section 10.12.5.1.

An INIT in the disabled state keeps the x2APIC in the disabled state.

State Changes From xAPIC Mode to x2APIC Mode

After APIC register states have been initialized by software in xAPIC mode, a transition from xAPIC mode to x2APIC mode does not affect most of the APIC register states, except the following:

- The Logical Destination Register is not preserved.
- Any APIC ID value written to the memory-mapped local APIC ID register is not preserved.
- The high half of the Interrupt Command Register is not preserved.

...

15. Updates to Chapter 11, Volume 3A

Change bars show changes to Chapter 11 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A:* System Programming Guide, Part 1.

...

11.5.1 Cache Control Registers and Bits

...

- **PCD and PWT flags in paging-structure entries** — Control the memory type used to access paging structures and pages (see Section 4.9, "Paging and Memory Typing").
- **PCD and PWT flags in control register CR3** — Control the memory type used to access the first paging structure of the current paging-structure hierarchy (see Section 4.9, "Paging and Memory Typing").

- **G (global) flag in the page-directory and page-table entries (introduced to the IA-32 architecture in the P6 family processors)** — Controls the flushing of TLB entries for individual pages. See Section 4.10, "Caching Translation Information," for more information about this flag.
- **PGE (page global enable) flag in control register CR4** — Enables the establishment of global pages with the G flag. See Section 4.10, "Caching Translation Information," for more information about this flag.
- **Memory type range registers (MTRRs) (introduced in P6 family processors)** — Control the type of caching used in specific regions of physical memory. Any of the caching types described in Section 11.3, "Methods of Caching Available," can be selected. See Section 11.11, "Memory Type Range Registers (MTRRs)," for a detailed description of the MTRRs.
- **Page Attribute Table (PAT) MSR (introduced in the Pentium III processor)** — Extends the memory typing capabilities of the processor to permit memory types to be assigned on a page-by-page basis (see Section 11.12, "Page Attribute Table (PAT)").
- **Third-Level Cache Disable flag, bit 6 of the IA32_MISC_ENABLE MSR (Available only in processors based on Intel NetBurst microarchitecture)** — Allows the L3 cache to be disabled and enabled, independently of the L1 and L2 caches.

...

11.5.4 Disabling and Enabling the L3 Cache

On processors based on Intel NetBurst microarchitecture, the third-level cache can be disabled by bit 6 of the IA32_MISC_ENABLE MSR. The third-level cache disable flag (bit 6 of the IA32_MISC_ENABLE MSR) allows the L3 cache to be disabled and enabled, independently of the L1 and L2 caches. Prior to using this control to disable or enable the L3 cache, software should disable and flush all the processor caches, as described earlier in Section 11.5.3, "Preventing Caching," to prevent of loss of information stored in the L3 cache. After the L3 cache has been disabled or enabled, caching for the whole processor can be restored.

Newer Intel 64 processor with L3 do not support IA32_MISC_ENABLE[bit 6], the procedure described in Section 11.5.3, "Preventing Caching," apply to the entire cache hierarchy.

...

16. Updates to Chapter 13, Volume 3A

Change bars show changes to Chapter 13 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A:* System Programming Guide, Part 1.

...

13.6 XSAVE/XRSTOR AND PROCESSOR EXTENDED STATE MANAGEMENT

The features associated with managing processor extended states include

- An extensible data layout for existing and future processor state extensions. The layout of the XSAVE/XRSTOR area extends from the 512-byte FXSAVE/FXRSTOR layout to provide compatibility and migration path from managing the legacy FXSAVE/FXRSTOR area. Specifically, the XSAVE/XRSTOR area layout consists of:
  - The FXSAVE/FXRSTOR area (512 bytes, the layout is identical to the FXSAVE/FXRSTOR area),
  - The XSAVE header area (64 bytes),
  - A finite set of save areas, each corresponding to a processor extended state (see *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2B*, XSAVE instruction). The number of save areas, the offset and the size of each save area is enumerated by CPUID leaf function 0DH.
- CPUID Enhancement: CPUID instruction provides information on
  - CPUID.01H.ECX.XSAVE[bit 26]. A feature flag indicating the processor's support of XSAVE/XRSTOR architecture extensions
  - CPUID.01H.ECX.OSXSAVE[bit 27]. A feature flag indicating whether OS has enabled extensible state management and communicating that the OS supports processor extended state management.
  - CPUID leaf function 0DH enumerates the list of processor states (including legacy x87 FPU, SSE states and processor extended states), the offset and size of individual save area for each processor extended state.
- Control register enhancement and dedicated register for enabling each processor extended state: CR4. OSXSAVE[bit 18] and XCR0 are described in Chapter 2, "System Architecture Overview". XCR0 can be read at all privilege levels but written only at ring 0.
- Instructions to manage XCR0 and the XSAVE/XRSTOR area (see *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2B*):
  - XGETBV: reads XCR0.
  - XSETBV: writes to XCR0, ring 0 only.
  - XRSTOR: restores from memory the processor states specified by a bit vector mask specified in EDX:EAX.
  - XSAVE: saves the current processor states to memory according to a bit vector mask in EDX:EAX.

13.6.1 XSAVE Header

The header section includes a "XSTATE_BV" bit vector field. If the value of a bit in HEADER.XSTATE_BV is 1, it indicates that the corresponding processor extended state was written to the respective save area in memory by the XSAVE instruction.

If software modifies the save area image of a particular processor state component directly, it is responsible to update the corresponding bit in HEADER.XSTATE_BV to 1. Otherwise, directly modified state information in a save area image may be ignored by XRSTOR.

The order of bit vectors in XSTATE_BV matches those of XCR0. Although XCR0 has only two bits initially defined for state management, the general relationship between the value of XSTATE_BV and the corresponding processor state in the XSAVE/XRSTOR layout is depicted in Figure 13-2.

...

The value of each bit in HEADER.XSTATE_BV may affect the action performed by XRSTOR, depending on the logical value of the respective bits in XCR0, the restore bit mask (EDX:EAX input to XRSTOR), and HEADER.XSTATE_BV. When an XRSTOR instruction is executed with a restore bit mask selecting the i'th bit vector (and the corresponding XCR0 bit is enabled), a value of "1" in the corresponding bit of HEADER.XSTATE_BV causes the processor state to be updated with contents of the save area read from the memory image. A value of "0" in HEADER.XSTATE_BV causes the processor state to be initialized by hardware supplied values instead of from memory (See the operation detail of XRSTOR in *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2B*).

...

13.7 INTEROPERABILITY OF XSAVE/XRSTOR AND FXSAVE/FXRSTOR

FXSAVE instruction writes x87 FPU and SSE state information to a 512-byte FXSAVE, FXRSTOR save area. FXRSTOR restores the processor's x87 FPU and SSE states from FXSAVE/FXRSTOR save area image. XSAVE/XRSTOR instructions support x87 FPU and SSE states using the same layout as the FXSAVE/FXRSTOR area to provide interoperability of FXSAVE versus XSAVE, and FXRSTOR versus XRSTOR. XSAVE/XRSTOR provides the additional flexibility for system software to manage SSE state independent of x87 FPU states. Thus system software that had been using FXSAVE/FXRSTOR to manage x87 FPU and SSE states can transition to XSAVE/XRSTOR to manage x87 FPU, SSE and other processor extended states in a systematic and forward-looking manner.

It is also possible for system software to adopt an alternate approach of using FXSAVE/FXRSTOR for x87 and SSE state management, and implementing forward processor extended state management using XSAVE/XRSTOR. In this case, system software must specify the bit vector mask in EDX:EAX appropriately when executing XSAVE/XRSTOR instructions.

For instance, when using the XSAVE instruction, the OS can supply a bit vector in EDX:EAX with the two least significant bits corresponding to x87 FPU and SSE state equal to 0. Then, the XSAVE instruction will not write the processor's x87 FPU and SSE state into memory. Similarly for the XRSTOR instruction a bit vector mask in EDX:EAX with the least two significant bit equal to 0 will cause the XRSTOR instruction to not restore nor initialize the processor's x87 FPU and SSE state.

The processor's action as a result of executing XRSTOR, on the x87 FPU state, MXCSR, and XMM registers, are listed in Table 13-4 (Both bit 1 and bit 0 of XCR0 are presumed to be 1). The x87 FPU or 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 the SSE state, independent of the x87 FPU state. The action of XSAVE is listed in Table 13-5.

...

13.8 DETECTION, ENUMERATION, ENABLING PROCESSOR EXTENDED STATE SUPPORT

An OS can determine if the XSAVE/XRSTOR/XGETBV/XSETBV instructions and XCR0 are available in the processor by checking the value of CPUID.1.ECX.XSAVE to be 1. The OS must set CR4.OSXSAVE to 1 to enable the new instructions. The OS uses XSETBV to enable the processor state component (setting the corresponding bit in XCR0 to 1) that it will manage using XSAVE/XRSTOR. Bit 0 of XCR0 must be set to 1. The value of CR4.OSXSAVE is reflected in CPUID.01H:ECX.OSXSAVE (bit 27) to communicate the setting to non-privileged software.

...

The bits that must be enabled in XCR0 and the size of the memory region needed to save processor extended state information must be enumerated by CPUID leaf 0DH with ECX = 0 as input. However, the recommended usage by system software to use XSAVE/ XSAVEOPT/XRSTOR is to:

- Use mask (EDX:EAX) with all bits set to 1.
- Alternately use the master bit vector mask EDX:EAX reported by CPUID.(EAX=0D, ECX=0H). This provides a more constrained list of features than using all 1's in the mask.

In either case, system software is required to allocate a memory buffer according to the size reported by CPUID.(EAX=0DH, ECX=0H):ECX. The value reported by CPUID.(EAX=0DH, ECX=0H):ECX always includes the size of the header. Clear the entire buffer prior to being used by XSAVE.

...

13.8.1 Application Programming Model and Processor Extended States

New instruction set extensions may be introduced over time and operating on a processor extended state that must be enabled in XCR0. The general application programming model for using such instruction set extensions are:

- Check if OS has enabled processor extended state management. If CPUID.01H:ECX.OSXSAVE is 1, the OS has enabled the XSAVE/XRSTOR/XSETBV/ XGETBV instructions and XCR0, and it has indicated support for the processor extended state management.

  Applications do not need to check the value of CPUID.01H:ECX.XSAVE because "CPUID.01H:ECX.OSXSAVE = 1" implies OS has successfully verified CPUID.01H:ECX.XSAVE = 1. CPUID.01H:ECX.OSXSAVE reflects the value of CR4.OSXSAVE, and this bit cannot be set to 1 unless CPUID.01H:ECX.XSAVE = 1.
- Check whether the processor extended state component associated with a given instruction set extension is enabled by the OS. The bits of EDX:EAX returned by XGETBV as 1 indicate which processor extended state components have been enabled by OS. Note, the CR4.OSFXSR is not used by OS to enable instruction extensions requiring processor extended state support.
- Check the target instruction set extension is supported in the processor. Each new instruction set extension is expected to provide a feature flag in CPUID when it is introduced.

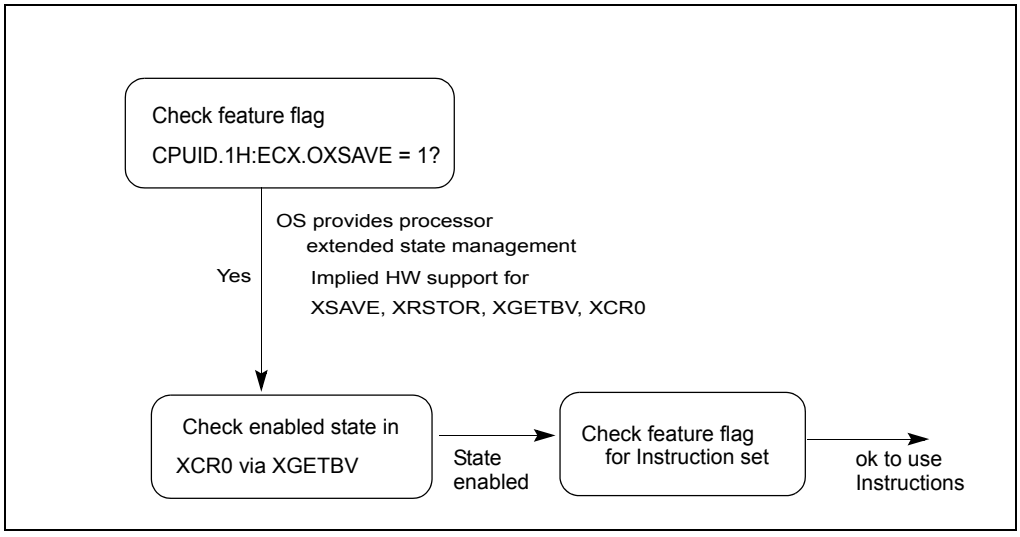


Figure 13-4 Application Detection of New Instruction Extensions and Processor Extended State

If all three requirements are met, applications can use the target new instruction set extensions. If any of the above requirements are not met, an attempt to execute an instruction operating on a processor extended state corresponding to bit offset higher than 1 in XCR0 will cause a #UD exception.

Newer instruction extensions operating on SSE state, but not on any processor extended states corresponding bits in XCR0 with an offset higher than 1, follow the programming model described by Section 13.1 through Section 13.5. XCR0 is not required to enable OS support for SSE state management, but CR4.OSFXSR is required.

13-9 INTEL ADVANCED VECTOR EXTENSIONS (INTEL AVX) AND YMM STATE

Intel AVX instructions comprises of 256-bit and 128-bit instructions that operates on YMM states. The following sections describes system software support requirements for 256-bit YMM states.

For processors that support YMM states, the YMM state exists in all operating modes. However, the available instruction interfaces to access YMM states may vary in different modes. XSAVE/XRSTOR and XSAVEOPT instructions can operate in all operating modes.

13.10 YMM STATE MANAGEMENT

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

An OS must enable its YMM state management to support AVX and any 256-bit extensions that operate on YMM registers. Otherwise, an attempt to execute an instruction in

AVX extensions (including an enhanced 128-bit SIMD instructions using VEX encoding) will cause a #UD exception.

13.10.1 Detection of YMM State Support

Detection of hardware support for new processor extended state is provided by the main CPUID leaf function 0DH with index ECX = 0. Specifically, the return value in EDX:EAX of CPUID.(EAX=0DH, ECX=0) provides a 64-bit wide bit vector of hardware support of processor state components, 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.

13.10.2 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 XCR0 by checking CPUID.1.ECX.XSAVE[bit 26]=1.
- Verify the processor supports YMM state (i.e. bit 2 of XCR0 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 *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A*).
- Set CR4.OSXSAVE[bit 18]=1 to enable the use of XSETBV/XGETBV instructions to write/read XCR0.
- Supply an appropriate mask via EDX:EAX to execute XSETBV to enable the processor state components 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 XCR0 is listed in Table 13-6).

  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 13-6 XCR0 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.
63:3	Reserved; must be 0.

13.10.3 Enabling of SIMD Floating-Exception Support

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

The effect of CR4 setting that affects AVX enabling is listed in Table 13-7.

Table 13-7 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 XCR0, XSAVE/XRSTOR to manage processor extended state. Must be set to '1' to enable AVX.
CR4.OSXMMEXCPT[bit 10]	Must be set to 1 to enable SIMD floating-point exceptions. This applies to AVX operating on YMM states, and legacy 128-bit SIMD floating-point instructions operating on XMM states.
CR4.OSFXSR[bit 9]	Ignored by AVX instructions operating on YMM states. Must be set to 1 to enable SIMD instructions operating on XMM state.

13.10.4 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 XCR0, it can allocate the buffer size according to CPUID.(EAX=0DH, ECX=0):ECX.

After the memory buffer for XSAVE is allocated, the entire buffer must 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 13-8. The register fields of the first 512 byte of the XSAVE area are identical to those of the FXSAVE/FXRSTOR area.

Table 13-8 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 13-9):

Table 13-9 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 13-10.

Table 13-10 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

13.10.5 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 13-4 (Both bit 1 and bit 2 of XCR0 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. MXCSR save/restore will not be bound to any future states.

Table 13-11 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 13-12.

Table 13-12 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 13-13.

Table 13-13 XSAVE Action on MXCSR, XMM, YMM Register

EDX:EAX		XCR0		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

13.10.6 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.

13.10.6.1 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.

17. Updates to Chapter 14, Volume 3A

Change bars show changes to Chapter 14 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A:* System Programming Guide, Part 1.

...

14.3.2.1 Discover Hardware Support and Enabling of Opportunistic Processor Operation

If an Intel 64 processor has hardware support for opportunistic processor performance operation, the power-on default state of IA32_MISC_ENABLE[38] indicates the presence of such hardware support. For Intel 64 processors that support opportunistic processor performance operation, the default value is 1, indicating its presence. For processors that do not support opportunistic processor performance operation, the default value is

0. The power-on default value of IA32_MISC_ENABLE[38] allows BIOS to detect the presence of hardware support of opportunistic processor performance operation.

IA32_MISC_ENABLE[38] is shared across all logical processors in a physical package. It is written by BIOS during platform initiation to enable/disable opportunistic processor operation in conjunction of OS power management capabilities, see Section 14.3.2.2. BIOS can set IA32_MISC_ENABLE[38] with 1 to disable opportunistic processor performance operation; it must clear the default value of IA32_MISC_ENABLE[38] to 0 to enable opportunistic processor performance operation. OS and applications must use CPUID leaf 06H if it needs to detect processors that has opportunistic processor operation enabled.

When CPUID is executed with EAX = 06H on input, Bit 1 of EAX in Leaf 06H (i.e. CPUID.06H:EAX[1]) indicates opportunistic processor performance operation, such as IDA, has been enabled by BIOS.

Opportunistic processor performance operation can be disabled by setting bit 38 of IA32_MISC_ENABLE. This mechanism is intended for BIOS only. If IA32_MISC_ENABLE[38] is set, CPUID.06H:EAX[1] will return 0.

...

18. Updates to Chapter 15, Volume 3A

Change bars show changes to Chapter 15 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A:* System Programming Guide, Part 1.

...

15.5.1 CMCI Local APIC Interface

The operation of CMCI is depicted in Figure 15-9.

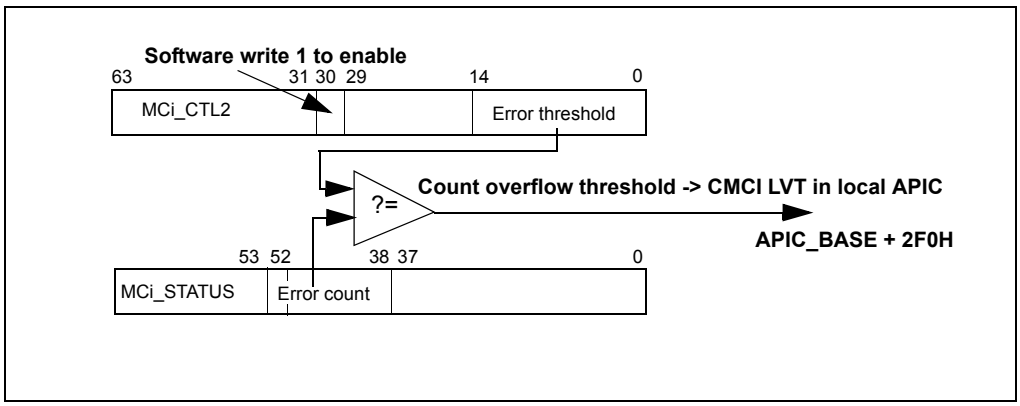


Figure 15-9 CMCI Behavior

CMCI interrupt delivery is configured by writing to the LVT CMCI register entry in the local APIC register space at default address of APIC_BASE + 2F0H. A CMCI interrupt can be delivered to more than one logical processors if multiple logical processors are affected by the associated MC errors. For example, if a corrected bit error in a cache shared by two logical processors caused a CMCI, the interrupt will be delivered to both

logical processors sharing that microarchitectural sub-system. Similarly, package level errors may cause CMCI to be delivered to all logical processors within the package. However, system level errors will not be handled by CMCI.

See Section 10.5.1, "Local Vector Table" for details regarding the LVT CMCI register.

...

19. Updates to Chapter 16, Volume 3A

Change bars show changes to Chapter 16 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A:* System Programming Guide, Part 1.

...

16.4.3 Single-Stepping on Branches

When software sets both the BTF flag (bit 1) in the IA32_DEBUGCTL MSR and the TF flag in the EFLAGS register, the processor generates a single-step debug exception only after instructions that cause a branch.[1] This mechanism allows a debugger to single-step on control transfers caused by branches. This "branch single stepping" helps isolate a bug to a particular block of code before instruction single-stepping further narrows the search. The processor clears the BTF flag when it generates a debug exception. The debugger must set the BTF flag before resuming program execution to continue single-stepping on branches.

...

16.4.8 LBR Stack

The last branch record stack and top-of-stack (TOS) pointer MSRs are supported across Intel 64 and IA-32 processor families. However, the number of MSRs in the LBR stack and the valid range of TOS pointer value can vary between different processor families. Table 16-3 lists the LBR stack size and TOS pointer range for several processor families according to the CPUID signatures of DisplayFamily_DisplayModel encoding (see CPUID instruction in Chapter 3 of *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A*).

Table 16-3 LBR Stack Size and TOS Pointer Range

DisplayFamily_DisplayModel	Size of LBR Stack	Range of TOS Pointer
06_2AH	16	0 to 15
06_1AH, 06_1EH, 06_1FH, 06_2EH, 06_25H, 06_2CH	16	0 to 15
06_17H, 06_1DH	4	0 to 3

1. Executions of CALL, IRET, and JMP that cause task switches never cause single-step debug exceptions (regardless of the value of the BTF flag). A debugger desiring debug exceptions on switches to a task should set the T flag (debug trap flag) in the TSS of that task. See Section 7.2.1, "Task-State Segment (TSS)."

Table 16-3 LBR Stack Size and TOS Pointer Range (Continued)

DisplayFamily_DisplayModel	Size of LBR Stack	Range of TOS Pointer
06_0FH	4	0 to 3
06_1CH	8	0 to 7

...

16.6 LAST BRANCH, INTERRUPT, AND EXCEPTION RECORDING FOR PROCESSORS BASED ON INTEL® MICROARCHITECTURE CODE NAME NEHALEM

The processors based on Intel® microarchitecture code name Nehalem and Intel® microarchitecture code name Westmere support last branch interrupt and exception recording. These capabilities are similar to those found in Intel Core 2 processors and adds additional capabilities:

- **Debug Trace and Branch Recording Control —** The IA32_DEBUGCTL MSR provides bit fields for software to configure mechanisms related to debug trace, branch recording, branch trace store, and performance counter operations. See Section 16.4.1 for a description of the flags. See Figure 16-11 for the MSR layout.
- **Last branch record (LBR) stack —** There are 16 MSR pairs that store the source and destination addresses related to recently executed branches. See Section 16.6.1.
- **Monitoring and single-stepping of branches, exceptions, and interrupts —** See Section 16.4.2 and Section 16.4.3. In addition, the ability to freeze the LBR stack on a PMI request is available.
- **Branch trace messages —** The IA32_DEBUGCTL MSR provides bit fields for software to enable each logical processor to generate branch trace messages. See Section 16.4.4. However, not all BTM messages are observable using the Intel® QPI link.
- **Last exception records —** See Section 16.8.3.
- **Branch trace store and CPL-qualified BTS —** See Section 16.4.6 and Section 16.4.5.
- **FREEZE_LBRS_ON_PMI flag (bit 11) —** see Section 16.4.7.
- **FREEZE_PERFMON_ON_PMI flag (bit 12) —** see Section 16.4.7.
- **UNCORE_PMI_EN (bit 13) —** When set. this logical processor is enabled to receive an counter overflow interrupt form the uncore.
- **FREEZE_WHILE_SMM_EN (bit 14) —** FREEZE_WHILE_SMM_EN is supported if IA32_PERF_CAPABILITIES.FREEZE_WHILE_SMM[Bit 12] is reporting 1. See Section 16.4.1.

...

16.6.2 Filtering of Last Branch Records

MSR_LBR_SELECT is cleared to zero at RESET, and LBR filtering is disabled, i.e. all branches will be captured. MSR_LBR_SELECT provides bit fields to specify the conditions of subsets of branches that will not be captured in the LBR. The layout of MSR_LBR_SELECT is shown in Table 16-9.

Table 16-9 MSR_LBR_SELECT for Intel microarchitecture code name Nehalem

Bit Field	Bit Offset	Access	Description
CPL_EQ_0	0	R/W	When set, do not capture branches occurring in ring 0
CPL_NEQ_0	1	R/W	When set, do not capture branches occurring in ring >0
JCC	2	R/W	When set, do not capture conditional branches
NEAR_REL_CALL	3	R/W	When set, do not capture near relative calls
NEAR_IND_CALL	4	R/W	When set, do not capture near indirect calls
NEAR_RET	5	R/W	When set, do not capture near returns
NEAR_IND_JMP	6	R/W	When set, do not capture near indirect jumps
NEAR_REL_JMP	7	R/W	When set, do not capture near relative jumps
FAR_BRANCH	8	R/W	When set, do not capture far branches
Reserved	63:9		Must be zero

16.7 LAST BRANCH, INTERRUPT, AND EXCEPTION RECORDING FOR PROCESSORS BASED ON INTEL® MICROARCHITECTURE CODE NAME SANDY BRIDGE

Generally, all of the last branch record, interrupt and exception recording facility described in Section 16.7, “Last Branch, Interrupt, and Exception Recording for Processors based on Intel® Microarchitecture code name Sandy Bridge”, apply to processors based on Intel® microarchitecture code name Sandy Bridge.

