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

Documentation Changes

April 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: 252046-031

Intel may make changes to specifications and product descriptions at any time, without notice.

64-bit computing on Intel architecture requires a computer system with a processor, chipset, BIOS, operating system, device drivers and applications enabled for Intel® 64 architecture. Performance will vary depending on your hardware and software configurations. Consult with your system vendor for more information.

Designers must not rely on the absence or characteristics of any features or instructions marked “reserved” or “undefined.” Intel reserves these for future definition and shall have no responsibility whatsoever for conflicts or incompatibilities arising from future changes to them.

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

I^{2}C is a two-wire communications bus/protocol developed by Philips. SMBus is a subset of the I^{2}C bus/protocol and was developed by Intel. Implementations of the I^{2}C bus/protocol may require licenses from various entities, including Philips Electronics N.V. and North American Philips Corporation.

Contents

Revision History 4

Preface 7

Summary Tables of Changes 8

Documentation Changes 9

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
-031	• Removed Documentation Changes 1-29 • Added Documentation Changes 1-29	April 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 1, Volume 1
2	Updates to Chapter 2, Volume 1
3	Updates to Chapter 5, Volume 1
4	Updates to Chapter 8, Volume 1
5	Updates to Appendix D, Volume 1
6	Updates to Chapter 1, Volume 2A
7	Updates to Chapter 2, Volume 2A
8	Updates to Chapter 3, Volume 2A
9	Updates to Chapter 4, Volume 2B
10	Updates to Chapter 5, Volume 2B
11	Updates to Appendix A, Volume 2B
12	Updates to Appendix B, Volume 2B
13	Updates to Appendix C, Volume 2B
14	Updates to Chapter 1, Volume 3A
15	Updates to Chapter 2, Volume 3A
16	Updates to Chapter 4, Volume 3A
17	Updates to Chapter 7, Volume 3A
18	Updates to Chapter 10, Volume 3A
19	Updates to Chapter 13, Volume 3A
20	Updates to Chapter 15, Volume 3A
21	Updates to Chapter 16, Volume 3A
22	Updates to Chapter 19, Volume 3A
23	Updates to Chapter 22, Volume 3B
24	Updates to Chapter 23, Volume 3B
25	Updates to Chapter 25, 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 E, Volume 3B

Documentation Changes

1. Updates to Chapter 1, Volume 1

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

...

1.1 INTEL® 64 AND IA-32 PROCESSORS COVERED IN THIS MANUAL

This manual set includes information pertaining primarily to the most recent Intel 64 and IA-32 processors, which include:

- Pentium® processors
- P6 family processors
- Pentium® 4 processors
- Pentium® M processors
- Intel® Xeon® processors
- Pentium® D processors
- Pentium® processor Extreme Editions
- 64-bit Intel® Xeon® processors
- Intel® Core™ Duo processor
- Intel® Core™ Solo processor
- Dual-Core Intel® Xeon® processor LV
- Intel® Core™2 Duo processor
- Intel® Core™2 Quad processor Q6000 series
- Intel® Xeon® processor 3000, 3200 series
- Intel® Xeon® processor 5000 series
- Intel® Xeon® processor 5100, 5300 series
- Intel® Core™2 Extreme processor X7000 and X6800 series
- Intel® Core™2 Extreme processor QX6000 series
- Intel® Xeon® processor 7100 series
- Intel® Pentium® Dual-Core processor
- Intel® Xeon® processor 7200, 7300 series
- Intel® Xeon® processor 5200, 5400, 7400 series
- Intel® Core™2 Extreme processor QX9000 and X9000 series
- Intel® Core™2 Quad processor Q9000 series
- Intel® Core™2 Duo processor E8000, T9000 series
- Intel® Atom™ processor family

- Intel® Core™ i7 processor
- Intel® Core™ i5 processor
- Intel® Xeon® processor E7-8800/4800/2800 product families

P6 family processors are IA-32 processors based on the P6 family microarchitecture. This includes the Pentium® Pro, Pentium® II, Pentium® III, and Pentium® III Xeon® processors.

The Pentium® 4, Pentium® D, and Pentium® processor Extreme Editions are based on the Intel NetBurst® microarchitecture. Most early Intel® Xeon® processors are based on the Intel NetBurst® microarchitecture. Intel Xeon processor 5000, 7100 series are based on the Intel NetBurst® microarchitecture.

The Intel® Core™ Duo, Intel® Core™ Solo and dual-core Intel® Xeon® processor LV are based on an improved Pentium® M processor microarchitecture.

The Intel® Xeon® processor 3000, 3200, 5100, 5300, 7200 and 7300 series, Intel® Pentium® dual-core, Intel® Core™2 Duo, Intel® Core™2 Quad, and Intel® Core™2 Extreme processors are based on Intel® Core™ microarchitecture.

The Intel® Xeon® processor 5200, 5400, 7400 series, Intel® Core™2 Quad processor Q9000 series, and Intel® Core™2 Extreme processor QX9000, X9000 series, Intel® Core™2 processor E8000 series are based on Enhanced Intel® Core™ microarchitecture.

The Intel® Atom™ processor family is based on the Intel® Atom™ microarchitecture and supports Intel 64 architecture.

The Intel® Core™ i7 processor and the Intel® Core™ i5 processor are based on the Intel® microarchitecture code name Nehalem and support Intel 64 architecture.

Processors based on Intel® microarchitecture code name Westmere support Intel 64 architecture.

P6 family, Pentium® M, Intel® Core™ Solo, Intel® Core™ Duo processors, dual-core Intel® Xeon® processor LV, and early generations of Pentium 4 and Intel Xeon processors support IA-32 architecture. The Intel® Atom™ processor Z5xx series support IA-32 architecture.

The Intel® Xeon® processor E7-8800/4800/2800 product families, Intel® Xeon® processor 3000, 3200, 5000, 5100, 5200, 5300, 5400, 7100, 7200, 7300, 7400 series, Intel® Core™2 Duo, Intel® Core™2 Extreme processors, Intel Core 2 Quad processors, Pentium® D processors, Pentium® Dual-Core processor, newer generations of Pentium 4 and Intel Xeon processor family support Intel® 64 architecture.

IA-32 architecture is the instruction set architecture and programming environment for Intel's 32-bit microprocessors.

Intel® 64 architecture is the instruction set architecture and programming environment which is the superset of Intel's 32-bit and 64-bit architectures. It is compatible with the IA-32 architecture.

...

2. Updates to Chapter 2, Volume 1

Change bars show changes to Chapter 2 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1:* Basic Architecture.

...

2.1.17 2010 Intel® Core™ Processor Family (2010)

2010 Intel Core processor family spans Intel Core i7, i5 and i3 processors. They are based on Intel® microarchitecture code name Westmere using 32 nm process technology. The innovative features can include:

- Deliver smart performance using Intel Hyper-Threading Technology plus Intel Turbo Boost Technology.
- Enhanced Intel Smart Cache and integrated memory controller.
- Intelligent power gating.
- Repartitioned platform with on-die integration of 45nm integrated graphics.
- Range of instruction set support up to AESNI, PCLMULQDQ, SSE4.2 and SSE4.1.

...

2.1.19 Second Generation Intel® Core™ Processor Family (2011)

Second Generation Intel Core processor family spans Intel Core i7, i5 and i3 processors based on Intel® microarchitecture code name Sandy Bridge. They are built from 32 nm process technology and have innovative features including:

- Intel Turbo Boost Technology for Intel Core i5 and i7 processors
- Intel Hyper-Threading Technology.
- Enhanced Intel Smart Cache and integrated memory controller.
- Processor graphics and built-in visual features like Intel® Quick Sync Video, Intel® Insider™ etc.
- Range of instruction set support up to AVX, AESNI, PCLMULQDQ, SSE4.2 and SSE4.1.

...

2.2.6 Intel® microarchitecture code name Sandy Bridge

Intel® microarchitecture code name Sandy Bridge builds on the successes of Intel® Core™ microarchitecture and Intel microarchitecture code name Nehalem. It offers the following innovative features:

- Intel Advanced Vector Extensions (Intel AVX)
  - — 256-bit floating-point instruction set extensions to the 128-bit Intel Streaming SIMD Extensions, providing up to 2X performance benefits relative to 128-bit code.
  - — Non-destructive destination encoding offers more flexible coding techniques.
  - — Supports flexible migration and co-existence between 256-bit AVX code, 128-bit AVX code and legacy 128-bit SSE code.
- Enhanced front-end and execution engine
  - — New decoded Icache component that improves front-end bandwidth and reduces branch misprediction penalty.
  - — Advanced branch prediction.
  - — Additional macro-fusion support.

  - Larger dynamic execution window.
  - Multi-precision integer arithmetic enhancements (ADC/SBB, MUL/IMUL).
  - LEA bandwidth improvement.
  - Reduction of general execution stalls (read ports, writeback conflicts, bypass latency, partial stalls).
  - Fast floating-point exception handling.
  - XSAVE/XRSTORE performance improvements and XSAVEOPT new instruction.
- Cache hierarchy improvements for wider data path
  - Doubling of bandwidth enabled by two symmetric ports for memory operation.
  - Simultaneous handling of more in-flight loads and stores enabled by increased buffers.
  - Internal bandwidth of two loads and one store each cycle.
  - Improved prefetching.
  - High bandwidth low latency LLC architecture.
  - High bandwidth ring architecture of on-die interconnect.

For additional information on Intel® Advanced Vector Extensions (AVX), see Section 5.13, "Intel® Advanced Vector Extensions (AVX)" and Chapter 13, "Programming with AVX" in *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*.

2.2.7 SIMD Instructions

Beginning with the Pentium II and Pentium with Intel MMX technology processor families, six extensions have been introduced into the Intel 64 and IA-32 architectures to perform single-instruction multiple-data (SIMD) operations. These extensions include the MMX technology, SSE extensions, SSE2 extensions, SSE3 extensions, Supplemental Streaming SIMD Extensions 3, and SSE4. Each of these extensions provides a group of instructions that perform SIMD operations on packed integer and/or packed floating-point data elements.

SIMD integer operations can use the 64-bit MMX or the 128-bit XMM registers. SIMD floating-point operations use 128-bit XMM registers. Figure 2-4 shows a summary of the various SIMD extensions (MMX technology, SSE, SSE2, SSE3, SSSE3, and SSE4), the data types they operate on, and how the data types are packed into MMX and XMM registers.

The Intel MMX technology was introduced in the Pentium II and Pentium with MMX technology processor families. MMX instructions perform SIMD operations on packed byte, word, or doubleword integers located in MMX registers. These instructions are useful in applications that operate on integer arrays and streams of integer data that lend themselves to SIMD processing.

SSE extensions were introduced in the Pentium III processor family. SSE instructions operate on packed single-precision floating-point values contained in XMM registers and on packed integers contained in MMX registers. Several SSE instructions provide state management, cache control, and memory ordering operations. Other SSE instructions are targeted at applications that operate on arrays of single-precision floating-point data elements (3-D geometry, 3-D rendering, and video encoding and decoding applications).

SSE2 extensions were introduced in Pentium 4 and Intel Xeon processors. SSE2 instructions operate on packed double-precision floating-point values contained in XMM registers and on packed integers contained in MMX and XMM registers. SSE2 integer instructions extend IA-32 SIMD operations by adding new 128-bit SIMD integer operations and by expanding existing 64-bit SIMD integer operations to 128-bit XMM capability. SSE2 instructions also provide new cache control and memory ordering operations.

SSE3 extensions were introduced with the Pentium 4 processor supporting Hyper-Threading Technology (built on 90 nm process technology). SSE3 offers 13 instructions that accelerate performance of Streaming SIMD Extensions technology, Streaming SIMD Extensions 2 technology, and x87-FP math capabilities.

SSSE3 extensions were introduced with the Intel Xeon processor 5100 series and Intel Core 2 processor family. SSSE3 offer 32 instructions to accelerate processing of SIMD integer data.

SSE4 extensions offer 54 instructions. 47 of them are referred to as SSE4.1 instructions. SSE4.1 are introduced with Intel Xeon processor 5400 series and Intel Core 2 Extreme processor QX9650. The other 7 SSE4 instructions are referred to as SSE4.2 instructions.

AESNI and PCLMULQDQ introduce 7 new instructions. Six of them are primitives for accelerating algorithms based on AES encryption/decryption standard, referred to as AESNI.

The PCLMULQDQ instruction accelerates general-purpose block encryption, which can perform carry-less multiplication for two binary numbers up to 64-bit wide.

Intel 64 architecture allows four generations of 128-bit SIMD extensions to access up to 16 XMM registers. IA-32 architecture provides 8 XMM registers.

Intel® Advanced Vector Extensions offers comprehensive architectural enhancements over previous generations of Streaming SIMD Extensions. Intel AVX introduces the following architectural enhancements:

- Support for 256-bit wide vectors and SIMD register set.
- 256-bit floating-point instruction set enhancement with up to 2X performance gain relative to 128-bit Streaming SIMD extensions.
- Instruction syntax support for generalized three-operand syntax to improve instruction programming flexibility and efficient encoding of new instruction extensions.
- Enhancement of legacy 128-bit SIMD instruction extensions to support three operand syntax and to simplify compiler vectorization of high-level language expressions.
- Support flexible deployment of 256-bit AVX code, 128-bit AVX code, legacy 128-bit code and scalar code.

In addition to performance considerations, programmers should also be cognizant of the implications of VEX-encoded AVX instructions with the expectations of system software components that manage the processor state components enabled by XCR0. For additional information see Section 2.3.10.1, “Vector Length Transition and Programming Considerations” in *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 2A*.

See also:

- Section 5.4, “MMX™ Instructions,” and Chapter 9, “Programming with Intel® MMX™ Technology”
- Section 5.5, “SSE Instructions,” and Chapter 10, “Programming with Streaming SIMD Extensions (SSE)”

- Section 5.6, “SSE2 Instructions,” and Chapter 11, “Programming with Streaming SIMD Extensions 2 (SSE2)”
- Section 5.7, “SSE3 Instructions”, Section 5.8, “Supplemental Streaming SIMD Extensions 3 (SSSE3) Instructions”, Section 5.9, “SSE4 Instructions”, and Chapter 12, “Programming with SSE3, SSSE3, SSE4 and AESNI”

...

Table 2-2 Key Features of Most Recent Intel 64 Processors

Intel Processor	Date Intro-duced	Microarchitec-ture	Top-Bin Fre-quency at Intro-duction	Transistors	Register Sizes	System Bus/ QPI Link Speed	Max. Extern . Addr. Space	On-Die Caches

64-bit Intel Xeon Processor with 800 MHz System Bus	2004	Intel NetBurst Microarchitecture; Intel Hyper-Threading Technology; Intel 64 Architecture	3.60 GHz	125 M	GP: 32, 64
					FPU: 80
					MMX: 64
					XMM: 128
						6.4 GB/s	64 GB	12K µop Execution Trace Cache;
								16 KB L1;
								1 MB L2

64-bit Intel Xeon Processor MP with 8MB L3	2005	Intel NetBurst Microarchitecture; Intel Hyper-Threading Technology; Intel 64 Architecture	3.33 GHz	675M	GP: 32, 64
					FPU: 80
					MMX: 64
					XMM: 128
						5.3 GB/s[1]	1024 GB (1 TB)	12K µop Execution Trace Cache;
								16 KB L1;
								1 MB L2,
								8 MB L3

Intel Pentium 4 Processor Extreme Edition Supporting Hyper-Threading Technology	2005	Intel NetBurst Microarchitecture; Intel Hyper-Threading Technology; Intel 64 Architecture	3.73 GHz	164 M	GP: 32, 64
					FPU: 80
					MMX: 64
					XMM: 128
						8.5 GB/s	64 GB	12K µop Execution Trace Cache;
								16 KB L1;
								2 MB L2

Intel Pentium Processor Extreme Edition 840	2005	Intel NetBurst Microarchitecture; Intel Hyper-Threading Technology; Intel 64 Architecture;
		Dual-core[2]
			3.20 GHz	230 M	GP: 32, 64
					FPU: 80
					MMX: 64
					XMM: 128
						6.4 GB/s	64 GB	12K µop Execution Trace Cache;
								16 KB L1;
								1MB L2 (2MB Total)

Dual-Core Intel Xeon Processor 7041	2005	Intel NetBurst Microarchitecture; Intel Hyper-Threading Technology; Intel 64 Architecture;
		Dual-core[3]
			3.00 GHz	321M	GP: 32, 64
					FPU: 80
					MMX: 64
					XMM: 128
						6.4 GB/s	64 GB	12K µop Execution Trace Cache;
								16 KB L1;
								2MB L2 (4MB Total)

Intel Pentium 4 Processor 672	2005	Intel NetBurst Microarchitecture; Intel Hyper-Threading Technology; Intel 64 Architecture;
		Intel Virtualization Technology.
			3.80 GHz	164 M	GP: 32, 64
					FPU: 80
					MMX: 64
					XMM: 128
						6.4 GB/s	64 GB	12K µop Execution Trace Cache;
								16 KB L1;
								2MB L2

Intel Pentium Processor Extreme Edition 955	2006	Intel NetBurst Microarchitecture;
		Intel 64 Architecture;
		Dual Core;
		Intel Virtualization Technology.
			3.46 GHz	376M	GP: 32, 64
					FPU: 80
					MMX: 64
					XMM: 128
						8.5 GB/s	64 GB	12K µop Execution Trace Cache;
								16 KB L1;
								2MB L2 (4MB Total)

Intel Core 2 Extreme Processor X6800	2006	Intel Core Microarchitecture;
		Dual Core;
		Intel 64 Architecture;
		Intel Virtualization Technology.
			2.93 GHz	291M	GP: 32,64
					FPU: 80
					MMX: 64
					XMM: 128
						8.5 GB/s	64 GB	L1: 64 KB
								L2: 4MB (4MB Total)

Intel Xeon Processor 5160	2006	Intel Core Microarchitecture;
		Dual Core;
		Intel 64 Architecture;
		Intel Virtualization Technology.
			3.00 GHz	291M	GP: 32, 64
					FPU: 80
					MMX: 64
					XMM: 128
						10.6 GB/s	64 GB	L1: 64 KB
								L2: 4MB (4MB Total)

Table 2-2 Key Features of Most Recent Intel 64 Processors (Continued)

Intel Processor	Date Introduced	Microarchitec-ture	Top-Bin Frequency at Introduction	Transistors	Register Sizes	System Bus/ QPI Link Speed	Max. Extern. Addr. Space	On-Die Caches

Intel Xeon Processor 7140	2006	Intel NetBurst Microarchitecture;
		Dual Core;
		Intel 64 Architecture;
		Intel Virtualization Technology.
			3.40 GHz	1.3 B	GP: 32, 64
					FPU: 80
					MMX: 64
					XMM: 128
						12.8 GB/s	64 GB	L1: 64 KB
								L2: 1MB (2MB Total)
								L3: 16 MB (16MB Total)

Intel Core 2 Extreme Processor QX6700	2006	Intel Core Microarchitecture;
		Quad Core;
		Intel 64 Architecture;
		Intel Virtualization Technology.
			2.66 GHz	582M	GP: 32,64
					FPU: 80
					MMX: 64
					XMM: 128
						8.5 GB/s	64 GB	L1: 64 KB
								L2: 4MB (4MB Total)

Quad-core Intel Xeon Processor 5355	2006	Intel Core Microarchitecture;
		Quad Core;
		Intel 64 Architecture;
		Intel Virtualization Technology.
			2.66 GHz	582 M	GP: 32, 64
					FPU: 80
					MMX: 64
					XMM: 128
						10.6 GB/s	256 GB	L1: 64 KB
								L2: 4MB (8 MB Total)

Intel Core 2 Duo Processor E6850	2007	Intel Core Microarchitecture;
		Dual Core;
		Intel 64 Architecture;
		Intel Virtualization Technology;
		Intel Trusted Execution Technology
			3.00 GHz	291 M	GP: 32, 64
					FPU: 80
					MMX: 64
					XMM: 128
						10.6 GB/s	64 GB	L1: 64 KB
								L2: 4MB (4MB Total)

Intel Xeon Processor 7350	2007	Intel Core Microarchitecture;
		Quad Core;
		Intel 64 Architecture;
		Intel Virtualization Technology.
			2.93 GHz	582 M	GP: 32, 64
					FPU: 80
					MMX: 64
					XMM: 128
						8.5 GB/s	1024 GB	L1: 64 KB
								L2: 4MB (8MB Total)

Intel Xeon Processor 5472	2007	Enhanced Intel Core Microarchitecture;
		Quad Core;
		Intel 64 Architecture;
		Intel Virtualization Technology.
			3.00 GHz	820 M	GP: 32, 64
					FPU: 80
					MMX: 64
					XMM: 128
						12.8 GB/s	256 GB	L1: 64 KB
								L2: 6MB (12MB Total)

Intel Atom Processor	2008	Intel Atom Microarchitecture;
		Intel 64 Architecture;
		Intel Virtualization Technology.
			2.0 - 1.60 GHz	47 M	GP: 32, 64
					FPU: 80
					MMX: 64
					XMM: 128
						Up to 4.2 GB/s	Up to 64GB	L1: 56 KB[4]
								L2: 512KB

Intel Xeon Processor 7460	2008	Enhanced Intel Core Microarchitecture; Six Cores;
		Intel 64 Architecture;
		Intel Virtualization Technology.
			2.67 GHz	1.9 B	GP: 32, 64
					FPU: 80
					MMX: 64
					XMM: 128
						8.5 GB/s	1024 GB	L1: 64 KB
								L2: 3MB (9MB Total)
								L3: 16MB