One difference of note is that MSR_LBR_SELECT is shared between two logical processors in the same core. In Intel microarchitecture code name Sandy Bridge, each logical processor has its own MSR_LBR_SELECT. The filtering semantics for “Near_ind_jmp“ and “Near_rel_jmp“ has been enhanced, see Table 16-10.

Table 16-10 MSR_LBR_SELECT for Intel microarchitecture code name Sandy Bridge

Bit Field	Bit Offset	Access	Description
CPL_EQ_0	0	R/W	When set, do not capture branches occurring in ring 0
CPL_NEQ_0	1	R/W	When set, do not capture branches occurring in ring >0
JCC	2	R/W	When set, do not capture conditional branches
NEAR_REL_CALL	3	R/W	When set, do not capture near relative calls
NEAR_IND_CALL	4	R/W	When set, do not capture near indirect calls

Table 16-10 MSR_LBR_SELECT for (Continued)Intel microarchitecture code name

Bit Field	Bit Offset	Access	Description
NEAR_RET	5	R/W	When set, do not capture near returns
NEAR_IND_JMP	6	R/W	When set, do not capture near indirect jumps except near indirect calls and near returns
NEAR_REL_JMP	7	R/W	When set, do not capture near relative jumps except near relative calls.
FAR_BRANCH	8	R/W	When set, do not capture far branches
Reserved	63:9		Must be zero

...

20. Updates to Chapter 21, Volume 3B

Change bars show changes to Chapter 21 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3B:* System Programming Guide, Part 2.

...

21.1 OVERVIEW

A logical processor uses **virtual-machine control data structures** (**VMCSs**) while it is in VMX operation. These manage transitions into and out of VMX non-root operation (VM entries and VM exits) as well as processor behavior in VMX non-root operation. This structure is manipulated by the new instructions VMCLEAR, VMPTRLD, VMREAD, and VMWRITE.

A VMM can use a different VMCS for each virtual machine that it supports. For a virtual machine with multiple logical processors (virtual processors), the VMM can use a different VMCS for each virtual processor.

A logical processor associates a region in memory with each VMCS. This region is called the **VMCS region**.[1] Software references a specific VMCS using the 64-bit physical address of the region (a **VMCS pointer**). VMCS pointers must be aligned on a 4-KByte boundary (bits 11:0 must be zero). These pointers must not set bits beyond the processor's physical-address width.[2,3]

A logical processor may maintain a number of VMCSs that are **active**. The processor may optimize VMX operation by maintaining the state of an active VMCS in memory, on the processor, or both. At any given time, at most one of the active VMCSs is the **current** VMCS. (This document frequently uses the term "the VMCS" to refer to the current

1. The amount of memory required for a VMCS region is at most 4 KBytes. The exact size is implementation specific and can be determined by consulting the VMX capability MSR IA32_VMX_BASIC to determine the size of the VMCS region (see Appendix G.1).
2. Software can determine a processor's physical-address width by executing CPUID with 80000008H in EAX. The physical-address width is returned in bits 7:0 of EAX.
3. If IA32_VMX_BASIC[48] is read as 1, these pointers must not set any bits in the range 63:32; see Appendix G.1.

VMCS.) The VMLAUNCH, VMREAD, VMRESUME, and VMWRITE instructions operate only on the current VMCS.

The following items describe how a logical processor determines which VMCSs are active and which is current:

- The memory operand of the VMPTRLD instruction is the address of a VMCS. After execution of the instruction, that VMCS is both active and current on the logical processor. Any other VMCS that had been active remains so, but no other VMCS is current.
- The memory operand of the VMCLEAR instruction is also the address of a VMCS. After execution of the instruction, that VMCS is neither active nor current on the logical processor. If the VMCS had been current on the logical processor, the logical processor no longer has a current VMCS.

...

21.10.5 VMXON Region

Before executing VMXON, software allocates a region of memory (called the VMXON region)[1] that the logical processor uses to support VMX operation. The physical address of this region (the VMXON pointer) is provided in an operand to VMXON. The VMXON pointer is subject to the limitations that apply to VMCS pointers:

- The VMXON pointer must be 4-KByte aligned (bits 11:0 must be zero).
- The VMXON pointer must not set any bits beyond the processor's physical-address width.[2,3]

Before executing VMXON, software should write the VMCS revision identifier (see Section 21.2) to the VMXON region. It need not initialize the VMXON region in any other way. Software should use a separate region for each logical processor and should not access or modify the VMXON region of a logical processor between execution of VMXON and VMXOFF on that logical processor. Doing otherwise may lead to unpredictable behavior (including behaviors identified in Section 21.10.1).

...

21. Updates to Chapter 22, Volume 3B

Change bars show changes to Chapter 22 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3B:* System Programming Guide, Part 2.

...

1. The amount of memory required for the VMXON region is the same as that required for a VMCS region. This size is implementation specific and can be determined by consulting the VMX capability MSR IA32_VMX_BASIC (see Appendix G.1).
2. Software can determine a processor's physical-address width by executing CPUID with 80000008H in EAX. The physical-address width is returned in bits 7:0 of EAX.
3. If IA32_VMX_BASIC[48] is read as 1, the VMXON pointer must not set any bits in the range 63:32; see Appendix G.1.

22.1.3 Instructions That Cause VM Exits Conditionally

Certain instructions cause VM exits in VMX non-root operation depending on the setting of the VM-execution controls. The following instructions can cause "fault-like" VM exits based on the conditions described:

- **CLTS**. The CLTS instruction causes a VM exit if the bits in position 3 (corresponding to CR0.TS) are set in both the CR0 guest/host mask and the CR0 read shadow.
- **HLT.** The HLT instruction causes a VM exit if the "HLT exiting" VM-execution control is 1.
- **IN, INS/INSB/INSW/INSD, OUT, OUTS/OUTSB/OUTSW/OUTSD.** The behavior of each of these instructions is determined by the settings of the "unconditional I/O exiting" and "use I/O bitmaps" VM-execution controls:
  - If both controls are 0, the instruction executes normally.
  - If the "unconditional I/O exiting" VM-execution control is 1 and the "use I/O bitmaps" VM-execution control is 0, the instruction causes a VM exit.
  - If the "use I/O bitmaps" VM-execution control is 1, the instruction causes a VM exit if it attempts to access an I/O port corresponding to a bit set to 1 in the appropriate I/O bitmap (see Section 21.6.4). If an I/O operation "wraps around" the 16-bit I/O-port space (accesses ports FFFFH and 0000H), the I/O instruction causes a VM exit (the "unconditional I/O exiting" VM-execution control is ignored if the "use I/O bitmaps" VM-execution control is 1).

  See Section 22.1.1 for information regarding the priority of VM exits relative to faults that may be caused by the INS and OUTS instructions.
- **INVLPG.** The INVLPG instruction causes a VM exit if the "INVLPG exiting" VM-execution control is 1.
- **LGDT, LIDT, LLDT, LTR, SGDT, SIDT, SLDT, STR.** These instructions cause VM exits if the "descriptor-table exiting" VM-execution control is 1.[1]
- **LMSW.** In general, the LMSW instruction causes a VM exit if it would write, for any bit set in the low 4 bits of the CR0 guest/host mask, a value different than the corresponding bit in the CR0 read shadow. LMSW never clears bit 0 of CR0 (CR0.PE); thus, LMSW causes a VM exit if either of the following are true:
  - The bits in position 0 (corresponding to CR0.PE) are set in both the CR0 guest/mask and the source operand, and the bit in position 0 is clear in the CR0 read shadow.
  - For any bit position in the range 3:1, the bit in that position is set in the CR0 guest/mask and the values of the corresponding bits in the source operand and the CR0 read shadow differ.

...

1. "Descriptor-table exiting" is a secondary processor-based VM-execution control. If bit 31 of the primary processor-based VM-execution controls is 0, VMX non-root operation functions as if the "descriptor-table exiting" VM-execution control were 0. See Section 21.6.2.

22.4 CHANGES TO INSTRUCTION BEHAVIOR IN VMX NON-ROOT OPERATION

The behavior of some instructions is changed in VMX non-root operation. Some of these changes are determined by the settings of certain VM-execution control fields. The following items detail such changes:

...

- **RDMSR.** Section 22.1.3 identifies when executions of the RDMSR instruction cause VM exits. If such an execution causes neither a fault due to CPL > 0 nor a VM exit, the instruction's behavior may be modified for certain values of ECX:
  - If ECX contains 10H (indicating the IA32_TIME_STAMP_COUNTER MSR), the value returned by the instruction is determined by the setting of the "use TSC offsetting" VM-execution control as well as the TSC offset:
    - If the control is 0, the instruction operates normally, loading EAX:EDX with the value of the IA32_TIME_STAMP_COUNTER MSR.
    - If the control is 1, the instruction loads EAX:EDX with the sum (using signed addition) of the value of the IA32_TIME_STAMP_COUNTER MSR and the value of the TSC offset (interpreted as a signed value).

    The 1-setting of the "use TSC-offsetting" VM-execution control does not effect executions of RDMSR if ECX contains 6E0H (indicating the IA32_TSC_DEADLINE MSR). Such executions return the APIC-timer deadline relative to the actual timestamp counter without regard to the TSC offset.
  - If ECX contains 808H (indicating the TPR MSR), instruction behavior is determined by the setting of the "virtualize x2APIC mode" VM-execution control:[1]
    - If the control is 0, the instruction operates normally. If the local APIC is in x2APIC mode, EAX[7:0] is loaded with the value of the APIC's task-priority register (EDX and EAX[31:8] are cleared to 0). If the local APIC is not in x2APIC mode, a general-protection fault occurs.
    - If the control is 1, the instruction loads EAX:EDX with the value of bytes 87H:80H of the virtual-APIC page. This occurs even if the local APIC is not in x2APIC mode (no general-protection fault occurs because the local APIC is not x2APIC mode).

...

- **WRMSR.** Section 22.1.3 identifies when executions of the WRMSR instruction cause VM exits. If such an execution neither a fault due to CPL > 0 nor a VM exit, the instruction's behavior may be modified for certain values of ECX:
  - If ECX contains 79H (indicating IA32_BIOS_UPDT_TRIG MSR), no microcode update is loaded, and control passes to the next instruction. This implies that microcode updates cannot be loaded in VMX non-root operation.
  - If ECX contains 808H (indicating the TPR MSR) and either EDX or EAX[31:8] is non-zero, a general-protection fault occurs (this is true even if the logical processor is not in VMX non-root operation). Otherwise, instruction behavior is

1. "Virtualize x2APIC mode" is a secondary processor-based VM-execution control. If bit 31 of the primary processor-based VM-execution controls is 0, VMX non-root operation functions as if the "virtualize x2APIC mode" VM-execution control were 0. See Section 21.6.2.

determined by the setting of the "virtualize x2APIC mode" VM-execution control and the value of the TPR-threshold VM-execution control field:

- If the control is 0, the instruction operates normally. If the local APIC is in x2APIC mode, the value of EAX[7:0] is written to the APIC's task-priority register. If the local APIC is not in x2APIC mode, a general-protection fault occurs.
- If the control is 1, the instruction stores the value of EAX:EDX to bytes 87H:80H of the virtual-APIC page. This store occurs even if the local APIC is not in x2APIC mode (no general-protection fault occurs because the local APIC is not x2APIC mode). The store may cause a VM exit to occur after the instruction completes (see Section 22.1.3).
- The 1-setting of the "use TSC-offsetting" VM-execution control does not effect executions of WRMSR if ECX contains 10H (indicating the IA32_TIME_STAMP_COUNTER MSR). Such executions modify the actual timestamp counter without regard to the TSC offset.
- The 1-setting of the "use TSC-offsetting" VM-execution control does not effect executions of WRMSR if ECX contains 6E0H (indicating the IA32_TSC_DEADLINE MSR). Such executions modify the APIC-timer deadline relative to the actual timestamp counter without regard to the TSC offset.

...

22. Updates to Chapter 23, Volume 3B

Change bars show changes to Chapter 23 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3B:* System Programming Guide, Part 2.

...

23.2.1.1 VM-Execution Control Fields

VM entries perform the following checks on the VM-execution control fields:[1]

- Reserved bits in the pin-based VM-execution controls must be set properly. Software may consult the VMX capability MSRs to determine the proper settings (see Appendix G.3.1).
- Reserved bits in the primary processor-based VM-execution controls must be set properly. Software may consult the VMX capability MSRs to determine the proper settings (see Appendix G.3.2).
- If the "activate secondary controls" primary processor-based VM-execution control is 1, reserved bits in the secondary processor-based VM-execution controls must be set properly. Software may consult the VMX capability MSRs to determine the proper settings (see Appendix G.3.3).

  If the "activate secondary controls" primary processor-based VM-execution control is 0 (or if the processor does not support the 1-setting of that control), no checks are performed on the secondary processor-based VM-execution controls. The

1. If the "activate secondary controls" primary processor-based VM-execution control is 0, VM entry operates as if each secondary processor-based VM-execution control were 0.

logical processor operates as if all the secondary processor-based VM-execution controls were 0.

- The CR3-target count must not be greater than 4. Future processors may support a different number of CR3-target values. Software should read the VMX capability MSR IA32_VMX_MISC to determine the number of values supported (see Appendix G.6).
- If the "use I/O bitmaps" VM-execution control is 1, bits 11:0 of each I/O-bitmap address must be 0. Neither address should set any bits beyond the processor's physical-address width.[1,2]
- If the "use MSR bitmaps" VM-execution control is 1, bits 11:0 of the MSR-bitmap address must be 0. The address should not set any bits beyond the processor's physical-address width.[3]
- If the "use TPR shadow" VM-execution control is 1, the virtual-APIC address must satisfy the following checks:
  - Bits 11:0 of the address must be 0.
  - The address should not set any bits beyond the processor's physical-address width.[4]

  The following items describe the treatment of bytes 81H-83H on the virtual-APIC page (see Section 21.6.8) if all of the above checks are satisfied and the "use TPR shadow" VM-execution control is 1, treatment depends upon the setting of the "virtualize APIC accesses" VM-execution control:[5]
  - If the "virtualize APIC accesses" VM-execution control is 0, the bytes may be cleared. (If the bytes are not cleared, they are left unmodified.)
  - If the "virtualize APIC accesses" VM-execution control is 1, the bytes are cleared.
  - If the VM entry fails, the any clearing of the bytes may or may not occur. This is true either if the failure causes control to pass to the instruction following the VM-entry instruction or if it cause processor state to be loaded from the host-state area of the VMCS. Behavior may be implementation-specific.
- If the "use TPR shadow" VM-execution control is 1, bits 31:4 of the TPR threshold VM-execution control field must be 0.
- The following check is performed if the "use TPR shadow" VM-execution control is 1 and the "virtualize APIC accesses" VM-execution control is 0: the value of bits 3:0 of the TPR threshold VM-execution control field should not be greater than the value of bits 7:4 in byte 80H on the virtual-APIC page (see Section 21.6.8).
- If the "NMI exiting" VM-execution control is 0, the "virtual NMIs" VM-execution control must be 0.

---

1. Software can determine a processor's physical-address width by executing CPUID with 80000008H in EAX. The physical-address width is returned in bits 7:0 of EAX.
2. If IA32_VMX_BASIC[48] is read as 1, these addresses must not set any bits in the range 63:32; see Appendix G.1.
3. If IA32_VMX_BASIC[48] is read as 1, this address must not set any bits in the range 63:32; see Appendix G.1.
4. If IA32_VMX_BASIC[48] is read as 1, this address must not set any bits in the range 63:32; see Appendix G.1.
5. "Virtualize APIC accesses" is a secondary processor-based VM-execution control. If bit 31 of the primary processor-based VM-execution controls is 0, VM entry functions as if the "virtualize APIC accesses" VM-execution control were 0. See Section 21.6.2.

- If the "virtual NMIs" VM-execution control is 0, the "NMI-window exiting" VM-execution control must be 0.
- If the "virtualize APIC-accesses" VM-execution control is 1, the APIC-access address must satisfy the following checks:
  - Bits 11:0 of the address must be 0.
  - The address should not set any bits beyond the processor's physical-address width.[1]
- If the "virtualize x2APIC mode" VM-execution control is 1, the "use TPR shadow" VM-execution control must be 1 and the "virtualize APIC accesses" VM-execution control must be 0.[2]
- If the "enable VPID" VM-execution control is 1, the value of the VPID VM-execution control field must not be 0000H.
- If the "enable EPT" VM-execution control is 1, the EPTP VM-execution control field (see Table 21-8 in Section 21.6.11) must satisfy the following checks:[3]
  - The EPT memory type (bits 2:0) must be a value supported by the logical processor as indicated in the IA32_VMX_EPT_VPID_CAP MSR (see Appendix G.10).
  - Bits 5:3 (1 less than the EPT page-walk length) must be 3, indicating an EPT page-walk length of 4; see Section 25.2.2.
  - Reserved bits 11:6 and 63:N (where N is the processor's physical-address width) must all be 0.
  - If the "unrestricted guest" VM-execution control is 1, the "enable EPT" VM-execution control must also be 1.[4]

23.2.1.2 VM-Exit Control Fields

VM entries perform the following checks on the VM-exit control fields.

- Reserved bits in the VM-exit controls must be set properly. Software may consult the VMX capability MSRs to determine the proper settings (see Appendix G.4).
- If "activate VMX-preemption timer" VM-execution control is 0, the "save VMX-preemption timer value" VM-exit control must also be 0.
- The following checks are performed for the VM-exit MSR-store address if the VM-exit MSR-store count field is non-zero:
  - The lower 4 bits of the VM-exit MSR-store address must be 0. The address should not set any bits beyond the processor's physical-address width.[5]

---

1. If IA32_VMX_BASIC[48] is read as 1, this address must not set any bits in the range 63:32; see Appendix G.1.
2. "Virtualize APIC accesses" and "virtualize x2APIC mode" are both secondary processor-based VM-execution controls. If bit 31 of the primary processor-based VM-execution controls is 0, VM entry functions as if both these controls were 0. See Section 21.6.2.
3. "Enable EPT" is a secondary processor-based VM-execution control. If bit 31 of the primary processor-based VM-execution controls is 0, VM entry functions as if the "enable EPT" VM-execution control were 0. See Section 21.6.2.
4. "Unrestricted guest" and "enable EPT" are both secondary processor-based VM-execution controls. If bit 31 of the primary processor-based VM-execution controls is 0, VM entry functions as if both these controls were 0. See Section 21.6.2.

— The address of the last byte in the VM-exit MSR-store area should not set any bits beyond the processor's physical-address width. The address of this last byte is VM-exit MSR-store address + (MSR count * 16) – 1. (The arithmetic used for the computation uses more bits than the processor's physical-address width.)

If IA32_VMX_BASIC[48] is read as 1, neither address should set any bits in the range 63:32; see Appendix G.1.

- The following checks are performed for the VM-exit MSR-load address if the VM-exit MSR-load count field is non-zero:
  — The lower 4 bits of the VM-exit MSR-load address must be 0. The address should not set any bits beyond the processor's physical-address width.
  — The address of the last byte in the VM-exit MSR-load area should not set any bits beyond the processor's physical-address width. The address of this last byte is VM-exit MSR-load address + (MSR count * 16) – 1. (The arithmetic used for the computation uses more bits than the processor's physical-address width.)

  If IA32_VMX_BASIC[48] is read as 1, neither address should set any bits in the range 63:32; see Appendix G.1.

23.2.1.3 VM-Entry Control Fields

VM entries perform the following checks on the VM-entry control fields.

- Reserved bits in the VM-entry controls must be set properly. Software may consult the VMX capability MSRs to determine the proper settings (see Appendix G.5).
- Fields relevant to VM-entry event injection must be set properly. These fields are the VM-entry interruption-information field (see Table 21-12 in Section 21.8.3), the VM-entry exception error code, and the VM-entry instruction length. If the valid bit (bit 31) in the VM-entry interruption-information field is 1, the following must hold:
  — The field's interruption type (bits 10:8) is not set to a reserved value. Value 1 is reserved on all logical processors; value 7 (other event) is reserved on logical processors that do not support the 1-setting of the "monitor trap flag" VM-execution control.
  — The field's vector (bits 7:0) is consistent with the interruption type:
    - If the interruption type is non-maskable interrupt (NMI), the vector is 2.
    - If the interruption type is hardware exception, the vector is at most 31.
    - If the interruption type is other event, the vector is 0 (pending MTF VM exit).
  — The field's deliver-error-code bit (bit 11) is 1 if and only if (1) either (a) the "unrestricted guest" VM-execution control is 0; or (b) bit 0 (corresponding to CR0.PE) is set in the CR0 field in the guest-state area; (2) the interruption type is hardware exception; and (3) the vector indicates an exception that would normally deliver an error code (8 = #DF; 10 = TS; 11 = #NP; 12 = #SS; 13 = #GP; 14 = #PF; or 17 = #AC).
  — Reserved bits in the field (30:12) are 0.
  — If the deliver-error-code bit (bit 11) is 1, bits 31:15 of the VM-entry exception error-code field are 0.

5. Software can determine a processor's physical-address width by executing CPUID with 80000008H in EAX. The physical-address width is returned in bits 7:0 of EAX.

— If the interruption type is software interrupt, software exception, or privileged software exception, the VM-entry instruction-length field is in the range 1–15.

- The following checks are performed for the VM-entry MSR-load address if the VM-entry MSR-load count field is non-zero:
  — The lower 4 bits of the VM-entry MSR-load address must be 0. The address should not set any bits beyond the processor's physical-address width.[1]
  — The address of the last byte in the VM-entry MSR-load area should not set any bits beyond the processor's physical-address width. The address of this last byte is VM-entry MSR-load address + (MSR count * 16) – 1. (The arithmetic used for the computation uses more bits than the processor's physical-address width.)

  If IA32_VMX_BASIC[48] is read as 1, neither address should set any bits in the range 63:32; see Appendix G.1.
- If the processor is not in SMM, the "entry to SMM" and "deactivate dual-monitor treatment" VM-entry controls must be 0.
- The "entry to SMM" and "deactivate dual-monitor treatment" VM-entry controls cannot both be 1.

…

23.3.1.5 Checks on Guest Non-Register State

The following checks are performed on fields in the guest-state area corresponding to non-register state:

…

- VMCS link pointer. The following checks apply if the field contains a value other than FFFFFFFF_FFFFFFFFH:
  — Bits 11:0 must be 0.
  — Bits beyond the processor's physical-address width must be 0.[2,3]
  — The 32 bits located in memory referenced by the value of the field (as a physical address) must contain the processor's VMCS revision identifier (see Section 21.2).
  — If the processor is not in SMM or the "entry to SMM" VM-entry control is 1, the field must not contain the current VMCS pointer.
  — If the processor is in SMM and the "entry to SMM" VM-entry control is 0, the field must not contain the VMXON pointer.

23.3.1.6 Checks on Guest Page-Directory-Pointer-Table Entries

If CR0.PG =1, CR4.PAE = 1, and IA32_EFER.LMA = 0, the logical processor also uses **PAE paging** (see Section 4.4 in the *Intel® 64 and IA-32 Architectures Software Devel-*

1. Software can determine a processor's physical-address width by executing CPUID with 80000008H in EAX. The physical-address width is returned in bits 7:0 of EAX.
2. Software can determine a processor's physical-address width by executing CPUID with 80000008H in EAX. The physical-address width is returned in bits 7:0 of EAX.
3. If IA32_VMX_BASIC[48] is read as 1, this field must not set any bits in the range 63:32; see Appendix G.1.

oper's Manual, Volume 3A).[1] When PAE paging is in use, the physical address in CR3 references a table of **page-directory-pointer-table entries** (PDPTEs). A MOV to CR3 when PAE paging is in use checks the validity of the PDPTEs.

A VM entry is to a guest that uses PAE paging if (1) bit 31 (corresponding to CR0.PG) is set in the CR0 field in the guest-state area; (2) bit 5 (corresponding to CR4.PAE) is set in the CR4 field; and (3) the "IA-32e mode guest" VM-entry control is 0. Such a VM entry checks the validity of the PDPTEs:

- If the "enable EPT" VM-execution control is 0, VM entry checks the validity of the PDPTEs referenced by the CR3 field in the guest-state area if either (1) PAE paging was not in use before the VM entry; or (2) the value of CR3 is changing as a result of the VM entry. VM entry may check their validity even if neither (1) nor (2) hold.[2]
- If the "enable EPT" VM-execution control is 1, VM entry checks the validity of the PDPTE fields in the guest-state area (see Section 21.4.2).

A VM entry to a guest that does not use PAE paging does not check the validity of any PDPTEs.

A VM entry that checks the validity of the PDPTEs uses the same checks that are used when CR3 is loaded with MOV to CR3 when PAE paging is in use.[3] If MOV to CR3 would cause a general-protection exception due to the PDPTEs that would be loaded (e.g., because a reserved bit is set), the VM entry fails.

...

23. Updates to Chapter 24, Volume 3B

Change bars show changes to Chapter 24 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3B:* System Programming Guide, Part 2.

...

24.2.1 Basic VM-Exit Information

Section 21.9.1 defines the basic VM-exit information fields. The following items detail their use.

...