Intel Atom Processor 330	2008	Intel Atom Microarchitecture;
		Intel 64 Architecture;
		Dual core;
		Intel Virtualization Technology.
			1.60 GHz	94 M	GP: 32, 64
					FPU: 80
					MMX: 64
					XMM: 128
						Up to 4.2 GB/s	Up to 64GB	L1: 56 KB[5]
								L2: 512KB (1MB Total)

Intel Core i7-965 Processor Extreme Edition	2008	Intel microarchitecture code name Nehalem;
		Quadcore;
		HyperThreading Technology; Intel QPI;
		Intel 64 Architecture;
		Intel Virtualization Technology.
			3.20 GHz	731 M	GP: 32, 64
					FPU: 80
					MMX: 64
					XMM: 128
						QPI: 6.4 GT/s; Memory: 25 GB/s	64 GB	L1: 64 KB
								L2: 256KB
								L3: 8MB

Table 2-2 Key Features of Most Recent Intel 64 Processors (Continued)

Intel Processor	Date Intro-duced	Micro-architec-ture	Top-Bin Fre-quency at Intro-duction	Tran-sistor s	Register Sizes	System Bus/ QPI Link Speed	Max. Extern . Addr. Space	On-Die Caches
Intel Core i7-620M Processor	2010	Intel Turbo Boost Technology, Intel microarchitecture code name Westmere; Dualcore; HyperThreading Technology; Intel 64 Architecture; Intel Virtualization Technology., Integrated graphics	2.66 GHz	383 M	GP: 32, 64 FPU: 80 MMX: 64 XMM: 128		64 GB	L1: 64 KB L2: 256KB L3: 4MB
Intel Xeon-Processor 5680	2010	Intel Turbo Boost Technology, Intel microarchitecture code name Westmere; Six core; HyperThreading Technology; Intel 64 Architecture; Intel Virtualization Technology.	3.33 GHz	1.1B	GP: 32, 64 FPU: 80 MMX: 64 XMM: 128	QPI: 6.4 GT/s; 32 GB/s	1 TB	L1: 64 KB L2: 256KB L3: 12MB
Intel Xeon-Processor 7560	2010	Intel Turbo Boost Technology, Intel microarchitecture code name Nehalem; Eight core; HyperThreading Technology; Intel 64 Architecture; Intel Virtualization Technology.	2.26 GHz	2.3B	GP: 32, 64 FPU: 80 MMX: 64 XMM: 128	QPI: 6.4 GT/s; Memory: 76 GB/s	16 TB	L1: 64 KB L2: 256KB L3: 24MB
Intel Core i7-2600K Processor	2011	Intel Turbo Boost Technology, Intel microarchitecture code name Sandy Bridge; Four core; HyperThreading Technology; Intel 64 Architecture; Intel Virtualization Technology., Processor graphics, Quicksync Video	3.40 GHz	995M	GP: 32, 64 FPU: 80 MMX: 64 XMM: 128 YMM: 256	DMI: 5 GT/s; Memory: 21 GB/s	64 GB	L1: 64 KB L2: 256KB L3: 8MB
Intel Xeon-Processor E3-1280	2011	Intel Turbo Boost Technology, Intel microarchitecture code name Sandy Bridge; Four core; HyperThreading Technology; Intel 64 Architecture; Intel Virtualization Technology.	3.50 GHz		GP: 32, 64 FPU: 80 MMX: 64 XMM: 128 YMM: 256	DMI: 5 GT/s; Memory: 21 GB/s	1 TB	L1: 64 KB L2: 256KB L3: 8MB
Intel Xeon-Processor E7-8870	2011	Intel Turbo Boost Technology, Intel microarchitecture code name Westmere; Ten core; HyperThreading Technology; Intel 64 Architecture; Intel Virtualization Technology.	2.40 GHz	2.2B	GP: 32, 64 FPU: 80 MMX: 64 XMM: 128	QPI: 6.4 GT/s; Memory: 102 GB/s	16 TB	L1: 64 KB L2: 256KB L3: 30MB

NOTES:

1. The 64-bit Intel Xeon Processor MP with an 8-MByte L3 supports a multi-processor platform with a dual system bus; this creates a platform bandwidth with 10.6 GBytes.
2. In Intel Pentium Processor Extreme Edition 840, the size of on-die cache is listed for each core. The total size of L2 in the physical package in 2 MBytes.
3. In Dual-Core Intel Xeon Processor 7041, the size of on-die cache is listed for each core. The total size of L2 in the physical package in 4 MBytes.

4. In Intel Atom Processor, the size of L1 instruction cache is 32 KBytes, L1 data cache is 24 KBytes.
5. In Intel Atom Processor, the size of L1 instruction cache is 32 KBytes, L1 data cache is 24 KBytes.

...

3. 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.

...

This chapter provides an abridged overview of Intel 64 and IA-32 instructions. Instructions are divided into the following groups:

- General purpose
- x87 FPU
- x87 FPU and SIMD state management
- Intel MMX technology
- SSE extensions
- SSE2 extensions
- SSE3 extensions
- SSSE3 extensions
- SSE4 extensions
- AESNI and PCLMULQDQ
- Intel AVX extensions
- System instructions
- IA-32e mode: 64-bit mode instructions
- VMX instructions
- SMX instructions

Table 5-1 lists the groups and IA-32 processors that support each group. More recent instruction set extensions are listed in Table 5-2. Within these groups, most instructions are collected into functional subgroups.

Table 5-1 Instruction Groups in Intel 64 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

Table 5-1 Instruction Groups in Intel 64 and IA-32 Processors (Continued)

Instruction Set Architecture	Intel 64 and IA-32 Processor Support
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 3xxx, 5100, 5200, 5300, 5400, 5500, 5600, 7300, 7400, 7500 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
IA-32e mode: 64-bit mode instructions	Intel 64 processors
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

Table 5-2 Recent Instruction Set Extensions in Intel 64 and IA-32 Processors

Instruction Set Architecture	Processor Generation Introduction
SSE4.1 Extensions	Intel Xeon processor 3100, 3300, 5200, 5400, 7400, 7500 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 X3400, X3500, X5500, X6500, X7500 series.
AESNI, PCLMULQDQ	InteL Xeon processor E7 family, Intel Xeon processors X3600, X5600, Intel Core i7 980X processor; Use CPUID to verify presence of AESNI and PCLMULQDQ across Intel Core processor families.
Intel AVX	Intel Xeon processor E3 series; Intel Core i7, i5, i3 processor 2xxx series.

...

4. Updates to Chapter 8, Volume 1

Change bars show changes to Chapter 8 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1:* Basic Architecture.

...

8.1.8 x87 FPU Instruction and Data (Operand) Pointers

The x87 FPU stores pointers to the instruction and data (operand) for the last non-control instruction executed. These are the x87 FPU instruction pointer and x87 FPU operand (data) pointers; software can save these pointers to provide state information for exception handlers. The pointers are illustrated in Figure 8-1 (the figure illustrates the pointers as used outside 64-bit mode; see below).

Note that the value in the x87 FPU data pointer register is always a pointer to a memory operand, If the last non-control instruction that was executed did not have a memory operand, the value in the data pointer register is undefined (reserved).

The contents of the x87 FPU instruction and data pointer registers remain unchanged when any of the control instructions (FCLEX/FNCLEX, FLDCW, FSTCW/FNSTCW, FSTSW/FNSTSW, FSTENV/FNSTENV, FLDENV, and WAIT/FWAIT) are executed.

For all the x87 FPUs and NPXs except the 8087, the x87 FPU instruction pointer points to any prefixes that preceded the instruction. For the 8087, the x87 FPU instruction pointer points only to the actual opcode.

The x87 FPU instruction and data pointers each consists of an offset and a segment selector. On processors that support IA-32e mode, each offset comprises 64 bits; on other processors, each offset comprises 32 bits. Each segment selector comprises 16 bits.

The pointers are accessed by the FINIT/FNINIT, FLDENV, FRSTOR, FSAVE/FNSAVE, FSTENV/FNSTENV, FXRSTOR, FXSAVE, XRSTOR, XSAVE, and XSAVEOPT instructions as follows:

- FINIT/FNINIT. Each instruction clears each 64-bit offset and 16-bit segment selector.
- FLDENV, FRSTOR. These instructions use the memory formats given in Figures 8-9 through 8-12:
  - For each 64-bit offset, each instruction loads the lower 32 bits from memory and clears the upper 32 bits.
  - If CR0.PE = 1, each instruction loads each 16-bit segment selector from memory; otherwise, it clears each 16-bit segment selector.
- FSAVE/FNSAVE, FSTENV/FNSTENV. These instructions use the memory formats given in Figures 8-9 through 8-12.
  - Each instruction saves the lower 32 bits of each 64-bit offset into memory. the upper 32 bits are not saved.
  - If CR0.PE = 1, each instruction saves each 16-bit segment selector into memory.
  - After saving these data into memory, FSAVE/FNSAVE clears each 64-bit offset and 16-bit segment selector.

- FXRSTOR, XRSTOR. These instructions load data from a memory image whose format depend on operating mode and the REX prefix. The memory formats are given in Tables 3-48, 3-51, and 3-52 in Chapter 3, "Instruction Set Reference, A-M," of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A*.
  - — Outside of 64-bit mode or if REX.W = 0, the instructions operate as follows:
    - For each 64-bit offset, each instruction loads the lower 32 bits from memory and clears the upper 32 bits.
    - Each instruction loads each 16-bit segment selector from memory.
  - — In 64-bit mode with REX.W = 1, the instructions operate as follows:
    - Each instruction loads each 64-bit offset from memory.
    - Each instruction clears each 16-bit segment selector.
- FXSAVE, XSAVE, and XSAVEOPT. These instructions store data into a memory image whose format depend on operating mode and the REX prefix. The memory formats are given in Tables 3-48, 3-51, and 3-52 in Chapter 3, "Instruction Set Reference, A-M," of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A*.
  - — Outside of 64-bit mode or if REX.W = 0, the instructions operate as follows:
    - Each instruction saves the lower 32 bits of each 64-bit offset into memory. The upper 32 bits are not saved.
    - Each instruction saves each 16-bit segment selector into memory.
  - — In 64-bit mode with REX.W = 1, each instruction saves each 64-bit offset into memory. The 16-bit segment selectors are not saved.

...

5. Updates to Appendix D, Volume 1

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

...

Example D-1 Incorrect Error Synchronization

```
FILD    COUNT       ;x87 FPU instruction
INC     COUNT       ;integer instruction alters operand
FSQRT               ;subsequent x87 FPU instruction -- error
                    ;from previous x87 FPU instruction detected here
```

Example D-2 Proper Error Synchronization

```
FILD    COUNT       ;x87 FPU instruction
FSQRT               ;subsequent x87 FPU instruction -- error from
                    ;previous x87 FPU instruction detected here

INC             COUNT               ;integer instruction alters operand
```

...

D.3.5 Need for Storing State of IGNNE# Circuit If Using x87 FPU and SMM

The recommended circuit (see Figure D-1) for MS-DOS compatibility x87 FPU exception handling for Intel486 processors and beyond contains two flip flops. When the x87 FPU exception handler accesses I/O port 0F0H it clears the IRQ13 interrupt request output from Flip Flop #1 and also clocks out the IGNNE# signal (active) from Flip Flop #2.

The assertion of IGNNE# may be used by the handler if needed to execute any x87 FPU instruction while ignoring the pending x87 FPU errors. The problem here is that the state of Flip Flop #2 is effectively an additional (but hidden) status bit that can affect processor behavior, and so ideally should be saved upon entering SMM, and restored before resuming to normal operation. If this is not done, and also the SMM code saves the x87 FPU state, AND an x87 FPU error handler is being used which relies on IGNNE# assertion, then (very rarely) the x87 FPU handler will nest inside itself and malfunction. The following example shows how this can happen.

Suppose that the x87 FPU exception handler includes the following sequence:

```
FNSTSW save_sw      ; save the x87 FPU status word
                    ; using a no-wait x87 FPU instruction
OUT    0F0H, AL     ; clears IRQ13 & activates IGNNE#
 . . . .
FLDCW  new_cw       ; loads new CW ignoring x87 FPU errors,
                    ; since IGNNE# is assumed active; or any
                    ; other x87 FPU instruction that is not a no-wait
                    ; type will cause the same problem
 . . . .
FCLEX               ; clear the x87 FPU error conditions & thus
                    ; turn off FERR# & reset the IGNNE# FF
```

The problem will only occur if the processor enters SMM between the OUT and the FLDCW instructions. But if that happens, AND the SMM code saves the x87 FPU state using FNSAVE, then the IGNNE# Flip Flop will be cleared (because FNSAVE clears the x87 FPU errors and thus de-asserts FERR#). When the processor returns from SMM it will restore the x87 FPU state with FRSTOR, which will re-assert FERR#, but the IGNNE# Flip Flop will not get set. Then when the x87 FPU error handler executes the FLDCW instruction, the active error condition will cause the processor to re-enter the x87 FPU error handler from the beginning. This may cause the handler to malfunction.

To avoid this problem, Intel recommends two measures:

1. Do not use the x87 FPU for calculations inside SMM code. (The normal power management, and sometimes security, functions provided by SMM have no need for x87 FPU calculations; if they are needed for some special case, use scaling or emulation instead.) This eliminates the need to do FNSAVE/FRSTOR inside SMM code, except when going into a 0 V suspend state (in which, in order to save power, the CPU is turned off completely, requiring its complete state to be saved).
2. The system should not call upon SMM code to put the processor into 0 V suspend while the processor is running x87 FPU calculations, or just after an interrupt has occurred. Normal power management protocol avoids this by going into power down states only after timed intervals in which no system activity occurs.

...

6. Updates to Chapter 1, Volume 2A

Change bars show changes to Chapter 1 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A:* Instruction Set Reference, A-M.

...

1.1 IA-32 PROCESSORS COVERED IN THIS MANUAL

This manual set includes information pertaining primarily to the most recent Intel 64 and IA-32 processors, which include:

- Pentium® processors
- P6 family processors
- Pentium® 4 processors
- Pentium® M processors
- Intel® Xeon® processors
- Pentium® D processors
- Pentium® processor Extreme Editions
- 64-bit Intel® Xeon® processors
- Intel® Core™ Duo processor
- Intel® Core™ Solo processor
- Dual-Core Intel® Xeon® processor LV
- Intel® Core™2 Duo processor
- Intel® Core™2 Quad processor Q6000 series
- Intel® Xeon® processor 3000, 3200 series
- Intel® Xeon® processor 5000 series
- Intel® Xeon® processor 5100, 5300 series
- Intel® Core™2 Extreme processor X7000 and X6800 series
- Intel® Core™2 Extreme QX6000 series
- Intel® Xeon® processor 7100 series
- Intel® Pentium® Dual-Core processor
- Intel® Xeon® processor 7200, 7300 series
- Intel® Xeon® processor 5200, 5400, 7400 series
- Intel® Core™2 Extreme processor QX9000 and X9000 series
- Intel® Core™2 Quad processor Q9000 series
- Intel® Core™2 Duo processor E8000, T9000 series
- Intel® Atom™ processor family
- Intel® Core™ i7 processor
- Intel® Core™ i5 processor

- Intel® Xeon® processor E7-8800/4800/2800 product families

P6 family processors are IA-32 processors based on the P6 family microarchitecture. This includes the Pentium® Pro, Pentium® II, Pentium® III, and Pentium® III Xeon® processors.

The Pentium® 4, Pentium® D, and Pentium® processor Extreme Editions are based on the Intel NetBurst® microarchitecture. Most early Intel® Xeon® processors are based on the Intel NetBurst® microarchitecture. Intel Xeon processor 5000, 7100 series are based on the Intel NetBurst® microarchitecture.

The Intel® Core™ Duo, Intel® Core™ Solo and dual-core Intel® Xeon® processor LV are based on an improved Pentium® M processor microarchitecture.

The Intel® Xeon® processor 3000, 3200, 5100, 5300, 7200, and 7300 series, Intel® Pentium® dual-core, Intel® Core™2 Duo, Intel® Core™2 Quad, and Intel® Core™2 Extreme processors are based on Intel® Core™ microarchitecture.

The Intel® Xeon® processor 5200, 5400, 7400 series, Intel® Core™2 Quad processor Q9000 series, and Intel® Core™2 Extreme processors QX9000, X9000 series, Intel® Core™2 processor E8000 series are based on Enhanced Intel® Core™ microarchitecture.

The Intel® Atom™ processor family is based on the Intel® Atom™ microarchitecture and supports Intel 64 architecture.

The Intel® Core™i7 processor and the Intel® Core™i5 processor are based on the Intel® microarchitecture code name Nehalem and support Intel 64 architecture.

Processors based on Intel® microarchitecture code name Westmere support Intel 64 architecture.

P6 family, Pentium® M, Intel® Core™ Solo, Intel® Core™ Duo processors, dual-core Intel® Xeon® processor LV, and early generations of Pentium 4 and Intel Xeon processors support IA-32 architecture. The Intel® Atom™ processor Z5xx series support IA-32 architecture.

The Intel® Xeon® processor E7-8800/4800/2800 product families, Intel® Xeon® processor 3000, 3200, 5000, 5100, 5200, 5300, 5400, 7100, 7200, 7300, 7400 series, Intel® Core™2 Duo, Intel® Core™2 Extreme, Intel® Core™2 Quad processors, Pentium® D processors, Pentium® Dual-Core processor, newer generations of Pentium 4 and Intel Xeon processor family support Intel® 64 architecture.

IA-32 architecture is the instruction set architecture and programming environment for Intel's 32-bit microprocessors.

Intel® 64 architecture is the instruction set architecture and programming environment which is the superset of Intel's 32-bit and 64-bit architectures. It is compatible with the IA-32 architecture.

...

7. 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.

...

2.3.10.1 Vector Length Transition and Programming Considerations

An instruction encoded with a VEX.128 prefix that loads a YMM register operand operates as follows:

- Data is loaded into bits 127:0 of the register
- Bits above bit 127 in the register are cleared.

Thus, such an instruction clears bits 255:128 of a destination YMM register on processors with a maximum vector-register width of 256 bits. In the event that future processors extend the vector registers to greater widths, an instruction encoded with a VEX.128 or VEX.256 prefix will also clear any bits beyond bit 255. (This is in contrast with legacy SSE instructions, which have no VEX prefix; these modify only bits 127:0 of any destination register operand.)

Programmers should bear in mind that instructions encoded with VEX.128 and VEX.256 prefixes will clear any future extensions to the vector registers. A calling function that uses such extensions should save their state before calling legacy functions. This is not possible for involuntary calls (e.g., into an interrupt-service routine). It is recommended that software handling involuntary calls accommodate this by not executing instructions encoded with VEX.128 and VEX.256 prefixes. In the event that it is not possible or desirable to restrict these instructions, then software must take special care to avoid actions that would, on future processors, zero the upper bits of vector registers.

Processors that support further vector-register extensions (defining bits beyond bit 255) will also extend the XSAVE and XRSTOR instructions to save and restore these extensions. To ensure forward compatibility, software that handles involuntary calls and that uses instructions encoded with VEX.128 and VEX.256 prefixes should first save and then restore the vector registers (with any extensions) using the XSAVE and XRSTOR instructions with save/restore masks that set bits that correspond to all vector-register extensions. Ideally, software should rely on a mechanism that is cognizant of which bits to set. (E.g., an OS mechanism that sets the save/restore mask bits for all vector-register extensions that are enabled in XCR0.) Saving and restoring state with instructions other than XSAVE and XRSTOR will, on future processors with wider vector registers, corrupt the extended state of the vector registers - even if doing so functions correctly on processors supporting 256-bit vector registers. (The same is true if XSAVE and XRSTOR are used with a save/restore mask that does not set bits corresponding to all supported extensions to the vector registers.)

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 subsections 2.4.1 through 2.4.8. 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. For instructions that operate on MMX registers, see Section 19.25.3, “Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A*.

...

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
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,

Exception Class	Instruction
	(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, VLDMXCSR*, VSTMXCSR
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

...

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		

...

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
Type 5		VEXTRACTPS, VINSERTPS, VMOVD, VMOVQ, VMOVLPD, VMOVLPS, VMOVHPD, VMOVHPS, VPEXTRB, VPEXTRD, VPEXTRW, VPEXTRQ, VPINSRB, VPINSRD, VPINSRW, VPINSRQ, VPMOVSX/ZX, VLDMXCSR, VSTMXCSR
Type 6	VEXTRACTF128, VPERM2F128, VBROADCASTSD, VBROADCASTF128, VINSERTF128,	
Type 7		VMOVLHPS, VMOVHLPS, VPMOVMSKB, VPSLLDQ, VPSRLDQ, VPSLLW, VPSLLD, VPSLLQ, VPSRAW, VPSRAD, VPSRLW, VPSRLD, VPSRLQ
Type 8		

...

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.

...

8. 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.

...

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** — 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.
- **ptr16:16, ptr16:32** — A far pointer, typically to a code segment different from that of the instruction. The notation *16:16* indicates that the value of the pointer has two parts. The value to the left of the colon is a 16-bit selector or value destined for the code segment register. The value to the right corresponds to the offset within the destination segment. The ptr16:16 symbol is used when the instruction's operand-size attribute is 16 bits; the ptr16:32 symbol is used when the operand-size attribute is 32 bits.
- **r8** — One of the byte general-purpose registers: AL, CL, DL, BL, AH, CH, DH, BH, BPL, SPL, DIL and SIL; or one of the byte registers (R8L - R15L) available when using REX.R and 64-bit mode.

- **r16** — One of the word general-purpose registers: AX, CX, DX, BX, SP, BP, SI, DI; or one of the word registers (R8-R15) available when using REX.R and 64-bit mode.
- **r32** — One of the doubleword general-purpose registers: EAX, ECX, EDX, EBX, ESP, EBP, ESI, EDI; or one of the doubleword registers (R8D - R15D) available when using REX.R in 64-bit mode.

...

ADC—Add with Carry

Opcode	Instruction	Op/ En	64-bit Mode	Compat/ Leg Mode	Description
14 *ib*	ADC AL, *imm8*	D	Valid	Valid	Add with carry *imm8* to AL.
15 *iw*	ADC AX, *imm16*	D	Valid	Valid	Add with carry *imm16* to AX.
15 *id*	ADC EAX, *imm32*	D	Valid	Valid	Add with carry *imm32* to EAX.
REX.W + 15 *id*	ADC RAX, *imm32*	D	Valid	N.E.	Add with carry *imm32 sign extended to 64-bits* to RAX.
80 /2 *ib*	ADC *r/m8, imm8*	C	Valid	Valid	Add with carry *imm8* to *r/m8.*
REX + 80 /2 *ib*	ADC r/m8*, imm8	C	Valid	N.E.	Add with carry *imm8* to *r/m8.*
81 /2 *iw*	ADC *r/m16, imm16*	C	Valid	Valid	Add with carry *imm16* to *r/m16.*
81 /2 *id*	ADC *r/m32, imm32*	C	Valid	Valid	Add with CF *imm32* to *r/m32.*
REX.W + 81 /2 *id*	ADC *r/m64, imm32*	C	Valid	N.E.	Add with CF *imm32* sign extended to 64-bits to *r/m64.*
83 /2 *ib*	ADC *r/m16, imm8*	C	Valid	Valid	Add with CF sign-extended *imm8* to *r/m16.*
83 /2 *ib*	ADC *r/m32, imm8*	C	Valid	Valid	Add with CF sign-extended *imm8* into *r/m32.*
REX.W + 83 /2 *ib*	ADC *r/m64, imm8*	C	Valid	N.E.	Add with CF sign-extended *imm8* into *r/m64.*
10 /*r*	ADC *r/m8, r8*	B	Valid	Valid	Add with carry byte register to *r/m8.*
REX + 10 /*r*	ADC r/m8*, r8*	B	Valid	N.E.	Add with carry byte register to *r/m64.*
11 /*r*	ADC *r/m16, r16*	B	Valid	Valid	Add with carry *r16* to *r/m16.*
11 /*r*	ADC *r/m32, r32*	B	Valid	Valid	Add with CF *r32* to *r/m32.*
REX.W + 11 /*r*	ADC *r/m64, r64*	B	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.

Opcode	Instruction	Op/ En	64-bit Mode	Compat/ Leg Mode	Description
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.

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)	ModRM:reg (r)	NA	NA
C	ModRM:r/m (r, w)	imm8	NA	NA
D	AL/AX/EAX/RAX	imm8	NA	NA

...

ADD—Add

Opcode	Instruction	Op/ En	64-bit Mode	Compat/ Leg Mode	Description
04 *ib*	ADD AL, *imm8*	D	Valid	Valid	Add *imm8* to AL.
05 *iw*	ADD AX, *imm16*	D	Valid	Valid	Add *imm16* to AX.
05 *id*	ADD EAX, *imm32*	D	Valid	Valid	Add *imm32* to EAX.
REX.W + 05 *id*	ADD RAX, *imm32*	D	Valid	N.E.	Add *imm32 sign-extended to 64-bits* to RAX.
80 /0 *ib*	ADD *r/m8, imm8*	C	Valid	Valid	Add *imm8* to *r/m8.*
REX + 80 /0 *ib*	ADD *r/m8**, *imm8*	C	Valid	N.E.	Add *sign*-extended *imm8* to *r/m64.*
81 /0 *iw*	ADD *r/m16, imm16*	C	Valid	Valid	Add *imm16* to *r/m16.*
81 /0 *id*	ADD *r/m32, imm32*	C	Valid	Valid	Add *imm32* to *r/m32.*
REX.W + 81 /0 *id*	ADD *r/m64, imm32*	C	Valid	N.E.	Add *imm32 sign-extended to 64-bits* to *r/m64.*
83 /0 *ib*	ADD *r/m16, imm8*	C	Valid	Valid	Add *sign*-extended *imm8* to *r/m16.*
83 /0 *ib*	ADD *r/m32, imm8*	C	Valid	Valid	Add *sign*-extended *imm8* to *r/m32.*
REX.W + 83 /0 *ib*	ADD *r/m64, imm8*	C	Valid	N.E.	Add *sign*-extended *imm8* to *r/m64.*
00 */r*	ADD *r/m8, r8*	B	Valid	Valid	Add *r8* to *r/m8.*

Opcode	Instruction	Op/ En	64-bit Mode	Compat/ Leg Mode	Description
REX + 00 /*r*	ADD *r/m8**, *r8**	B	Valid	N.E.	Add *r8* to *r/m8.*
01 /*r*	ADD *r/m16, r16*	B	Valid	Valid	Add *r16* to *r/m16.*
01 /*r*	ADD *r/m32, r32*	B	Valid	Valid	Add r32 to *r/m32.*
REX.W + 01 /*r*	ADD *r/m64, r64*	B	Valid	N.E.	Add r64 to *r/m64.*
02 /*r*	ADD *r8, r/m8*	A	Valid	Valid	Add *r/m8* to *r8.*
REX + 02 /*r*	ADD *r8**, *r/m8**	A	Valid	N.E.	Add *r/m8* to *r8.*
03 /*r*	ADD *r16, r/m16*	A	Valid	Valid	Add *r/m16* to *r16.*
03 /*r*	ADD *r32, r/m32*	A	Valid	Valid	Add *r/m32* to *r32.*
REX.W + 03 /*r*	ADD *r64, r/m64*	A	Valid	N.E.	Add *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.

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)	ModRM:reg (r)	NA	NA
C	ModRM:r/m (r, w)	imm8	NA	NA
D	AL/AX/EAX/RAX	imm8	NA	NA

...

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.

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

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; additionally

#UD	If VEX.vvvv != 1111B.

...

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.

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

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; additionally

#UD	If VEX.vvvv != 1111B.

...

AND—Logical AND

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

Opcode	Instruction	Op/ En	64-bit Mode	Compat/ Leg Mode	Description
83 /4 *ib*	AND *r/m32, imm8*	C	Valid	Valid	*r/m32* AND *imm8 (sign-extended).*
REX.W + 83 /4 *ib*	AND *r/m64, imm8*	C	Valid	N.E.	*r/m64* AND *imm8 (sign-extended).*
20 */r*	AND *r/m8, r8*	B	Valid	Valid	*r/m8* AND *r8.*
REX + 20 */r*	AND *r/m8**, *r8**	B	Valid	N.E.	*r/m64* AND *r8 (sign-extended).*
21 */r*	AND *r/m16, r16*	B	Valid	Valid	*r/m16* AND *r16.*
21 */r*	AND *r/m32, r32*	B	Valid	Valid	*r/m32* AND *r32.*
REX.W + 21 */r*	AND *r/m64, r64*	B	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.

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)	ModRM:reg (r)	NA	NA
C	ModRM:r/m (r, w)	imm8	NA	NA
D	AL/AX/EAX/RAX	imm8	NA	NA

...

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 4:0 of imm8 as a comparison predicate.

...

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	Version Information: Type, Family, Model, and Stepping ID (see Figure 3-9)
	EBX	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
	ECX EDX	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.

Table 3-17 Information Returned by CPUID Instruction (Continued)

Initial EAX Value	Information Provided about the Processor	
02H	EAX EBX ECX EDX	Cache and TLB Information (see Table 3-22) 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 3-227.
	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*

Table 3-17 Information Returned by CPUID Instruction (Continued)

Initial EAX Value	Information Provided about the Processor	
	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
	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	

Table 3-17 Information Returned by CPUID Instruction (Continued)

Initial EAX Value	Information Provided about the Processor	
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
	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

Table 3-17 Information Returned by CPUID Instruction (Continued)

Initial EAX Value	Information Provided about the Processor	
	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. ** Software must not use EBX[15:0] to enumerate processor topology of the system. This value in this field (EBX[15:0]) is only intended for display/diagnostic purposes. The actual number of logical processors available to BIOS/OS/Applications may be different from the value of EBX[15:0], depending on software and platform hardware configurations. *** The value of the "level type" field is not related to level numbers in any way, higher "level type" values do not mean higher levels. Level type field has the following encoding: 0 : invalid 1 : SMT 2 : Core 3-255 : Reserved
	Processor Extended State Enumeration Main Leaf (EAX = 0DH, ECX = 0)	
0DH		**NOTES:** Leaf 0DH main leaf (ECX = 0).

Table 3-17 Information Returned by CPUID Instruction (Continued)

Initial EAX Value	Information Provided about the Processor	
	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. Bit 00: legacy x87 Bit 01: 128-bit SSE Bit 02: 256-bit AVX Bits 31- 03: 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	Bits 31-01: Reserved Bit 00: XSAVEOPT is available;
	EBX	Reserved
	ECX	Reserved
	EDX	Reserved
	Processor Extended State Enumeration Sub-leaves (EAX = 0DH, ECX = n, n > 1)	
0DH		**NOTES:** Leaf 0DH output depends on the initial value in ECX. If ECX contains an invalid sub leaf index, EAX/EBX/ECX/EDX return 0. Each valid sub-leaf index maps to a valid bit in the XCR0 register starting at bit position 2
	EAX	Bits 31-0: The size in bytes (from the offset specified in EBX) of the save area for an extended state feature associated with a valid sub-leaf index, *n*. This field reports 0 if the sub-leaf index, *n*, is invalid*.
	EBX	Bits 31-0: The offset in bytes of this extended state component's save area from the beginning of the XSAVE/XRSTOR area. This field reports 0 if the sub-leaf index, *n*, is invalid*.
	ECX	This field reports 0 if the sub-leaf index, *n*, is invalid*; otherwise it is reserved.
	EDX	This field reports 0 if the sub-leaf index, *n*, is invalid*; otherwise it is reserved.
	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	

Table 3-17 Information Returned by CPUID Instruction (Continued)

Initial EAX Value	Information Provided about the Processor	
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	Processor Brand String
	EBX	Processor Brand String Continued
	ECX	Processor Brand String Continued
	EDX	Processor Brand String Continued
80000003H	EAX	Processor Brand String Continued
	EBX	Processor Brand String Continued
	ECX	Processor Brand String Continued
	EDX	Processor Brand String Continued
80000004H	EAX	Processor Brand String Continued
	EBX	Processor Brand String Continued
	ECX	Processor Brand String Continued
	EDX	Processor Brand String Continued
80000005H	EAX	Reserved = 0
	EBX	Reserved = 0
	ECX	Reserved = 0
	EDX	Reserved = 0
80000006H	EAX	Reserved = 0
	EBX	Reserved = 0
	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

Table 3-17 Information Returned by CPUID Instruction (Continued)

Initial EAX Value	Information Provided about the Processor
	NOTES: * L2 associativity field encodings: 00H - Disabled 01H - Direct mapped 02H - 2-way 04H - 4-way 06H - 8-way 08H - 16-way 0FH - Fully associative
80000007H	EAX Reserved = 0 EBX Reserved = 0 ECX Reserved = 0 EDX 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.

...

CVTPD2PI—Convert Packed Double-Precision FP Values to Packed Dword Integers

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
66 0F 2D /*r*	CVTPD2PI *mm, xmm/m128*	A	Valid	Valid	Convert two packed double-precision floating-point values from *xmm/m128* to two packed signed doubleword integers in *mm*.

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 MMX technology 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.

This instruction causes a transition from x87 FPU to MMX technology operation (that is, the x87 FPU top-of-stack pointer is set to 0 and the x87 FPU tag word is set to all 0s [valid]). If this instruction is executed while an x87 FPU floating-point exception is pending, the exception is handled before the CVTPD2PI instruction is executed.

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

Operation

```
DEST[31:0] ← Convert_Double_Precision_Floating_Point_To_Integer32(SRC[63:0]);
DEST[63:32] ← Convert_Double_Precision_Floating_Point_To_Integer32(SRC[127:64]);
```

Intel C/C++ Compiler Intrinsic Equivalent

```
CVTPD1PI __m64 _mm_cvtpd_pi32(__m128d a)
```

SIMD Floating-Point Exceptions

Invalid, Precision.

Other Exceptions

See Table 19-4, “Exception Conditions for Legacy SIMD/MMX Instructions with FP Exception and 16-Byte Alignment,” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A*.

...

CVTPI2PD—Convert Packed Dword Integers to Packed Double-Precision FP Values

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
66 0F 2A /*r*	CVTPI2PD *xmm, mm/m64**	A	Valid	Valid	Convert two packed signed doubleword integers from *mm/mem64* to two packed double-precision floating-point values in *xmm*.

NOTES:

*Operation is different for different operand sets; see the Description section.

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).

The source operand can be an MMX technology register or a 64-bit memory location. The destination operand is an XMM register. In addition, depending on the operand configuration:

- **For operands *xmm, mm*:** the instruction causes a transition from x87 FPU to MMX technology operation (that is, the x87 FPU top-of-stack pointer is set to 0 and the x87 FPU tag word is set to all 0s [valid]). If this instruction is executed while an x87 FPU floating-point exception is pending, the exception is handled before the CVTPI2PD instruction is executed.
- **For operands *xmm, m64*:** the instruction does not cause a transition to MMX technology and does not take x87 FPU exceptions.

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

Operation

```
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]);
```

Intel C/C++ Compiler Intrinsic Equivalent

```
CVTPI2PD __m128d _mm_cvtpi32_pd(__m64 a)
```

SIMD Floating-Point Exceptions

Precision.

Other Exceptions

See Table 19-6, “Exception Conditions for Legacy SIMD/MMX Instructions with XMM and without FP Exception,” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A*.

...

CVTPI2PS—Convert Packed Dword Integers to Packed Single-Precision FP Values

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
OF 2A /*r*	CVTPI2PS *xmm, mm/m64*	A	Valid	Valid	Convert two signed doubleword integers from *mm/m64* to two single-precision floating-point values in *xmm*.

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 single-precision floating-point values in the destination operand (first operand).

The source operand can be an MMX technology register or a 64-bit memory location. The destination operand is an XMM register. The results are stored in the low quadword of the destination operand, and the high quadword remains unchanged. When a conversion is inexact, the value returned is rounded according to the rounding control bits in the MXCSR register.

This instruction causes a transition from x87 FPU to MMX technology operation (that is, the x87 FPU top-of-stack pointer is set to 0 and the x87 FPU tag word is set to all 0s [valid]). If this instruction is executed while an x87 FPU floating-point exception is pending, the exception is handled before the CVTPI2PS instruction is executed.

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

Operation

```
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]);
(* High quadword of destination unchanged *)
```

Intel C/C++ Compiler Intrinsic Equivalent

CVTPI2PS __m128 _mm_cvtpi32_ps(__m128 a, __m64 b)

SIMD Floating-Point Exceptions

Precision.

Other Exceptions

See Table 19-5, “Exception Conditions for Legacy SIMD/MMX Instructions with XMM and FP Exception,” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A*.

...

CVTPS2PI—Convert Packed Single-Precision FP Values to Packed Dword Integers

Opcode	**Instruction**	**Op/ En**	**64-Bit Mode**	**Compat/ Leg Mode**	**Description**
0F 2D /*r*	CVTPS2PI *mm, xmm/m64*	A	Valid	Valid	Convert two packed single-precision floating-point values from *xmm/m64* to two packed signed doubleword integers in *mm*.

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 single-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 MMX technology register. When the source operand is an XMM register, the two single-precision floating-point values are 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.

CVTPS2PI causes a transition from x87 FPU to MMX technology operation (that is, the x87 FPU top-of-stack pointer is set to 0 and the x87 FPU tag word is set to all 0s [valid]). If this instruction is executed while an x87 FPU floating-point exception is pending, the exception is handled before the CVTPS2PI instruction is executed.

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

Operation

```
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]);
```

Intel C/C++ Compiler Intrinsic Equivalent

CVTPS2PI __m64 _mm_cvtps_pi32(__m128 a)

SIMD Floating-Point Exceptions

Invalid, Precision.

Other Exceptions

See Table 19-5, "Exception Conditions for Legacy SIMD/MMX Instructions with XMM and FP Exception," in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*.

...

CVTTPD2PI—Convert with Truncation Packed Double-Precision FP Values to Packed Dword Integers

Opcode/ Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
66 0F 2C /*r* CVTTPD2PI *mm, xmm/m128*	A	Valid	Valid	Convert two packer double-precision floating-point values from *xmm/m128* to two packed signed doubleword integers in *mm* 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 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 MMX technology 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, and if this exception is masked, the indefinite integer value (80000000H) is returned.

This instruction causes a transition from x87 FPU to MMX technology operation (that is, the x87 FPU top-of-stack pointer is set to 0 and the x87 FPU tag word is set to all 0s [valid]). If this instruction is executed while an x87 FPU floating-point exception is pending, the exception is handled before the CVTTPD2PI instruction is executed.

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

Operation

```
DEST[31:0] ← Convert_Double_Precision_Floating_Point_To_Integer32_Truncate(SRC[63:0]);
```

```
DEST[63:32] ← Convert_Double_Precision_Floating_Point_To_Integer32_
                        Truncate(SRC[127:64]);
```

Intel C/C++ Compiler Intrinsic Equivalent

CVTTPD1PI __m64 _mm_cvttpd_pi32(__m128d a)

SIMD Floating-Point Exceptions

Invalid, Precision.

Other Mode Exceptions

See Table 19-4, "Exception Conditions for Legacy SIMD/MMX Instructions with FP Exception and 16-Byte Alignment," in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*.

...

CVTTPS2PI—Convert with Truncation Packed Single-Precision FP Values to Packed Dword Integers

Opcode/ Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
0F 2C /r CVTTPS2PI *mm, xmm/m64*	A	Valid	Valid	Convert two single-precision floating-point values from *xmm/m64* to two signed doubleword signed integers in *mm* 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 packed single-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 64-bit memory location. The destination operand is an MMX technology register. When the source operand is an XMM register, the two single-precision floating-point values are 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, and if this exception is masked, the indefinite integer value (80000000H) is returned.

This instruction causes a transition from x87 FPU to MMX technology operation (that is, the x87 FPU top-of-stack pointer is set to 0 and the x87 FPU tag word is set to all 0s [valid]). If this instruction is executed while an x87 FPU floating-point exception is pending, the exception is handled before the CVTTPS2PI instruction is executed.

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

Operation

```
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]);
```

Intel C/C++ Compiler Intrinsic Equivalent

CVTTPS2PI __m64 _mm_cvttps_pi32(__m128 a)

SIMD Floating-Point Exceptions

Invalid, Precision.

Other Exceptions

See Table 19-5, "Exception Conditions for Legacy SIMD/MMX Instructions with XMM and FP Exception," in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*.

...

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;
```

DPPS (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.

...

Table 3-57 Layout of the 64-bit-mode FXSAVE Map (REX.W = 0)

15 14	13 12	11 10	9 8	7 6	5	4	3 2	1 0	
Reserved	CS	FPU IP		FOP	Reserved	FTW	FSW	FCW	**0**
MXCSR_MASK		MXCSR		Reserved	DS		FPU DP		**16**
Reserved			ST0/MM0						**32**
Reserved			ST1/MM1						**48**
Reserved			ST2/MM2						**64**
Reserved			ST3/MM3						**80**
Reserved			ST4/MM4						**96**
Reserved			ST5/MM5						**112**
Reserved			ST6/MM6						**128**
Reserved			ST7/MM7						**144**
XMM0									**160**
XMM1									**176**
XMM2									**192**
XMM3									**208**
XMM4									**224**
XMM5									**240**
XMM6									**256**
XMM7									**272**
XMM8									**288**
XMM9									**304**
XMM10									**320**
XMM11									**336**
XMM12									**352**
XMM13									**368**
XMM14									**384**
XMM15									**400**
Reserved									**416**
Reserved									**432**
Reserved									**448**
Available									**464**
Available									**480**
Available									**496**

…

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.

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

Operation

```
MXCSR ← m32;
```

C/C++ Compiler Intrinsic Equivalent

```
_mm_setcsr(unsigned int i)
```

Numeric Exceptions

None.

Other Exceptions

See Exceptions Type 5; additionally

#GP	For an attempt to set reserved bits in MXCSR.
#UD	If VEX.vvvv != 1111B.

...

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 memory location specified by the effective address in the DI/EDI/RDI register (the default segment register is DS, but this may be overridden with a segment-override prefix). 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

[1] ModRM.MOD = 011B required

in conjunction with MASKMOVDQU instructions if multiple processors might use different memory types to read/write the destination memory locations.

Behavior with a mask of all 0s is as follows:

- No data will be written to memory.
- Signaling of breakpoints (code or data) is not guaranteed; different processor implementations may signal or not signal these breakpoints.
- Exceptions associated with addressing memory and page faults may still be signaled (implementation dependent).
- If the destination memory region is mapped as UC or WP, enforcement of associated semantics for these memory types is not guaranteed (that is, is reserved) and is implementation-specific.

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.

...

MASKMOVQ—Store Selected Bytes of Quadword

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
0F F7 /r	MASKMOVQ *mm1, mm2*	A	Valid	Valid	Selectively write bytes from *mm1* to memory location using the byte mask in *mm2*. The default memory location is specified by DS:DI/EDI/RDI.

Instruction Operand Encoding

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 a 64-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 MMX technology registers. The memory location specified by the effective address in the DI/EDI/RDI register (the default segment register is DS, but this may be overridden with a segment-override prefix). 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 MASKMOVQ 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 MASKMOVQ instructions if multiple processors might use different memory types to read/write the destination memory locations.

This instruction causes a transition from x87 FPU to MMX technology state (that is, the x87 FPU top-of-stack pointer is set to 0 and the x87 FPU tag word is set to all 0s [valid]).

The behavior of the MASKMOVQ instruction with a mask of all 0s is as follows:

- No data will be written to memory.
- Transition from x87 FPU to MMX technology state will occur.
- Exceptions associated with addressing memory and page faults may still be signaled (implementation dependent).
- Signaling of breakpoints (code or data) is not guaranteed (implementation dependent).
- If the destination memory region is mapped as UC or WP, enforcement of associated semantics for these memory types is not guaranteed (that is, is reserved) and is implementation-specific.

The MASKMOVQ instruction can be used to improve performance for algorithms that need to merge data on a byte-by-byte basis. It 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, the memory address is specified by DS:RDI.

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 6th bytes in source operand *)
IF (MASK[63] = 1)
    THEN DEST[DI/EDI +15] ← SRC[63:56] ELSE (* Memory location unchanged *); FI;
```

Intel C/C++ Compiler Intrinsic Equivalent

```
void _mm_maskmove_si64(__m64d, __m64n, char * p)
```

Other Exceptions

See Table 19-8, "Exception Conditions for Legacy SIMD/MMX Instructions without FP Exception," in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*.