- **Guest-linear address.** For some VM exits, this field receives a linear address that pertains to the VM exit. The field is set for different VM exits as follows:

1. On processors that support Intel 64 architecture, the physical-address extension may support more than 36 physical-address bits. Software can determine the number physical-address bits supported by executing CPUID with 80000008H in EAX. The physical-address width is returned in bits 7:0 of EAX.
2. "Enable EPT" is a secondary processor-based VM-execution control. If bit 31 of the primary processor-based VM-execution controls is 0, VM entry functions as if the "enable EPT" VM-execution control were 0. See Section 21.6.2.
3. This implies that (1) bits 11:9 in each PDPTE are ignored; and (2) if bit 0 (present) is clear in one of the PDPTEs, bits 63:1 of that PDPTE are ignored.

— VM exits due to attempts to execute LMSW with a memory operand. In these cases, this field receives the linear address of that operand. Bits 63:32 are cleared if the logical processor was not in 64-bit mode before the VM exit.

— VM exits due to attempts to execute INS or OUTS for which the relevant segment is usable (if the relevant segment is not usable, the value is undefined). (ES is always the relevant segment for INS; for OUTS, the relevant segment is DS unless overridden by an instruction prefix.) The linear address is the base address of relevant segment plus (E)DI (for INS) or (E)SI (for OUTS). Bits 63:32 are cleared if the logical processor was not in 64-bit mode before the VM exit.

— VM exits due to EPT violations that set bit 7 of the exit qualification (see Table 24-7; these are all EPT violations except those resulting from an attempt to load the PDPTEs as of execution of the MOV CR instruction). The linear address may translate to the guest-physical address whose access caused the EPT violation. Alternatively, translation of the linear address may reference a paging-structure entry whose access caused the EPT violation. Bits 63:32 are cleared if the logical processor was not in 64-bit mode before the VM exit.

— For all other VM exits, the field is undefined.

...

24. Updates to Chapter 26, Volume 3B

Change bars show changes to Chapter 26 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3B:* System Programming Guide, Part 2.

...

26.14.4 VMXOFF and SMI Unblocking

The VMXOFF instruction can be executed only with the default treatment (see Section 26.15.1) and only outside SMM. If SMIs are blocked when VMXOFF is executed, VMXOFF unblocks them unless IA32_SMM_MONITOR_CTL[bit 2] is 1 (see Section 26.15.5 for details regarding this MSR).[1] Section 26.15.7 identifies a case in which SMIs may be blocked when VMXOFF is executed.

Not all processors allow this bit to be set to 1. Software should consult the VMX capability MSR IA32_VMX_MISC (see Appendix G.6) to determine whether this is allowed.

...

26.15.4.1 Checks on the Executive-VMCS Pointer Field

VM entries that return from SMM perform the following checks on the executive-VMCS pointer field in the current VMCS:

- Bits 11:0 must be 0.
- The pointer must not set any bits beyond the processor's physical-address width.[2,3]

1. Setting IA32_SMM_MONITOR_CTL[bit 2] to 1 prevents VMXOFF from unblocking SMIs regardless of the value of the register's valid bit (bit 0).

- The 32 bits located in memory referenced by the physical address in the pointer must contain the processor's VMCS revision identifier (see Section 21.2).

The checks above are performed before the checks described in Section 23.2.1.1 and before any of the following checks:

- If the "deactivate dual-monitor treatment" VM-entry control is 0, the launch state of the executive VMCS (the VMCS referenced by the executive-VMCS pointer field) must be launched (see Section 21.10.3).
- If the "deactivate dual-monitor treatment" VM-entry control is 1, the executive-VMCS pointer field must contain the VMXON pointer (see Section 26.15.7).[1]

...

26.15.5 Enabling the Dual-Monitor Treatment

Code and data for the SMM monitor reside in a region of SMRAM called the **monitor segment** (MSEG). Code running in SMM determines the location of MSEG and establishes its content. This code is also responsible for enabling the dual-monitor treatment.

SMM code enables the dual-monitor treatment and determines the location of MSEG by writing to IA32_SMM_MONITOR_CTL MSR (index 9BH). The MSR has the following format:

- Bit 0 is the register's valid bit. The SMM monitor may be invoked using VMCALL only if this bit is 1. Because VMCALL is used to activate the dual-monitor treatment (see Section 26.15.6), the dual-monitor treatment cannot be activated if the bit is 0. This bit is cleared when the logical processor is reset.
- Bit 1 is reserved.
- Bit 2 determines whether executions of VMXOFF unblock SMIs under the default treatment of SMIs and SMM. Executions of VMXOFF unblock SMIs unless bit 2 is 1 (the value of bit 0 is irrelevant). See Section 26.14.4.

  Certain leaf functions of the GETSEC instruction clear this bit (see Chapter 6, "Safer Mode Extensions Reference," in *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2B*)
- Bits 11:3 are reserved.
- Bits 31:12 contain a value that, when shifted right 12 bits, is the physical address of MSEG (the MSEG base address).
- Bits 63:32 are reserved.

...

26.15.6.1 Initial Checks

An execution of VMCALL attempts to activate the dual-monitor treatment if (1) the processor supports the dual-monitor treatment;[2] (2) the logical processor is in VMX root

2. Software can determine a processor's physical-address width by executing CPUID with 80000008H in EAX. The physical-address width is returned in bits 7:0 of EAX.
3. If IA32_VMX_BASIC[48] is read as 1, this pointer must not set any bits in the range 63:32; see Appendix G.1.
1. An SMM monitor can determine the VMXON pointer by reading the executive-VMCS pointer field in the current VMCS after the SMM VM exit that activates the dual-monitor treatment.

operation; (3) the logical processor is outside SMM and the valid bit is set in the IA32_SMM_MONITOR_CTL MSR; (4) the logical processor is not in virtual-8086 mode and not in compatibility mode; (5) CPL = 0; and (6) the dual-monitor treatment is not active.

The VMCS that manages SMM VM exit caused by this VMCALL is the current VMCS established by the executive monitor. The VMCALL performs the following checks on the current VMCS in the order indicated:

1. There must be a current VMCS pointer.
2. The launch state of the current VMCS must be clear.
3. The VM-exit control fields must be valid:
   — Reserved bits in the VM-exit controls must be set properly. Software may consult the VMX capability MSR IA32_VMX_EXIT_CTLS to determine the proper settings (see Appendix G.4).
   — The following checks are performed for the VM-exit MSR-store address if the VM-exit MSR-store count field is non-zero:
   - The lower 4 bits of the VM-exit MSR-store address must be 0. The address should not set any bits beyond the processor's physical-address width.[1]
   - The address of the last byte in the VM-exit MSR-store area should not set any bits beyond the processor's physical-address width. The address of this last byte is VM-exit MSR-store address + (MSR count * 16) – 1. (The arithmetic used for the computation uses more bits than the processor's physical-address width.)

   If IA32_VMX_BASIC[48] is read as 1, neither address should set any bits in the range 63:32; see Appendix G.1.

...

26.15.7 Deactivating the Dual-Monitor Treatment

An SMM monitor may deactivate the dual monitor treatment and return the processor to default treatment of SMIs and SMM (see Section 26.14). It does this by executing a VM entry with the "deactivate dual-monitor treatment" VM-entry control set to 1.

As noted in Section 23.2.1.3 and Section 26.15.4.1, an attempt to deactivate the dual-monitor treatment fails in the following situations: (1) the processor is not in SMM; (2) the "entry to SMM" VM-entry control is 1; or (3) the executive-VMCS pointer does not contain the VMXON pointer (the VM entry is to VMX non-root operation).

As noted in Section 26.15.4.9, VM entries that deactivate the dual-monitor treatment ignore the SMI bit in the interruptibility-state field of the guest-state area. Instead, the blocking of SMIs following such a VM entry depends on whether the logical processor is in SMX operation:[2]

- If the logical processor is in SMX operation, SMIs are blocked after VM entry. SMIs may later be unblocked by the VMXOFF instruction (see Section 26.14.4) or by certain leaf functions of the GETSEC instruction (see Chapter 6, "Safer Mode

2. Software should consult the VMX capability MSR IA32_VMX_BASIC (see Appendix G.1) to determine whether the dual-monitor treatment is supported.
1. Software can determine a processor's physical-address width by executing CPUID with 80000008H in EAX. The physical-address width is returned in bits 7:0 of EAX.

Extensions Reference," in *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2B*).

- If the logical processor is outside SMX operation, SMIs are unblocked after VM entry.

...

26.16 SMI AND PROCESSOR EXTENDED STATE MANAGEMENT

On processors that support processor extended states using XSAVE/XRSTOR (see Chapter 13, "System Programming for Instruction Set Extensions and Processor Extended States"), the processor does not save any XSAVE/XRSTOR related state on an SMI. It is the responsibility of the SMM handler code to properly preserve the state information (including CR4.OSXSAVE, XCR0, and possibly processor extended states using XSAVE/XRSTOR). Therefore, the SMM handler must follow the rules described in Chapter 13.

...

25. Updates to Chapter 28, Volume 3B

Change bars show changes to Chapter 28 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3B:* System Programming Guide, Part 2.

...

28.3.5.3 Response to Uses of INVLPG

Operating-systems can use INVLPG to flush entries from the TLB. This instruction takes a linear address as an operand and software expects any cached translations for the address to be flushed. A VMM should set the processor-based VM-execution control "INVLPG exiting" to 1 so that any attempts by a privileged guest to execute INVLPG will trap to the VMM. The VMM can then modify the active page-table hierarchy to emulate the desired effect of the INVLPG.

The following steps are performed. Note that these steps are performed only if the guest invocation of INVLPG would not fault and only if the guest software is running at privilege level 0:

1. Locate the relevant active PDE using the upper 10 bits of the operand address and the current value of CR3. If the PDE refers to a 4-MByte page (PS = 1), then set P = 0 in the PDE.
2. If the PDE is marked present and refers to a page table (PS = 0), locate the relevant active PTE using the next 10 bits of the operand address (bits 21–12) and the page-table base address in the PDE. Set P = 0 in the PTE. Examine all PTEs in the page table; if they are now all marked not-present, de-allocate the page table and set P = 0 in the PDE (this step may be optional).

---

2. A logical processor is in SMX operation if GETSEC[SEXIT] has not been executed since the last execution of GETSEC[SENTER]. A logical processor is outside SMX operation if GETSEC[SENTER] has not been executed or if GETSEC[SEXIT] was executed after the last execution of GETSEC[SENTER]. See Chapter 6, "Safer Mode Extensions Reference," in *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2B*.

...

26. Updates to Chapter 30, Volume 3B

Change bars show changes to Chapter 30 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3B:* System Programming Guide, Part 2.

...

30.2.2.3 Full-Width Writes to Performance Counter Registers

The general-purpose performance counter registers IA32_PMCx are writable via WRMSR instruction. However, the value written into IA32_PMCx by WRMSR is the signed extended 64-bit value of the EAX[31:0] input of WRMSR.

A processor that supports full-width writes to the general-purpose performance counters enumerated by CPUID.0AH:EAX[15:8] will set IA32_PERF_CAPABILITIES[13] to enumerate its full-width-write capability See Figure 30-37.

If IA32_PERF_CAPABILITIES.FW_WRITE[bit 13] =1, each IA32_PMCi is accompanied by a corresponding alias address starting at 4C1H for IA32_A_PMC0.

If IA32_A_PMCi is present, the 64-bit input value (EDX:EAX) of WRMSR to IA32_A_PMCi will cause IA32_PMCi to be updated by:

IA32_PMCi[63:32] ← SignExtend(EDX[N-32:0]);

IA32_PMCi[31:0] ← EAX[31:0];

...

Table 30-11 Requirements to Program PEBS

	For Processors based on Intel Core microarchitecture	For Processors based on Intel NetBurst microarchitecture
Verify PEBS support of processor/OS	▪ IA32_MISC_ENABLE.EMON_AVAILABE (bit 7) is set. ▪ IA32_MISC_ENABLE.PEBS_UNAVAILABE (bit 12) is clear.	
Ensure counters are in disabled	On initial set up or changing event configurations, write MSR_PERF_GLOBAL_CTRL MSR (0x38F) with 0. On subsequent entries: ▪ Clear all counters if "Counter Freeze on PMI" is not enabled. ▪ If IA32_DebugCTL.Freeze is enabled, counters are automatically disabled. Counters MUST be stopped before writing.[1]	Optional
Disable PEBS.	Clear ENABLE PMC0 bit in IA32_PEBS_ENABLE MSR (0x3F1).	Optional

Table 30-11 Requirements to Program PEBS (Continued)

	For Processors based on Intel Core microarchitecture	For Processors based on Intel NetBurst microarchitecture
Check overflow conditions.	Check MSR_PERF_GLOBAL_STATUS MSR (0x 38E) handle any overflow conditions.	Check OVF flag of each CCCR for overflow condition
Clear overflow status.	Clear MSR_PERF_GLOBAL_STATUS MSR (0x 38E) using IA32_PERF_GLOBAL_OVF_CTRL MSR (0x390).	Clear OVF flag of each CCCR.
Write "sample-after" values.	Configure the counter(s) with the sample after value.	
Configure specific counter configuration MSR.	▪ Set local enable bit 22 - 1. ▪ Do NOT set local counter PMI/ INT bit, bit 20 - 0. ▪ Event programmed must be PEBS capable.	▪ Set appropriate OVF_PMI bits - 1. ▪ Only CCCR for MSR_IQ_COUNTER4 support PEBS.
Allocate buffer for PEBS states.	Allocate a buffer in memory for the precise information.	
Program the IA32_DS_AREA MSR.	Program the IA32_DS_AREA MSR.	
Configure the PEBS buffer management records.	Configure the PEBS buffer management records in the DS buffer management area.	
Configure/Enable PEBS.	Set Enable PMC0 bit in IA32_PEBS_ENABLE MSR (0x3F1).	Configure MSR_PEBS_ENABLE, MSR_PEBS_MATRIX_VERT and MSR_PEBS_MATRIX_HORZ as needed.
Enable counters.	Set Enable bits in MSR_PERF_GLOBAL_CTRL MSR (0x38F).	Set each CCCR enable bit 12 - 1.

NOTES:
1. Counters read while enabled are not guaranteed to be precise with event counts that occur in timing proximity to the RDMSR.

…

30.6 PERFORMANCE MONITORING FOR PROCESSORS BASED ON INTEL® MICROARCHITECTURE CODE NAME NEHALEM

Intel Core i7 processor family[1] supports architectural performance monitoring capability with version ID 3 (see Section 30.2.2.2) and a host of non-architectural monitoring capabilities. The Intel Core i7 processor family is based on Intel® microarchitecture code name Nehalem, and provides four general-purpose performance counters (IA32_PMC0, IA32_PMC1, IA32_PMC2, IA32_PMC3) and three fixed-function performance counters (IA32_FIXED_CTR0, IA32_FIXED_CTR1, IA32_FIXED_CTR2) in the processor core.

Non-architectural performance monitoring in Intel Core i7 processor family uses the IA32_PERFEVTSELx MSR to configure a set of non-architecture performance monitoring events to be counted by the corresponding general-purpose performance counter. The list of non-architectural performance monitoring events is listed in Table A-10. Non-architectural performance monitoring events fall into two broad categories:

- Performance monitoring events in the processor core: These include many events that are similar to performance monitoring events available to processor based on Intel Core microarchitecture. Additionally, there are several enhancements in the performance monitoring capability for detecting microarchitectural conditions in the processor core or in the interaction of the processor core to the off-core sub-systems in the physical processor package. The off-core sub-systems in the physical processor package is loosely referred to as "uncore".
- Performance monitoring events in the uncore: The uncore sub-system is shared by more than one processor cores in the physical processor package. It provides additional performance monitoring facility outside of IA32_PMCx and performance monitoring events that are specific to the uncore sub-system.

Architectural and non-architectural performance monitoring events in Intel Core i7 processor family support thread qualification using bit 21 of IA32_PERFEVTSELx MSR.

The bit fields within each IA32_PERFEVTSELx MSR are defined in Figure 30-6 and described in Section 30.2.1.1 and Section 30.2.2.2.

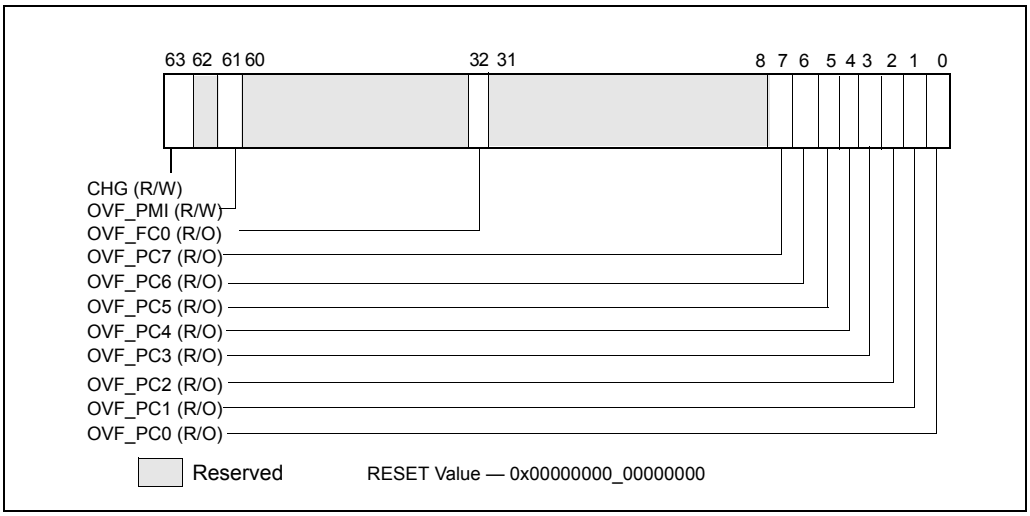


Figure 30-13 IA32_PERF_GLOBAL_STATUS MSR in Intel microarchitecture code name Nehalem

...

1. Intel Xeon processor 5500 series and 3400 series are also based on Intel microarchitecture code name Nehalem, so the performance monitoring facilities described in this section generally also apply.

30.6.1.1 Precise Event Based Sampling (PEBS)

All four general-purpose performance counters, IA32_PMCx, can be used for PEBS if the performance event supports PEBS. Software uses IA32_MISC_ENABLE[7] and IA32_MISC_ENABLE[12] to detect whether the performance monitoring facility and PEBS functionality are supported in the processor. The MSR IA32_PEBS_ENABLE provides 4 bits that software must use to enable which IA32_PMCx overflow condition will cause the PEBS record to be captured.

Additionally, the PEBS record is expanded to allow latency information to be captured. The MSR IA32_PEBS_ENABLE provides 4 additional bits that software must use to enable latency data recording in the PEBS record upon the respective IA32_PMCx overflow condition. The layout of IA32_PEBS_ENABLE for processors based on Intel microarchitecture code name Nehalem is shown in Figure 30-14.

When a counter is enabled to capture machine state (PEBS_EN_PMCx = 1), the processor will write machine state information to a memory buffer specified by software as detailed below. When the counter IA32_PMCx overflows from maximum count to zero, the PEBS hardware is armed.

...

Programming PEBS Facility

Only a subset of non-architectural performance events in the processor support PEBS. The subset of precise events are listed in Table 30-10. In addition to using IA32_PERFEVTSELx to specify event unit/mask settings and setting the EN_PMCx bit in the IA32_PEBS_ENABLE register for the respective counter, the software must also initialize the DS_BUFFER_MANAGEMENT_AREA data structure in memory to support capturing PEBS records for precise events.

NOTE

PEBS events are only valid when the following fields of IA32_PERFEVTSELx are all zero: AnyThread, Edge, Invert, CMask.

...

30.6.1.2 Load Latency Performance Monitoring Facility

The load latency facility provides software a means to characterize the average load latency to different levels of cache/memory hierarchy. This facility requires processor supporting enhanced PEBS record format in the PEBS buffer, see Table 30-12. The facility measures latency from micro-operation (uop) dispatch to when data is globally observable (GO).

To use this feature software must assure:

- One of the IA32_PERFEVTSELx MSR is programmed to specify the event unit MEM_INST_RETIRED, and the LATENCY_ABOVE_THRESHOLD event mask must be specified (IA32_PerfEvtSelX[15:0] = 0x100H). The corresponding counter IA32_PMCx will accumulate event counts for architecturally visible loads which exceed the programmed latency threshold specified separately in a MSR. Stores are ignored when this event is programmed. The CMASK or INV fields of the IA32_PerfEvtSelX register used for counting load latency must be 0. Writing other values will result in undefined behavior.
- The MSR_PEBS_LD_LAT_THRESHOLD MSR is programmed with the desired latency threshold in core clock cycles. Loads with latencies greater than this value are

eligible for counting and latency data reporting. The minimum value that may be programmed in this register is 3 (the minimum detectable load latency is 4 core clock cycles).

- The PEBS enable bit in the IA32_PEBS_ENABLE register is set for the corresponding IA32_PMCx counter register. This means that both the PEBS_EN_CTRX and LL_EN_CTRX bits must be set for the counter(s) of interest. For example, to enable load latency on counter IA32_PMC0, the IA32_PEBS_ENABLE register must be programmed with the 64-bit value 0x00000001.00000001.

When the load-latency facility is enabled, load operations are randomly selected by hardware and tagged to carry information related to data source locality and latency. Latency and data source information of tagged loads are updated internally.

When a PEBS assist occurs, the last update of latency and data source information are captured by the assist and written as part of the PEBS record. The PEBS sample after value (SAV), specified in PEBS CounterX Reset, operates orthogonally to the tagging mechanism. Loads are randomly tagged to collect latency data. The SAV controls the number of tagged loads with latency information that will be written into the PEBS record field by the PEBS assists. The load latency data written to the PEBS record will be for the last tagged load operation which retired just before the PEBS assist was invoked.

The load-latency information written into a PEBS record (see Table 30-12, bytes AFH:98H) consists of:

- **Data Linear Address**: This is the linear address of the target of the load operation.
- **Latency Value**: This is the elapsed cycles of the tagged load operation between dispatch to GO, measured in processor core clock domain.
- **Data Source** : The encoded value indicates the origin of the data obtained by the load instruction. The encoding is shown in Table 30-13. In the descriptions local memory refers to system memory physically attached to a processor package, and remote memory referrals to system memory physically attached to another processor package.

...

30.8 PERFORMANCE MONITORING FOR PROCESSORS BASED ON INTEL® MICROARCHITECTURE CODE NAME SANDY BRIDGE

Intel Core i7 processors 2xxx series are based on Intel microarchitecture code name Sandy Bridge, this section describes the performance monitoring facilities provided in the processor core. The core PMU supports architectural performance monitoring capability with version ID 3 (see Section 30.2.2.2) and a host of non-architectural monitoring capabilities.

Architectural performance monitoring events and non-architectural monitoring events are programmed using fixed counters and programmable counters/event select MSRS described in Section 30.2.2.2.

The core PMU's capability is similar to those described in Section 30.6.1 and Section 30.7, with some differences and enhancements relative to Intel microarchitecture code name Westmere summarized in Table 30-18.

Table 30-18 Core PMU Comparison

Box	Sandy Bridge	Westmere	Comment
# of Fixed counters per thread	3	3	Use CPUID to enumerate # of counters
# of general-purpose counters per core	8	8	
Counter width (R,W)	R:48 , W: 32/48	R:48, W:32	see Section 30.2.2.3
# of programmable counters per thread	4 or (8 if a core not shared by two threads)	4	Use CPUID to enumerate # of counters
PEBS Events	See Table 30-20	See Table 30-10	IA32_PMC4-IA32_PMC7 do not support PEBS.
PEBS-Load Latency	Data source/ STLB/Lock encoding; See Section 30.8.4.2	Data source encoding	
PEBS-Precise Store	Section 30.8.4.3	No	
PEBS-PDIR	yes (using precise INST_RETIRED.ALL)	No PDIR, no INST_RETIRED.ALL	
Off-core Response Event	MSR 1A6H and 1A7H; Extended request and response types	MSR 1A6H and 1A7H, limited types	Nehalem supports 1A6H only.

30.8.1 Global Counter Control Facilities In Intel® microarchitecture code name Sandy Bridge

The number of general-purpose performance counters visible to a logical processor can vary across Processors based on Intel microarchitecture code name Sandy Bridge. Software must use CPUID to determine the number performance counters/event select registers (See Section 30.2.1.1).