...

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:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA

...

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:r/m (r)	NA	NA

...

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

[1] ModRM.MOD = 011B is not permitted

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.

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_si128( __m128i *p, __m128i 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*.

...

[1] ModRM.MOD = 011B is not permitted

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*.

...

[1] ModRM.MOD = 011B is not permitted

MOVNTQ—Store of Quadword Using Non-Temporal Hint

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
0F E7 /*r*	MOVNTQ *m64, mm*	A	Valid	Valid	Move quadword from *mm* to *m64* using non-temporal hint.

Instruction Operand Encoding

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

Description

Moves the quadword in the source operand (second operand) to the destination operand (first operand) using a non-temporal hint to minimize cache pollution during the write to memory. The source operand is an MMX technology register, which is assumed to contain packed integer data (packed bytes, words, or doublewords). The destination operand is a 64-bit memory location.

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 MOVNTQ instructions if multiple processors might use different memory types to read/write the destination memory locations.

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

Operation

```
DEST ← SRC;
```

Intel C/C++ Compiler Intrinsic Equivalent

MOVNTQ void _mm_stream_pi(__m64 * p, __m64 a)

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Table 19-8, "Exception Conditions for Legacy SIMD/MMX Instructions without FP Exception," in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*.

...

MOVQ—Move Quadword

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
0F 6F /*r*	MOVQ *mm, mm/ m64*	A	Valid	Valid	Move quadword from *mm/ m64* to *mm*.
0F 7F /*r*	MOVQ *mm/m64, mm*	B	Valid	Valid	Move quadword from *mm* to *mm/m64*.
F3 0F 7E	MOVQ *xmm1, xmm2/m64*	A	Valid	Valid	Move quadword from *xmm2/mem64* to *xmm1*.
66 0F D6	MOVQ *xmm2/ m64, xmm1*	B	Valid	Valid	Move quadword from *xmm1* to *xmm2/mem64*.

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 quadword from the source operand (second operand) to the destination operand (first operand). The source and destination operands can be MMX technology registers, XMM registers, or 64-bit memory locations. This instruction can be used to move a quadword between two MMX technology registers or between an MMX technology register and a 64-bit memory location, or to move data between two XMM registers or between an XMM register and a 64-bit memory location. The instruction cannot be used to transfer data between memory locations.

When the source operand is an XMM register, the low quadword is moved; when the destination operand is an XMM register, the quadword is stored to the low quadword of the register, and the high quadword is cleared to all 0s.

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

Operation

```
MOVQ instruction when operating on MMX technology registers and memory locations:
    DEST ← SRC;

MOVQ instruction when source and destination operands are XMM registers:
    DEST[63:0] ← SRC[63:0];
    DEST[127:64] ← 0000000000000000H;

MOVQ instruction when source operand is XMM register and destination
operand is memory location:
    DEST ← SRC[63:0];
```

```
MOVQ instruction when source operand is memory location and destination
operand is XMM register:
    DEST[63:0] ← SRC;
    DEST[127:64] ← 0000000000000000H;
```

Flags Affected

None.

Intel C/C++ Compiler Intrinsic Equivalent

MOVQ   m128i _mm_mov_epi64(__m128i a)

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Table 19-8, "Exception Conditions for Legacy SIMD/MMX Instructions without FP Exception," in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*.

...

9. 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.

...

4.1.8 Diagram Comparison and Aggregation Process

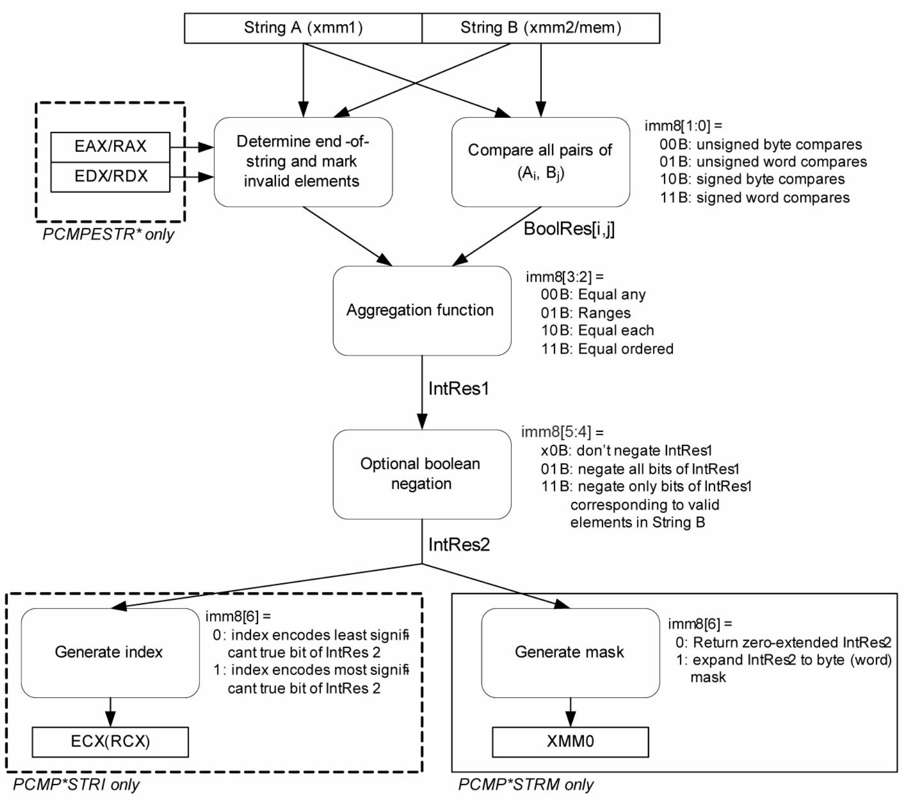
String A (xmm1)
String B (xmm2/mem)
EAX/RAX
EDX/RDX
PCMPESTR* only
Determine end -of-string and mark invalid elements
Compare all pairs of (A$_i$, B$_j$)
imm8[1:0] =
00B: unsigned byte compares
01B: unsigned word compares
10B: signed byte compares
11B: signed word compares
BoolRes[i,j]
Aggregation function
imm8[3:2] =
00B: Equal any
01B: Ranges
10B: Equal each
11B: Equal ordered
IntRes1
Optional boolean negation
imm8[5:4] =
x0B: don't negate IntRes1
01B: negate all bits of IntRes1
11B: negate only bits of IntRes1 corresponding to valid elements in String B
IntRes2
Generate index
imm8[6] =
0: index encodes least significant true bit of IntRes 2
1: index encodes most significant true bit of IntRes 2
ECX(RCX)
PCMP*STRI only
Generate mask
imm8[6] =
0: Return zero-extended IntRes2
1: expand IntRes2 to byte (word) mask
XMM0
PCMP*STRM only

Figure 4-1 Operation of PCMPSTRx and PCMPESTRx

...

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” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 2A* and Section 19.25.3, “Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A*.

...

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" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A* and Section 19.25.3, "Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*.

...

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” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 2A* and Section 19.25.3, “Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A*.

...

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” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 2A* and Section 19.25.3, “Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A*.

...

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" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A* and Section 19.25.3, "Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*.

...

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" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A* and Section 19.25.3, "Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*.

...

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.128.6 60F.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.128.6 6.0F.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" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A* and Section 19.25.3, "Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*.

...

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” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 2A* and Section 19.25.3, “Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A*.

...

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” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 2A* and Section 19.25.3, “Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A*.

...

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” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 2A* and Section 19.25.3, “Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A*.

...

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” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 2A* and Section 19.25.3, “Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A*.

...

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” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 2A* and Section 19.25.3, “Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A*.

...

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 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.

...

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 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, bits (VLMAX-1:128) of XMM0 are zeroed. VEX.vvvv is reserved and must be 1111b, VEX.L must be 0, otherwise the instruction will #UD.

...

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” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 2A* and Section 19.25.3, “Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A*.

...

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

...

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, bits (VLMAX-1:128) of XMM0 are zeroed. VEX.vvvv is reserved and must be 1111b, VEX.L must be 0, otherwise the instruction will #UD.

...

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	SSE2	Extract the word specified by *imm8* from *xmm* and move it to *reg*, bits 15-0. The upper bits of r32 or r64 is zeroed.
66 0F 3A 15 /r ib PEXTRW *reg/m16, xmm, imm8*	B	V/V	SSE4_1	Extract the word specified by *imm8* from *xmm* and copy it to lowest 16 bits of *reg or m16*. Zero-extend the result in the destination, r32 or r64.
VEX.128.66.0F.W0 C5 /r ib VPEXTRW reg, xmm1, imm8	A	V[2]/V	AVX	Extract the word specified by imm8 from xmm1 and move it to reg, bits 15:0. Zero-extend the result. The upper bits of r64/r32 is filled with zeros.
VEX.128.66.0F3A.W0 15 /r ib VPEXTRW reg/m16, xmm2, imm8	B	V/V	AVX	Extract a word integer value from xmm2 at the source word offset specified by imm8 into reg or m16. The upper bits of r64/r32 is filled with zeros.

NOTES:

1. See note in Section 2.4, “Instruction Exception Specification” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 2A* and Section 19.25.3, “Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A*.

2. In 64-bit mode, VEX.W1 is ignored for VPEXTRW (similar to legacy REX.W=1 prefix in PEXTRW).

...

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 integers horizontally, pack to MM1.
66 0F 38 01 /r PHADDW xmm1, xmm2/m128	A	V/V	SSSE3	Add 16-bit integers horizontally, pack to XMM1.
0F 38 02 /r PHADDD mm1, mm2/m64	A	V/V	SSSE3	Add 32-bit integers horizontally, pack to MM1.
66 0F 38 02 /r PHADDD xmm1, xmm2/m128	A	V/V	SSSE3	Add 32-bit 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 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 integers horizontally, pack to xmm1.

NOTES:

1. See note in Section 2.4, "Instruction Exception Specification" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A* and Section 19.25.3, "Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*.

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.

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, 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.

...

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" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A* and Section 19.25.3, "Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*.

...

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" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A* and Section 19.25.3, "Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*.

...

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" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A* and Section 19.25.3, "Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*.

...

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" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A* and Section 19.25.3, "Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*.
2. In 64-bit mode, VEX.W1 is ignored for VPINSRW (similar to legacy REX.W=1 prefix in PINSRW).

...

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" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A* and Section 19.25.3, "Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*.

...

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” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 2A* and Section 19.25.3, “Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A*.

...

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" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A* and Section 19.25.3, "Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*.

...

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" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A* and Section 19.25.3, "Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*.

...

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" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A* and Section 19.25.3, "Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*.

...

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" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A* and Section 19.25.3, "Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*.

...

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" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A* and Section 19.25.3, "Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*.

...

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" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A* and Section 19.25.3, "Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*.

...

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” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 2A* and Section 19.25.3, “Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A*.

...

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" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A* and Section 19.25.3, "Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*.

...

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” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 2A* and Section 19.25.3, “Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A*.

...

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” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 2A* and Section 19.25.3, “Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A*.

...

POP—Pop a Value from the Stack

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
8F /0	POP r/*m16*	A	Valid	Valid	Pop top of stack into *m16*; increment stack pointer.
8F /0	POP r/*m32*	A	N.E.	Valid	Pop top of stack into *m32*; increment stack pointer.
8F /0	POP r/*m64*	A	Valid	N.E.	Pop top of stack into *m64*; increment stack pointer. Cannot encode 32-bit operand size.
58+ *rw*	POP *r16*	B	Valid	Valid	Pop top of stack into *r16*; increment stack pointer.
58+ *rd*	POP *r32*	B	N.E.	Valid	Pop top of stack into *r32*; increment stack pointer.
58+ *rd*	POP *r64*	B	Valid	N.E.	Pop top of stack into *r64*; increment stack pointer. Cannot encode 32-bit operand size.
1F	POP DS	C	Invalid	Valid	Pop top of stack into DS; increment stack pointer.
07	POP ES	C	Invalid	Valid	Pop top of stack into ES; increment stack pointer.
17	POP SS	C	Invalid	Valid	Pop top of stack into SS; increment stack pointer.
0F A1	POP FS	C	Valid	Valid	Pop top of stack into FS; increment stack pointer by 16 bits.
0F A1	POP FS	C	N.E.	Valid	Pop top of stack into FS; increment stack pointer by 32 bits.
0F A1	POP FS	C	Valid	N.E.	Pop top of stack into FS; increment stack pointer by 64 bits.
0F A9	POP GS	C	Valid	Valid	Pop top of stack into GS; increment stack pointer by 16 bits.
0F A9	POP GS	C	N.E.	Valid	Pop top of stack into GS; increment stack pointer by 32 bits.
0F A9	POP GS	C	Valid	N.E.	Pop top of stack into GS; increment stack pointer by 64 bits.

Instruction Operand Encoding

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

Description

Loads the value from the top of the stack to the location specified with the destination operand (or explicit opcode) and then increments the stack pointer. The destination operand can be a general-purpose register, memory location, or segment register.

Address and operand sizes are determined and used as follows:

- Address size. The D flag in the current code-segment descriptor determines the default address size; it may be overridden by an instruction prefix (67H).

  The address size is used only when writing to a destination operand in memory.
- Operand size. The D flag in the current code-segment descriptor determines the default operand size; it may be overridden by instruction prefixes (66H or REX.W).

  The operand size (16, 32, or 64 bits) determines the amount by which the stack pointer is incremented (2, 4 or 8).
- Stack-address size. Outside of 64-bit mode, the B flag in the current stack-segment descriptor determines the size of the stack pointer (16 or 32 bits); in 64-bit mode, the size of the stack pointer is always 64 bits.

  The stack-address size determines the width of the stack pointer when reading from the stack in memory and when incrementing the stack pointer. (As stated above, the amount by which the stack pointer is incremented is determined by the operand size.)

If the destination operand is one of the segment registers DS, ES, FS, GS, or SS, the value loaded into the register must be a valid segment selector. In protected mode, popping a segment selector into a segment register automatically causes the descriptor information associated with that segment selector to be loaded into the hidden (shadow) part of the segment register and causes the selector and the descriptor information to be validated (see the "Operation" section below).

A NULL value (0000-0003) may be popped into the DS, ES, FS, or GS register without causing a general protection fault. However, any subsequent attempt to reference a segment whose corresponding segment register is loaded with a NULL value causes a general protection exception (#GP). In this situation, no memory reference occurs and the saved value of the segment register is NULL.

The POP instruction cannot pop a value into the CS register. To load the CS register from the stack, use the RET instruction.

If the ESP register is used as a base register for addressing a destination operand in memory, the POP instruction computes the effective address of the operand after it increments the ESP register. For the case of a 16-bit stack where ESP wraps to 0H as a result of the POP instruction, the resulting location of the memory write is processor-family-specific.

The POP ESP instruction increments the stack pointer (ESP) before data at the old top of stack is written into the destination.

A POP SS instruction inhibits all interrupts, including the NMI interrupt, until after execution of the next instruction. This action allows sequential execution of POP SS and MOV

ESP, EBP instructions without the danger of having an invalid stack during an interrupt[1]. However, use of the LSS instruction is the preferred method of loading the SS and ESP registers.

In 64-bit mode, using a REX prefix in the form of REX.R permits access to additional registers (R8-R15). When in 64-bit mode, POPs using 32-bit operands are not encodable and POPs to DS, ES, SS are not valid. See the summary chart at the beginning of this section for encoding data and limits.

...

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" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A* and Section 19.25.3, "Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*.

...

[1] If a code instruction breakpoint (for debug) is placed on an instruction located immediately after a POP SS instruction, the breakpoint may not be triggered. However, in a sequence of instructions that POP the SS register, only the first instruction in the sequence is guaranteed to delay an interrupt.

In the following sequence, interrupts may be recognized before POP ESP executes:

```
POP SS
POP SS
POP ESP
```

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” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 2A* and Section 19.25.3, “Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A*.

...

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" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A* and Section 19.25.3, "Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*.

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
```

...

PSHUFW—Shuffle Packed Words

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
0F 70 /*r* ib	PSHUFW *mm1, mm2/m64, imm8*	A	Valid	Valid	Shuffle the words in *mm2/ m64* based on the encoding in *imm8* and store the result in *mm1*.

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 source operand (second operand) and inserts them in 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 PSHUFW instruction, each 2-bit field in the order operand selects the contents of one word location in the destination operand. The encodings of the order operand fields select words from the source operand to be copied to the destination operand.

The source operand can be an MMX technology register or a 64-bit memory location. The destination operand is an MMX technology register. The order operand is an 8-bit immediate. Note that this instruction permits a word in the source operand to be copied to more than one word 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).

Operation

```
DEST[15:0] ← (SRC >> (ORDER[1:0] * 16))[15:0];
DEST[31:16] ← (SRC >> (ORDER[3:2] * 16))[15:0];
DEST[47:32] ← (SRC >> (ORDER[5:4] * 16))[15:0];
DEST[63:48] ← (SRC >> (ORDER[7:6] * 16))[15:0];
```

Intel C/C++ Compiler Intrinsic Equivalent

```
PSHUFW    __m64 _mm_shuffle_pi16(__m64 a, int n)
```

Flags Affected

None.

Numeric Exceptions

None.

Other Exceptions

See Table 19-7, “Exception Conditions for SIMD Instructions with Memory Reference,” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A*.

...

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” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 2A* and Section 19.25.3, “Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A*.

...

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.

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
VEX.NDS.128.66.0F.WIG F1 /r VPSLLW xmm1, xmm2, xmm3/m128	C	V/V	AVX	Shift words in xmm2 left by amount specified in xmm3/m128 while shifting in 0s.
VEX.NDD.128.66.0F.WIG 71 /6 ib VPSLLW xmm1, xmm2, imm8	D	V/V	AVX	Shift words in xmm2 left by imm8 while shifting in 0s.
VEX.NDS.128.66.0F.WIG F2 /r VPSLLD xmm1, xmm2, xmm3/m128	C	V/V	AVX	Shift doublewords in xmm2 left by amount specified in xmm3/m128 while shifting in 0s.
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” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 2A* and Section 19.25.3, “Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A*.

...

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

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
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” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 2A* and Section 19.25.3, “Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A*.

...

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.
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.

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
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" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A* and Section 19.25.3, "Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*.

...

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*.
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” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 2A* and Section 19.25.3, “Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A*.

...

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" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A* and Section 19.25.3, "Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*.

...

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” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 2A* and Section 19.25.3, “Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A*.

...

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" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A* and Section 19.25.3, "Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*.

...

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.

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.

The first source register is specified by the ModR/M *reg* field.

128-bit versions: The first source register is an XMM register. The second source register can be an XMM register or a 128-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

(V)PTEST (128-bit version)

```
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;
```

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);
```

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.

...

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.

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
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” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 2A* and Section 19.25.3, “Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A*.

...

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.

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
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” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 2A* and Section 19.25.3, “Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers” in the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A*.

...

PUSH—Push Word, Doubleword or Quadword Onto the Stack

Opcode*	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
FF /6	PUSH *r/m16*	A	Valid	Valid	Push *r/m16.*
FF /6	PUSH *r/m32*	A	N.E.	Valid	Push *r/m32.*
FF /6	PUSH *r/m64*	A	Valid	N.E.	Push *r/m64.*
50+*rw*	PUSH *r16*	B	Valid	Valid	Push *r16.*
50+*rd*	PUSH *r32*	B	N.E.	Valid	Push *r32.*
50+*rd*	PUSH *r64*	B	Valid	N.E.	Push *r64.*
6A	PUSH *imm8*	C	Valid	Valid	Push *imm8.*
68	PUSH *imm16*	C	Valid	Valid	Push *imm16.*
68	PUSH *imm32*	C	Valid	Valid	Push *imm32.*
0E	PUSH CS	D	Invalid	Valid	Push CS.
16	PUSH SS	D	Invalid	Valid	Push SS.
1E	PUSH DS	D	Invalid	Valid	Push DS.
06	PUSH ES	D	Invalid	Valid	Push ES.

Opcode*	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
0F A0	PUSH FS	D	Valid	Valid	Push FS.
0F A8	PUSH GS	D	Valid	Valid	Push GS.

NOTES:

* See IA-32 Architecture Compatibility section below.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
A	ModRM:r/m (r)	NA	NA	NA
B	reg (r)	NA	NA	NA
C	imm8/16/32	NA	NA	NA
D	NA	NA	NA	NA

Description

Decrements the stack pointer and then stores the source operand on the top of the stack. Address and operand sizes are determined and used as follows:

- Address size. The D flag in the current code-segment descriptor determines the default address size; it may be overridden by an instruction prefix (67H).

  The address size is used only when referencing a source operand in memory.

- Operand size. The D flag in the current code-segment descriptor determines the default operand size; it may be overridden by instruction prefixes (66H or REX.W).

  The operand size (16, 32, or 64 bits) determines the amount by which the stack pointer is decremented (2, 4 or 8).