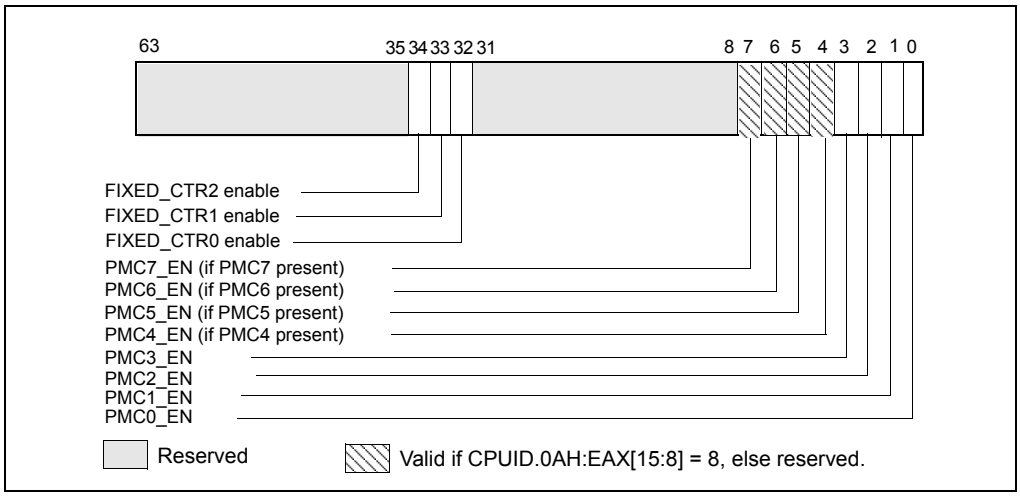


Figure 30-25 IA32_PERF_GLOBAL_CTRL MSR in Intel microarchitecture code name Sandy Bridge

Figure 30-10 depicts the layout of IA32_PERF_GLOBAL_CTRL MSR. The enable bits (PMC4_EN, PMC5_EN, PMC6_EN, PMC7_EN) corresponding to IA32_PMC4-IA32_PMC7 are valid only if CPUID.0AH:EAX[15:8] reports a value of '8'. If CPUID.0AH:EAX[15:8] = 4, attempts to set the invalid bits will cause #GP.

Each enable bit in IA32_PERF_GLOBAL_CTRL is AND'ed with the enable bits for all privilege levels in the respective IA32_PERFEVTSELx or IA32_PERF_FIXED_CTR_CTRL MSRs to start/stop the counting of respective counters. Counting is enabled if the AND'ed results is true; counting is disabled when the result is false.
IA32_PERF_GLOBAL_STATUS MSR provides single-bit status used by software to query the overflow condition of each performance counter. The MSR also provides additional status bit to indicate overflow conditions when counters are programmed for precise-event-based sampling (PEBS). The IA32_PERF_GLOBAL_STATUS MSR also provides a 'sticky bit' to indicate changes to the state of performance monitoring hardware (see Figure 30-26). A value of 1 in each bit of the PMCx_OVF field indicates an overflow condition has occurred in the associated counter.

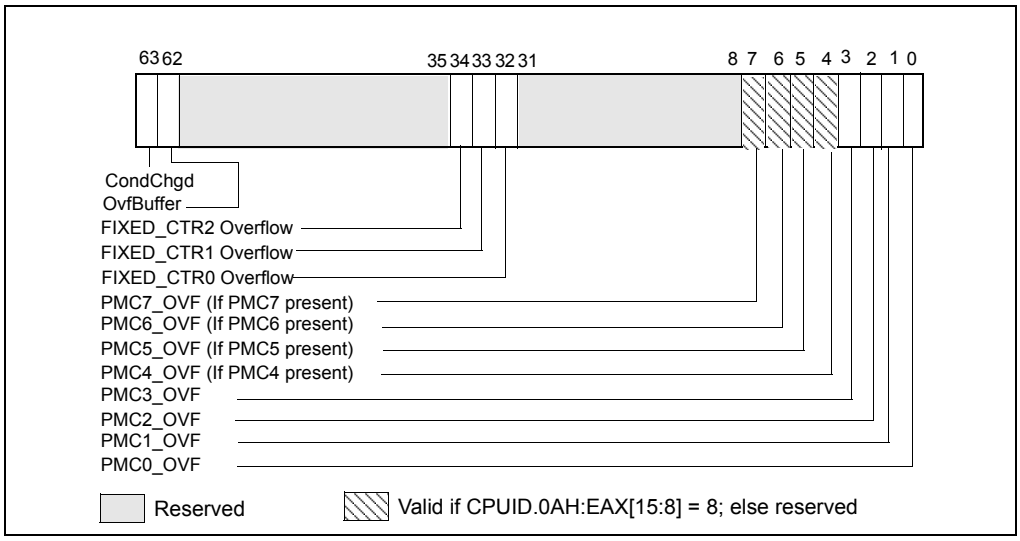


Figure 30-26 IA32_PERF_GLOBAL_STATUS MSR in Intel microarchitecture code name Sandy Bridge

When a performance counter is configured for PEBS, an overflow condition in the counter generates a performance-monitoring interrupt this signals a PEBS event. On a PEBS event, the processor stores data records in the buffer area (see Section 16.4.9), clears the counter overflow status, and sets the OvfBuffer bit in IA32_PERF_GLOBAL_STATUS.

IA32_PERF_GLOBAL_OVF_CTL MSR allows software to clear overflow the indicators for general-purpose or fixed-function counters via a single WRMSR (see Figure 30-27). Clear overflow indications when:

- Setting up new values in the event select and/or UMASK field for counting or sampling
- Reloading counter values to continue sampling
- Disabling event counting or sampling

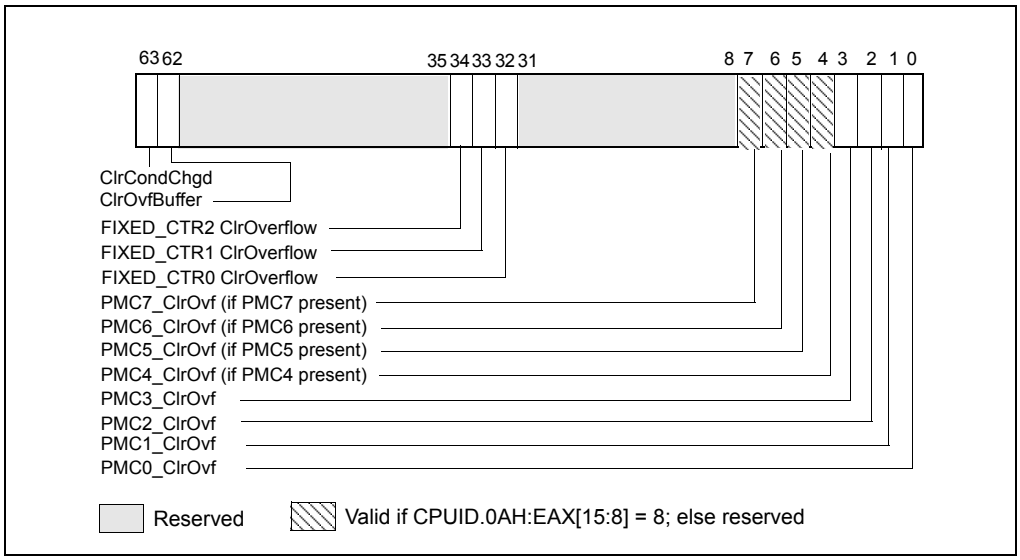


Figure 30-27 IA32_PERF_GLOBAL_OVF_CTRL MSR in Intel microarchitecture code name Sandy Bridge

30.8.2 Counter Coalescence

In processors based on Intel microarchitecture code name Sandy Bridge, each processor core implements eight general-purpose counters. CPUID.0AH:EAX[15:8] will report either 4 or 8 depending specific processor's product features.

If a processor core is shared by two logical processors, each logical processors can access 4 counters (IA32_PMC0-IA32_PMC3). This is the same as in the prior generation for processors based on Intel microarchitecture code name Nehalem.

If a processor core is not shared by two logical processors, all eight general-purpose counters are visible, and CPUID.0AH:EAX[15:8] reports 8. IA32_PMC4-IA32_PMC7 occupy MSR addresses 0C5H through 0C8H. Each counter is accompanied by an event select MSR (IA32_PERFEVTSEL4-IA32_PERFEVTSEL7).

If CPUID.0AH:EAX[15:8] report 4, access to IA32_PMC4-IA32_PMC7, IA32_PMC4-IA32_PMC7 will cause #GP. Writing 1's to bit position 7:4 of IA32_PERF_GLOBAL_CTRL, IA32_PERF_GLOBAL_STATUS, or IA32_PERF_GLOBAL_OVF_CTL will also cause #GP.

30.8.3 Full Width Writes to Performance Counters

Processors based on Intel microarchitecture code name Sandy Bridge support full-width writes to the general-purpose counters, IA32_PMCx. Support of full-width writes are enumerated by IA32_PERF_CAPABILITIES.FW_WRITES[13] (see Section 30.2.2.3).

The default behavior of IA32_PMCx is unchanged, i.e. WRMSR to IA32_PMCx results in a sign-extended 32-bit value of the input EAX written into IA32_PMCx. Full-width writes must issue WRMSR to a dedicated alias MSR address for each IA32_PMCx.

Software must check the presence of full-width write capability and the presence of the alias address IA32_A_PMCx by testing IA32_PERF_CAPABILITIES[13].

30.8.4 PEBS Support in Intel® microarchitecture code name Sandy Bridge

Processors based on Intel microarchitecture code name Sandy Bridge support PEBS, similar to those offered in prior generation, with several enhanced features. The key components and differences of PEBS facility relative to Intel microarchitecture code name Westmere is summarized in Table 30-19.

Table 30-19 PEBS Facility Comparison

Box	Sandy Bridge	Westmere	Comment
Valid IA32_PMCx	PMC0-PMC3	PMC0-PMC3	No PEBS on PMC4-PMC7
PEBS Buffer Programming	Section 30.6.1.1	Section 30.6.1.1	Unchanged
IA32_PEBS_ENABLE Layout	Figure 30-28	Figure 30-14	
PEBS record layout	Physical Layout same as Table 30-12	Table 30-12	Enhanced fields at offsets 98H, A0H, A8H
PEBS Events	See Table 30-20	See Table 30-10	IA32_PMC4-IA32_PMC7 do not support PEBS.
PEBS-Load Latency	See Table 30-21	Table 30-13	
PEBS-Precise Store	yes; see Section 30.8.4.3	No	IA32_PMC3 only
PEBS-PDIR	yes	No	IA32_PMC1 only
SAMPLING Restriction	Small SAV(CountDown) value incur higher overhead than prior generation.		

Only IA32_PMC0 through IA32_PMC3 support PEBS.

NOTE

PEBS events are only valid when the following fields of IA32_PERFEVTSELx are all zero: AnyThread, Edge, Invert, CMask.

In IA32_PEBS_ENABLE MSR, bit 63 is defined as PS_ENABLE: When set, this enables IA32_PMC3 to capture precise store information. Only IA32_PMC3 supports the precise store facility.

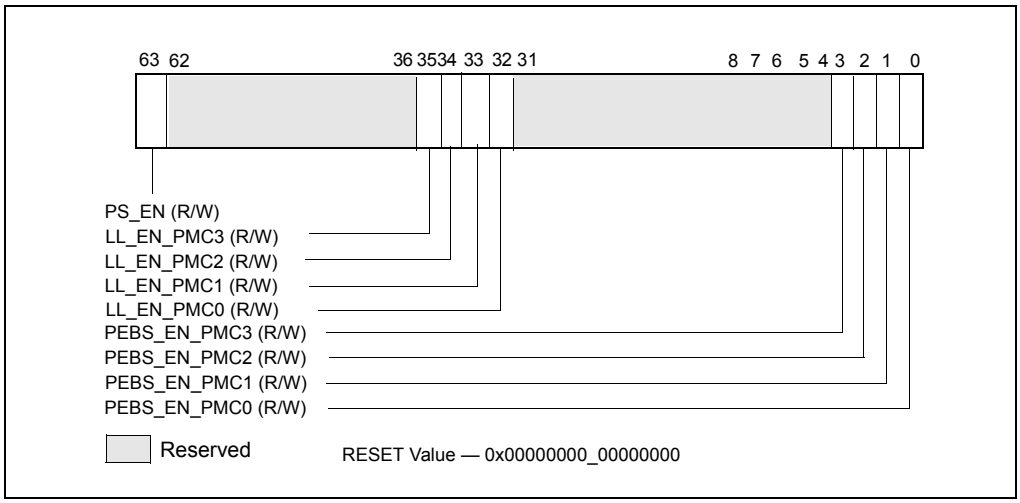


Figure 30-28 Layout of IA32_PEBS_ENABLE MSR

30.8.4.1 PEBS Record Format

The layout of PEBS records physically identical to those shown in Table 30-12, but the fields at offset 98H, A0H and A8H have been enhanced to support additional PEBS capabilities.

- Load/Store Data Linear Address (Offset 98H): This field will contain the linear address of the source of the load, or linear address of the destination of the store.
- Data Source /Store Status (Offset A0H):When load latency is enabled, this field will contain three piece of information (including an encoded value indicating the source which satisfied the load operation). The source field encodings are detailed in Table 30-13. When precise store is enabled, this field will contain information indicating the status of the store, as detailed in Table 19.
- Latency Value/0 (Offset A8H): When load latency is enabled, this field contains the latency in cycles to service the load. This field is not meaningful when precise store is enabled and will be written to zero in that case. Upon writing the PEBS record, microcode clears the overflow status bits in the IA32_PERF_GLOBAL_STATUS corresponding to those counters that both overflowed and were enabled in the IA32_PEBS_ENABLE register. The status bits of other counters remain unaffected.

The number PEBS events has expanded. The list of PEBS events supported in Intel microarchitecture code name Sandy Bridge is shown in Table 30-20.

Table 30-20 PEBS Performance Events for Intel microarchitecture code name Sandy Bridge

Event Name	Event Select	Sub-event	UMask
INST_RETIRED	C0H	All	01H
UOPS_RETIRED	C2H	All	01H
		Retire_Slots	02H

Table 30-20 PEBS Performance Events for Intel microarchitecture (Continued)code name Sandy Bridge

Event Name	Event Select	Sub-event	UMask
BR_INST_RETIRED	C4H	Conditional	01H
		Near_Call	02H
		All_branches	04H
		Near_Return	08H
		Not_Taken	10H
		Near_Taken	20H
		Far_Branches	40H
BR_MISP_RETIRED	C5H	Conditional	01H
		Near_Call	02H
		All_branches	04H
		Not_Taken	10H
		Taken	20H
MEM_TRANS_RETIRED	CDH	Load_Latency	01H
		Precise_Store	02H
MEM_UOP_RETIRED	D0H	Load	01H
		Store	02H
		STLB_Miss	10H
		Lock	20H
		SPLIT	40H
		ALL	80H
MEM_LOAD_UOPS_RETIRED	D1H	L1_Hit	01H
		L2_Hit	02H
		L3_Hit	04H
		Hit_LFB	40H
MEM_LOAD_UOPS_LLC_HIT_RETIRED	D2H	XSNP_Miss	01H
		XSNP_Hit	02H
		XSNP_Hitm	04H
		XSNP_None	08H
MEM_LOAD_UOPS_MISC_RETIRED	D4H	LLC_Miss	02H

30.8.4.2 Load Latency Performance Monitoring Facility

The load latency facility in Intel microarchitecture code name Sandy Bridge is similar to that in prior microarchitecture. It provides software a means to characterize the average load latency to different levels of cache/memory hierarchy. This facility requires processor supporting enhanced PEBS record format in the PEBS buffer, see Table 30-12 and Section 30.8.4.1. The facility measures latency from micro-operation (uop) dispatch to when data is globally observable (GO).

To use this feature software must assure:

- One of the IA32_PERFEVTSELx MSR is programmed to specify the event unit MEM_TRANS_RETIRED, and the LATENCY_ABOVE_THRESHOLD event mask must be specified (IA32_PerfEvtSelX[15:0] = 0x1CDH). The corresponding counter IA32_PMCx will accumulate event counts for architecturally visible loads which exceed the programmed latency threshold specified separately in a MSR. Stores are ignored when this event is programmed. The CMASK or INV fields of the IA32_PerfEvtSelX register used for counting load latency must be 0. Writing other values will result in undefined behavior.
- The MSR_PEBS_LD_LAT_THRESHOLD MSR is programmed with the desired latency threshold in core clock cycles. Loads with latencies greater than this value are eligible for counting and latency data reporting. The minimum value that may be programmed in this register is 3 (the minimum detectable load latency is 4 core clock cycles).
- The PEBS enable bit in the IA32_PEBS_ENABLE register is set for the corresponding IA32_PMCx counter register. This means that both the PEBS_EN_CTRX and LL_EN_CTRX bits must be set for the counter(s) of interest. For example, to enable load latency on counter IA32_PMC0, the IA32_PEBS_ENABLE register must be programmed with the 64-bit value 0x00000001.00000001.
- When Load latency event is enabled, no other PEBS event can be configured with other counters.

When the load-latency facility is enabled, load operations are randomly selected by hardware and tagged to carry information related to data source locality and latency. Latency and data source information of tagged loads are updated internally. The MEM_TRANS_RETIRED event for load latency counts only tagged retired loads. If a load is cancelled it will not be counted and the internal state of the load latency facility will not be updated. In this case the hardware will tag the next available load.

When a PEBS assist occurs, the last update of latency and data source information are captured by the assist and written as part of the PEBS record. The PEBS sample after value (SAV), specified in PEBS CounterX Reset, operates orthogonally to the tagging mechanism. Loads are randomly tagged to collect latency data. The SAV controls the number of tagged loads with latency information that will be written into the PEBS record field by the PEBS assists. The load latency data written to the PEBS record will be for the last tagged load operation which retired just before the PEBS assist was invoked.

The physical layout of the PEBS records is the same as shown in Table 30-12. The specificity of Data Source entry at offset A0H has been enhanced to report three piece of information.

Table 30-21 Layout of Data Source Field of Load Latency Record

Field	Position	Description
Source	3:0	See Table 30-13
STLB_MISS	4	0: The load did not miss the STLB (hit the DTLB or STLB). 1: The load missed the STLB.
Lock	5	0: The load was not part of a locked transaction. 1: The load was part of a locked transaction.
Reserved	63:6	

The layout of MSR_PEBS_LD_LAT_THRESHOLD is the same as shown in Figure 30-16.

30.8.4.3 Precise Store Facility

Processors based on Intel microarchitecture code name Sandy Bridge offer a precise store capability that complements the load latency facility. It provides a means to profile sore memory references in the system.

Precise stores leverage the PEBS facility and provide additional information about sampled stores. Having precise memory reference events with linear address information for both loads and stores can help programmers improve data structure layout, eliminate remote node references, and identify cache-line conflicts in NUMA systems.

Only IA32_PMC3 can be used to capture precise store information. After enabling this facility, counter overflows will initiate the generation of PEBS records as previously described in PEBS. Upon counter overflow hardware captures the linear address and other status information of the next store that retires. This information is then written to the PEBS record.

To enable the precise store facility, software must complete the following steps. Please note that the precise store facility relies on the PEBS facility, so the PEBS configuration requirements must be completed before attempting to capture precise store information.

- Complete the PEBS configuration steps.
- Program the MEM_TRANS_RETIRED.PRECISE_STORE event in IA32_PERFEVTSEL3. Only counter 3 (IA32_PMC3) supports collection of precise store information.
- Set IA32_PEBS_ENABLE[3] and IA32_PEBS_ENABLE[63]. This enables IA32_PMC3 as a PEBS counter and enables the precise store facility, respectively.

The precise store information written into a PEBS record affects entries at offset 98H, A0H and A8H of Table 30-12. The specificity of Data Source entry at offset A0H has been enhanced to report three piece of information.

Table 30-22 Layout of Precise Store Information In PEBS Record

Field	Offset	Description
Store Data Linear Address	98H	The linear address of the destination of the store.
Store Status	A0H	**DCU Hit** (Bit 0): The store hit the data cache closest to the core (lowest latency cache) if this bit is set, otherwise the store missed the data cache. **STLB Miss** (bit 4): The load missed the STLB if set, otherwise the store hit the STLB **Locked Access** (bit 5): The store was part of a locked access if set, otherwise the store was not part of a locked access.
Lock	A0H	0: The load was not part of a locked transaction. 1: The load was part of a locked transaction.
Reserved	A8H	Always 0

30.8.4.4 Precise Distribution of Instructions Retired (PDIR)

Upon triggering a PEBS assist, there will be a finite delay between the time the counter overflows and when the microcode starts to carry out its data collection obligations. INST_RETIRED is a very common event that is used to sample where performance bottleneck happened and to help identify its location in instruction address space. Even

if the delay is constant in core clock space, it invariably manifest as variable "skids" in instruction address space. This creates a challenge for programmers to profile a workload and pinpoint the location of bottlenecks.

The core PMU in processors based on Intel microarchitecture code name Sandy Bridge include a facility referred to as precise distribution of Instruction Retired (PDIR).

The PDIR facility mitigates the "slid" problem by providing an early indication of when the INST_RETIRED counter is about to overflow, allowing the machine to more precisely trap on the instruction that actually caused the counter overflow thus eliminating skid.

PDIR applies only to the INST_RETIRED.ALL precise event, and must use IA32_PMC1 with PerfEvtSel1 property configured and bit 1 in the IA32_PEBS_ENABLE set to 1. INST_RETIRED.ALL is a non-architectural performance event, it is not supported in prior generation microarchitectures.

30.8.4.5 Off-core Response Performance Monitoring

The core PMU in processors based on Intel microarchitecture code name Sandy Bridge provides off-core response facility similar to prior generation. Off-core response can be programed only with a specific pair of event select and counter MSR, and with specific event codes and predefine mask bit value in a dedicated MSR to specify attributes of the off-core transaction. Two event codes are dedicated for off-core response event programming. Each event code for off-core response monitoring requires programming an associated configuration MSR, MSR_OFFCORE_RSP_x. Table 30-23 lists the event code, mask value and additional off-core configuration MSR that must be programmed to count off-core response events using IA32_PMCx.

Table 30-23 Off-Core Response Event Encoding

Counter	Event code	UMask	Required Off-core Response MSR
PMC0	0xB7	0x01	MSR_OFFCORE_RSP_0 (address 0x1A6)
PMC3	0xBB	0x01	MSR_OFFCORE_RSP_1 (address 0x1A7)

The layout of MSR_OFFCORE_RSP_0 and MSR_OFFCORE_RSP_1 are shown in Figure 30-29 and Figure 30-30. Bits 15:0 specifies the request type of a transaction request to the uncore. Bits 30:16 specifies supplier information, bits 37:31 specifies snoop response information.

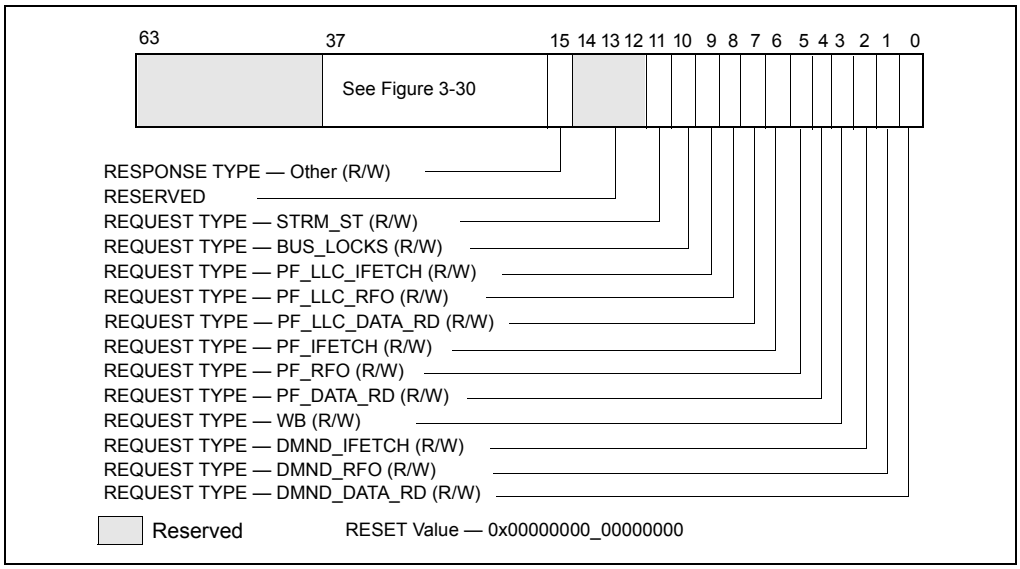


Figure 30-29 Request_Type Fields for MSR_OFFCORE_RSP_x

Table 30-24 MSR_OFFCORE_RSP_x Request_Type Field Definition

Bit Name	Offset	Description
DMND_DATA_RD	0	(R/W). Counts the number of demand and DCU prefetch data reads of full and partial cachelines as well as demand data page table entry cacheline reads. Does not count L2 data read prefetches or instruction fetches.
DMND_RFO	1	(R/W). Counts the number of demand and DCU prefetch reads for ownership (RFO) requests generated by a write to data cacheline. Does not count L2 RFO prefetches.
DMND_IFETCH	2	(R/W). Counts the number of demand and DCU prefetch instruction cacheline reads. Does not count L2 code read prefetches.
WB	3	(R/W). Counts the number of writeback (modified to exclusive) transactions.
PF_DATA_RD	4	(R/W). Counts the number of data cacheline reads generated by L2 prefetchers.
PF_RFO	5	(R/W). Counts the number of RFO requests generated by L2 prefetchers.
PF_IFETCH	6	(R/W). Counts the number of code reads generated by L2 prefetchers.
PF_LLC_DATA_RD	7	(R/W). L2 prefetcher to L3 for loads.
PF_LLC_RFO	8	(R/W). RFO requests generated by L2 prefetcher
PF_LLC_IFETCH	9	(R/W). L2 prefetcher to L3 for instruction fetches.
BUS_LOCKS	10	(R/W). Bus lock and split lock requests
STRM_ST	11	(R/W). Streaming store requests

Table 30-24 MSR_OFFCORE_RSP_x Request_Type Field Definition (Continued)

Bit Name	Offset	Description
OTHER	15	(R/W). Any other request that crosses IDI, including I/O.

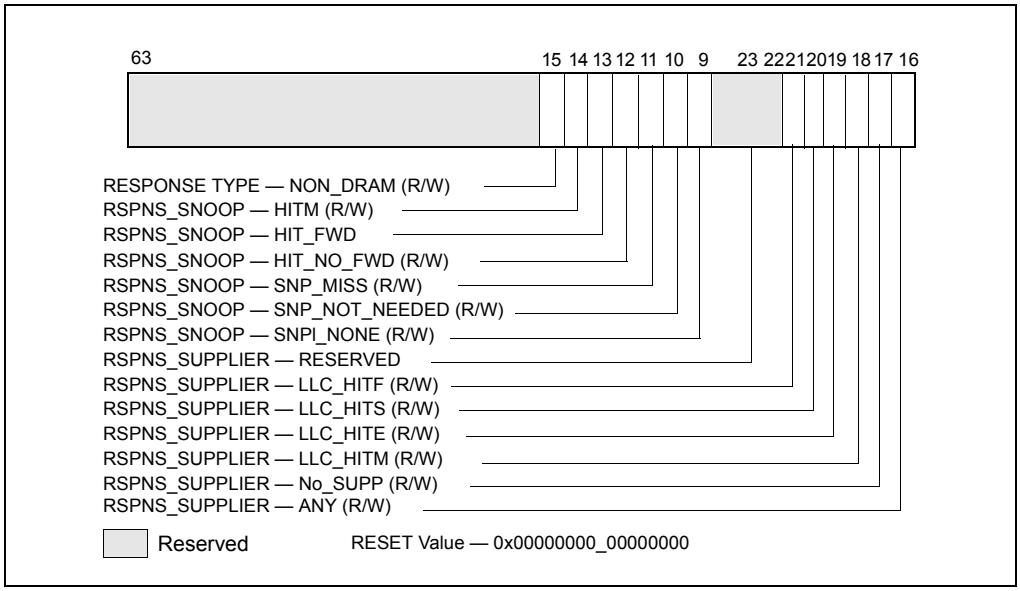


Figure 30-30 Response_Type Fields for MSR_OFFCORE_RSP_x

To properly program this extra register, software must set at least one request type bit and a valid response type pattern. Otherwise, the event count reported will be zero. It is permissible and useful to set multiple request and response type bits in order to obtain various classes of off-core response events.

Table 30-25 MSR_OFFCORE_RSP_x Response Type Field Definition

Subtype	Bit Name	Offset	Description
Common	Any	16	(R/W). Catch all value for any response types.
Supplier Info	NO_SUPP	17	(R/W). No Supplier Information available
	LLC_HITM	18	(R/W). M-state initial lookup stat in L3.
	LLC_HITE	19	(R/W). E-state
	LLC_HITS	20	(R/W). S-state
	LLC_HITF	21	(R/W). F-state
	Reserved	30:22	Reserved
Snoop Info	SNP_NONE	31	(R/W). No details on snoop-related information
	SNP_NOT_NEEDED	32	(R/W). No snoop was needed to satisfy the request.

Table 30-25 MSR_OFFCORE_RSP_x Response Type Field Definition (Continued)

Subtype	Bit Name	Offset	Description
	SNP_MISS	33	(R/W). A snoop was needed and it missed all snooped caches: -For LLC Hit, ReslHitI was returned by all cores -For LLC Miss, RspI was returned by all sockets and data was returned from DRAM.
	SNP_NO_FWD	34	(R/W). A snoop was needed and it hits in at least one snooped cache. Hit denotes a cache-line was valid before snoop effect. This includes: -Snoop Hit w/ Invalidation (LLC Hit, RFO) -Snoop Hit, Left Shared (LLC Hit/Miss, IFetch/Data_RD) -Snoop Hit w/ Invalidation and No Forward (LLC Miss, RFO Hit S) In the LLC Miss case, data is returned from DRAM.
	SNP_FWD	35	(R/W). A snoop was needed and data was forwarded from a remote socket. This includes: -Snoop Forward Clean, Left Shared (LLC Hit/Miss, IFetch/Data_RD/RFT).
	HITM	36	(R/W). A snoop was needed and it HitM-ed in local or remote cache. HitM denotes a cache-line was in modified state before effect as a results of snoop. This includes: -Snoop HitM w/ WB (LLC miss, IFetch/Data_RD) -Snoop Forward Modified w/ Invalidation (LLC Hit/Miss, RFO) -Snoop MtoS (LLC Hit, IFetch/Data_RD).
	NON_DRAM	37	(R/W). Target was non-DRAM system address. This includes MMIO transactions.

To specify a complete offcore response filter, software must properly program bits in the request and response type fields. A valid request type must have at least one bit set in the non-reserved bits of 15:0. A valid response type must be a non-zero value of the following expression:

ANY | [('OR' of Supplier Info Bits) & ('OR' of Snoop Info Bits)]

If "ANY" bit is set, the supplier and snoop info bits are ignored.

...

27. Updates to Appendix A, Volume 3B

Change bars show changes to Appendix A of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3B:* System Programming Guide, Part 2.

...

This appendix lists the performance-monitoring events that can be monitored with the Intel 64 or IA-32 processors. The ability to monitor performance events and the events

that can be monitored in these processors are mostly model-specific, except for architectural performance events, described in Section A.1.

Non-architectural performance events (i.e. model-specific events) are listed for each generation of microarchitecture:

- Section A.2 - Processors based on Intel® microarchitecture code name Sandy Bridge
- Section A.3 - Processors based on Intel® microarchitecture code name Nehalem
- Section A.4 - Processors based on Intel® microarchitecture code name Westmere
- Section A.5 - Processors based on Enhanced Intel® Core™ microarchitecture
- Section A.6 - Processors based on Intel® Core™ microarchitecture
- Section A.7 - Processors based on Intel® Atom™ microarchitecture
- Section A.8 - Intel® Core™ Solo and Intel® Core™ Duo processors
- Section A.9 - Processors based on Intel NetBurst® microarchitecture
- Section A.10 - Pentium® M family processors
- Section A.11 - P6 family processors
- Section A.12 - Pentium® processors

...

A.2 PERFORMANCE MONITORING EVENTS FOR INTEL® CORE™ PROCESSOR 2XXX SERIES

Second generation Intel® Core™ Processor 2xxx Series are based on the Intel microarchitecture code name Sandy Bridge. They support the architectural and non-architectural performance-monitoring events listed in Table A-1 and Table A-3. The events in Table A-3 apply to processors with CPUID signature of DisplayFamily_DisplayModel encoding with the following values: 06_2AH.

Table A-2 Non-Architectural Performance Events In the Processor Core for Intel Core Processor 2xxx Series

Event Num.	Umask Value	Event Mask Mnemonic	Description	Comment
03H	01H	LD_BLOCKS.DATA_UNKNOWN	blocked loads due to store buffer blocks with unknown data.	
03H	02H	LD_BLOCKS.STORE_FORWARD	loads blocked by overlapping with store buffer that cannot be forwarded .	
03H	08H	LD_BLOCKS.NO_SR	# of Split loads blocked due to resource not available.	
03H	10H	LD_BLOCKS.ALL_BLOCK	Number of cases where any load is blocked but has no DCU miss.	
05H	01H	MISALIGN_MEM_REF.LOADS	Speculative cache-line split load uops dispatched to L1D.	
05H	02H	MISALIGN_MEM_REF.STORES	Speculative cache-line split Store-address uops dispatched to L1D.	

Table A-2 Non-Architectural Performance Events In the Processor Core for Intel Core Processor 2xxx Series

Event Num.	Umask Value	Event Mask Mnemonic	Description	Comment
07H	01H	LD_BLOCKS_PARTIAL.ADDRESS_ALIAS	False dependencies in MOB due to partial compare on address.	
07H	08H	LD_BLOCKS_PARTIAL.ALL_STA_BLOCK	The number of times that load operations are temporarily blocked because of older stores, with addresses that are not yet known. A load operation may incur more than one block of this type.	
08H	01H	DTLB_LOAD_MISSES.CAUSES_A_WALK	Misses in all TLB levels that cause a page walk of any page size.	
08H	02H	DTLB_LOAD_MISSES.WALK_COMPLETED	Misses in all TLB levels that caused page walk completed of any size.	
08H	04H	DTLB_LOAD_MISSES.WALK_DURATION	Cycle PMH is busy with a walk.	
08H	10H	DTLB_LOAD_MISSES.STLB_HIT	Number of cache load STLB hits. No page walk.	
0DH	03H	INT_MISC.RECOVERY_CYCLES	Cycles waiting to be recover after Machine Clears due to all other cases except JEClear. Set Cmask= 1.	Set Edge to count occurrences
0DH	40H	INT_MISC.RAT_STALL_CYCLES	Cycles RAT external stall is sent to IDQ for this thread.	
0EH	01H	UOPS_ISSUED.ANY	Increments each cycle the # of Uops issued by the RAT to RS. Set Cmask = 1, Inv = 1, Any= 1to count stalled cycles of this core.	Set Cmask = 1, Inv = 1to count stalled cycles
14H	01H	ARITH.FPU_DIV_ACTIVE	Cycles that the divider is active, includes INT and FP. Set 'edge =1, cmask=1' to count the number of divides.	
17H	01H	INSTS_WRITTEN_TO_IQ.INSTS	Counts the number of instructions written into the IQ every cycle.	
24H	01H	L2_RQSTS.ALL_DEMAND_DATA_RD_HIT	Demand Data Read requests that hit L2 cache	
24H	03H	L2_RQSTS.ALL_DEMAND_DATA_RD	Counts any data load requests to L2.	
24H	04H	L2_RQSTS.RFO_HITS	Counts the number of store RFO requests that hit the L2 cache.	
24H	08H	L2_RQSTS.RFO_MISS	Counts the number of store RFO requests that miss the L2 cache.	
24H	0CH	L2_RQSTS.RFO_ANY	Counts all L2 store RFO requests.	
24H	10H	L2_RQSTS.CODE_RD_HIT	Number of instruction fetches that hit the L2 cache.	

Table A-2 Non-Architectural Performance Events In the Processor Core for Intel Core Processor 2xxx Series

Event Num.	Umask Value	Event Mask Mnemonic	Description	Comment
24H	20H	L2_RQSTS.CODE_RD_MISS	Number of instruction fetches that missed the L2 cache.	
24H	30H	L2_RQSTS.ALL_CODE_RD	Counts all L2 code requests.	
24H	40H	L2_RQSTS.PF_HIT	Requests from L2 Hardware prefetcher that hit L2.	
24H	80H	L2_RQSTS.PF_MISS	Requests from L2 Hardware prefetcher that missed L2.	
24H	C0H	L2_RQSTS.ALL_PF	Any requests from L2 Hardware prefetchers	
27H	01H	L2_STORE_LOCK_RQSTS.MISS	RFOs that miss cache lines	
27H	04H	L2_STORE_LOCK_RQSTS.HIT_E	RFOs that hit cache lines in E state	
27H	08H	L2_STORE_LOCK_RQSTS.HIT_M	RFOs that hit cache lines in M state	
27H	0FH	L2_STORE_LOCK_RQSTS.ALL	RFOs that access cache lines in any state	
28H	04H	L2_L1D_WB_RQSTS.HIT_E	Not rejected writebacks from L1D to L2 cache lines in E state.	
28H	08H	L2_L1D_WB_RQSTS.HIT_M	Not rejected writebacks from L1D to L2 cache lines in M state.	
2EH	4FH	L3_LAT_CACHE.REFERENCE	This event counts requests originating from the core that reference a cache line in the last level cache.	see Table A-1
2EH	41H	L3_LAT_CACHE.MISS	This event counts each cache miss condition for references to the last level cache.	see Table A-1
3CH	00H	CPU_CLK_UNHALTED.THREAD_P	Counts the number of thread cycles while the thread is not in a halt state. The thread enters the halt state when it is running the HLT instruction. The core frequency may change from time to time due to power or thermal throttling.	see Table A-1
3CH	01H	CPU_CLK_UNHALTED.REF_P	Increments at the frequency of TSC when not halted.	see Table A-1
48H	01H	L1D_PEND_MISS.PENDING	Increments the number of outstanding L1D misses every cycle. Set Cmaks = 1 and Edge =1 to count occurrences.	Counter 2 only; Set Cmask = 1 to count cycles.

Table A-2 Non-Architectural Performance Events In the Processor Core for Intel Core Processor 2xxx Series

Event Num.	Umask Value	Event Mask Mnemonic	Description	Comment
49H	01H	DTLB_STORE_MISSES.MISS_CAUSES_A_WALK	Miss in all TLB levels causes an page walk of any page size (4K/2M/4M/1G).	
49H	02H	DTLB_STORE_MISSES.WALK_COMPLETED	Miss in all TLB levels causes a page walk that completes of any page size (4K/2M/4M/1G).	
49H	04H	DTLB_STORE_MISSES.WALK_DURATION	Cycles PMH is busy with this walk.	
49H	10H	DTLB_STORE_MISSES.STLB_HIT	Store operations that miss the first TLB level but hit the second and do not cause page walks	
4CH	01H	LOAD_HIT_PRE.SW_PF	Not SW-prefetch load dispatches that hit fill buffer allocated for S/W prefetch.	
4CH	02H	LOAD_HIT_PRE.HW_PF	Not SW-prefetch load dispatches that hit fill buffer allocated for H/W prefetch.	
4EH	02H	HW_PRE_REQ.DL1_MISS	Hardware Prefetch requests that miss the L1D cache. A request is being counted each time it access the cache & miss it, including if a block is applicable or if hit the Fill Buffer for example.	This accounts for both L1 streamer and IP-based (IPP) HW prefetchers.
51H	01H	L1D.REPLACEMENT	Counts the number of lines brought into the L1 data cache.	
51H	02H	L1D.ALLOCATED_IN_M	Counts the number of allocations of modified L1D cache lines.	
51H	04H	L1D.M_EVICT	Counts the number of modified lines evicted from the L1 data cache due to replacement.	
51H	08H	L1D.ALL_M_REPLACEMENT	Cache lines in M state evicted out of L1D due to Snoop HitM or dirty line replacement	
59H	20H	PARTIAL_RAT_STALLS.FLAGS_MERGE_UOP	Increments the number of flags-merge uops in flight each cycle. Set Cmask = 1 to count cycles.	
59H	40H	PARTIAL_RAT_STALLS.SLOW_LEA_WINDOW	Cycles with at least one slow LEA uop allocated.	
59H	80H	PARTIAL_RAT_STALLS.MUL_SINGLE_UOP	Number of Multiply packed/scalar single precision uops allocated.	
5BH	0CH	RESOURCE_STALLS2.ALL_FL_EMPTY	Cycles stalled due to free list empty	

Table A-2 Non-Architectural Performance Events In the Processor Core for Intel Core Processor 2xxx Series

Event Num.	Umask Value	Event Mask Mnemonic	Description	Comment
5BH	0FH	RESOURCE_STALLS2 .ALL_PRF_CONTROL	Cycles stalled due to control structures full for physical registers	
5BH	40H	RESOURCE_STALLS2 .BOB_FULL	Cycles Allocator is stalled due Branch Order Buffer.	
5BH	4FH	RESOURCE_STALLS2 .OOO_RSRC	Cycles stalled due to out of order resources full	
5CH	01H	CPL_CYCLES.RING0	Unhalted core cycles when the thread is in ring 0	Use Edge to count transition
5CH	02H	CPL_CYCLES.RING123	Unhalted core cycles when the thread is not in ring 0	
5EH	01H	RS_EVENTS.EMPTY_ CYCLES	Cycles the RS is empty for the thread.	
60H	01H	OFFCORE_REQUEST S_OUTSTANDING.DE MAND_DATA_RD	Offcore outstanding Demand Data Read transactions in SQ to uncore. Set Cmask=1 to count cycles.	
60H	02H	OFFCORE_REQUEST S_OUTSTANDING.DE MAND_CODE_RD	Offcore outstanding Code Read transactions in SQ to uncore. Set Cmask=1 to count cycles.	
60H	04H	OFFCORE_REQUEST S_OUTSTANDING.DE MAND_RFO	Offcore outstanding RFO store transactions in SQ to uncore. Set Cmask=1 to count cycles.	
60H	08H	OFFCORE_REQUEST S_OUTSTANDING.AL L_DATA_RD	Offcore outstanding cacheable data read transactions in SQ to uncore. Set Cmask=1 to count cycles.	
63H	01H	LOCK_CYCLES.SPLIT_ LOCK_UC_LOCK_DUR ATION	Cycles in which the L1D and L2 are locked, due to a UC lock or split lock.	
63H	02H	LOCK_CYCLES.CACHE _LOCK_DURATION	Cycles in which the L1D is locked.	
79H	02H	IDQ.EMPTY	Counts cycles the IDQ is empty.	
79H	04H	IDQ.MITE_UOPS	Increment each cycle # of uops delivered to IDQ from MITE path. Set Cmask = 1 to count cycles.	Can combine Umask 04H and 20H
79H	08H	IDQ.DSB_UOPS	Increment each cycle. # of uops delivered to IDQ from DSB path. Set Cmask = 1 to count cycles.	Can combine Umask 08H and 10H
79H	10H	IDQ.MS_DSB_UOPS	Increment each cycle # of uops delivered to IDQ when MS busy by DSB. Set Cmask = 1 to count cycles MS is busy. Set Cmask=1 and Edge =1 to count MS activations.	Can combine Umask 08H and 10H

Table A-2 Non-Architectural Performance Events In the Processor Core for Intel Core Processor 2xxx Series

Event Num.	Umask Value	Event Mask Mnemonic	Description	Comment
79H	20H	IDQ.MS_MITE_UOPS	Increment each cycle # of uops delivered to IDQ when MS is busy by MITE. Set Cmask = 1 to count cycles.	Can combine Umask 04H and 20H
79H	30H	IDQ.MS_UOPS	Increment each cycle # of uops delivered to IDQ from MS by either DSB or MITE. Set Cmask = 1 to count cycles.	Can combine Umask 04H, 08H and 30H
80H	02H	ICACHE.MISSES	Number of Instruction Cache, Streaming Buffer and Victim Cache Misses. Includes UC accesses.	
85H	01H	ITLB_MISSES.CAUSES_A_WALK	Misses in all ITLB levels that cause page walks	
85H	02H	ITLB_MISSES.WALK_COMPLETED	Misses in all ITLB levels that cause completed page walks	
85H	04H	ITLB_MISSES.WALK_DURATION	Cycle PMH is busy with a walk.	
85H	10H	ITLB_MISSES.STLB_HIT	Number of cache load STLB hits. No page walk.	
87H	01H	ILD_STALL.LCP	Stalls caused by changing prefix length of the instruction.	
87H	04H	ILD_STALL.IQ_FULL	Stall cycles due to IQ is full.	
88H	01H	BR_INST_EXEC.COND	Qualify conditional near branch instructions executed, but not necessarily retired.	Must combine with umask 40H, 80H
88H	02H	BR_INST_EXEC.DIRECT_JMP	Qualify all unconditional near branch instructions excluding calls and indirect branches.	Must combine with umask 40H, 80H
88H	04H	BR_INST_EXEC.INDIRECT_JMP_NON_CALL_RET	Qualify executed indirect near branch instructions that are not calls nor returns.	Must combine with umask 40H, 80H
88H	08H	BR_INST_EXEC.RETURN_NEAR	Qualify indirect near branches that have a return mnemonic.	Must combine with umask 40H, 80H
88H	10H	BR_INST_EXEC.DIRECT_NEAR_CALL	Qualify unconditional near call branch instructions, excluding non call branch, executed.	Must combine with umask 40H, 80H
88H	20H	BR_INST_EXEC.INDIRECT_NEAR_CALL	Qualify indirect near calls, including both register and memory indirect, executed.	Must combine with umask 40H, 80H
88H	40H	BR_INST_EXEC.NONTAKEN	Qualify non-taken near branches executed. Must combine with 01H,02H, 04H, 08H, 10H, 20H	

Table A-2 Non-Architectural Performance Events In the Processor Core for Intel Core Processor 2xxx Series

Event Num.	Umask Value	Event Mask Mnemonic	Description	Comment
88H	80H	BR_INST_EXEC.TAKEN	Qualify taken near branches executed. Must combine with 01H,02H, 04H, 08H, 10H, 20H	
88H	FFH	BR_INST_EXEC.ALL_BRANCHES	Counts all near executed branches (not necessarily retired).	
89H	01H	BR_MISP_EXEC.COND	Qualify conditional near branch instructions mispredicted.	Must combine with umask 40H, 80H
89H	04H	BR_MISP_EXEC.INDIRECT_JMP_NON_CALL_RET	Qualify mispredicted indirect near branch instructions that are not calls nor returns.	Must combine with umask 40H, 80H
89H	08H	BR_MISP_EXEC.RETURN_NEAR	Qualify mispredicted indirect near branches that have a return mnemonic.	Must combine with umask 40H, 80H
89H	10H	BR_MISP_EXEC.DIRECT_NEAR_CALL	Qualify mispredicted unconditional near call branch instructions, excluding non call branch, executed.	Must combine with umask 40H, 80H
89H	20H	BR_MISP_EXEC.INDIRECT_NEAR_CALL	Qualify mispredicted indirect near calls, including both register and memory indirect, executed.	Must combine with umask 40H, 80H
89H	40H	BR_MISP_EXEC.NONTAKEN	Qualify mispredicted non-taken near branches executed,. Must combine with 01H,02H, 04H, 08H, 10H, 20H	
89H	80H	BR_MISP_EXEC.TAKEN	Qualify mispredicted taken near branches executed. Must combine with 01H,02H, 04H, 08H, 10H, 20H	
89H	FFH	BR_MISP_EXEC.ALL_BRANCHES	Counts all near executed branches (not necessarily retired).	
9CH	01H	IDQ_UOPS_NOT_DELIVERED.CORE	Count number of non-delivered uops to RAT per thread.	Use Cmask to qualify uop b/w
A1H	01H	UOPS_DISPATCHED_PORT.PORT_0	Cycles which a Uop is dispatched on port 0.	
A1H	02H	UOPS_DISPATCHED_PORT.PORT_1	Cycles which a Uop is dispatched on port 1.	
A1H	04H	UOPS_DISPATCHED_PORT.PORT_2_LD	Cycles which a load uop is dispatched on port 2.	
A1H	08H	UOPS_DISPATCHED_PORT.PORT_2_STA	Cycles which a store address uop is dispatched on port 2.	
A1H	0CH	UOPS_DISPATCHED_PORT.PORT_2	Cycles which a Uop is dispatched on port 2.	
A1H	10H	UOPS_DISPATCHED_PORT.PORT_3_LD	Cycles which a load uop is dispatched on port 3.	

Table A-2 Non-Architectural Performance Events In the Processor Core for Intel Core Processor 2xxx Series

Event Num.	Umask Value	Event Mask Mnemonic	Description	Comment
A1H	20H	UOPS_DISPATCHED_PORT.PORT_3_STA	Cycles which a store address uop is dispatched on port 3.	
A1H	30H	UOPS_DISPATCHED_PORT.PORT_3	Cycles which a Uop is dispatched on port 3.	
A1H	40H	UOPS_DISPATCHED_PORT.PORT_4	Cycles which a Uop is dispatched on port 4.	
A1H	80H	UOPS_DISPATCHED_PORT.PORT_5	Cycles which a Uop is dispatched on port 5.	
A2H	01H	RESOURCE_STALLS.ANY	Cycles Allocation is stalled due to Resource Related reason.	
A2H	02H	RESOURCE_STALLS.LB	Counts the cycles of stall due to lack of load buffers.	
A2H	04H	RESOURCE_STALLS.RS	Cycles stalled due to no eligible RS entry available.	
A2H	08H	RESOURCE_STALLS.B	Cycles stalled due to no store buffers available. (not including draining form sync).	
A2H	10H	RESOURCE_STALLS.ROB	Cycles stalled due to re-order buffer full.	
A2H	20H	RESOURCE_STALLS.FCSW	Cycles stalled due to writing the FPU control word.	
A2H	40H	RESOURCE_STALLS.MXCSR	Cycles stalled due to the MXCSR register rename occurring to close to a previous MXCSR rename.	
A2H	80H	RESOURCE_STALLS.OTHER	Cycles stalled while execution was stalled due to other resource issues.	
ABH	01H	DSB2MITE_SWITCHES.COUNT	Number of DSB to MITE switches.	
ABH	02H	DSB2MITE_SWITCHES.PENALTY_CYCLES	Cycles SB to MITE switches caused delay.	
ACH	02H	DSB_FILL.OTHER_CANCEL	Cases of cancelling valid DSB fill not because of exceeding way limit	
ACH	08H	DSB_FILL.EXCEED_DSB_LINES	DSB Fill encountered > 3 DSB lines.	
ACH	0AH	DSB_FILL.ALL_CANCEL	Cases of cancelling valid Decode Stream Buffer (DSB) fill not because of exceeding way limit	
AEH	01H	ITLB.ITLB_FLUSH	Counts the number of ITLB flushes, includes 4k/2M/4M pages.	
B0H	01H	OFFCORE_REQUESTS.DEMAND_DATA_RD	Demand data read requests sent to uncore.	