  If the source operand is an immediate and its size is less than the operand size, a sign-extended value is pushed on the stack. If the source operand is a segment register (16 bits) and the operand size is greater than 16 bits, a zero-extended value is pushed on the stack.

- Stack-address size. Outside of 64-bit mode, the B flag in the current stack-segment descriptor determines the size of the stack pointer (16 or 32 bits); in 64-bit mode, the size of the stack pointer is always 64 bits.

  The stack-address size determines the width of the stack pointer when writing to the stack in memory and when decrementing the stack pointer. (As stated above, the amount by which the stack pointer is decremented is determined by the operand size.)

  If the operand size is less than the stack-address size, the PUSH instruction may result in a misaligned stack pointer (a stack pointer that is not aligned on a doubleword or quadword boundary).

The PUSH ESP instruction pushes the value of the ESP register as it existed before the instruction was executed. If a PUSH instruction uses a memory operand in which the ESP register is used for computing the operand address, the address of the operand is computed before the ESP register is decremented.

If the ESP or SP register is 1 when the PUSH instruction is executed in real-address mode, a stack-fault exception (#SS) is generated (because the limit of the stack segment is violated). Its delivery encounters a second stack-fault exception (for the same reason), causing generation of a double-fault exception (#DF). Delivery of the double-fault exception encounters a third stack-fault exception, and the logical processor enters shutdown mode. See the discussion of the double-fault exception in Chapter 6 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*.

IA-32 Architecture Compatibility

For IA-32 processors from the Intel 286 on, the PUSH ESP instruction pushes the value of the ESP register as it existed before the instruction was executed. (This is also true for Intel 64 architecture, real-address and virtual-8086 modes of IA-32 architecture.) For the Intel® 8086 processor, the PUSH SP instruction pushes the new value of the SP register (that is the value after it has been decremented by 2).

Operation

```
IF SRC is a segment register
    THEN
        IF operand size = 16
            THEN TEMP ← SRC;
            ELSE TEMP ← ZeroExtend(SRC);     (* extend to operand size *)
        FI;
ELSE IF SRC is immediate byte
    THEN TEMP ← SignExtend(SRC);             (* extend to operand size *)
ELSE IF SRC is immediate word                (* operand size is 16 *)
    THEN TEMP ← SRC;
ELSE IF SRC is immediate doubleword          (* operand size is 32 or 64 *)
    THEN
        IF operand size = 32
            THEN TEMP ← SRC;
            ELSE TEMP ← SignExtend(SRC);     (* extend to operand size of 64 *)
        FI;
ELSE IF SRC is in memory
    THEN TEMP ← SRC;                         (* use address and operand sizes *)
    ELSE TEMP ← SRC;                         (* SRC is register; use operand size *)
FI;
IF in 64-bit mode                            (* stack-address size = 64 *)
    THEN
        IF operand size = 64
            THEN
                RSP ← RSP – 8;
                Memory[RSP] ← TEMP;          (* Push quadword *)
            ELSE                             (* operand size = 16 *)
                RSP ← RSP – 2;
                Memory[RSP] ← TEMP;          (* Push word *)
        FI;
ELSE IF stack-address size = 32
    THEN
        IF operand size = 32
            THEN
                ESP ← ESP – 4;
                Memory[SS:ESP] ← TEMP;          (* Push doubleword *)
            ELSE                                (* operand size = 16 *)
                ESP ← ESP – 2;
                Memory[SS:ESP] ← TEMP;          (* Push word *)
        FI;
    ELSE                                        (* stack-address size = 16 *)
        IF operand size = 32
            THEN
                SP ← SP – 4;
                Memory[SS:SP] ← TEMP;           (* Push doubleword *)
            ELSE                                (* operand size = 16 *)
                SP ← SP – 2;
                Memory[SS:SP] ← TEMP;           (* Push word *)
        FI;
FI;
```

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 the DS, ES, FS, or GS register is used to access memory and it contains a NULL segment selector.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If the LOCK prefix is used.

Real-Address Mode Exceptions

#GP	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS	If a memory operand effective address is outside the SS segment limit. If the new value of the SP or ESP register is outside the stack segment limit.
#UD	If the LOCK prefix is used.

Virtual-8086 Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.

#UD If the 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 the stack address is in a non-canonical form.

#PF(fault-code) If a page fault occurs.

#AC(0) If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

#UD If the LOCK prefix is used.

...

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" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A* and Section 19.25.3, "Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*.

...

RCL/RCR/ROL/ROR-—Rotate

...

Operation

```
(* RCL and RCR instructions *)
SIZE ← OperandSize;
CASE (determine count) OF
    SIZE ← 8:     tempCOUNT ← (COUNT AND 1FH) MOD 9;
    SIZE ← 16:    tempCOUNT ← (COUNT AND 1FH) MOD 17;
    SIZE ← 32:    tempCOUNT ← COUNT AND 1FH;
    SIZE ← 64:    tempCOUNT ← COUNT AND 3FH;
ESAC;
```

```
(* RCL instruction operation *)
WHILE (tempCOUNT ≠ 0)
    DO
        tempCF ← MSB(DEST);
        DEST ← (DEST ∗ 2) + CF;
        CF ← tempCF;
        tempCOUNT ← tempCOUNT – 1;
    OD;
ELIHW;
IF COUNT = 1
    THEN OF ← MSB(DEST) XOR CF;
    ELSE OF is undefined;
FI;

(* RCR instruction operation *)
IF COUNT = 1
    THEN OF ← MSB(DEST) XOR CF;
    ELSE OF is undefined;
FI;
WHILE (tempCOUNT ≠ 0)
    DO
        tempCF ← LSB(SRC);
        DEST ← (DEST / 2) + (CF * $2^{SIZE}$);
        CF ← tempCF;
        tempCOUNT ← tempCOUNT – 1;
    OD;

(* ROL and ROR instructions *)
IF OperandSize = 64
    THEN COUNTMASK = 3FH;
    ELSE COUNTMASK = 1FH;
FI;

(* ROL instruction operation *)
IF (COUNT & COUNTMASK) > 0 (* Prevents updates to CF *)
    tempCOUNT ← (COUNT MOD SIZE)

    WHILE (tempCOUNT ≠ 0)
        DO
            tempCF ← MSB(DEST);
            DEST ← (DEST ∗ 2) + tempCF;
            tempCOUNT ← tempCOUNT – 1;
        OD;
    ELIHW;
    CF ← LSB(DEST);
    IF COUNT = 1
        THEN OF ← MSB(DEST) XOR CF;
        ELSE OF is undefined;
    FI;
```

```
FI;

(* ROR instruction operation *)
IF (COUNT & COUNTMASK) > 0 (* Prevents updates to CF *)
    tempCOUNT ← (COUNT MOD SIZE)

    WHILE (tempCOUNT ≠ 0)
        DO
            tempCF ← LSB(SRC);
            DEST ← (DEST / 2) + (tempCF ∗ $2^{SIZE}$);
            tempCOUNT ← tempCOUNT – 1;
        OD;
    ELIHW;
    CF ← MSB(DEST);
    IF COUNT = 1
        THEN OF ← MSB(DEST) XOR MSB − 1(DEST);
        ELSE OF is undefined;
    FI;
FI;
```

...

RDPMC—Read Performance-Monitoring Counters

...

Operation

(* Intel Core i7 processor family and Intel Xeon processor 3400, 5500 series*)

Most significant counter bit (MSCB) = 47

```
IF ((CR4.PCE = 1) or (CPL = 0) or (CR0.PE = 0))
    THEN IF (ECX[30] = 1 and ECX[29:0] in valid fixed-counter range)
        EAX ← IA32_FIXED_CTR(ECX)[30:0];
        EDX ← IA32_FIXED_CTR(ECX)[MSCB:32];
    ELSE IF (ECX[30] = 0 and ECX[29:0] in valid general-purpose counter range)
        EAX ← PMC(ECX[30:0])[31:0];
        EDX ← PMC(ECX[30:0])[MSCB:32];
    ELSE (* ECX is not valid or CR4.PCE is 0 and CPL is 1, 2, or 3 and CR0.PE is 1 *)
        #GP(0);
FI;
```

(* Intel Core 2 Duo processor family and Intel Xeon processor 3000, 5100, 5300, 7400 series*)

Most significant counter bit (MSCB) = 39

```
IF ((CR4.PCE = 1) or (CPL = 0) or (CR0.PE = 0))
    THEN IF (ECX[30] = 1 and ECX[29:0] in valid fixed-counter range)
        EAX ← IA32_FIXED_CTR(ECX)[30:0];
        EDX ← IA32_FIXED_CTR(ECX)[MSCB:32];
    ELSE IF (ECX[30] = 0 and ECX[29:0] in valid general-purpose counter range)
         EAX ← PMC(ECX[30:0])[31:0];
         EDX ← PMC(ECX[30:0])[MSCB:32];
    ELSE IF (ECX[30] = 0 and ECX[29:0] in valid special-purpose counter range)
         EAX ← PMC(ECX[30:0])[31:0]; (* 32-bit read *)
    ELSE (* ECX is not valid or CR4.PCE is 0 and CPL is 1, 2, or 3 and CR0.PE is 1 *)
         #GP(0);
FI;

(* P6 family processors and Pentium processor with MMX technology *)

IF (ECX = 0 or 1) and ((CR4.PCE = 1) or (CPL = 0) or (CR0.PE = 0))
    THEN
         EAX ← PMC(ECX)[31:0];
         EDX ← PMC(ECX)[39:32];
    ELSE (* ECX is not 0 or 1 or CR4.PCE is 0 and CPL is 1, 2, or 3 and CR0.PE is 1 *)
         #GP(0);
FI;
(* Processors with CPUID family 15 *)
IF ((CR4.PCE = 1) or (CPL = 0) or (CR0.PE = 0))
    THEN IF (ECX[30:0] = 0:17)
         THEN IF ECX[31] = 0
              THEN
                   EAX ← PMC(ECX[30:0])[31:0]; (* 40-bit read *)
                   EDX ← PMC(ECX[30:0])[39:32];
         ELSE (* ECX[31] = 1*)
              THEN
                   EAX ← PMC(ECX[30:0])[31:0]; (* 32-bit read *)
                   EDX ← 0;
         FI;
    ELSE IF (*64-bit Intel Xeon processor with L3 *)
         THEN IF (ECX[30:0] = 18:25 )
              EAX ← PMC(ECX[30:0])[31:0]; (* 32-bit read *)
              EDX ← 0;
         FI;
    ELSE IF (*Intel Xeon processor 7100 series with L3 *)
         THEN IF (ECX[30:0] = 18:25 )
              EAX ← PMC(ECX[30:0])[31:0]; (* 32-bit read *)
              EDX ← 0;
         FI;
    ELSE (* Invalid PMC index in ECX[30:0], see Table 4-15. *)
         GP(0);
    FI;
ELSE  (* CR4.PCE = 0 and (CPL = 1, 2, or 3) and CR0.PE = 1 *)
    #GP(0);
FI;

...
```

REP/REPE/REPZ/REPNE/REPNZ—Repeat String Operation Prefix

...

Description

Repeats a string instruction the number of times specified in the count register or until the indicated condition of the ZF flag is no longer met. The REP (repeat), REPE (repeat while equal), REPNE (repeat while not equal), REPZ (repeat while zero), and REPNZ (repeat while not zero) mnemonics are prefixes that can be added to one of the string instructions. The REP prefix can be added to the INS, OUTS, MOVS, LODS, and STOS instructions, and the REPE, REPNE, REPZ, and REPNZ prefixes can be added to the CMPS and SCAS instructions. (The REPZ and REPNZ prefixes are synonymous forms of the REPE and REPNE prefixes, respectively.) The behavior of the REP prefix is undefined when used with non-string instructions.

The REP prefixes apply only to one string instruction at a time. To repeat a block of instructions, use the LOOP instruction or another looping construct. All of these repeat prefixes cause the associated instruction to be repeated until the count in register is decremented to 0. See Table 4-13.

Table 4-13 Repeat Prefixes

Repeat Prefix	Termination Condition 1*	Termination Condition 2
REP	RCX or (E)CX = 0	None
REPE/REPZ	RCX or (E)CX = 0	ZF = 0
REPNE/REPNZ	RCX or (E)CX = 0	ZF = 1

NOTES:

* Count register is CX, ECX or RCX by default, depending on attributes of the operating modes.

The REPE, REPNE, REPZ, and REPNZ prefixes also check the state of the ZF flag after each iteration and terminate the repeat loop if the ZF flag is not in the specified state. When both termination conditions are tested, the cause of a repeat termination can be determined either by testing the count register with a JECXZ instruction or by testing the ZF flag (with a JZ, JNZ, or JNE instruction).

When the REPE/REPZ and REPNE/REPNZ prefixes are used, the ZF flag does not require initialization because both the CMPS and SCAS instructions affect the ZF flag according to the results of the comparisons they make.

A repeating string operation can be suspended by an exception or interrupt. When this happens, the state of the registers is preserved to allow the string operation to be resumed upon a return from the exception or interrupt handler. The source and destination registers point to the next string elements to be operated on, the EIP register points to the string instruction, and the ECX register has the value it held following the last successful iteration of the instruction. This mechanism allows long string operations to proceed without affecting the interrupt response time of the system.

When a fault occurs during the execution of a CMPS or SCAS instruction that is prefixed with REPE or REPNE, the EFLAGS value is restored to the state prior to the execution of the instruction. Since the SCAS and CMPS instructions do not use EFLAGS as an input, the processor can resume the instruction after the page fault handler.

Use the REP INS and REP OUTS instructions with caution. Not all I/O ports can handle the rate at which these instructions execute. Note that a REP STOS instruction is the fastest way to initialize a large block of memory.

In 64-bit mode, the operand size of the count register is associated with the address size attribute. Thus the default count register is RCX; REX.W has no effect on the address size and the count register. In 64-bit mode, if 67H is used to override address size attribute, the count register is ECX and any implicit source/destination operand will use the corresponding 32-bit index register. See the summary chart at the beginning of this section for encoding data and limits.

Operation

```
IF AddressSize = 16
   THEN
      Use CX for CountReg;
      Implicit Source/Dest operand for memory use of SI/DI;
   ELSE IF AddressSize = 64
      THEN Use RCX for CountReg;
      Implicit Source/Dest operand for memory use of RSI/RDI;
   ELSE
      Use ECX for CountReg;
      Implicit Source/Dest operand for memory use of ESI/EDI;
FI;
WHILE CountReg ≠ 0
    DO
        Service pending interrupts (if any);
        Execute associated string instruction;
        CountReg ← (CountReg – 1);
        IF CountReg = 0
            THEN exit WHILE loop; FI;
        IF (Repeat prefix is REPZ or REPE) and (ZF = 0)
        or (Repeat prefix is REPNZ or REPNE) and (ZF = 1)
            THEN exit WHILE loop; FI;
    OD;
```

Flags Affected

None; however, the CMPS and SCAS instructions do set the status flags in the EFLAGS register.

Exceptions (All Operating Modes)

Exceptions may be generated by an instruction associated with the prefix.

64-Bit Mode Exceptions

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

...

RET—Return from Procedure

Opcode*	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
C3	RET	A	Valid	Valid	Near return to calling procedure.
CB	RET	A	Valid	Valid	Far return to calling procedure.
C2 *iw*	RET *imm16*	B	Valid	Valid	Near return to calling procedure and pop *imm16* bytes from stack.
CA *iw*	RET *imm16*	B	Valid	Valid	Far return to calling procedure and pop *imm16* bytes from stack.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
A	NA	NA	NA	NA
B	imm16	NA	NA	NA

Description

Transfers program control to a return address located on the top of the stack. The address is usually placed on the stack by a CALL instruction, and the return is made to the instruction that follows the CALL instruction.

The optional source operand specifies the number of stack bytes to be released after the return address is popped; the default is none. This operand can be used to release parameters from the stack that were passed to the called procedure and are no longer needed. It must be used when the CALL instruction used to switch to a new procedure uses a call gate with a non-zero word count to access the new procedure. Here, the source operand for the RET instruction must specify the same number of bytes as is specified in the word count field of the call gate.

The RET instruction can be used to execute three different types of returns:

- **Near return** — A return to a calling procedure within the current code segment (the segment currently pointed to by the CS register), sometimes referred to as an intrasegment return.
- **Far return** — A return to a calling procedure located in a different segment than the current code segment, sometimes referred to as an intersegment return.
- **Inter-privilege-level far return** — A far return to a different privilege level than that of the currently executing program or procedure.

The inter-privilege-level return type can only be executed in protected mode. See the section titled “Calling Procedures Using Call and RET” in Chapter 6 of the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1*, for detailed information on near, far, and inter-privilege-level returns.

When executing a near return, the processor pops the return instruction pointer (offset) from the top of the stack into the EIP register and begins program execution at the new instruction pointer. The CS register is unchanged.

When executing a far return, the processor pops the return instruction pointer from the top of the stack into the EIP register, then pops the segment selector from the top of the stack into the CS register. The processor then begins program execution in the new code segment at the new instruction pointer.

The mechanics of an inter-privilege-level far return are similar to an intersegment return, except that the processor examines the privilege levels and access rights of the code and stack segments being returned to determine if the control transfer is allowed to be made. The DS, ES, FS, and GS segment registers are cleared by the RET instruction during an inter-privilege-level return if they refer to segments that are not allowed to be accessed at the new privilege level. Since a stack switch also occurs on an inter-privilege level return, the ESP and SS registers are loaded from the stack.

If parameters are passed to the called procedure during an inter-privilege level call, the optional source operand must be used with the RET instruction to release the parameters on the return. Here, the parameters are released both from the called procedure's stack and the calling procedure's stack (that is, the stack being returned to).

In 64-bit mode, the default operation size of this instruction is the stack-address size, i.e. 64 bits.