Table A-2 Non-Architectural Performance Events In the Processor Core for Intel Core Processor 2xxx Series

Event Num.	Umask Value	Event Mask Mnemonic	Description	Comment
B0H	02H	OFFCORE_REQUESTS.DEMAND_CODE_RD	Offcore code read requests, Include Cacheable and Un-cacheables.	
B0H	04H	OFFCORE_REQUESTS.DEMAND_RFO	Demand RFO read requests sent to uncore., including regular RFOs, locks, ItoM	
B0H	08H	OFFCORE_REQUESTS.ALL_DATA_RD	Data read requests sent to uncore (demand and prefetch).	
B1H	01H	UOPS_DISPATCHED.THREAD	Counts total number of uops to be dispatched per-thread each cycle. Set Cmask = 1, INV =1 to count stall cycles.	
B1H	02H	UOPS_DISPATCHED.CORE	Counts total number of uops to be dispatched per-core each cycle.	Do not need to set ANY
B2H	01H	OFFCORE_REQUESTS_BUFFER.SQ_FULL	Offcore requests buffer cannot take more entries for this thread core.	
B6H	01H	AGU_BYPASS_CANCEL.COUNT	Counts executed load operations with all the following traits: 1. addressing of the format [base + offset], 2. the offset is between 1 and 2047, 3. the address specified in the base register is in one page and the address [base+offset] is in another page.	
B7H	01H	OFF_CORE_RESPONSE_0	see Section 30.8.4.5, "Off-core Response Performance Monitoring"; PMC0 only.	Requires programming MSR 01A6H
BBH	01H	OFF_CORE_RESPONSE_1	See Section 30.8.4.5, "Off-core Response Performance Monitoring". PMC3 only.	Requires programming MSR 01A7H
BDH	01H	TLB_FLUSH.DTLB_THREAD	DTLB flush attempts of the thread-specific entries	
BDH	20H	TLB_FLUSH.STLB_ANY	Count number of STLB flush attempts	
BFH	01H	L1D_BLOCKS.BANK_CONFLICT	Dispatched loads cancelled due to L1D bank conflicts with other load ports	
BFH	05H	L1D_BLOCKS.BANK_CONFLICT_CYCLES	Cycles when dispatched loads are cancelled due to L1D bank conflicts with other load ports	
C0H	00H	INST_RETIRED.ANY_P	See Table A-1	

Table A-2 Non-Architectural Performance Events In the Processor Core for Intel Core Processor 2xxx Series

Event Num.	Umask Value	Event Mask Mnemonic	Description	Comment
C0H	01H	INST_RETIRED.PREC_DIST	Precise instruction retired event with HW to reduce effect of PEBS shaddow in IP distribution	PMC1 only
C1H	02H	OTHER_ASSISTS.ITLB_MISS_RETIRED	Instructions that experienced an ITLB miss.	
C1H	10H	OTHER_ASSISTS.AVX_TO_SSE	Number of transitions from AVX-256 to legacy SSE when penalty applicable.	
C1H	20H	OTHER_ASSISTS.SSE_TO_AVX	Number of transitions from SSE to AVX-256 when penalty applicable.	
C2H	01H	UOPS_RETIRED.ALL	Counts the number of micro-ops retired, Use cmask=1 and invert to count active cycles or stalled cycles.	Supports PEBS
C2H	02H	UOPS_RETIRED.RETIRE_SLOTS	Counts the number of retirement slots used each cycle.	
C3H	02H	MACHINE_CLEARS.MEMORY_ORDERING	Counts the number of machine clears due to memory order conflicts.	
C3H	04H	MACHINE_CLEARS.SMC	Counts the number of times that a program writes to a code section.	
C3H	20H	MACHINE_CLEARS.MASKMOV	Counts the number of executed AVX masked load operations that refer to an illegal address range with the mask bits set to 0.	
C4H	00H	BR_INST_RETIRED.ALL_BRANCHES	See Table A-1	
C4H	01H	BR_INST_RETIRED.CONDITIONAL	Counts the number of conditional branch instructions retired.	Supports PEBS
C4H	02H	BR_INST_RETIRED.NEAR_CALL	Direct and indirect near call instructions retired.	
C4H	04H	BR_INST_RETIRED.ALL_BRANCHES	Counts the number of branch instructions retired.	
C4H	08H	BR_INST_RETIRED.NEAR_RETURN	Counts the number of near return instructions retired.	
C4H	10H	BR_INST_RETIRED.NOT_TAKEN	Counts the number of not taken branch instructions retired.	
C4H	20H	BR_INST_RETIRED.NEAR_TAKEN	Number of near taken branches retired.	
C4H	40H	BR_INST_RETIRED.FAR_BRANCH	Number of far branches retired.	
C5H	00H	BR_MISP_RETIRED.ALL_BRANCHES	See Table A-1	

Table A-2 Non-Architectural Performance Events In the Processor Core for Intel Core Processor 2xxx Series

Event Num.	Umask Value	Event Mask Mnemonic	Description	Comment
C5H	01H	BR_MISP_RETIRED.CONDITIONAL	Mispredicted conditional branch instructions retired.	Supports PEBS
C5H	02H	BR_MISP_RETIRED.NEAR_CALL	Direct and indirect mispredicted near call instructions retired.	
C5H	04H	BR_MISP_RETIRED.ALL_BRANCHES	Mispredicted macro branch instructions retired.	
C5H	10H	BR_MISP_RETIRED.NOT_TAKEN	Mispredicted not taken branch instructions retired.	
C5H	20H	BR_MISP_RETIRED.TAKEN	Mispredicted taken branch instructions retired.	
CAH	02H	FP_ASSIST.X87_OUTPUT	Number of X87 assists due to output value.	
CAH	04H	FP_ASSIST.X87_INPUT	Number of X87 assists due to input value.	
CAH	08H	FP_ASSIST.SIMD_OUTPUT	Number of SIMD FP assists due to Output values	
CAH	10H	FP_ASSIST.SIMD_INPUT	Number of SIMD FP assists due to input values	
CAH	1EH	FP_ASSIST.ANY	Cycles with any input/output SSE* or FP assists	
CBH	01H	HW_INTERRUPTS.RECEIVED	Number of hardware interrupts received by the processor.	
CCH	20H	ROB_MISC_EVENTS.LBR_INSERTS	Count cases of saving new LBR records by hardware.	
CDH	01H	MEM_TRANS_RETIRED.LOAD_LATENCY	Count Loads with specified latency threshold. PMC3 only.	Specify threshold in MSR 0x3F6
D0H	01H	MEM_UOP_RETIRED.LOADS	Qualify retired memory uops that are loads. Combine with umask 10H, 20H, 40H, 80H.	Supports PEBS
D0H	02H	MEM_UOP_RETIRED.STORES	Qualify retired memory uops that are stores. Combine with umask 10H, 20H, 40H, 80H.	
D0H	10H	MEM_UOP_RETIRED.STLB_MISS	Qualify retired memory uops with STLB miss. Must combine with umask 01H, 02H, to produce counts.	
D0H	20H	MEM_UOP_RETIRED.LOCK	Qualify retired memory uops with lock. Must combine with umask 01H, 02H, to produce counts.	
D0H	40H	MEM_UOP_RETIRED.SPLIT	Qualify retired memory uops with line split. Must combine with umask 01H, 02H, to produce counts.	

Table A-2 Non-Architectural Performance Events In the Processor Core for Intel Core Processor 2xxx Series

Event Num.	Umask Value	Event Mask Mnemonic	Description	Comment
D0H	80H	MEM_UOP_RETIRED.ALL	Qualify any retired memory uops. Must combine with umask 01H, 02H, to produce counts.	
D1H	01H	MEM_LOAD_UOPS_RETIRED.L1_HIT	Retired load uops with L1 cache hits as data sources.	Supports PEBS
D1H	02H	MEM_LOAD_UOPS_RETIRED.L2_HIT	Retired load uops with L2 cache hits as data sources.	
D1H	04H	MEM_LOAD_UOPS_RETIRED.LLC_HIT	Retired load uops which data sources were data hits in LLC without snoops required.	
D1H	40H	MEM_LOAD_UOPS_RETIRED.HIT_LFB	Retired load uops which data sources were load uops missed L1 but hit FB due to preceding miss to the same cache line with data not ready.	
D2H	01H	MEM_LOAD_UOPS_LLC_HIT_RETIRED.XSNP_MISS	Retired load uops which data sources were LLC hit and cross-core snoop missed in on-pkg core cache.	Supports PEBS
D2H	02H	MEM_LOAD_UOPS_LLC_HIT_RETIRED.XSNP_HIT	Retired load uops which data sources were LLC and cross-core snoop hits in on-pkg core cache.	
D2H	04H	MEM_LOAD_UOPS_LLC_HIT_RETIRED.XSNP_HITM	Retired load uops which data sources were HitM responses from shared LLC.	
D2H	08H	MEM_LOAD_UOPS_LLC_HIT_RETIRED.XSNP_NONE	Retired load uops which data sources were hits in LLC without snoops required.	
D4H	02H	MEM_LOAD_UOPS_MISC_RETIRED.LLC_MISS	Retired load uops with unknown information as data source in cache serviced the load.	Supports PEBS.
F0H	01H	L2_TRANS.DEMAND_DATA_RD	Demand Data Read requests that access L2 cache	
F0H	02H	L2_TRANS.RFO	RFO requests that access L2 cache	
F0H	04H	L2_TRANS.CODE_RD	L2 cache accesses when fetching instructions	
F0H	08H	L2_TRANS.ALL_PF	L2 or LLC HW prefetches that access L2 cache	including rejects.
F0H	10H	L2_TRANS.L1D_WB	L1D writebacks that access L2 cache	
F0H	20H	L2_TRANS.L2_FILL	L2 fill requests that access L2 cache	
F0H	40H	L2_TRANS.L2_WB	L2 writebacks that access L2 cache	
F0H	80H	L2_TRANS.ALL_REQUESTS	Transactions accessing L2 pipe	

Table A-2 Non-Architectural Performance Events In the Processor Core for Intel Core Processor 2xxx Series

Event Num.	Umask Value	Event Mask Mnemonic	Description	Comment
F1H	01H	L2_LINES_IN.I	L2 cache lines in I state filling L2	Counting does not cover rejects.
F1H	02H	L2_LINES_IN.S	L2 cache lines in S state filling L2	Counting does not cover rejects.
F1H	04H	L2_LINES_IN.E	L2 cache lines in E state filling L2	Counting does not cover rejects.
F1H	07H	L2_LINES_IN.ALL	L2 cache lines filling L2	Counting does not cover rejects.
F2H	01H	L2_LINES_OUT.DEMAND_CLEAN	Clean L2 cache lines evicted by demand	
F2H	02H	L2_LINES_OUT.DEMAND_DIRTY	Dirty L2 cache lines evicted by demand	
F2H	04H	L2_LINES_OUT.PF_CLEAN	Clean L2 cache lines evicted by L2 prefetch	
F2H	08H	L2_LINES_OUT.PF_DIRTY	Dirty L2 cache lines evicted by L2 prefetch	
F2H	0AH	L2_LINES_OUT.DIRTY_ALL	Dirty L2 cache lines filling the L2	Counting does not cover rejects.
F4H	10H	SQ_MISC.SPLIT_LOCK	Split locks in SQ	

...

Table A-3 Non-Architectural Performance Events In the Processor Core for Intel Core i7 Processor and Intel Xeon Processor 5500 Series

Event Num.	Umask Value	Event Mask Mnemonic	Description	Comment
04H	07H	SB_DRAIN.ANY	Counts the number of store buffer drains.	
06H	04H	STORE_BLOCKS.AT_RET	Counts number of loads delayed with at-Retirement block code. The following loads need to be executed at retirement and wait for all senior stores on the same thread to be drained: load splitting across 4K boundary (page split), load accessing uncacheable (UC or USWC) memory, load lock, and load with page table in UC or USWC memory region.	
06H	08H	STORE_BLOCKS.L1D_BLOCK	Cacheable loads delayed with L1D block code.	
07H	01H	PARTIAL_ADDRESS_ALIAS	Counts false dependency due to partial address aliasing.	

Table A-3 Non-Architectural Performance Events In the Processor Core for Intel Core i7 Processor and Intel Xeon Processor 5500 Series

Event Num.	Umask Value	Event Mask Mnemonic	Description	Comment
08H	01H	DTLB_LOAD_MISSES.ANY	Counts all load misses that cause a page walk.	
08H	02H	DTLB_LOAD_MISSES.WALK_COMPLETED	Counts number of completed page walks due to load miss in the STLB.	
08H	10H	DTLB_LOAD_MISSES.STLB_HIT	Number of cache load STLB hits.	
08H	20H	DTLB_LOAD_MISSES.PDE_MISS	Number of DTLB cache load misses where the low part of the linear to physical address translation was missed.	
08H	40H	DTLB_LOAD_MISSES.PDP_MISS	Number of DTLB cache load misses where the high part of the linear to physical address translation was missed.	
08H	80H	DTLB_LOAD_MISSES.LARGE_WALK_COMPLETED	Counts number of completed large page walks due to load miss in the STLB.	
0BH	01H	MEM_INST_RETIRED.LOADS	Counts the number of instructions with an architecturally-visible load retired on the architected path.	In conjunction with ld_lat facility
0BH	02H	MEM_INST_RETIRED.STORES	Counts the number of instructions with an architecturally-visible store retired on the architected path.	In conjunction with ld_lat facility
…				

…

Table A-5 Non-Architectural Performance Events In Next Generation Processor Core (Intel microarchitecture code name Westmere)

Event Num.	Umask Value	Event Mask Mnemonic	Description	Comment
03H	02H	LOAD_BLOCK.OVERLAP_STORE	Loads that partially overlap an earlier store.	
04H	07H	SB_DRAIN.ANY	All Store buffer stall cycles.	
05H	02H	MISALIGN_MEMORY.STORE	All store referenced with misaligned address.	

Table A-5 Non-Architectural Performance Events In Next Generation Processor Core (Intel microarchitecture code name Westmere)

Event Num.	Umask Value	Event Mask Mnemonic	Description	Comment
06H	04H	STORE_BLOCKS.AT_RET	Counts number of loads delayed with at-Retirement block code. The following loads need to be executed at retirement and wait for all senior stores on the same thread to be drained: load splitting across 4K boundary (page split), load accessing uncacheable (UC or USWC) memory, load lock, and load with page table in UC or USWC memory region.	
06H	08H	STORE_BLOCKS.L1D_BLOCK	Cacheable loads delayed with L1D block code.	
07H	01H	PARTIAL_ADDRESS_ALIAS	Counts false dependency due to partial address aliasing.	
08H	01H	DTLB_LOAD_MISSES.ANY	Counts all load misses that cause a page walk.	
08H	02H	DTLB_LOAD_MISSES.WALK_COMPLETED	Counts number of completed page walks due to load miss in the STLB.	
08H	04H	DTLB_LOAD_MISSES.WALK_CYCLES	Cycles PMH is busy with a page walk due to a load miss in the STLB.	
08H	10H	DTLB_LOAD_MISSES.STLB_HIT	Number of cache load STLB hits.	
08H	20H	DTLB_LOAD_MISSES.PDE_MISS	Number of DTLB cache load misses where the low part of the linear to physical address translation was missed.	
0BH	01H	MEM_INST_RETIRED.LOADS	Counts the number of instructions with an architecturally-visible load retired on the architected path.	In conjunction with ld_lat facility
0BH	02H	MEM_INST_RETIRED.STORES	Counts the number of instructions with an architecturally-visible store retired on the architected path.	In conjunction with ld_lat facility
…				

…

28. Updates to Appendix B, Volume 3B

Change bars show changes to Appendix B of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3B:* System Programming Guide, Part 2.

...

Table B-1 CPUID Signature Values of DisplayFamily_DisplayModel

DisplayFamily_DisplayModel	Processor Families/Processor Number Series
06_2AH	Second Generation Intel Core Processor 2xxx Series
06_2DH	Next Generation Intel Xeon Processor
06_1AH	Intel Core i7 Processor, Intel Xeon Processor 5500 series
06_1EH, 06_1FH	Intel Core i7 and i5 Processor,
06_2EH	Intel Xeon Processor 7500 series
06_25H, 06_2CH	Intel Xeon Processor 5600 series, Intel Core i7, i5 and i3 Processor
06_1DH	Intel Xeon Processor MP 7400 series
06_17H	Intel Xeon Processor 5200, 5400 series, Intel Core 2 Quad processors 8000, 9000 series
06_0FH	Intel Xeon Processor 3000, 3200, 5100, 5300, 7300 series, Intel Core 2 Quad processor 6000 series, Intel Core 2 Extreme 6000 series, Intel Core 2 Duo 4000, 5000, 6000, 7000 series processors, Intel Pentium dual-core processors
06_0EH	Intel Core Duo, Intel Core Solo processors
06_0DH	Intel Pentium M processor
06_1CH	Intel Atom processor
0F_06H	Intel Xeon processor 7100, 5000 Series, Intel Xeon Processor MP, Intel Pentium 4, Pentium D processors
0F_03H, 0F_04H	Intel Xeon Processor, Intel Xeon Processor MP, Intel Pentium 4, Pentium D processors
06_09H	Intel Pentium M processor
0F_02H	Intel Xeon Processor, Intel Xeon Processor MP, Intel Pentium 4 processors
0F_0H, 0F_01H	Intel Xeon Processor, Intel Xeon Processor MP, Intel Pentium 4 processors
06_7H, 06_08H, 06_0AH, 06_0BH	Intel Pentium III Xeon Processor, Intel Pentium III Processor
06_03H, 06_05H	Intel Pentium II Xeon Processor, Intel Pentium II Processor
06_01H	Intel Pentium Pro Processor
05_01H, 05_02H, 05_04H	Intel Pentium Processor, Intel Pentium Processor with MMX Technology

...

B.2 ARCHITECTURAL MSRS

Many MSRs have carried over from one generation of IA-32 processors to the next and to Intel 64 processors. A subset of MSRs and associated bit fields, which do not change on future processor generations, are now considered architectural MSRs. For historical reasons (beginning with the Pentium 4 processor), these "architectural MSRs" were given the prefix "IA32_". Table B-2 lists the architectural MSRs, their addresses, their current names, their names in previous IA-32 processors, and bit fields that are considered architectural. MSR addresses outside Table B-2 and certain bitfields in an MSR address that may overlap with architectural MSR addresses are model-specific. Code that accesses a machine specified MSR and that is executed on a processor that does not support that MSR will generate an exception.

Architectural MSR or individual bit fields in an architectural MSR may be introduced or transitioned at the granularity of certain processor family/model or the presence of certain CPUID feature flags. The right-most column of Table B-2 provides information on the introduction of each architectural MSR or its individual fields. This information is expressed either as signature values of "DF_DM" (see Table B-1) or via CPUID flags.

Certain bit field position may be related to the maximum physical address width, the value of which is expressed as "MAXPHYWID" in Table B-2. "MAXPHYWID" is reported by CPUID.8000_0008H leaf.

MSR address range between 40000000H - 400000FFH is marked as a specially reserved range. All existing and future processors will not implement any features using any MSR in this range.

Table B-2 IA-32 Architectural MSRs

Register Address		Architectural MSR Name and bit fields (Former MSR Name)	MSR/Bit Description	Introduced as Architectural MSR
Hex	Decimal			
…				
9BH	155	IA32_SMM_MONITOR_CTL	SMM Monitor Configuration (R/W)	If CPUID.01H: ECX[bit 5 or bit 6] = 1
		0	Valid (R/W)	
		1	Reserved	
		2	Controls SMI unblocking by VMXOFF (see Section 26.14.4)	If IA32_VMX_MISC[bit 28])
		11:3	Reserved	
		31:12	MSEG Base (R/W)	
		63:32	Reserved	
…				
C5H	197	IA32_PMC4	General Performance Counter 4 (R/W)	If CPUID.0AH: EAX[15:8] > 4
C6H	198	IA32_PMC5	General Performance Counter 5 (R/W)	If CPUID.0AH: EAX[15:8] > 5
C7H	199	IA32_PMC6	General Performance Counter 6 (R/W)	If CPUID.0AH: EAX[15:8] > 6

Table B-2 IA-32 Architectural MSRs (Continued)

Register Address		Architectural MSR Name and bit fields (Former MSR Name)	MSR/Bit Description	Introduced as Architectural MSR
Hex	Decimal			
C8H	200	IA32_PMC7	General Performance Counter 7 (R/W)	If CPUID.0AH: EAX[15:8] > 7
...				
210H	528	IA32_MTRR_PHYSBASE8	MTRRphysBase8	if IA32_MTRR_CAP [7:0] > 8
211H	529	IA32_MTRR_PHYSMASK8	MTRRphysMask8	if IA32_MTRR_CAP [7:0] > 8
212H	530	IA32_MTRR_PHYSBASE9	MTRRphysBase9	if IA32_MTRR_CAP [7:0] > 9
213H	531	IA32_MTRR_PHYSMASK9	MTRRphysMask9	if IA32_MTRR_CAP [7:0] > 9
...				
345H	837	IA32_PERF_CAPABILITIES	RO	If CPUID.01H: ECX[15] = 1
		5:0	LBR format	
		6	PEBS Trap	
		7	PEBSSaveArchRegs	
		11:8	PEBS Record Format	
		12	1: Freeze while SMM is supported	
		13	1: Full width of counter writable via IA32_A_PMCx	
		63:14	Reserved	
...				
4C1H	1217	IA32_A_PMC0	Full Width Writable IA32_PMC0 Alias (R/W)	(If CPUID.0AH: EAX[15:8] > 0) & IA32_PERF_CAP ABILITIES[13] = 1
4C2H	1218	IA32_A_PMC1	Full Width Writable IA32_PMC1 Alias (R/W)	(If CPUID.0AH: EAX[15:8] > 1) & IA32_PERF_CAP ABILITIES[13] = 1
4C3H	1219	IA32_A_PMC2	Full Width Writable IA32_PMC2 Alias (R/W)	(If CPUID.0AH: EAX[15:8] > 2) & IA32_PERF_CAP ABILITIES[13] = 1

Table B-2 IA-32 Architectural MSRs (Continued)

Register Address		Architectural MSR Name and bit fields (Former MSR Name)	MSR/Bit Description	Introduced as Architectural MSR
Hex	Decimal			
4C4H	1220	IA32_A_PMC3	Full Width Writable IA32_PMC3 Alias (R/W)	(If CPUID.0AH: EAX[15:8] > 3) & IA32_PERF_CAPABILITIES[13] = 1
4C5H	1221	IA32_A_PMC4	Full Width Writable IA32_PMC4 Alias (R/W)	(If CPUID.0AH: EAX[15:8] > 4) & IA32_PERF_CAPABILITIES[13] = 1
4C6H	1222	IA32_A_PMC5	Full Width Writable IA32_PMC5 Alias (R/W)	(If CPUID.0AH: EAX[15:8] > 5) & IA32_PERF_CAPABILITIES[13] = 1
4C7H	1223	IA32_A_PMC6	Full Width Writable IA32_PMC6 Alias (R/W)	(If CPUID.0AH: EAX[15:8] > 6) & IA32_PERF_CAPABILITIES[13] = 1
4C8H	1224	IA32_A_PMC7	Full Width Writable IA32_PMC7 Alias (R/W)	(If CPUID.0AH: EAX[15:8] > 7) & IA32_PERF_CAPABILITIES[13] = 1
…				

…

Table B-5 MSRs in Processors Based on Intel Microarchitecture code name Nehalem

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
...				
34H	52	MSR_SMI_ COUNT	Thread	**SMI Counter. (R/O).**
		31:0		**SMI Count. (R/O)** Count SMIs
		63:32		Reserved.
...				
CEH	206	MSR_PLATFORM_I NFO	Package	
		7:0		Reserved.
		15:8	Package	**Maximum Non-Turbo Ratio. (R/O)** The is the ratio of the frequency that invariant TSC runs at. The invariant TSC frequency can be computed by multiplying this ratio by 133.33 MHz.
		27:16		Reserved.
		28	Package	**Programmable Ratio Limit for Turbo Mode. (R/O)** When set to 1, indicates that Programmable Ratio Limits for Turbo mode is enabled, and when set to 0, indicates Programmable Ratio Limits for Turbo mode is disabled.
		29	Package	**Programmable TDC-TDP Limit for Turbo Mode. (R/O)** When set to 1, indicates that TDC/TDP Limits for Turbo mode are programmable, and when set to 0, indicates TDC and TDP Limits for Turbo mode are not programmable.
		39:30		Reserved.
		47:40	Package	**Maximum Efficiency Ratio. (R/O)** The is the minimum ratio (maximum efficiency) that the processor can operates, in units of 133.33MHz.
		63:48		Reserved.
E2H	226	MSR_PKG_CST_CO NFIG_CONTROL	Core	**C-State Configuration Control** (R/W) Note: C-state values are processor specific C-state code names, unrelated to MWAIT extension C-state parameters or ACPI C-States.

		2:0		**Package C-State limit. (R/W)** Specifies the lowest processor-specific C-state code name (consuming the least power). for the package. The default is set as factory-configured package C-state limit. The following C-state code name encodings are supported: 000b: C0 (no package C-sate support) 001b: C1 (Behavior is the same as 000b) 010b: C3 011b: C6 100b: C7 101b and 110b: Reserved 111: No package C-state limit. Note: This field cannot be used to limit package C-state to C3.
		9:3		**Reserved.**
		10		**I/O MWAIT Redirection Enable. (R/W)** When set, will map IO_read instructions sent to IO register specified by MSR_PMG_IO_CAPTURE_BASE to MWAIT instructions
		14:11		**Reserved.**
		15		**CFG Lock. (R/WO)** When set, lock bits 15:0 of this register until next reset
		23:16		**Reserved.**
		24		**Interrupt filtering enable. (R/W)** When set, processor cores in a deep C-State will wake only when the event message is destined for that core. When 0, all processor cores in a deep C-State will wake for an event message
		25		**C3 state auto demotion enable. (R/W)** When set, the processor will conditionally demote C6/C7 requests to C3 based on uncore auto-demote information
		26		**C1 state auto demotion enable. (R/W)** When set, the processor will conditionally demote C3/C6/C7 requests to C1 based on uncore auto-demote information
		63:27		Reserved.
...				