Operation

```
(* Near return *)
IF instruction = near return
   THEN;
       IF OperandSize = 32
           THEN
               IF top 4 bytes of stack not within stack limits
                   THEN #SS(0); FI;
               EIP ← Pop();
           ELSE
               IF OperandSize = 64
                   THEN
                       IF top 8 bytes of stack not within stack limits
                           THEN #SS(0); FI;
                       RIP ← Pop();
                   ELSE (* OperandSize = 16 *)
                       IF top 2 bytes of stack not within stack limits
                           THEN #SS(0); FI;
                       tempEIP ← Pop();
                       tempEIP ← tempEIP AND 0000FFFFH;
                       IF tempEIP not within code segment limits
                           THEN #GP(0); FI;
                       EIP ← tempEIP;
               FI;
       FI;

   IF instruction has immediate operand
       THEN (* Release parameters from stack *)
           IF StackAddressSize = 32
               THEN
                   ESP ← ESP + SRC;
               ELSE
```

```
                        IF StackAddressSize = 64
                            THEN
                                RSP ← RSP + SRC;
                            ELSE (* StackAddressSize = 16 *)
                                SP ← SP + SRC;
                        FI;
                FI;
        FI;
FI;

(* Real-address mode or virtual-8086 mode *)
IF ((PE = 0) or (PE = 1 AND VM = 1)) and instruction = far return
    THEN
        IF OperandSize = 32
            THEN
                IF top 8 bytes of stack not within stack limits
                    THEN #SS(0); FI;
                EIP ← Pop();
                CS ← Pop(); (* 32-bit pop, high-order 16 bits discarded *)
            ELSE (* OperandSize = 16 *)
                IF top 4 bytes of stack not within stack limits
                    THEN #SS(0); FI;
                tempEIP ← Pop();
                tempEIP ← tempEIP AND 0000FFFFH;
                IF tempEIP not within code segment limits
                    THEN #GP(0); FI;
                EIP ← tempEIP;
                CS ← Pop(); (* 16-bit pop *)
        FI;
    IF instruction has immediate operand
        THEN (* Release parameters from stack *)
            SP ← SP + (SRC AND FFFFH);
    FI;
FI;

(* Protected mode, not virtual-8086 mode *)
IF (PE = 1 and VM = 0 and IA32_EFER.LMA = 0) and instruction = far return
    THEN
        IF OperandSize = 32
            THEN
                IF second doubleword on stack is not within stack limits
                    THEN #SS(0); FI;
            ELSE (* OperandSize = 16 *)
                IF second word on stack is not within stack limits
                    THEN #SS(0); FI;
        FI;
    IF return code segment selector is NULL
        THEN #GP(0); FI;
    IF return code segment selector addresses descriptor beyond descriptor table limit
        THEN #GP(selector); FI;
    Obtain descriptor to which return code segment selector points from descriptor table;
    IF return code segment descriptor is not a code segment
        THEN #GP(selector); FI;
    IF return code segment selector RPL < CPL
        THEN #GP(selector); FI;
    IF return code segment descriptor is conforming
    and return code segment DPL > return code segment selector RPL
        THEN #GP(selector); FI;
    IF return code segment descriptor is non-conforming and return code
    segment DPL ≠ return code segment selector RPL
        THEN #GP(selector); FI;
    IF return code segment descriptor is not present
        THEN #NP(selector); FI:
    IF return code segment selector RPL > CPL
        THEN GOTO RETURN-OUTER-PRIVILEGE-LEVEL;
        ELSE GOTO RETURN-TO-SAME-PRIVILEGE-LEVEL;
    FI;
FI;

RETURN-SAME-PRIVILEGE-LEVEL:
    IF the return instruction pointer is not within the return code segment limit
        THEN #GP(0); FI;
    IF OperandSize = 32
        THEN
            EIP ← Pop();
            CS ← Pop(); (* 32-bit pop, high-order 16 bits discarded *)
        ELSE (* OperandSize = 16 *)
            EIP ← Pop();
            EIP ← EIP AND 0000FFFFH;
            CS ← Pop(); (* 16-bit pop *)
    FI;
    IF instruction has immediate operand
        THEN (* Release parameters from stack *)
            IF StackAddressSize = 32
                THEN
                    ESP ← ESP + SRC;
                ELSE (* StackAddressSize = 16 *)
                    SP ← SP + SRC;
            FI;
    FI;

RETURN-OUTER-PRIVILEGE-LEVEL:
    IF top (16 + SRC) bytes of stack are not within stack limits (OperandSize = 32)
    or top (8 + SRC) bytes of stack are not within stack limits (OperandSize = 16)
            THEN #SS(0); FI;
    Read return segment selector;
    IF stack segment selector is NULL
        THEN #GP(0); FI;
    IF return stack segment selector index is not within its descriptor table limits
        THEN #GP(selector); FI;
```

```
Read segment descriptor pointed to by return segment selector;
IF stack segment selector RPL ≠ RPL of the return code segment selector
or stack segment is not a writable data segment
or stack segment descriptor DPL ≠ RPL of the return code segment selector
        THEN #GP(selector); FI;
IF stack segment not present
    THEN #SS(StackSegmentSelector); FI;
IF the return instruction pointer is not within the return code segment limit
    THEN #GP(0); FI;
CPL ← ReturnCodeSegmentSelector(RPL);
IF OperandSize = 32
    THEN
        EIP ← Pop();
        CS ← Pop(); (* 32-bit pop, high-order 16 bits discarded; segment descriptor loaded *)
        CS(RPL) ← CPL;
        IF instruction has immediate operand
            THEN (* Release parameters from called procedure’s stack *)
                IF StackAddressSize = 32
                    THEN
                        ESP ← ESP + SRC;
                    ELSE (* StackAddressSize = 16 *)
                        SP ← SP + SRC;
                FI;
        FI;
        tempESP ← Pop();
        tempSS ← Pop(); (* 32-bit pop, high-order 16 bits discarded; seg. descriptor loaded *)
        ESP ← tempESP;
        SS ← tempSS;
    ELSE (* OperandSize = 16 *)
        EIP ← Pop();
        EIP ← EIP AND 0000FFFFH;
        CS ← Pop(); (* 16-bit pop; segment descriptor loaded *)
        CS(RPL) ← CPL;
        IF instruction has immediate operand
            THEN (* Release parameters from called procedure’s stack *)
                IF StackAddressSize = 32
                    THEN
                        ESP ← ESP + SRC;
                    ELSE (* StackAddressSize = 16 *)
                        SP ← SP + SRC;
                FI;
        FI;
        tempESP ← Pop();
        tempSS ← Pop(); (* 16-bit pop; segment descriptor loaded *)
        ESP ← tempESP;
        SS ← tempSS;
FI;

FOR each of segment register (ES, FS, GS, and DS)
    DO
```

```
                IF segment register points to data or non-conforming code segment
                and CPL > segment descriptor DPL (* DPL in hidden part of segment register *)
                    THEN SegmentSelector ← 0; (* Segment selector invalid *)
                FI;
            OD;

        IF instruction has immediate operand
            THEN (* Release parameters from calling procedure's stack *)
                IF StackAddressSize = 32
                    THEN
                        ESP ← ESP + SRC;
                    ELSE (* StackAddressSize = 16 *)
                        SP ← SP + SRC;
                FI;
        FI;

    (* IA-32e Mode *)
        IF (PE = 1 and VM = 0 and IA32_EFER.LMA = 1) and instruction = far return
            THEN
                IF OperandSize = 32
                    THEN
                        IF second doubleword on stack is not within stack limits
                            THEN #SS(0); FI;
                        IF first or second doubleword on stack is not in canonical space
                            THEN #SS(0); FI;
                    ELSE
                        IF OperandSize = 16
                            THEN
                                IF second word on stack is not within stack limits
                                    THEN #SS(0); FI;
                                IF first or second word on stack is not in canonical space
                                    THEN #SS(0); FI;
                            ELSE (* OperandSize = 64 *)
                                IF first or second quadword on stack is not in canonical space
                                    THEN #SS(0); FI;
                        FI
                FI;
            IF return code segment selector is NULL
                THEN GP(0); FI;
            IF return code segment selector addresses descriptor beyond descriptor table limit
                THEN GP(selector); FI;
            IF return code segment selector addresses descriptor in non-canonical space
                THEN GP(selector); FI;
            Obtain descriptor to which return code segment selector points from descriptor table;
            IF return code segment descriptor is not a code segment
                THEN #GP(selector); FI;
            IF return code segment descriptor has L-bit = 1 and D-bit = 1
                THEN #GP(selector); FI;
            IF return code segment selector RPL < CPL
                THEN #GP(selector); FI;
            IF return code segment descriptor is conforming
            and return code segment DPL > return code segment selector RPL
                THEN #GP(selector); FI;
            IF return code segment descriptor is non-conforming
            and return code segment DPL ≠ return code segment selector RPL
                THEN #GP(selector); FI;
            IF return code segment descriptor is not present
                THEN #NP(selector); FI:
            IF return code segment selector RPL > CPL
                THEN GOTO IA-32E-MODE-RETURN-OUTER-PRIVILEGE-LEVEL;
                ELSE GOTO IA-32E-MODE-RETURN-SAME-PRIVILEGE-LEVEL;
            FI;
        FI;

    IA-32E-MODE-RETURN-SAME-PRIVILEGE-LEVEL:
    IF the return instruction pointer is not within the return code segment limit
        THEN #GP(0); FI;
    IF the return instruction pointer is not within canonical address space
        THEN #GP(0); FI;
    IF OperandSize = 32
        THEN
            EIP ← Pop();
            CS ← Pop(); (* 32-bit pop, high-order 16 bits discarded *)
        ELSE
            IF OperandSize = 16
                THEN
                    EIP ← Pop();
                    EIP ← EIP AND 0000FFFFH;
                    CS ← Pop(); (* 16-bit pop *)
                ELSE (* OperandSize = 64 *)
                    RIP ← Pop();
                    CS ← Pop(); (* 64-bit pop, high-order 48 bits discarded *)
            FI;
    FI;
    IF instruction has immediate operand
        THEN (* Release parameters from stack *)
            IF StackAddressSize = 32
                THEN
                    ESP ← ESP + SRC;
                ELSE
                    IF StackAddressSize = 16
                        THEN
                            SP ← SP + SRC;
                        ELSE (* StackAddressSize = 64 *)
                            RSP ← RSP + SRC;
                    FI;
            FI;
    FI;

    IA-32E-MODE-RETURN-OUTER-PRIVILEGE-LEVEL:
```

```
IF top (16 + SRC) bytes of stack are not within stack limits (OperandSize = 32)
or top (8 + SRC) bytes of stack are not within stack limits (OperandSize = 16)
    THEN #SS(0); FI;
IF top (16 + SRC) bytes of stack are not in canonical address space (OperandSize = 32)
or top (8 + SRC) bytes of stack are not in canonical address space (OperandSize = 16)
or top (32 + SRC) bytes of stack are not in canonical address space (OperandSize = 64)
    THEN #SS(0); FI;
Read return stack segment selector;
IF stack segment selector is NULL
    THEN
        IF new CS descriptor L-bit = 0
            THEN #GP(selector);
        IF stack segment selector RPL = 3
            THEN #GP(selector);
FI;
IF return stack segment descriptor is not within descriptor table limits
        THEN #GP(selector); FI;
IF return stack segment descriptor is in non-canonical address space
        THEN #GP(selector); FI;
Read segment descriptor pointed to by return segment selector;
IF stack segment selector RPL ≠ RPL of the return code segment selector
or stack segment is not a writable data segment
or stack segment descriptor DPL ≠ RPL of the return code segment selector
    THEN #GP(selector); FI;
IF stack segment not present
    THEN #SS(StackSegmentSelector); FI;
IF the return instruction pointer is not within the return code segment limit
    THEN #GP(0); FI:
IF the return instruction pointer is not within canonical address space
    THEN #GP(0); FI;
CPL ← ReturnCodeSegmentSelector(RPL);
IF OperandSize = 32
    THEN
        EIP ← Pop();
        CS ← Pop(); (* 32-bit pop, high-order 16 bits discarded, segment descriptor loaded *)
        CS(RPL) ← CPL;
        IF instruction has immediate operand
            THEN (* Release parameters from called procedure's stack *)
                IF StackAddressSize = 32
                    THEN
                        ESP ← ESP + SRC;
                    ELSE
                        IF StackAddressSize = 16
                            THEN
                                SP ← SP + SRC;
                            ELSE (* StackAddressSize = 64 *)
                                RSP ← RSP + SRC;
                        FI;
                FI;
        FI;
        tempESP ← Pop();
        tempSS ← Pop(); (* 32-bit pop, high-order 16 bits discarded, segment descriptor loaded *)
        ESP ← tempESP;
        SS ← tempSS;
    ELSE
        IF OperandSize = 16
            THEN
                EIP ← Pop();
                EIP ← EIP AND 0000FFFFH;
                CS ← Pop(); (* 16-bit pop; segment descriptor loaded *)
                CS(RPL) ← CPL;
                IF instruction has immediate operand
                    THEN (* Release parameters from called procedure's stack *)
                        IF StackAddressSize = 32
                            THEN
                                ESP ← ESP + SRC;
                            ELSE
                                IF StackAddressSize = 16
                                    THEN
                                        SP ← SP + SRC;
                                    ELSE (* StackAddressSize = 64 *)
                                        RSP ← RSP + SRC;
                                FI;
                        FI;
                FI;
                tempESP ← Pop();
                tempSS ← Pop(); (* 16-bit pop; segment descriptor loaded *)
                ESP ← tempESP;
                SS ← tempSS;
            ELSE (* OperandSize = 64 *)
                RIP ← Pop();
                CS ← Pop(); (* 64-bit pop; high-order 48 bits discarded; seg. descriptor loaded *)
                CS(RPL) ← CPL;
                IF instruction has immediate operand
                    THEN (* Release parameters from called procedure's stack *)
                        RSP ← RSP + SRC;
                FI;
                tempESP ← Pop();
                tempSS ← Pop(); (* 64-bit pop; high-order 48 bits discarded; seg. desc. loaded *)
                ESP ← tempESP;
                SS ← tempSS;
        FI;
FI;

FOR each of segment register (ES, FS, GS, and DS)
    DO
        IF segment register points to data or non-conforming code segment
        and CPL > segment descriptor DPL; (* DPL in hidden part of segment register *)
            THEN SegmentSelector ← 0; (* SegmentSelector invalid *)
        FI;
    OD;

IF instruction has immediate operand
    THEN (* Release parameters from calling procedure's stack *)
        IF StackAddressSize = 32
            THEN
                ESP ← ESP + SRC;
            ELSE
                IF StackAddressSize = 16
                    THEN
                        SP ← SP + SRC;
                    ELSE (* StackAddressSize = 64 *)
                        RSP ← RSP + SRC;
                FI;
        FI;
FI;
```

...

STMXCSR—Store MXCSR Register State

Opcode*/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
0F AE /3 STMXCSR *m32*	A	V/V	SSE	Store contents of MXCSR register to *m32*.
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.

Note: In VEX-encoded versions, VEX.vvvv is reserved and must be 1111b, 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 5; additionally

#UD	If VEX.L= 1, If VEX.vvvv != 1111B.

...

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.

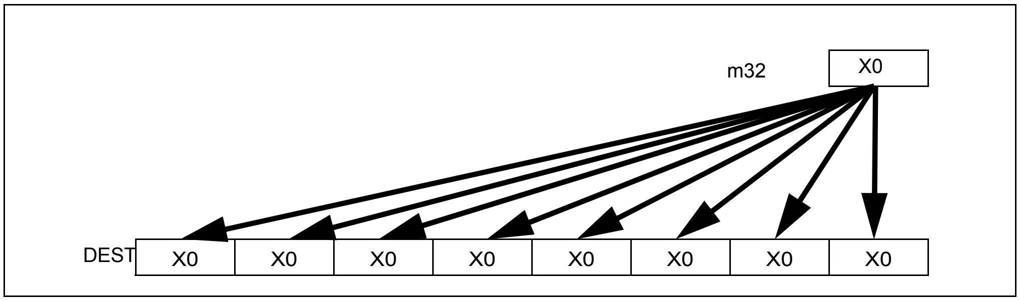

Figure 4-21 VBROADCASTSS Operation (VEX.256 encoded version)

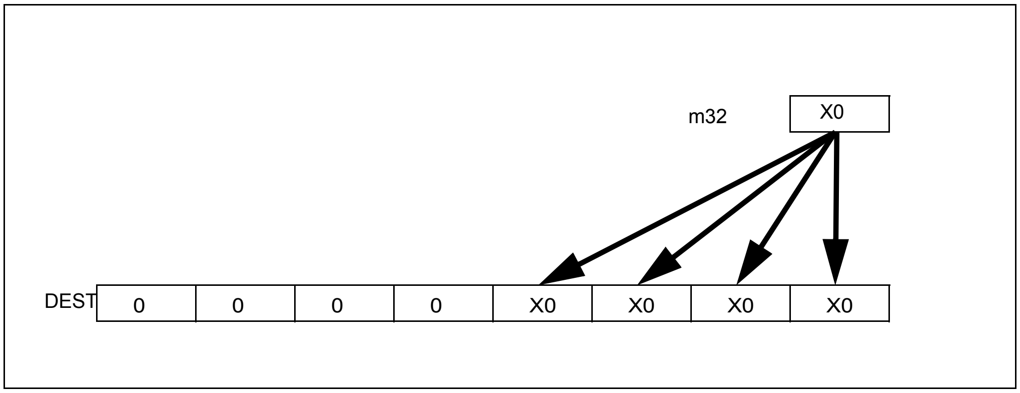

Figure 4-22 VBROADCASTSS Operation (128-bit version)

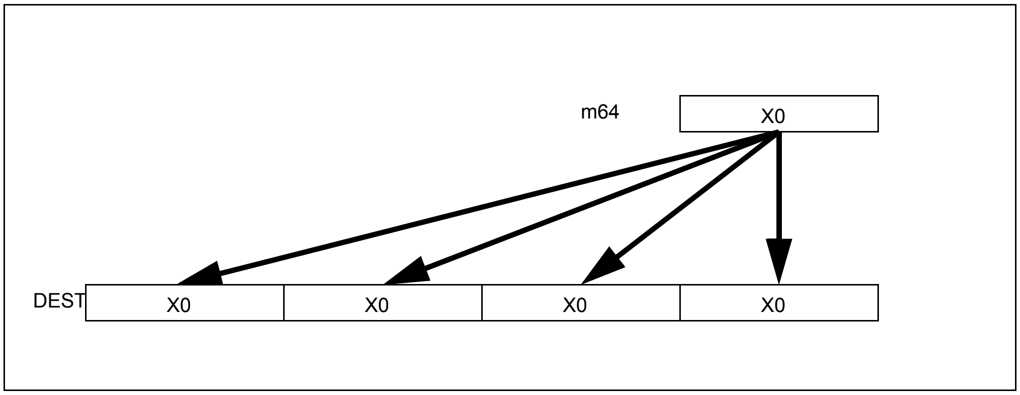

Figure 4-23 VBROADCASTSD Operation

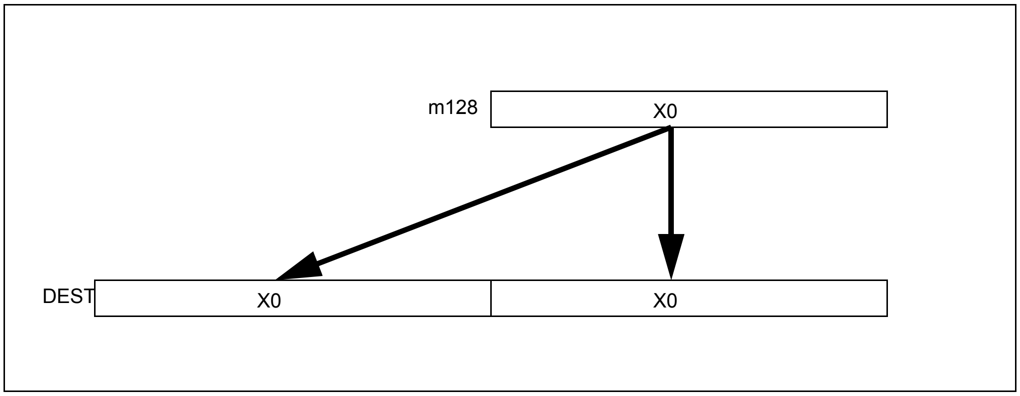

Figure 4-24 VBROADCASTF128 Operation

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.

...

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.

...

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.

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.

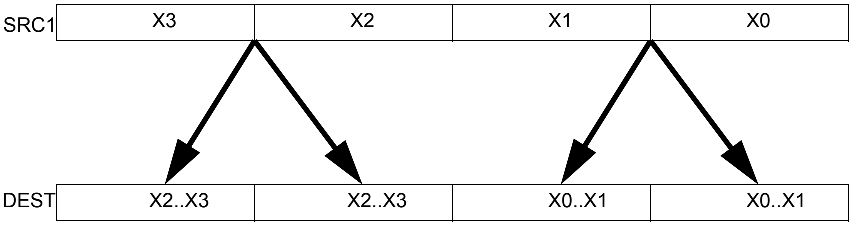

Figure 4-25 VPERMILPD operation

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 4-26) 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.

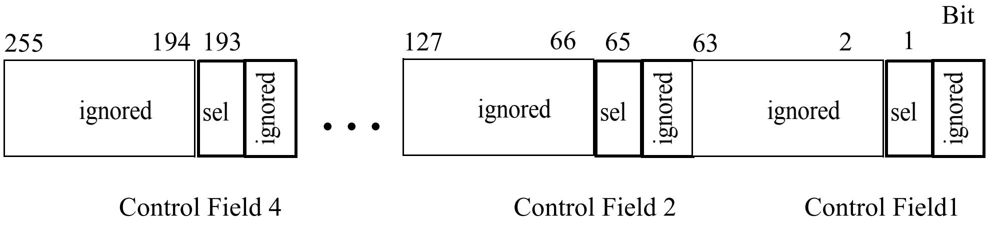

Figure 4-26 VPERMILPD Shuffle Control

(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.

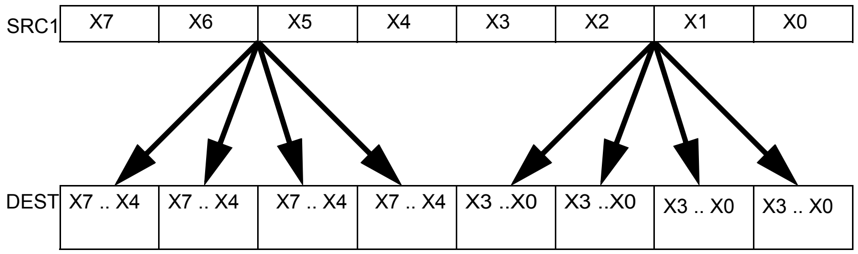

Figure 4-27 VPERMILPS Operation

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 4-28) 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.

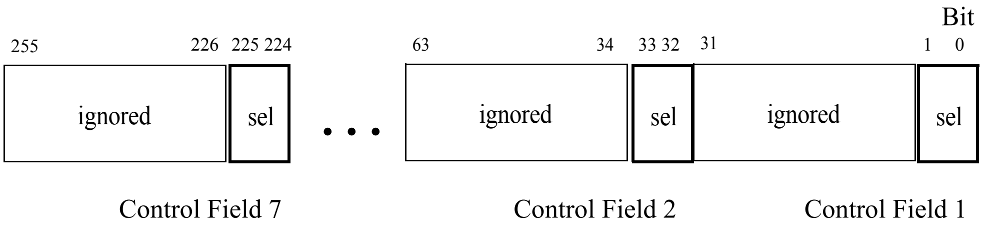

Figure 4-28 VPERMILPS Shuffle Control

(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.

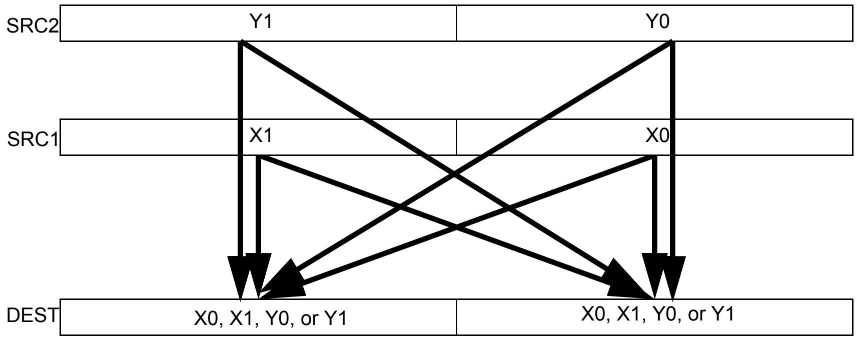

Figure 4-29 VPERM2F128 Operation

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.

...

VTESTPD/VTESTPS—Packed Bit Test

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
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

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

VTESTPS (128-bit version)

```
TEMP[127:0] ← SRC[127:0] AND DEST[127:0]
IF (TEMP[31] = TEMP[63] = TEMP[95] = TEMP[127] = 0)
    THEN ZF ←1;
    ELSE ZF ← 0;

TEMP[127:0] ← SRC[127:0] AND NOT DEST[127:0]
IF (TEMP[31] = TEMP[63] = TEMP[95] = TEMP[127] = 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 (128-bit version)

```
TEMP[127:0] ← SRC[127:0] AND DEST[127:0]
```

```
IF ( TEMP[63] = TEMP[127] = 0)
    THEN ZF ←1;
    ELSE ZF ← 0;

TEMP[127:0] ← SRC[127:0] AND NOT DEST[127:0]
IF ( TEMP[63] = TEMP[127] = 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

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.