19AH	410	IA32_CLOCK_ MODULATION	Thread	**Clock Modulation. (R/W)** see Table B-2 IA32_CLOCK_MODULATION MSR was originally named IA32_THERM_CONTROL MSR.
		0		Reserved
		3:1		**On demand Clock Modulation Duty Cycle (R/W).**
		4		**On demand Clock Modulation Enable (R/W).**
		63:5		Reserved.
…				
1A2H	418	MSR_ TEMPERATURE_TA RGET	Thread	
		15:0		Reserved.
		23:16		**Temperature Target. (R)** The minimum temperature at which PROCHOT# will be asserted. The value is degree C.
		63:24		Reserved
1A6H	422	MSR_OFFCORE_RS P_0	Thread	**Offcore Response Event Select Register** (R/W)
1AAH	426	MSR_MISC_PWR_ MGMT		
		0	Package	**EIST Hardware Coordination Disable (R/W).** When 0, enables hardware coordination of EIST request from processor cores; When 1, disables hardware coordination of EIST requests.
		1	Thread	**Energy/Performance Bias Enable. (R/W)** This bit makes the IA32_ENERGY_PERF_BIAS register (MSR 1B0h) visible to software with Ring 0 privileges. This bit's status (1 or 0) is also reflected by CPUID.(EAX=06h):ECX[3].
		63:2		Reserved
1ACH	428	MSR_TURBO_POW ER_CURRENT_LIMI T		
		14:0	Package	**TDP Limit (R/W)** TDP limit in 1/8 Watt granularity
		15	Package	**TDP Limit Override Enable (R/W)** A value = 0 indicates override is not active, and a value = 1 indicates active
		30:16	Package	**TDC Limit (R/W)** TDC limit in 1/8 Amp granularity

		31	Package	**TDC Limit Override Enable (R/W)** A value = 0 indicates override is not active, and a value = 1 indicates active
		63:32		Reserved
1ADH	429	MSR_TURBO_RATIO_LIMIT	Package	**Maximum Ratio Limit of Turbo Mode.** RO if MSR_PLATFORM_INFO.[28] = 0, RW if MSR_PLATFORM_INFO.[28] = 1
		7:0	Package	**Maximum Ratio Limit for 1C.** Maximum turbo ratio limit of 1 core active.
		15:8	Package	**Maximum Ratio Limit for 2C.** Maximum turbo ratio limit of 2 core active.
		23:16	Package	**Maximum Ratio Limit for 3C.** Maximum turbo ratio limit of 3 core active.
		31:24	Package	**Maximum Ratio Limit for 4C.** Maximum turbo ratio limit of 4 core active.
		63:32		Reserved.
…				
1FCH	508	MSR_POWER_CTL	Core	Power Control Register
		0		Reserved.
		1	Package	**C1E Enable. (R/W)** When set to '1', will enable the CPU to switch to the Minimum Enhanced Intel SpeedStep Technology operating point when all execution cores enter MWAIT (C1).
		63:2		Reserved
…				
210H	528	IA32_MTRR_PHYSBASE8	Thread	see Table B-2
211H	529	IA32_MTRR_PHYSMASK8	Thread	see Table B-2
212H	530	IA32_MTRR_PHYSBASE9	Thread	see Table B-2
213H	531	IA32_MTRR_PHYSMASK9	Thread	see Table B-2
…				
38EH	910	MSR_PERF_GLOBAL_STAUS	Thread	**(RO)**
		61		**UNC_Ovf**. Uncore overflowed if 1.
38FH	911	IA32_PERF_GLOBAL_CTRL	Thread	see Table B-2. See Section 30.4.2, "Global Counter Control Facilities."

390H	912	IA32_PERF_ GLOBAL_OVF_ CTRL	Thread	see Table B-2. See Section 30.4.2, "Global Counter Control Facilities."
390H	912	MSR_PERF_ GLOBAL_OVF_ CTRL	Thread	**(R/W)**
		61		**CLR_UNC_Ovf**. Set 1 to clear UNC_Ovf.
…				

…

B.5 MSRS IN THE INTEL XEON PROCESSOR 5600 SERIES (INTEL® MICROARCHITECTURE CODE NAME WESTMERE)

Intel Xeon processor 5600 series (Intel® microarchitecture code name Westmere) supports the MSR interfaces listed in Table B-5, Table B-6, plus additional MSR listed in Table B-8. These MSRs also apply to Intel Core i7, i5 and i3 processor family with CPUID signature DisplayFamily_DisplayModel of 06_25H and 06_2CH, see Table B-1.

Table B-8 Additional MSRs supported by Intel Processors (Intel microarchitecture code name Westmere)

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
1A7H	423	MSR_OFFCORE_RSP_1	Thread	**Offcore Response Event Select Register** (R/W)
1ADH	429	MSR_TURBO_RATIO_LIMIT	Package	**Maximum Ratio Limit of Turbo Mode.** RO if MSR_PLATFORM_INFO.[28] = 0, RW if MSR_PLATFORM_INFO.[28] = 1
		7:0	Package	**Maximum Ratio Limit for 1C.** Maximum turbo ratio limit of 1 core active.
		15:8	Package	**Maximum Ratio Limit for 2C.** Maximum turbo ratio limit of 2 core active.
		23:16	Package	**Maximum Ratio Limit for 3C.** Maximum turbo ratio limit of 3 core active.
		31:24	Package	**Maximum Ratio Limit for 4C.** Maximum turbo ratio limit of 4 core active.
		39:32	Package	**Maximum Ratio Limit for 5C.** Maximum turbo ratio limit of 5 core active.
		47:40	Package	**Maximum Ratio Limit for 6C.** Maximum turbo ratio limit of 6 core active.
		63:48		Reserved.

Table B-8 Additional MSRs supported by Intel Processors (Continued)(Intel microarchitecture code name Westmere)

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
1B0H	432	IA32_ENERGY_PERF_BIAS	Package	see Table B-2

...

B.6 MSRS IN NEXT GENERATION INTEL® PROCESSOR FAMILY (INTEL® MICROARCHITECTURE CODE NAME SANDY BRIDGE)

Table B-9 lists model-specific registers (MSRs) that are common to next generation Intel® processor family (Intel® microarchitecture code name Sandy Bridge). All architectural MSRs listed in Table B-2 are supported. These processors have a CPUID signature with DisplayFamily_DisplayModel of 06_2AH, 06_2DH, see Table B-1. Additional MSRs specific to 06_2AH are listed in Table B-10.

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
0H	0	IA32_P5_MC_ADDR	Thread	See Appendix B.11, "MSRs in Pentium Processors."
1H	1	IA32_P5_MC_TYPE	Thread	See Appendix B.11, "MSRs in Pentium Processors."
6H	6	IA32_MONITOR_FILTER_SIZE	Thread	See Section 8.10.5, "Monitor/Mwait Address Range Determination." andTable B-2
10H	16	IA32_TIME_STAMP_COUNTER	Thread	See Section 16.12, "Time-Stamp Counter." and see Table B-2
17H	23	IA32_PLATFORM_ID	Package	**Platform ID. (R)** See Table B-2.
1BH	27	IA32_APIC_BASE	Thread	See Section 10.4.4, "Local APIC Status and Location." and Table B-2
34H	52	MSR_SMI_COUNT	Thread	**SMI Counter. (R/O).**
		31:0		**SMI Count. (R/O)** Count SMIs
		63:32		Reserved.
3AH	58	IA32_FEATURE_CONTROL	Thread	**Control Features in Intel 64Processor. (R/W).** see Table B-2

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
79H	121	IA32_BIOS_ UPDT_TRIG	Core	**BIOS Update Trigger Register. (W)** see Table B-2
8BH	139	IA32_BIOS_ SIGN_ID	Thread	**BIOS Update Signature ID. (RO)** see Table B-2
C1H	193	IA32_PMC0	Thread	**Performance counter register.** see Table B-2
C2H	194	IA32_PMC1	Thread	**Performance counter register.** see Table B-2
C3H	195	IA32_PMC2	Thread	**Performance counter register.** see Table B-2
C4H	196	IA32_PMC3	Thread	**Performance counter register.** see Table B-2
C5H	197	IA32_PMC4	Core	**Performance counter register.** see Table B-2
C6H	198	IA32_PMC5	Core	**Performance counter register.** see Table B-2
C7H	199	IA32_PMC6	Core	**Performance counter register.** see Table B-2
C8H	200	IA32_PMC7	Core	**Performance counter register.** see Table B-2
CEH	206	MSR_PLATFORM_I NFO	Package	
		7:0		Reserved.
		15:8	Package	**Maximum Non-Turbo Ratio. (R/O)** The is the ratio of the frequency that invariant TSC runs at. Frequency = ratio * 100 MHz.
		27:16		Reserved.
		28	Package	**Programmable Ratio Limit for Turbo Mode. (R/O)** When set to 1, indicates that Programmable Ratio Limits for Turbo mode is enabled, and when set to 0, indicates Programmable Ratio Limits for Turbo mode is disabled.
		29	Package	**Programmable TDP Limit for Turbo Mode. (R/O)** When set to 1, indicates that TDP Limits for Turbo mode are programmable, and when set to 0, indicates TDP Limit for Turbo mode is not programmable.
		39:30		Reserved.
		47:40	Package	**Maximum Efficiency Ratio. (R/O)** The is the minimum ratio (maximum efficiency) that the processor can operates, in units of 100MHz.
		63:48		Reserved.

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
E2H	226	MSR_PKG_CST_CONFIG_CONTROL	Core	**C-State Configuration Control** (R/W) Note: C-state values are processor specific C-state code names, unrelated to MWAIT extension C-state parameters or ACPI C-States.
		2:0		**Package C-State limit. (R/W)** Specifies the lowest processor-specific C-state code name (consuming the least power). for the package. The default is set as factory-configured package C-state limit. The following C-state code name encodings are supported: 000b: C0/C1 (no package C-sate support) 001b: C2 010b: C6 no retention 011b: C6 retention 100b: C7 101b: C7s 111: No package C-state limit. Note: This field cannot be used to limit package C-state to C3.
		9:3		**Reserved.**
		10		**I/O MWAIT Redirection Enable. (R/W)** When set, will map IO_read instructions sent to IO register specified by MSR_PMG_IO_CAPTURE_BASE to MWAIT instructions
		14:11		**Reserved.**
		15		**CFG Lock. (R/WO)** When set, lock bits 15:0 of this register until next reset
		24:16		**Reserved.**
		25		**C3 state auto demotion enable. (R/W)** When set, the processor will conditionally demote C6/C7 requests to C3 based on uncore auto-demote information
		26		**C1 state auto demotion enable. (R/W)** When set, the processor will conditionally demote C3/C6/C7 requests to C1 based on uncore auto-demote information

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
		27		**Enable C3 undemotion (R/W)** When set, enables undemotion from demoted C3
		28		**Enable C1 undemotion (R/W)** When set, enables undemotion from demoted C1
		63:29		Reserved.
E4H	228	MSR_PMG_IO_CAPTURE_BASE	Core	**Power Management IO Redirection in C-state** (R/W)
		15:0		**LVL_2 Base Address. (R/W)** Specifies the base address visible to software for IO redirection. If IO MWAIT Redirection is enabled, reads to this address will be consumed by the power management logic and decoded to MWAIT instructions. When IO port address redirection is enabled, this is the IO port address reported to the OS/software
		18:16		**C-state Range. (R/W)** Specifies the encoding value of the maximum C-State code name to be included when IO read to MWAIT redirection is enabled by MSR_PMG_CST_CONFIG_CONTROL[bit10]: 000b - C3 is the max C-State to include 001b - C6 is the max C-State to include 010b - C7 is the max C-State to include
		63:19		Reserved.
E7H	231	IA32_MPERF	Thread	**Maximum Performance Frequency Clock Count. (RW)** see Table B-2
E8H	232	IA32_APERF	Thread	**Actual Performance Frequency Clock Count. (RW)** see Table B-2
FEH	254	IA32_MTRRCAP	Thread	see Table B-2
174H	372	IA32_SYSENTER_CS	Thread	see Table B-2
175H	373	IA32_SYSENTER_ESP	Thread	see Table B-2
176H	374	IA32_SYSENTER_EIP	Thread	see Table B-2
179H	377	IA32_MCG_CAP	Thread	see Table B-2
17AH	378	IA32_MCG_ STATUS	Thread	

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
		0		**RIPV.** When set, bit indicates that the instruction addressed by the instruction pointer pushed on the stack (when the machine check was generated) can be used to restart the program. If cleared, the program cannot be reliably restarted
		1		**EIPV.** When set, bit indicates that the instruction addressed by the instruction pointer pushed on the stack (when the machine check was generated) is directly associated with the error.
		2		**MCIP.** When set, bit indicates that a machine check has been generated. If a second machine check is detected while this bit is still set, the processor enters a shutdown state. Software should write this bit to 0 after processing a machine check exception.
		63:3		Reserved.
186H	390	IA32_ PERFEVTSEL0	Thread	see Table B-2
187H	391	IA32_ PERFEVTSEL1	Thread	see Table B-2
188H	392	IA32_ PERFEVTSEL2	Thread	see Table B-2
189H	393	IA32_ PERFEVTSEL3	Thread	see Table B-2
18AH	394	IA32_ PERFEVTSEL4	Core	see Table B-2; If CPUID.0AH:EAX[15:8] = 8
18BH	395	IA32_ PERFEVTSEL5	Core	see Table B-2; If CPUID.0AH:EAX[15:8] = 8
18CH	396	IA32_ PERFEVTSEL6	Core	see Table B-2; If CPUID.0AH:EAX[15:8] = 8
18DH	397	IA32_ PERFEVTSEL7	Core	see Table B-2; If CPUID.0AH:EAX[15:8] = 8
198H	408	IA32_PERF_STAT US	Package	see Table B-2
		15:0		Current Performance State Value.
		63:16		Reserved.

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
198H	408	MSR_PERF_STATUS	Package	
		47:32		Core Voltage (R/O) P-state core voltage can be computed by MSR_PERF_STATUS[37:32] * (float) 1/(2^13).
199H	409	IA32_PERF_CTL	Thread	see Table B-2
19AH	410	IA32_CLOCK_MODULATION	Thread	**Clock Modulation. (R/W)** see Table B-2 IA32_CLOCK_MODULATION MSR was originally named IA32_THERM_CONTROL MSR.
		3:0		**On demand Clock Modulation Duty Cycle (R/W).** In 6.25% increment
		4		**On demand Clock Modulation Enable (R/W).**
		63:5		Reserved.
19BH	411	IA32_THERM_INTERRUPT	Core	**Thermal Interrupt Control. (R/W)** see Table B-2
19CH	412	IA32_THERM_STATUS	Core	**Thermal Monitor Status. (R/W)** see Table B-2
1A0	416	IA32_MISC_ENABLE		**Enable Misc. Processor Features. (R/W)** Allows a variety of processor functions to be enabled and disabled.
		0	Thread	**Fast-Strings Enable.** see Table B-2
		6:1		Reserved.
		7	Thread	**Performance Monitoring Available. (R)** see Table B-2
		10:8		Reserved.
		11	Thread	**Branch Trace Storage Unavailable. (RO)** see Table B-2
		12	Thread	**Precise Event Based Sampling Unavailable. (RO)** see Table B-2
		15:13		Reserved.
		16	Package	**Enhanced Intel SpeedStep Technology Enable. (R/W)** see Table B-2
		18	Thread	ENABLE MONITOR FSM. (R/W) see Table B-2
		21:19		Reserved.
		22	Thread	**Limit CPUID Maxval. (R/W)** see Table B-2
		23	Thread	**xTPR Message Disable. (R/W)** see Table B-2

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
		33:24		Reserved.
		34	Thread	**XD Bit Disable. (R/W)** see Table B-2
		37:35		Reserved.
		38	Package	**Turbo Mode Disable. (R/W)** When set to 1 on processors that support Intel Turbo Boost Technology, the turbo mode feature is disabled and the IDA_Enable feature flag will be clear (CPUID.06H: EAX[1]=0). When set to a 0 on processors that support IDA, CPUID.06H: EAX[1] reports the processor's support of turbo mode is enabled. **Note:** the power-on default value is used by BIOS to detect hardware support of turbo mode. If power-on default value is 1, turbo mode is available in the processor. If power-on default value is 0, turbo mode is not available.
		63:39		Reserved.
1A2H	418	MSR_ TEMPERATURE_TARGET	Unique	
		15:0		Reserved.
		23:16		**Temperature Target. (R)** The minimum temperature at which PROCHOT# will be asserted. The value is degree C.
		63:24		Reserved
1A6H	422	MSR_OFFCORE_RSP_0	Thread	**Offcore Response Event Select Register** (R/W)
1B0H	432	IA32_ENERGY_PERF_BIAS	Package	see Table B-2
1B1H	433	IA32_PACKAGE_THERM_STATUS	Package	see Table B-2
1B2H	434	IA32_PACKAGE_THERM_INTERRUPT	Package	see Table B-2
1C8H	456	MSR_LBR_SELECT	Thread	**Last Branch Record Filtering Select Register** (R/W) see Section 16.6.2, "Filtering of Last Branch Records."
1C9H	457	MSR_ LASTBRANCH_ TOS	Thread	**Last Branch Record Stack TOS. (R)** Contains an index (bits 0-3) that points to the MSR containing the most recent branch record. See MSR_LASTBRANCH_0_FROM_IP (at 680H).

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
1D9H	473	IA32_DEBUGCTL	Thread	**Debug Control. (R/W)** see Table B-2
1DDH	477	MSR_LER_FROM_LIP	Thread	**Last Exception Record From Linear IP. (R)** Contains a pointer to the last branch instruction that the processor executed prior to the last exception that was generated or the last interrupt that was handled.
1DEH	478	MSR_LER_TO_LIP	Thread	**Last Exception Record To Linear IP. (R)** This area contains a pointer to the target of the last branch instruction that the processor executed prior to the last exception that was generated or the last interrupt that was handled.
1F2H	498	IA32_SMRR_PHYSBASE	Core	see Table B-2
1F3H	499	IA32_SMRR_PHYSMASK	Core	see Table B-2
1FCH	508	MSR_POWER_CTL	Core	Power Control Register
200H	512	IA32_MTRR_PHYSBASE0	Thread	see Table B-2
201H	513	IA32_MTRR_PHYSMASK0	Thread	see Table B-2
202H	514	IA32_MTRR_PHYSBASE1	Thread	see Table B-2
203H	515	IA32_MTRR_PHYSMASK1	Thread	see Table B-2
204H	516	IA32_MTRR_PHYSBASE2	Thread	see Table B-2
205H	517	IA32_MTRR_PHYSMASK2	Thread	see Table B-2
206H	518	IA32_MTRR_PHYSBASE3	Thread	see Table B-2
207H	519	IA32_MTRR_PHYSMASK3	Thread	see Table B-2
208H	520	IA32_MTRR_PHYSBASE4	Thread	see Table B-2
209H	521	IA32_MTRR_PHYSMASK4	Thread	see Table B-2
20AH	522	IA32_MTRR_PHYSBASE5	Thread	see Table B-2
20BH	523	IA32_MTRR_PHYSMASK5	Thread	see Table B-2

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
20CH	524	IA32_MTRR_PHYSBASE6	Thread	see Table B-2
20DH	525	IA32_MTRR_PHYSMASK6	Thread	see Table B-2
20EH	526	IA32_MTRR_PHYSBASE7	Thread	see Table B-2
20FH	527	IA32_MTRR_PHYSMASK7	Thread	see Table B-2
210H	528	IA32_MTRR_PHYSBASE8	Thread	see Table B-2
211H	529	IA32_MTRR_PHYSMASK8	Thread	see Table B-2
212H	530	IA32_MTRR_PHYSBASE9	Thread	see Table B-2
213H	531	IA32_MTRR_PHYSMASK9	Thread	see Table B-2
250H	592	IA32_MTRR_FIX64K_00000	Thread	see Table B-2
258H	600	IA32_MTRR_FIX16K_80000	Thread	see Table B-2
259H	601	IA32_MTRR_FIX16K_A0000	Thread	see Table B-2
268H	616	IA32_MTRR_FIX4K_C0000	Thread	see Table B-2
269H	617	IA32_MTRR_FIX4K_C8000	Thread	see Table B-2
26AH	618	IA32_MTRR_FIX4K_D0000	Thread	see Table B-2
26BH	619	IA32_MTRR_FIX4K_D8000	Thread	see Table B-2
26CH	620	IA32_MTRR_FIX4K_E0000	Thread	see Table B-2
26DH	621	IA32_MTRR_FIX4K_E8000	Thread	see Table B-2
26EH	622	IA32_MTRR_FIX4K_F0000	Thread	see Table B-2
26FH	623	IA32_MTRR_FIX4K_F8000	Thread	see Table B-2
277H	631	IA32_PAT	Thread	see Table B-2
280H	640	IA32_MC0_CTL2	Core	see Table B-2
281H	641	IA32_MC1_CTL2	Core	see Table B-2

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
282H	642	IA32_MC2_CTL2	Core	see Table B-2
283H	643	IA32_MC3_CTL2	Core	see Table B-2
284H	644	MSR_MC4_CTL2	Package	Always 0 (CMCI not supported)
2FFH	767	IA32_MTRR_DEF_ TYPE	Thread	**Default Memory Types. (R/W)** see Table B-2
309H	777	IA32_FIXED_CTR0	Thread	**Fixed-Function Performance Counter Register 0. (R/W)** see Table B-2
30AH	778	IA32_FIXED_CTR1	Thread	**Fixed-Function Performance Counter Register 1. (R/W)** see Table B-2
30BH	779	IA32_FIXED_CTR2	Thread	**Fixed-Function Performance Counter Register 2. (R/W)** see Table B-2
345H	837	IA32_PERF_CAPA BILITIES	Thread	see Table B-2. See Section 16.4.1, "IA32_DEBUGCTL MSR."
		5:0		LBR Format. see Table B-2.
		6		PEBS Record Format.
		7		PEBSSaveArchRegs. see Table B-2.
		11:8		PEBS_REC_FORMAT. see Table B-2.
		12		SMM_FREEZE. see Table B-2.
		63:13		Reserved.
38DH	909	IA32_FIXED_CTR_ CTRL	Thread	**Fixed-Function-Counter Control Register. (R/W)** see Table B-2
38EH	910	IA32_PERF_ GLOBAL_STAUS	Thread	see Table B-2. See Section 30.4.2, "Global Counter Control Facilities."
38FH	911	IA32_PERF_ GLOBAL_CTRL	Thread	see Table B-2. See Section 30.4.2, "Global Counter Control Facilities."
390H	912	IA32_PERF_ GLOBAL_OVF_ CTRL	Thread	see Table B-2. See Section 30.4.2, "Global Counter Control Facilities."
3F1H	1009	MSR_PEBS_ ENABLE	Thread	see See Section 30.6.1.1, "Precise Event Based Sampling (PEBS)."
		0		Enable PEBS on IA32_PMC0. (R/W)
		1		Enable PEBS on IA32_PMC1. (R/W)
		2		Enable PEBS on IA32_PMC2. (R/W)
		3		Enable PEBS on IA32_PMC3. (R/W)
		31:4		Reserved
		32		Enable Load Latency on IA32_PMC0. (R/W)
		33		Enable Load Latency on IA32_PMC1. (R/W)
		34		Enable Load Latency on IA32_PMC2. (R/W)

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
		35		Enable Load Latency on IA32_PMC3. (R/W)
		63:36		Reserved
3F6H	1014	MSR_PEBS_ LD_LAT	Thread	see See Section 30.6.1.2, "Load Latency Performance Monitoring Facility."
		15:0		Minimum threshold latency value of tagged load operation that will be counted. (R/W)
		63:36		Reserved
3F8H	1016	MSR_PKG_C3_RES IDENCY	Package	Note: C-state values are processor specific C-state code names, unrelated to MWAIT extension C-state parameters or ACPI C-States.
		63:0		Package C3 Residency Counter. (R/O) Value since last reset that this package is in processor-specific C3 states. Count at the same frequency as the TSC.
3F9H	1017	MSR_PKG_C6_RES IDENCY	Package	Note: C-state values are processor specific C-state code names, unrelated to MWAIT extension C-state parameters or ACPI C-States.
		63:0		Package C6 Residency Counter. (R/O) Value since last reset that this package is in processor-specific C6 states. Count at the same frequency as the TSC.
3FAH	1018	MSR_PKG_C7_RES IDENCY	Package	Note: C-state values are processor specific C-state code names, unrelated to MWAIT extension C-state parameters or ACPI C-States.
		63:0		Package C7 Residency Counter. (R/O) Value since last reset that this package is in processor-specific C7 states. Count at the same frequency as the TSC.
3FCH	1020	MSR_CORE_C3_RE SIDENCY	Core	Note: C-state values are processor specific C-state code names, unrelated to MWAIT extension C-state parameters or ACPI C-States.
		63:0		CORE C3 Residency Counter. (R/O) Value since last reset that this core is in processor-specific C3 states. Count at the same frequency as the TSC.
3FDH	1021	MSR_CORE_C6_RE SIDENCY	Core	Note: C-state values are processor specific C-state code names, unrelated to MWAIT extension C-state parameters or ACPI C-States.