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.

...

XRSTOR—Restore Processor Extended States

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
0F AE /5	XRSTOR *mem*	A	Valid	Valid	Restore processor extended states from *memory*. The states are specified by EDX:EAX
REX.W+ 0F AE / 5	XRSTOR64 *mem*	A	Valid	N.E.	Restore processor extended states from *memory*. The states are specified by EDX:EAX

Instruction Operand Encoding

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

Description

Performs a full or partial restore of the enabled processor states using the state information stored in the memory address specified by the source operand. The implicit EDX:EAX register pair specifies a 64-bit restore mask.

The format of the XSAVE/XRSTOR area is shown in Table 4-18. The memory layout of the XSAVE/XRSTOR area may have holes between save areas written by the processor as a result of the processor not supporting certain processor extended states or system software not supporting certain processor extended states. There is no relationship between the order of XCR0 bits and the order of the state layout. States corresponding to higher and lower XCR0 bits may be intermingled in the layout.

Table 4-18 General Layout of XSAVE/XRSTOR Save Area

Save Areas	Offset (Byte)	Size (Bytes)
FPU/SSE SaveArea[1]	0	512
Header	512	64
Reserved (Ext_Save_Area_2)	CPUID.(EAX=0DH, ECX=2):EBX	CPUID.(EAX=0DH, ECX=2):EAX
Reserved(Ext_Save_Area_4)[2]	CPUID.(EAX=0DH, ECX=4):EBX	CPUID.(EAX=0DH, ECX=4):EAX
Reserved(Ext_Save_Area_3)	CPUID.(EAX=0DH, ECX=3):EBX	CPUID.(EAX=0DH, ECX=3):EAX
Reserved(...)	...	...

NOTES:

1. Bytes 464:511 are available for software use. XRSTOR ignores the value contained in bytes 464:511 of an XSAVE SAVE image.
2. State corresponding to higher and lower XCR0 bits may be intermingled in layout.

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 4-18) 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 4-20) 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.

The format of the header section (XSAVE.HEADER) of the XSAVE/XRSTOR area is shown in Table 4-19.

Table 4-19 XSAVE.HEADER Layout

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

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. While it's legal to set any bit in the EDX:EAX mask to 1, it is strongly recommended to set only the bits that are required to save/restore specific states.

...

XSAVE—Save Processor Extended States

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
0F AE /4	XSAVE *mem*	A	Valid	Valid	Save processor extended states to *memory*. The states are specified by EDX:EAX
REX.W+ 0F AE / 4	XSAVE64 *mem*	A	Valid	N.E.	Save processor extended states to *memory*. The states are specified by EDX:EAX

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 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 4-18. 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. While it's legal to set any bit in the EDX:EAX mask to 1, it is strongly recommended to set only the bits that are required to save/restore specific states. 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.

...

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

XSAVEOPT 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 4-18.

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. While it's legal to set any bit in the EDX:EAX mask to 1, it is strongly recommended to set only the bits that are required to save/restore specific states. 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 4-18. 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.

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.

...

WRMSR—Write to Model Specific Register

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
0F 30	WRMSR	A	Valid	Valid	Write the value in EDX:EAX to MSR specified by ECX.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
A	NA	NA	NA	NA

Description

Writes the contents of registers EDX:EAX into the 64-bit model specific register (MSR) 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 MSR and the contents of the EAX register are copied to low-order 32 bits of the MSR. (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 MSR 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 MSR address in ECX will also cause a general protection exception. The processor will also generate a general protection exception if software attempts to write to bits in a reserved MSR.

When the WRMSR instruction is used to write to an MTRR, the TLBs are invalidated. This includes global entries (see "Translation Lookaside Buffers (TLBs)" in Chapter 3 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*).

MSRs control functions for testability, execution tracing, performance-monitoring and machine check errors. Appendix B, "Model-Specific Registers (MSRs)", in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3B*, lists all MSRs that can be read with this instruction and their addresses. Note that each processor family has its own set of MSRs.

The WRMSR instruction is a serializing instruction (see "Serializing Instructions" in Chapter 8 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*). Note that WRMSR to the IA32_TSC_DEADLINE MSR (MSR index 6E0H) and the X2APIC MSRs (MSR indices 802H to 83FH) are not serializing.

The CPUID instruction should be used to determine whether MSRs are supported (CPUID.01H:EDX[5] = 1) before using this instruction.

...

10. 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.

...

INVVPID— Invalidate Translations Based on VPID

Opcode	Instruction	Description
66 0F 38 81	INVVPID r64, m128	Invalidates entries in the TLBs and paging-structure caches based on VPID (in 64-bit mode)
66 0F 38 81	INVVPID r32, m128	Invalidates entries in the TLBs and paging-structure caches based on VPID (outside 64-bit mode)

Description

Invalidates mappings in the translation lookaside buffers (TLBs) and paging-structure caches based on **virtual-processor identifier** (VPID). (See Chapter 25, "Support for Address Translation" in *IA-32 Intel Architecture Software Developer's Manual, Volume 3B*.) Invalidation is based on the **INVVPID type** specified in the register operand and the **INVVPID descriptor** specified in the memory operand.

Outside IA-32e mode, the register operand is always 32 bits, regardless of the value of CS.D. In 64-bit mode, the register operand has 64 bits; however, if bits 63:32 of the register operand are not zero, INVVPID fails due to an attempt to use an unsupported INVVPID type (see below).

The INVVPID types supported by a logical processors are reported in the IA32_VMX_EPT_VPID_CAP MSR (see Appendix "VMX Capability Reporting Facility" in *IA-32 Intel Architecture Software Developer's Manual, Volume 3B*). There are four INVVPID types currently defined:

- Individual-address invalidation: If the INVVPID type is 0, the logical processor invalidates mappings for a single linear address and tagged with the VPID specified in the INVVPID descriptor. In some cases, it may invalidate mappings for other linear addresses (or with other VPIDs) as well.
- Single-context invalidation: If the INVVPID type is 1, the logical processor invalidates all mappings tagged with the VPID specified in the INVVPID descriptor. In some cases, it may invalidate mappings for other VPIDs as well.
- All-contexts invalidation: If the INVVPID type is 2, the logical processor invalidates all mappings tagged with all VPIDs except VPID 0000H. In some cases, it may invalidate translations with VPID 0000H as well.
- Single-context invalidation, retaining global translations: If the INVVPID type is 3, the logical processor invalidates all mappings tagged with the VPID specified in the INVVPID descriptor except global translations. In some cases, it may invalidate global translations (and mappings with other VPIDs) as well. See the "Caching Translation Information" section in Chapter 4 of the *IA-32 Intel Architecture Software Developer's Manual, Volumes 3A* for information about global translations.

If an unsupported INVVPID type is specified, the instruction fails.

INVVPID invalidates all the specified mappings for the indicated VPID(s) regardless of the EPTP and PCID values with which those mappings may be associated.

The INVVPID descriptor comprises 128 bits and consists of a VPID and a linear address as shown in Figure5-2.

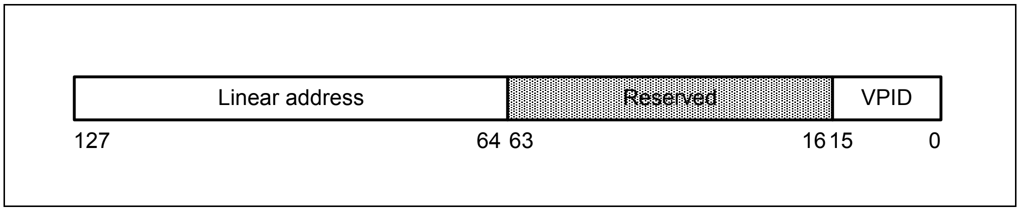

Figure5-2 INVVPID Descriptor

Operation

```
IF (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
        INVVPID_TYPE ← value of register operand;
        IF IA32_VMX_EPT_VPID_CAP MSR indicates that processor does not support
        INVVPID_TYPE
            THEN VMfail(Invalid operand to INVEPT/INVVPID);
            ELSE            // INVVPID_TYPE must be in the range 0–3
                INVVPID_DESC ← value of memory operand;
                IF INVVPID_DESC[63:16] ≠ 0
                    THEN VMfail(Invalid operand to INVEPT/INVVPID);
                    ELSE
                        CASE INVVPID_TYPE OF
                            0:                  // individual-address invalidation
                                VPID ← INVVPID_DESC[15:0];
                                IF VPID = 0
                                    THEN VMfail(Invalid operand to INVEPT/INVVPID);
                                    ELSE
                                        GL_ADDR ← INVVPID_DESC[127:64];
                                        IF (GL_ADDR is not in a canonical form)
                                            THEN
                                                VMfail(Invalid operand to INVEPT/INVVPID);
                                            ELSE
                                                Invalidate mappings for GL_ADDR tagged with VPID;
                                                VMsucceed;
                                        FI;
                                FI;
                                BREAK;
                            1:                  // single-context invalidation
                                VPID ← INVVPID_DESC[15:0];
                                IF VPID = 0
                                    THEN VMfail(Invalid operand to INVEPT/INVVPID);
                                    ELSE
                                        Invalidate all mappings tagged with VPID;
                                        VMsucceed;
                                FI;
                                BREAK;
                        2:                  // all-context invalidation
                                Invalidate all mappings tagged with all non-zero VPIDs;
                                VMsucceed;
                                BREAK;
                        3:                  // single-context invalidation retaining globals
                                VPID ← INVVPID_DESC[15:0];
                                IF VPID = 0
                                        THEN VMfail(Invalid operand to INVEPT/INVVPID);
                                        ELSE
                                                Invalidate all mappings tagged with VPID except global
translations;
                                                VMsucceed;
                                FI;
                                BREAK;
                ESAC;
        FI;
    FI;
FI;
```

Flags Affected

See the operation section and Section 5.2.

Protected Mode Exceptions

#GP(0)	If the current privilege level is not 0.
	If the memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
	If the DS, ES, FS, or GS register contains an unusable segment.
	If the source operand is located in an execute-only code segment.
#PF(fault-code)	If a page fault occurs in accessing the memory operand.
#SS(0)	If the memory operand effective address is outside the SS segment limit.
	If the SS register contains an unusable segment.
#UD	If not in VMX operation.
	If the logical processor does not support VPIDs (IA32_VMX_PROCBASED_CTLS2[37]=0).
	If the logical processor supports VPIDs (IA32_VMX_PROCBASED_CTLS2[37]=1) but does not support the INVVPID instruction (IA32_VMX_EPT_VPID_CAP[32]=0).

Real-Address Mode Exceptions

#UD	A logical processor cannot be in real-address mode while in VMX operation and the INVVPID instruction is not recognized outside VMX operation.

Virtual-8086 Mode Exceptions

#UD	The INVVPID instruction is not recognized in virtual-8086 mode.

Compatibility Mode Exceptions

#UD	The INVVPID instruction is not recognized in compatibility mode.

64-Bit Mode Exceptions

#GP(0)	If the current privilege level is not 0. If the memory operand is in the CS, DS, ES, FS, or GS segments and the memory address is in a non-canonical form.
#PF(fault-code)	If a page fault occurs in accessing the memory operand.
#SS(0)	If the memory destination operand is in the SS segment and the memory address is in a non-canonical form.
#UD	If not in VMX operation. If the logical processor does not support VPIDs (IA32_VMX_PROCBASED_CTLS2[37]=0). If the logical processor supports VPIDs (IA32_VMX_PROCBASED_CTLS2[37]=1) but does not support the INVVPID instruction (IA32_VMX_EPT_VPID_CAP[32]=0).

...

VMCALL—Call to VM Monitor

Opcode	Instruction	Description
0F 01 C1	VMCALL	Call to VM monitor by causing VM exit.

Description

This instruction allows guest software can make a call for service into an underlying VM monitor. The details of the programming interface for such calls are VMM-specific; this instruction does nothing more than cause a VM exit, registering the appropriate exit reason.

Use of this instruction in VMX root operation invokes an SMM monitor (see Section 26.15.2 in *IA-32 Intel Architecture Software Developer's Manual, Volume 3B*). This invocation will activate the dual-monitor treatment of system-management interrupts (SMIs) and system-management mode (SMM) if it is not already active (see Section 26.15.6 in *IA-32 Intel Architecture Software Developer's Manual, Volume 3B*).

Operation

```
IF not in VMX operation
    THEN #UD;
ELSIF in VMX non-root operation
    THEN VM exit;
ELSIF (RFLAGS.VM = 1) or (IA32_EFER.LMA = 1 and CS.L = 0)
    THEN #UD;
ELSIF CPL > 0
    THEN #GP(0);
```

```
ELSIF in SMM or the logical processor does not support the dual-monitor treatment of SMIs and SMM or
the valid bit in the IA32_SMM_MONITOR_CTL MSR is clear
    THEN VMfail (VMCALL executed in VMX root operation);
ELSIF dual-monitor treatment of SMIs and SMM is active
    THEN perform an SMM VM exit (see Section 26.15.2
     of the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3B);
ELSIF current-VMCS pointer is not valid
    THEN VMfailInvalid;
ELSIF launch state of current VMCS is not clear
    THEN VMfailValid(VMCALL with non-clear VMCS);
ELSIF VM-exit control fields are not valid (see Section 26.15.6.1 of the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3B)
    THEN VMfailValid (VMCALL with invalid VM-exit control fields);
ELSE
    enter SMM;
    read revision identifier in MSEG;
    IF revision identifier does not match that supported by processor
         THEN
              leave SMM;
              VMfailValid(VMCALL with incorrect MSEG revision identifier);
         ELSE
              read SMM-monitor features field in MSEG (see Section 26.15.6.2,
              in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3B);
              IF features field is invalid
                   THEN
                        leave SMM;
                        VMfailValid(VMCALL with invalid SMM-monitor features);
                   ELSE activate dual-monitor treatment of SMIs and SMM (see Section 26.15.6
                   in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3B);
              FI;
         FI;
FI;
```

Flags Affected

See the operation section and Section 5.2.

Protected Mode Exceptions

#GP(0)	If the current privilege level is not 0 and the logical processor is in VMX root operation.
#UD	If executed outside VMX operation.

Real-Address Mode Exceptions

#UD	If executed outside VMX operation.

Virtual-8086 Mode Exceptions

#UD	If executed outside VMX non-root operation.

Compatibility Mode Exceptions

#UD	If executed outside VMX non-root operation.

64-Bit Mode Exceptions

#UD	If executed outside VMX non-root operation.

...

VMXON—Enter VMX Operation

Opcode	Instruction	Description
F3 0F C7 /6	VMXON m64	Enter VMX root operation.

Description

Puts the logical processor in VMX operation with no current VMCS, blocks INIT signals, disables A20M, and clears any address-range monitoring established by the MONITOR instruction.[1]

The operand of this instruction is a 4KB-aligned physical address (the VMXON pointer) that references the VMXON region, which the logical processor may use to support VMX operation. This operand is always 64 bits and is always in memory.

Operation

```
IF (register operand) or (CR0.PE = 0) or (CR4.VMXE = 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[2]) or
        (bit 0 (lock bit) of IA32_FEATURE_CONTROL MSR is clear) or
        (in SMX operation[3] 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
                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[4]
```

[1] See the information on MONITOR/MWAIT in Chapter 8, "Multiple-Processor Management," of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*.

[2] See Section 19.8 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3B*.

[3] 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."

[4] 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.

```
                    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;
```

Flags Affected

See the operation section and Section 5.2.

Protected Mode Exceptions

#GP(0) If executed outside VMX operation with CPL>0 or with invalid CR0 or CR4 fixed bits.

If executed in A20M mode.

If the memory source operand effective address is outside the CS, DS, ES, FS, or GS segment limit.

If the DS, ES, FS, or GS register contains an unusable segment.

If the source operand is located in an execute-only code segment.

#PF(fault-code) If a page fault occurs in accessing the memory source operand.

#SS(0) If the memory source operand effective address is outside the SS segment limit.

If the SS register contains an unusable segment.

#UD If operand is a register.

If executed with CR4.VMXE = 0.

Real-Address Mode Exceptions

#UD The VMXON instruction is not recognized in real-address mode.

Virtual-8086 Mode Exceptions

#UD The VMXON instruction is not recognized in virtual-8086 mode.

...

XSETBV—Set Extended Control Register

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
0F 01 D1	XSETBV	A	Valid	Valid	Write the value in EDX:EAX to the XCR specified by ECX.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
A	NA	NA	NA	NA

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 (corresponding to x87 state) must be set to 1; the instruction will cause a #GP(0) if an attempt is made to clear this bit. Additionally, bit 1 of XCR0 (corresponding to AVX state) and bit 2 of XCR0 (corresponding to SSE state) must be set to 1 when using AVX registers; the instruction will cause a #GP(0) if an attempt is made to set XCR0[2:1] = 10.

Operation

```
XCR[ECX] ← EDX:EAX;
```

Flags Affected

None.

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.
	If an attempt is made to set XCR0[2:1] = 10.
#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.

If an attempt is made to set XCR0[2:1] = 10.

#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.

Virtual-8086 Mode Exceptions

#GP(0) The XSETBV instruction is not recognized in virtual-8086 mode.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

Same exceptions as in protected mode.

...

11. Updates to Appendix A, Volume 2B

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

...

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^{o64}	CLTS	SYSRET^{o64}
1		vmovups Vps, Wps	vmovups Wps, Vps	vmovlps Vq, Hq, Mq vmovhlps Vq, Hq, Uq	vmovlps Mq, Vq	vunpcklps Vps, Wq Vx, Hx, Wx	vunpckhps Vps, Wq Vx, Hx, Wx	vmovhps^{v1} Vdq, Hq, Mq vmovlhps Vdq, Hq, Uq	vmovhps^{v1} Mq, Vq
	66	vmovupd Vpd, Wpd	vmovupd Wpd,Vpd	vmovlpd Vq, Hq, Mq	vmovlpd Mq, Vq	vunpcklpd Vpd,WqVx,Hx,Wx	vunpckhpd Vpd,WqVx,Hx,Wx	vmovhpd^{v1} Vdq, Hq, Mq	vmovhpd^{v1} Mq, Vq
	F3	vmovss Vss, Wss Mss, Hss, Vss	vmovss Wss, Vss Mss, Hss, Vss	vmovsldup Vq, Wq Vx, Wx				vmovshdup Vq, Wq Vx, Wx	
	F2	vmovsd Vsd, Wsd Vsd, Hsd, Msd	vmovsd Vsd, Wsd Vsd, Hsd, Msd	vmovddup Vq, Wq 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, Ups	vsqrtps Vps, Wps	vrsqrtps Vps, Wps	vrcpps Vps, Wps	vandps Vps, Hps, Wps	vandnps Vps, Hps, Wps	vorps Vps, Hps, Wps	vxorps Vps, Hps, Wps
	66	vmovmskpd Gy,Upd	vsqrtpd Vpd,Wpd			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^{v} vzeroall(L)^{v}
	66	vpshufd Vdq,Wdq,Ib				vpcmpeqb Vdq, Hdq, Wdq	vpcmpeqw Vdq, Hdq, Wdq	vpcmpeqd Vdq, Hdq, Wdq	
	F3	vpshufhw Vdq,Wdq,Ib							
	F2	vpshuflw Vdq,Wdq,Ib							

...

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	pblendvb Vdq, Wdq				blendvps Vdq, Wdq Vx, Wx	blendvpd Vdq, Wdq Vx, Wx		vptest Vdq, Wdq 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-5 Three-byte Opcode Map: 00H — F7H (First two bytes are 0F 3AH) *

	pfx	0	1	2	3	4	5	6	7
0	66					vpermilps^{V} Vx, Wx, Ib	vpermilpd^{V} Vx, Wx, Ib	vperm2f128^{V} Vqq,Hqq,Wqq,Ib	
1	66					vpextrb Rd/Mb, Vdq, Ib	vpextrw Rd/Mw, Vdq, Ib	vpextrd/q Ey, Vdq, Ib	vextractps Ed, Vdq, Ib
2	66	vpinsrb Vdq,Hdq,Ry/ Mb,Ib	vinsertps Vdq,Hdq,Udq/ Md,Ib	vpinsrd/q Vdq,Hdq,Ey,Ib					
3									
4	66	vdpps Vdq,Wdq,Ib Vx,Hx,Wx,Ib	vdppd Vdq,Wdq,Ib Vx,Hx,Wx,Ib	vmpsadbw Vdq,Hdq,Wdq,Ib		vpclmulqdq Vdq,Hdq,Wdq,Ib			
5									
6	66	vpcmpestrm Vdq, 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 Vdq,Wdq,Ib Vx,Wx,Ib	vroundpd Vdq,Wdq,Ib Vx,Wx,Ib	vroundss Vss,Hss,Wss,Ib	vroundsd Vss,Hss,Wss,Ib	vblendps Vdq,Wdq,Ib Vx,Hx,Wx,Ib	vblendpd Vdq,Wdq,Ib Vx,Hx,Wx,Ib	vpblendw Vdq,Hdq,Wdq,Ib	vpalignr Vdq,Hdq,Wdq,Ib
1	66	vinsertf128^{V} Vqq,Hqq,Wqq,Ib	vextractf128^{V} Wdq,Vqq,Ib						
2									
3									
4	66			vblendvps^{V}Vx,Hx ,Wx,Lx,Ib	vblendvpd^{V}Vx,Hx ,Wx,Lx,Ib	vpblendvb^{V}Vdq, 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.

...

12. Updates to Appendix B, Volume 2B

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

...

B.5.3 MMX Instruction Formats and Encodings Table

Table B-19 shows the formats and encodings of the integer instructions.