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
		63:0		CORE C6 Residency Counter. (R/O) Value since last reset that this core is in processor-specific C6 states. Count at the same frequency as the TSC.
3FEH	1022	MSR_CORE_C7_RESIDENCY	Core	Note: C-state values are processor specific C-state code names, unrelated to MWAIT extension C-state parameters or ACPI C-States.
		63:0		CORE C7 Residency Counter. (R/O) Value since last reset that this core is in processor-specific C7 states. Count at the same frequency as the TSC.
400H	1024	IA32_MC0_CTL	Core	See Section 15.3.2.1, "IA32_MCi_CTL MSRs."
401H	1025	IA32_MC0_ STATUS	Core	See Section 15.3.2.2, "IA32_MCi_STATUS MSRS." and Appendix E.
402H	1026	IA32_MC0_ADDR	Core	See Section 15.3.2.3, "IA32_MCi_ADDR MSRs."
403H	1027	IA32_MC0_MISC	Core	See Section 15.3.2.4, "IA32_MCi_MISC MSRs."
404H	1028	IA32_MC1_CTL	Core	See Section 15.3.2.1, "IA32_MCi_CTL MSRs."
405H	1029	IA32_MC1_ STATUS	Core	See Section 15.3.2.2, "IA32_MCi_STATUS MSRS." and Appendix E.
406H	1030	IA32_MC1_ADDR	Core	See Section 15.3.2.3, "IA32_MCi_ADDR MSRs."
407H	1031	IA32_MC1_MISC	Core	See Section 15.3.2.4, "IA32_MCi_MISC MSRs."
408H	1032	IA32_MC2_CTL	Core	See Section 15.3.2.1, "IA32_MCi_CTL MSRs."
409H	1033	IA32_MC2_ STATUS	Core	See Section 15.3.2.2, "IA32_MCi_STATUS MSRS." and Appendix E.
40AH	1034	IA32_MC2_ADDR	Core	See Section 15.3.2.3, "IA32_MCi_ADDR MSRs."
40BH	1035	IA32_MC2_MISC	Core	See Section 15.3.2.4, "IA32_MCi_MISC MSRs."
40CH	1036	IA32_MC3_CTL	Core	See Section 15.3.2.1, "IA32_MCi_CTL MSRs."
40DH	1037	IA32_MC3_ STATUS	Core	See Section 15.3.2.2, "IA32_MCi_STATUS MSRS." and Appendix E.
40EH	1038	IA32_MC3_ADDR	Core	See Section 15.3.2.3, "IA32_MCi_ADDR MSRs."
40FH	1039	IA32_MC3_MISC	Core	See Section 15.3.2.4, "IA32_MCi_MISC MSRs."
410H	1040	MSR_MC4_CTL	Core	See Section 15.3.2.1, "IA32_MCi_CTL MSRs."
		0		**PCU Hardware Error. (R/W)** When set, enables signaling of PCU hardware detected errors.
		1		**PCU Controller Error. (R/W)** When set, enables signaling of PCU controller detected errors

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
		2		**PCU Firmware Error. (R/W)** When set, enables signaling of PCU firmware detected errors
		63:2		Reserved.
411H	1041	IA32_MC4_ STATUS	Core	See Section 15.3.2.2, "IA32_MCi_STATUS MSRS." and Appendix E.
480H	1152	IA32_VMX_BASIC	Thread	**Reporting Register of Basic VMX Capabilities. (R/O)** see Table B-2. See Appendix G.1, "Basic VMX Information"
481H	1153	IA32_VMX_PINBA SED_CTLS	Thread	**Capability Reporting Register of Pin-based VM-execution Controls. (R/O)** see Table B-2. See Appendix G.3, "VM-Execution Controls"
482H	1154	IA32_VMX_PROCB ASED_CTLS	Thread	**Capability Reporting Register of Primary Processor-based VM-execution Controls. (R/O)** See Appendix G.3, "VM-Execution Controls"
483H	1155	IA32_VMX_EXIT_ CTLS	Thread	**Capability Reporting Register of VM-exit Controls. (R/O)** see Table B-2. See Appendix G.4, "VM-Exit Controls"
484H	1156	IA32_VMX_ ENTRY_CTLS	Thread	**Capability Reporting Register of VM-entry Controls. (R/O)** see Table B-2. See Appendix G.5, "VM-Entry Controls"
485H	1157	IA32_VMX_MISC	Thread	**Reporting Register of Miscellaneous VMX Capabilities. (R/O)** see Table B-2. See Appendix G.6, "Miscellaneous Data"
486H	1158	IA32_VMX_CR0_ FIXED0	Thread	**Capability Reporting Register of CR0 Bits Fixed to 0. (R/O)** see Table B-2. See Appendix G.7, "VMX-Fixed Bits in CR0"
487H	1159	IA32_VMX_CR0_ FIXED1	Thread	**Capability Reporting Register of CR0 Bits Fixed to 1. (R/O)** see Table B-2. See Appendix G.7, "VMX-Fixed Bits in CR0"
488H	1160	IA32_VMX_CR4_FI XED0	Thread	**Capability Reporting Register of CR4 Bits Fixed to 0. (R/O)** see Table B-2. See Appendix G.8, "VMX-Fixed Bits in CR4"
489H	1161	IA32_VMX_CR4_FI XED1	Thread	**Capability Reporting Register of CR4 Bits Fixed to 1. (R/O)** see Table B-2. See Appendix G.8, "VMX-Fixed Bits in CR4"
48AH	1162	IA32_VMX_ VMCS_ENUM	Thread	**Capability Reporting Register of VMCS Field Enumeration. (R/O).** see Table B-2. See Appendix G.9, "VMCS Enumeration"

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
48BH	1163	IA32_VMX_PROCBASED_CTLS2	Thread	**Capability Reporting Register of Secondary Processor-based VM-execution Controls. (R/O)** See Appendix G.3, "VM-Execution Controls"
4C1H	1217	IA32_A_PMC0	Thread	see Table B-2
4C2H	1218	IA32_A_PMC1	Thread	see Table B-2
4C3H	1219	IA32_A_PMC2	Thread	see Table B-2
4C4H	1220	IA32_A_PMC3	Thread	see Table B-2
4C5H	1221	IA32_A_PMC4	Core	see Table B-2
4C6H	1222	IA32_A_PMC5	Core	see Table B-2
4C7H	1223	IA32_A_PMC6	Core	see Table B-2
C8H	200	IA32_A_PMC7	Core	see Table B-2
600H	1536	IA32_DS_AREA	Thread	**DS Save Area. (R/W).** see Table B-2 See Section 30.9.4, "Debug Store (DS) Mechanism."
606H	1542	MSR_RAPL_POWER_UNIT	Package	**Unit Multipliers used in RAPL Interfaces** (R/O) See Section 14.7.1, "RAPL Interfaces."
60AH	1546	MSR_PKGC3_IRTL	Package	**Package C3 Interrupt Response Limit** (R/W) Note: C-state values are processor specific C-state code names, unrelated to MWAIT extension C-state parameters or ACPI C-States.
		9:0		**Interrupt response time limit. (R/W)** Specifies the limit that should be used to decide if the package should be put into a package C3 state.
		12:10		**Time Unit. (R/W)** Specifies the encoding value of time unit of the interrupt response time limit. The following time unit encodings are supported: 000b: 1 ns 001b: 32 ns 010b: 1024 ns 011b: 32768 ns 100b: 1048576 ns 101b: 33554432 ns
		14:13		**Reserved.**

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
		15		**Valid. (R/W)** Indicates whether the values in bits 12:0 are valid and can be used by the processor for package C-sate management.
		63:16		Reserved.
60BH	1547	MSR_PKGC6_IRTL	Package	**Package C6 Interrupt Response Limit** (R/W) This MSR defines the budget allocated for the package to exit from C6 to a C0 state, where interrupt request can be delivered to the core and serviced. Additional core-exit latency amy be applicable depending on the actual C-state the core is in. Note: C-state values are processor specific C-state code names, unrelated to MWAIT extension C-state parameters or ACPI C-States.
		9:0		**Interrupt response time limit. (R/W)** Specifies the limit that should be used to decide if the package should be put into a package C6 state.
		12:10		**Time Unit. (R/W)** Specifies the encoding value of time unit of the interrupt response time limit. The following time unit encodings are supported: 000b: 1 ns 001b: 32 ns 010b: 1024 ns 011b: 32768 ns 100b: 1048576 ns 101b: 33554432 ns
		14:13		**Reserved.**
		15		**Valid. (R/W)** Indicates whether the values in bits 12:0 are valid and can be used by the processor for package C-sate management.
		63:16		Reserved.

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
60CH	1548	MSR_PKGC7_IRTL	Package	**Package C7 Interrupt Response Limit** (R/W) This MSR defines the budget allocated for the package to exit from C7 to a C0 state, where interrupt request can be delivered to the core and serviced. Additional core-exit latency amy be applicable depending on the actual C-state the core is in. Note: C-state values are processor specific C-state code names, unrelated to MWAIT extension C-state parameters or ACPI C-States.
		9:0		**Interrupt response time limit. (R/W)** Specifies the limit that should be used to decide if the package should be put into a package C7 state.
		12:10		**Time Unit. (R/W)** Specifies the encoding value of time unit of the interrupt response time limit. The following time unit encodings are supported: 000b: 1 ns 001b: 32 ns 010b: 1024 ns 011b: 32768 ns 100b: 1048576 ns 101b: 33554432 ns
		14:13		**Reserved.**
		15		**Valid. (R/W)** Indicates whether the values in bits 12:0 are valid and can be used by the processor for package C-sate management.
		63:16		Reserved.
60DH	1549	MSR_PKG_C2_RESIDENCY	Package	Note: C-state values are processor specific C-state code names, unrelated to MWAIT extension C-state parameters or ACPI C-States.
		63:0		Package C2 Residency Counter. (R/O) Value since last reset that this package is in processor-specific C2 states. Count at the same frequency as the TSC.
610H	1552	MSR_PKG_RAPL_POWER_LIMIT	Package	**PKG RAPL Power Limit Control** (R/W) See Section 14.7.3, "Package RAPL Domain."
611H	1553	MSR_PKG_ENERY_STATUS	Package	**PKG Energy Status** (R/O) See Section 14.7.3, "Package RAPL Domain."

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
613H	1555	MSR_PKG_PERF_STATUS	Package	**PKG Performance Throttling Status** (R/O) See Section 14.7.3, "Package RAPL Domain."
614H	1556	MSR_PKG_POWER_INFO	Package	**PKG RAPL Parameters** (R/W) See Section 14.7.3, "Package RAPL Domain."
638H	1592	MSR_PP0_POWER_LIMIT	Package	**PP0 RAPL Power Limit Control** (R/W) See Section 14.7.4, "PP0/PP1 RAPL Domains."
639H	1593	MSR_PP0_ENERY_STATUS	Package	**PP0 Energy Status** (R/O) See Section 14.7.4, "PP0/PP1 RAPL Domains."
63AH	1594	MSR_PP0_POLICY	Package	**PP0 Balance Policy** (R/W) See Section 14.7.4, "PP0/PP1 RAPL Domains."
63BH	1595	MSR_PP0_PERF_STATUS	Package	**PP0 Performance Throttling Status** (R/O) See Section 14.7.4, "PP0/PP1 RAPL Domains."
680H	1664	MSR_LASTBRANCH_0_FROM_IP	Thread	**Last Branch Record 0 From IP. (R/W)** One of sixteen pairs of last branch record registers on the last branch record stack. This part of the stack contains pointers to the **source instruction** for one of the last sixteen branches, exceptions, or interrupts taken by the processor. See also: ▪ Last Branch Record Stack TOS at 1C9H ▪ Section 16.6.1, "LBR Stack."
681H	1665	MSR_LASTBRANCH_1_FROM_IP	Thread	**Last Branch Record 1 From IP. (R/W)** See description of MSR_LASTBRANCH_0_FROM_IP.
682H	1666	MSR_LASTBRANCH_2_FROM_IP	Thread	**Last Branch Record 2 From IP. (R/W)** See description of MSR_LASTBRANCH_0_FROM_IP.
683H	1667	MSR_LASTBRANCH_3_FROM_IP	Thread	**Last Branch Record 3 From IP. (R/W)** See description of MSR_LASTBRANCH_0_FROM_IP.
684H	1668	MSR_LASTBRANCH_4_FROM_IP	Thread	**Last Branch Record 4 From IP. (R/W)** See description of MSR_LASTBRANCH_0_FROM_IP.
685H	1669	MSR_LASTBRANCH_5_FROM_IP	Thread	**Last Branch Record 5 From IP. (R/W)** See description of MSR_LASTBRANCH_0_FROM_IP.
686H	1670	MSR_LASTBRANCH_6_FROM_IP	Thread	**Last Branch Record 6 From IP. (R/W)** See description of MSR_LASTBRANCH_0_FROM_IP.
687H	1671	MSR_LASTBRANCH_7_FROM_IP	Thread	**Last Branch Record 7 From IP. (R/W)** See description of MSR_LASTBRANCH_0_FROM_IP.

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
688H	1672	MSR_ LASTBRANCH_8_FROM_IP	Thread	**Last Branch Record 8 From IP. (R/W)** See description of MSR_LASTBRANCH_0_FROM_IP.
689H	1673	MSR_ LASTBRANCH_9_FROM_IP	Thread	**Last Branch Record 9 From IP. (R/W)** See description of MSR_LASTBRANCH_0_FROM_IP.
68AH	1674	MSR_ LASTBRANCH_10_FROM_IP	Thread	**Last Branch Record 10 From IP. (R/W)** See description of MSR_LASTBRANCH_0_FROM_IP.
68BH	1675	MSR_ LASTBRANCH_11_FROM_IP	Thread	**Last Branch Record 11 From IP. (R/W)** See description of MSR_LASTBRANCH_0_FROM_IP.
68CH	1676	MSR_ LASTBRANCH_12_FROM_IP	Thread	**Last Branch Record 12 From IP. (R/W)** See description of MSR_LASTBRANCH_0_FROM_IP.
68DH	1677	MSR_ LASTBRANCH_13_FROM_IP	Thread	**Last Branch Record 13 From IP. (R/W)** See description of MSR_LASTBRANCH_0_FROM_IP.
68EH	1678	MSR_ LASTBRANCH_14_FROM_IP	Thread	**Last Branch Record 14 From IP. (R/W)** See description of MSR_LASTBRANCH_0_FROM_IP.
68FH	1679	MSR_ LASTBRANCH_15_FROM_IP	Thread	**Last Branch Record 15 From IP. (R/W)** See description of MSR_LASTBRANCH_0_FROM_IP.
6C0H	1728	MSR_ LASTBRANCH_0_TO_LIP	Thread	**Last Branch Record 0 To IP. (R/W)** One of sixteen pairs of last branch record registers on the last branch record stack. This part of the stack contains pointers to the destination instruction for one of the last sixteen branches, exceptions, or interrupts taken by the processor.
6C1H	1729	MSR_ LASTBRANCH_1_TO_LIP	Thread	**Last Branch Record 1 To IP. (R/W)** See description of MSR_LASTBRANCH_0_TO_LIP.
6C2H	1730	MSR_ LASTBRANCH_2_TO_LIP	Thread	**Last Branch Record 2 To IP. (R/W)** See description of MSR_LASTBRANCH_0_TO_LIP.
6C3H	1731	MSR_ LASTBRANCH_3_TO_LIP	Thread	**Last Branch Record 3 To IP. (R/W)** See description of MSR_LASTBRANCH_0_TO_LIP.

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
6C4H	1732	MSR_ LASTBRANCH_4_ TO_LIP	Thread	**Last Branch Record 4 To IP. (R/W)** See description of MSR_LASTBRANCH_0_TO_LIP.
6C5H	1733	MSR_ LASTBRANCH_5_ TO_LIP	Thread	**Last Branch Record 5 To IP. (R/W)** See description of MSR_LASTBRANCH_0_TO_LIP.
6C6H	1734	MSR_ LASTBRANCH_6_ TO_LIP	Thread	**Last Branch Record 6 To IP. (R/W)** See description of MSR_LASTBRANCH_0_TO_LIP.
6C7H	1735	MSR_ LASTBRANCH_7_ TO_LIP	Thread	**Last Branch Record 7 To IP. (R/W)** See description of MSR_LASTBRANCH_0_TO_LIP.
6C8H	1736	MSR_ LASTBRANCH_8_ TO_LIP	Thread	**Last Branch Record 8 To IP. (R/W)** See description of MSR_LASTBRANCH_0_TO_LIP.
6C9H	1737	MSR_ LASTBRANCH_9_ TO_LIP	Thread	**Last Branch Record 9 To IP. (R/W)** See description of MSR_LASTBRANCH_0_TO_LIP.
6CAH	1738	MSR_ LASTBRANCH_10_ TO_LIP	Thread	**Last Branch Record 10 To IP. (R/W)** See description of MSR_LASTBRANCH_0_TO_LIP.
6CBH	1739	MSR_ LASTBRANCH_11_ TO_LIP	Thread	**Last Branch Record 11 To IP. (R/W)** See description of MSR_LASTBRANCH_0_TO_LIP.
6CCH	1740	MSR_ LASTBRANCH_12_ TO_LIP	Thread	**Last Branch Record 12 To IP. (R/W)** See description of MSR_LASTBRANCH_0_TO_LIP.
6CDH	1741	MSR_ LASTBRANCH_13_ TO_LIP	Thread	**Last Branch Record 13 To IP. (R/W)** See description of MSR_LASTBRANCH_0_TO_LIP.
6CEH	1742	MSR_ LASTBRANCH_14_ TO_LIP	Thread	**Last Branch Record 14 To IP. (R/W)** See description of MSR_LASTBRANCH_0_TO_LIP.
6CFH	1743	MSR_ LASTBRANCH_15_ TO_LIP	Thread	**Last Branch Record 15 To IP. (R/W)** See description of MSR_LASTBRANCH_0_TO_LIP.
6E0H	1760	IA32_TSC_DEADLI NE	Thread	See Table B-2.
C000_ 0080H		IA32_EFER	Thread	**Extended Feature Enables.** see Table B-2

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
C000_0081H		IA32_STAR	Thread	**System Call Target Address. (R/W).** see Table B-2
C000_0082H		IA32_LSTAR	Thread	**IA-32e Mode System Call Target Address. (R/W).** see Table B-2
C000_0084H		IA32_FMASK	Thread	**System Call Flag Mask. (R/W).** see Table B-2
C000_0100H		IA32_FS_BASE	Thread	**Map of BASE Address of FS. (R/W).** see Table B-2
C000_0101H		IA32_GS_BASE	Thread	**Map of BASE Address of GS. (R/W).** see Table B-2
C000_0102H		IA32_KERNEL_GS BASE	Thread	**Swap Target of BASE Address of GS. (R/W).** see Table B-2
C000_0103H		IA32_TSC_AUX	Thread	**AUXILIARY TSC Signature. (R/W).** see Table B-2 and Section 16.12.2, "IA32_TSC_AUX Register and RDTSCP Support."

…

B.6.1 MSRs In Second Generation Intel® Core Processor Family (Intel® microarchitecture code name Sandy Bridge)

Table B-10 lists selected model-specific registers (MSRs) that are specific to next generation for Intel® Core processor family (Intel® microarchitecture code name Sandy Bridge). These processors have a CPUID signature with DisplayFamily_DisplayModel of 06_2AH, see Table B-1.

Table B-10 MSRs supported by Second Generation Intel Core Processors (Intel microarchitecture code name Sandy Bridge)

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
1ADH	429	MSR_TURBO_RATIO_LIMIT	Package	**Maximum Ratio Limit of Turbo Mode.** RO if MSR_PLATFORM_INFO.[28] = 0, RW if MSR_PLATFORM_INFO.[28] = 1
		7:0	Package	**Maximum Ratio Limit for 1C.** Maximum turbo ratio limit of 1 core active.
		15:8	Package	**Maximum Ratio Limit for 2C.** Maximum turbo ratio limit of 2 core active.
		23:16	Package	**Maximum Ratio Limit for 3C.** Maximum turbo ratio limit of 3 core active.

Table B-10 MSRs supported by Second Generation Intel Core Processors (Continued)(Intel microarchitecture code name Sandy Bridge)

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
		31:24	Package	**Maximum Ratio Limit for 4C.** Maximum turbo ratio limit of 4 core active.
		63:32		Reserved.
640H	1600	MSR_PP1_POWER_LIMIT	Package	**PP1 RAPL Power Limit Control** (R/W) See Section 14.7.4, "PP0/PP1 RAPL Domains."
641H	1601	MSR_PP1_ENERY_STATUS	Package	**PP1 Energy Status** (R/O) See Section 14.7.4, "PP0/PP1 RAPL Domains."
642H	1602	MSR_PP1_POLICY	Package	**PP1 Balance Policy** (R/W) See Section 14.7.4, "PP0/PP1 RAPL Domains."

...

29. Updates to Appendix G, Volume 3B

Change bars show changes to Appendix G of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3B:* System Programming Guide, Part 2.

...

G.1 BASIC VMX INFORMATION

The IA32_VMX_BASIC MSR (index 480H) consists of the following fields:

- Bits 31:0 contain the 32-bit VMCS revision identifier used by the processor. Logical processors that use the same VMCS revision identifier use the same size for VMCS regions (see next item)
- Bits 44:32 report the number of bytes that software should allocate for the VMXON region and any VMCS region. It is a value greater than 0 and at most 4096 (bit 44 is set if and only if bits 43:32 are clear).
- Bit 48 indicates the width of the physical addresses that may be used for the VMXON region, each VMCS, and data structures referenced by pointers in a VMCS (I/O bitmaps, virtual-APIC page, MSR areas for VMX transitions). If the bit is 0, these addresses are limited to the processor's physical-address width.[1] If the bit is 1, these addresses are limited to 32 bits. This bit is always 0 for processors that support Intel 64 architecture.
- If bit 49 is read as 1, the logical processor supports the dual-monitor treatment of system-management interrupts and system-management mode. See Section 26.15 for details of this treatment.

1. On processors that support Intel 64 architecture, the pointer must not set bits beyond the processor's physical address width.

- Bits 53:50 report the memory type that the logical processor uses to access the VMCS for VMREAD and VMWRITE and to access the VMCS, data structures referenced by pointers in the VMCS (I/O bitmaps, virtual-APIC page, MSR areas for VMX transitions), and the MSEG header during VM entries, VM exits, and in VMX non-root operation.[1]

...

G.6 MISCELLANEOUS DATA

The IA32_VMX_MISC MSR (index 485H) consists of the following fields:

- Bits 4:0 report a value X that specifies the relationship between the rate of the VMX-preemption timer and that of the timestamp counter (TSC). Specifically, the VMX-preemption timer (if it is active) counts down by 1 every time bit X in the TSC changes due to a TSC increment.
- If bit 5 is read as 1, VM exits store the value of IA32_EFER.LMA into the "IA-32e mode guest" VM-entry control; see Section 24.2 for more details. This bit is read as 1 on any logical processor that supports the 1-setting of the "unrestricted guest" VM-execution control.
- Bits 8:6 report, as a bitmap, the activity states supported by the implementation:
  - Bit 6 reports (if set) the support for activity state 1 (HLT).
  - Bit 7 reports (if set) the support for activity state 2 (shutdown).
  - Bit 8 reports (if set) the support for activity state 3 (wait-for-SIPI).

  If an activity state is not supported, the implementation causes a VM entry to fail if it attempts to establish that activity state. All implementations support VM entry to activity state 0 (active).
- Bits 24:16 indicate the number of CR3-target values supported by the processor. This number is a value between 0 and 256, inclusive (bit 24 is set if and only if bits 23:16 are clear).
- Bits 27:25 is used to compute the recommended maximum number of MSRs that should appear in the VM-exit MSR-store list, the VM-exit MSR-load list, or the VM-entry MSR-load list. Specifically, if the value bits 27:25 of IA32_VMX_MISC is N, then 512 * (N + 1) is the recommended maximum number of MSRs to be included in each list. If the limit is exceeded, undefined processor behavior may result (including a machine check during the VMX transition).
- If bit 28 is read as 1, bit 2 of the IA32_SMM_MONITOR_CTL can be set to 1. VMXOFF unblocks SMIs unless IA32_SMM_MONITOR_CTL[bit 2] is 1 (see Section 26.14.4).
- Bits 63:32 report the 32-bit MSEG revision identifier used by the processor.
- Bits 15:9 and bits 31:29 are reserved and are read as 0.

...

1. If the MTRRs are disabled by clearing the E bit (bit 11) in the IA32_MTRR_DEF_TYPE MSR, the logical processor uses the UC memory type to access the indicated data structures, regardless of the value reported in bits 53:50 in the IA32_VMX_BASIC MSR. The processor will also use the UC memory type if the setting of CR0.CD on this logical processor (or another logical processor on the same physical processor) would cause it to do so for all memory accesses. The values of IA32_MTRR_DEF_TYPE.E and CR0.CD do not affect the value reported in IA32_VMX_BASIC[53:50].