Table B-19 MMX Instruction Formats and Encodings

Instruction and Format	Encoding
EMMS – Empty MMX technology state	0000 1111:01110111
MOVD – Move doubleword	
reg to mmxreg	0000 1111:0110 1110: 11 mmxreg reg
reg from mmxreg	0000 1111:0111 1110: 11 mmxreg reg
mem to mmxreg	0000 1111:0110 1110: mod mmxreg r/m

Table B-19 MMX Instruction Formats and Encodings (Continued)

Instruction and Format	Encoding
mem from mmxreg	0000 1111:0111 1110: mod mmxreg r/m
MOVQ – Move quadword	
mmxreg2 to mmxreg1	0000 1111:0110 1111: 11 mmxreg1 mmxreg2
mmxreg2 from mmxreg1	0000 1111:0111 1111: 11 mmxreg1 mmxreg2
mem to mmxreg	0000 1111:0110 1111: mod mmxreg r/m
mem from mmxreg	0000 1111:0111 1111: mod mmxreg r/m
PACKSSDW[1] – Pack dword to word data (signed with saturation)	
mmxreg2 to mmxreg1	0000 1111:0110 1011: 11 mmxreg1 mmxreg2
memory to mmxreg	0000 1111:0110 1011: mod mmxreg r/m
PACKSSWB[1] – Pack word to byte data (signed with saturation)	
mmxreg2 to mmxreg1	0000 1111:0110 0011: 11 mmxreg1 mmxreg2
memory to mmxreg	0000 1111:0110 0011: mod mmxreg r/m
PACKUSWB[1] – Pack word to byte data (unsigned with saturation)	
mmxreg2 to mmxreg1	0000 1111:0110 0111: 11 mmxreg1 mmxreg2
memory to mmxreg	0000 1111:0110 0111: mod mmxreg r/m
PADD – Add with wrap-around	
mmxreg2 to mmxreg1	0000 1111: 1111 11gg: 11 mmxreg1 mmxreg2
memory to mmxreg	0000 1111: 1111 11gg: mod mmxreg r/m
PADDS – Add signed with saturation	
mmxreg2 to mmxreg1	0000 1111: 1110 11gg: 11 mmxreg1 mmxreg2
memory to mmxreg	0000 1111: 1110 11gg: mod mmxreg r/m
PADDUS – Add unsigned with saturation	
mmxreg2 to mmxreg1	0000 1111: 1101 11gg: 11 mmxreg1 mmxreg2
memory to mmxreg	0000 1111: 1101 11gg: mod mmxreg r/m
PAND – Bitwise And	
mmxreg2 to mmxreg1	0000 1111:1101 1011: 11 mmxreg1 mmxreg2
memory to mmxreg	0000 1111:1101 1011: mod mmxreg r/m
PANDN – Bitwise AndNot	
mmxreg2 to mmxreg1	0000 1111:1101 1111: 11 mmxreg1 mmxreg2
memory to mmxreg	0000 1111:1101 1111: mod mmxreg r/m
PCMPEQ – Packed compare for equality	
mmxreg1 with mmxreg2	0000 1111:0111 01gg: 11 mmxreg1 mmxreg2
mmxreg with memory	0000 1111:0111 01gg: mod mmxreg r/m

Table B-19 MMX Instruction Formats and Encodings (Continued)

Instruction and Format	Encoding
PCMPGT – Packed compare greater (signed)	
mmxreg1 with mmxreg2	0000 1111:0110 01gg: 11 mmxreg1 mmxreg2
mmxreg with memory	0000 1111:0110 01gg: mod mmxreg r/m
PMADDWD – Packed multiply add	
mmxreg2 to mmxreg1	0000 1111:1111 0101: 11 mmxreg1 mmxreg2
memory to mmxreg	0000 1111:1111 0101: mod mmxreg r/m
PMULHUW – Packed multiplication, store high word (unsigned)	
mmxreg2 to mmxreg1	0000 1111: 1110 0100: 11 mmxreg1 mmxreg2
memory to mmxreg	0000 1111: 1110 0100: mod mmxreg r/m
PMULHW – Packed multiplication, store high word	
mmxreg2 to mmxreg1	0000 1111:1110 0101: 11 mmxreg1 mmxreg2
memory to mmxreg	0000 1111:1110 0101: mod mmxreg r/m
PMULLW – Packed multiplication, store low word	
mmxreg2 to mmxreg1	0000 1111:1101 0101: 11 mmxreg1 mmxreg2
memory to mmxreg	0000 1111:1101 0101: mod mmxreg r/m
POR – Bitwise Or	
mmxreg2 to mmxreg1	0000 1111:1110 1011: 11 mmxreg1 mmxreg2
memory to mmxreg	0000 1111:1110 1011: mod mmxreg r/m
PSLL[2] – Packed shift left logical	
mmxreg1 by mmxreg2	0000 1111:1111 00gg: 11 mmxreg1 mmxreg2
mmxreg by memory	0000 1111:1111 00gg: mod mmxreg r/m
mmxreg by immediate	0000 1111:0111 00gg: 11 110 mmxreg: imm8 data
PSRA[2] – Packed shift right arithmetic	
mmxreg1 by mmxreg2	0000 1111:1110 00gg: 11 mmxreg1 mmxreg2
mmxreg by memory	0000 1111:1110 00gg: mod mmxreg r/m
mmxreg by immediate	0000 1111:0111 00gg: 11 100 mmxreg: imm8 data
PSRL[2] – Packed shift right logical	
mmxreg1 by mmxreg2	0000 1111:1101 00gg: 11 mmxreg1 mmxreg2
mmxreg by memory	0000 1111:1101 00gg: mod mmxreg r/m
mmxreg by immediate	0000 1111:0111 00gg: 11 010 mmxreg: imm8 data

Table B-19 MMX Instruction Formats and Encodings (Continued)

Instruction and Format	Encoding
PSUB – Subtract with wrap-around	
mmxreg2 from mmxreg1	0000 1111:1111 10gg: 11 mmxreg1 mmxreg2
memory from mmxreg	0000 1111:1111 10gg: mod mmxreg r/m
PSUBS – Subtract signed with saturation	
mmxreg2 from mmxreg1	0000 1111:1110 10gg: 11 mmxreg1 mmxreg2
memory from mmxreg	0000 1111:1110 10gg: mod mmxreg r/m
PSUBUS – Subtract unsigned with saturation	
mmxreg2 from mmxreg1	0000 1111:1101 10gg: 11 mmxreg1 mmxreg2
memory from mmxreg	0000 1111:1101 10gg: mod mmxreg r/m
PUNPCKH – Unpack high data to next larger type	
mmxreg2 to mmxreg1	0000 1111:0110 10gg: 11 mmxreg1 mmxreg2
memory to mmxreg	0000 1111:0110 10gg: mod mmxreg r/m
PUNPCKL – Unpack low data to next larger type	
mmxreg2 to mmxreg1	0000 1111:0110 00gg: 11 mmxreg1 mmxreg2
memory to mmxreg	0000 1111:0110 00gg: mod mmxreg r/m
PXOR – Bitwise Xor	
mmxreg2 to mmxreg1	0000 1111:1110 1111: 11 mmxreg1 mmxreg2
memory to mmxreg	0000 1111:1110 1111: mod mmxreg r/m

NOTES:

1. The pack instructions perform saturation from signed packed data of one type to signed or unsigned data of the next smaller type.
2. The format of the shift instructions has one additional format to support shifting by immediate shift-counts. The shift operations are not supported equally for all data types.

...

B.8 SSE INSTRUCTION FORMATS AND ENCODINGS

The SSE instructions use the ModR/M format and are preceded by the 0FH prefix byte. In general, operations are not duplicated to provide two directions (that is, separate load and store variants).

The following three tables (Tables Table B-22, B-23, and Table B-24) show the formats and encodings for the SSE SIMD floating-point, SIMD integer, and cacheability and memory ordering instructions, respectively. Some SSE instructions require a mandatory prefix (66H, F2H, F3H) as part of the two-byte opcode. Mandatory prefixes are included in the tables.

Table B-22 Formats and Encodings of SSE Floating-Point Instructions

Instruction and Format	Encoding
ADDPS—Add Packed Single-Precision Floating-Point Values	
xmmreg2 to xmmreg1	0000 1111:0101 1000:11 xmmreg1 xmmreg2
mem to xmmreg	0000 1111:0101 1000: mod xmmreg r/m
ADDSS—Add Scalar Single-Precision Floating-Point Values	
xmmreg2 to xmmreg1	1111 0011:0000 1111:01011000:11 xmmreg1 xmmreg2
mem to xmmreg	1111 0011:0000 1111:01011000: mod xmmreg r/m
ANDNPS—Bitwise Logical AND NOT of Packed Single-Precision Floating-Point Values	
xmmreg2 to xmmreg1	0000 1111:0101 0101:11 xmmreg1 xmmreg2
mem to xmmreg	0000 1111:0101 0101: mod xmmreg r/m
ANDPS—Bitwise Logical AND of Packed Single-Precision Floating-Point Values	
xmmreg2 to xmmreg1	0000 1111:0101 0100:11 xmmreg1 xmmreg2
mem to xmmreg	0000 1111:0101 0100: mod xmmreg r/m
CMPPS—Compare Packed Single-Precision Floating-Point Values	
xmmreg2 to xmmreg1, imm8	0000 1111:1100 0010:11 xmmreg1 xmmreg2: imm8
mem to xmmreg, imm8	0000 1111:1100 0010: mod xmmreg r/m: imm8
CMPSS—Compare Scalar Single-Precision Floating-Point Values	
xmmreg2 to xmmreg1, imm8	1111 0011:0000 1111:1100 0010:11 xmmreg1 xmmreg2: imm8
mem to xmmreg, imm8	1111 0011:0000 1111:1100 0010: mod xmmreg r/ m: imm8
COMISS—Compare Scalar Ordered Single-Precision Floating-Point Values and Set EFLAGS	
xmmreg2 to xmmreg1	0000 1111:0010 1111:11 xmmreg1 xmmreg2
mem to xmmreg	0000 1111:0010 1111: mod xmmreg r/m
CVTPI2PS—Convert Packed Doubleword Integers to Packed Single-Precision Floating-Point Values	
mmreg to xmmreg	0000 1111:0010 1010:11 xmmreg1 mmreg1
mem to xmmreg	0000 1111:0010 1010: mod xmmreg r/m

Table B-22 Formats and Encodings of SSE Floating-Point Instructions (Continued)

Instruction and Format	Encoding
CVTPS2PI—Convert Packed Single-Precision Floating-Point Values to Packed Doubleword Integers	
xmmreg to mmreg	0000 1111:0010 1101:11 mmreg1 xmmreg1
mem to mmreg	0000 1111:0010 1101: mod mmreg r/m
CVTSI2SS—Convert Doubleword Integer to Scalar Single-Precision Floating-Point Value	
r32 to xmmreg1	1111 0011:0000 1111:00101010:11 xmmreg1 r32
mem to xmmreg	1111 0011:0000 1111:00101010: mod xmmreg r/m
CVTSS2SI—Convert Scalar Single-Precision Floating-Point Value to Doubleword Integer	
xmmreg to r32	1111 0011:0000 1111:0010 1101:11 r32 xmmreg
mem to r32	1111 0011:0000 1111:0010 1101: mod r32 r/m
CVTTPS2PI—Convert with Truncation Packed Single-Precision Floating-Point Values to Packed Doubleword Integers	
xmmreg to mmreg	0000 1111:0010 1100:11 mmreg1 xmmreg1
mem to mmreg	0000 1111:0010 1100: mod mmreg r/m
CVTTSS2SI—Convert with Truncation Scalar Single-Precision Floating-Point Value to Doubleword Integer	
xmmreg to r32	1111 0011:0000 1111:0010 1100:11 r32 xmmreg1
mem to r32	1111 0011:0000 1111:0010 1100: mod r32 r/m
DIVPS—Divide Packed Single-Precision Floating-Point Values	
xmmreg2 to xmmreg1	0000 1111:0101 1110:11 xmmreg1 xmmreg2
mem to xmmreg	0000 1111:0101 1110: mod xmmreg r/m
DIVSS—Divide Scalar Single-Precision Floating-Point Values	
xmmreg2 to xmmreg1	1111 0011:0000 1111:0101 1110:11 xmmreg1 xmmreg2
mem to xmmreg	1111 0011:0000 1111:0101 1110: mod xmmreg r/m
LDMXCSR—Load MXCSR Register State	
m32 to MXCSR	0000 1111:1010 1110:mod[A] 010 mem
MAXPS—Return Maximum Packed Single-Precision Floating-Point Values	
xmmreg2 to xmmreg1	0000 1111:0101 1111:11 xmmreg1 xmmreg2
mem to xmmreg	0000 1111:0101 1111: mod xmmreg r/m

Table B-22 Formats and Encodings of SSE Floating-Point Instructions (Continued)

Instruction and Format	Encoding
MAXSS—Return Maximum Scalar Double-Precision Floating-Point Value	
xmmreg2 to xmmreg1	1111 0011:0000 1111:0101 1111:11 xmmreg1 xmmreg2
mem to xmmreg	1111 0011:0000 1111:0101 1111: mod xmmreg r/m
MINPS—Return Minimum Packed Double-Precision Floating-Point Values	
xmmreg2 to xmmreg1	0000 1111:0101 1101:11 xmmreg1 xmmreg2
mem to xmmreg	0000 1111:0101 1101: mod xmmreg r/m
MINSS—Return Minimum Scalar Double-Precision Floating-Point Value	
xmmreg2 to xmmreg1	1111 0011:0000 1111:0101 1101:11 xmmreg1 xmmreg2
mem to xmmreg	1111 0011:0000 1111:0101 1101: mod xmmreg r/m
MOVAPS—Move Aligned Packed Single-Precision Floating-Point Values	
xmmreg2 to xmmreg1	0000 1111:0010 1000:11 xmmreg2 xmmreg1
mem to xmmreg1	0000 1111:0010 1000: mod xmmreg r/m
xmmreg1 to xmmreg2	0000 1111:0010 1001:11 xmmreg1 xmmreg2
xmmreg1 to mem	0000 1111:0010 1001: mod xmmreg r/m
MOVHLPS—Move Packed Single-Precision Floating-Point Values High to Low	
xmmreg2 to xmmreg1	0000 1111:0001 0010:11 xmmreg1 xmmreg2
MOVHPS—Move High Packed Single-Precision Floating-Point Values	
mem to xmmreg	0000 1111:0001 0110: mod xmmreg r/m
xmmreg to mem	0000 1111:0001 0111: mod xmmreg r/m
MOVLHPS—Move Packed Single-Precision Floating-Point Values Low to High	
xmmreg2 to xmmreg1	0000 1111:00010110:11 xmmreg1 xmmreg2
MOVLPS—Move Low Packed Single-Precision Floating-Point Values	
mem to xmmreg	0000 1111:0001 0010: mod xmmreg r/m
xmmreg to mem	0000 1111:0001 0011: mod xmmreg r/m

Table B-22 Formats and Encodings of SSE Floating-Point Instructions (Continued)

Instruction and Format	Encoding
MOVMSKPS—Extract Packed Single-Precision Floating-Point Sign Mask	
xmmreg to r32	0000 1111:0101 0000:11 r32 xmmreg
MOVSS—Move Scalar Single-Precision Floating-Point Values	
xmmreg2 to xmmreg1	1111 0011:0000 1111:0001 0000:11 xmmreg2 xmmreg1
mem to xmmreg1	1111 0011:0000 1111:0001 0000: mod xmmreg r/m
xmmreg1 to xmmreg2	1111 0011:0000 1111:0001 0001:11 xmmreg1 xmmreg2
xmmreg1 to mem	1111 0011:0000 1111:0001 0001: mod xmmreg r/m
MOVUPS—Move Unaligned Packed Single-Precision Floating-Point Values	
xmmreg2 to xmmreg1	0000 1111:0001 0000:11 xmmreg2 xmmreg1
mem to xmmreg1	0000 1111:0001 0000: mod xmmreg r/m
xmmreg1 to xmmreg2	0000 1111:0001 0001:11 xmmreg1 xmmreg2
xmmreg1 to mem	0000 1111:0001 0001: mod xmmreg r/m
MULPS—Multiply Packed Single-Precision Floating-Point Values	
xmmreg2 to xmmreg1	0000 1111:0101 1001:11 xmmreg1 xmmreg2
mem to xmmreg	0000 1111:0101 1001: mod xmmreg r/m
MULSS—Multiply Scalar Single-Precision Floating-Point Values	
xmmreg2 to xmmreg1	1111 0011:0000 1111:0101 1001:11 xmmreg1 xmmreg2
mem to xmmreg	1111 0011:0000 1111:0101 1001: mod xmmreg r/m
ORPS—Bitwise Logical OR of Single-Precision Floating-Point Values	
xmmreg2 to xmmreg1	0000 1111:0101 0110:11 xmmreg1 xmmreg2
mem to xmmreg	0000 1111:0101 0110: mod xmmreg r/m
RCPPS—Compute Reciprocals of Packed Single-Precision Floating-Point Values	
xmmreg2 to xmmreg1	0000 1111:0101 0011:11 xmmreg1 xmmreg2
mem to xmmreg	0000 1111:0101 0011: mod xmmreg r/m
RCPSS—Compute Reciprocals of Scalar Single-Precision Floating-Point Value	
xmmreg2 to xmmreg1	1111 0011:0000 1111:01010011:11 xmmreg1 xmmreg2

Table B-22 Formats and Encodings of SSE Floating-Point Instructions (Continued)

Instruction and Format	Encoding
mem to xmmreg	1111 0011:0000 1111:01010011: mod xmmreg r/m
RSQRTPS—Compute Reciprocals of Square Roots of Packed Single-Precision Floating-Point Values	
xmmreg2 to xmmreg1	0000 1111:0101 0010:11 xmmreg1 xmmreg2
mem to xmmreg	0000 1111:0101 0010: mode xmmreg r/m
RSQRTSS—Compute Reciprocals of Square Roots of Scalar Single-Precision Floating-Point Value	
xmmreg2 to xmmreg1	1111 0011:0000 1111:0101 0010:11 xmmreg1 xmmreg2
mem to xmmreg	1111 0011:0000 1111:0101 0010: mod xmmreg r/m
SHUFPS—Shuffle Packed Single-Precision Floating-Point Values	
xmmreg2 to xmmreg1, imm8	0000 1111:1100 0110:11 xmmreg1 xmmreg2: imm8
mem to xmmreg, imm8	0000 1111:1100 0110: mod xmmreg r/m: imm8
SQRTPS—Compute Square Roots of Packed Single-Precision Floating-Point Values	
xmmreg2 to xmmreg1	0000 1111:0101 0001:11 xmmreg1 xmmreg2
mem to xmmreg	0000 1111:0101 0001: mod xmmreg r/m
SQRTSS—Compute Square Root of Scalar Single-Precision Floating-Point Value	
xmmreg2 to xmmreg1	1111 0011:0000 1111:0101 0001:11 xmmreg1 xmmreg2
mem to xmmreg	1111 0011:0000 1111:0101 0001:mod xmmreg r/m
STMXCSR—Store MXCSR Register State	
MXCSR to mem	0000 1111:1010 1110:mod[A] 011 mem
SUBPS—Subtract Packed Single-Precision Floating-Point Values	
xmmreg2 to xmmreg1	0000 1111:0101 1100:11 xmmreg1 xmmreg2
mem to xmmreg	0000 1111:0101 1100:mod xmmreg r/m
SUBSS—Subtract Scalar Single-Precision Floating-Point Values	
xmmreg2 to xmmreg1	1111 0011:0000 1111:0101 1100:11 xmmreg1 xmmreg2
mem to xmmreg	1111 0011:0000 1111:0101 1100:mod xmmreg r/m

Table B-22 Formats and Encodings of SSE Floating-Point Instructions (Continued)

Instruction and Format	Encoding
UCOMISS—Unordered Compare Scalar Ordered Single-Precision Floating-Point Values and Set EFLAGS	
xmmreg2 to xmmreg1	0000 1111:0010 1110:11 xmmreg1 xmmreg2
mem to xmmreg	0000 1111:0010 1110: mod xmmreg r/m
UNPCKHPS—Unpack and Interleave High Packed Single-Precision Floating-Point Values	
xmmreg2 to xmmreg1	0000 1111:0001 0101:11 xmmreg1 xmmreg2
mem to xmmreg	0000 1111:0001 0101: mod xmmreg r/m
UNPCKLPS—Unpack and Interleave Low Packed Single-Precision Floating-Point Values	
xmmreg2 to xmmreg1	0000 1111:0001 0100:11 xmmreg1 xmmreg2
mem to xmmreg	0000 1111:0001 0100: mod xmmreg r/m
XORPS—Bitwise Logical XOR of Single-Precision Floating-Point Values	
xmmreg2 to xmmreg1	0000 1111:0101 0111:11 xmmreg1 xmmreg2
mem to xmmreg	0000 1111:0101 0111: mod xmmreg r/m

Table B-23 Formats and Encodings of SSE Integer Instructions

Instruction and Format	Encoding
PAVGB/PAVGW—Average Packed Integers	
mmreg2 to mmreg1	0000 1111:1110 0000:11 mmreg1 mmreg2
	0000 1111:1110 0011:11 mmreg1 mmreg2
mem to mmreg	0000 1111:1110 0000: mod mmreg r/m
	0000 1111:1110 0011: mod mmreg r/m
PEXTRW—Extract Word	
mmreg to reg32, imm8	0000 1111:1100 0101:11 r32 mmreg: imm8
PINSRW—Insert Word	
reg32 to mmreg, imm8	0000 1111:1100 0100:11 mmreg r32: imm8
m16 to mmreg, imm8	0000 1111:1100 0100: mod mmreg r/m: imm8
PMAXSW—Maximum of Packed Signed Word Integers	
mmreg2 to mmreg1	0000 1111:1110 1110:11 mmreg1 mmreg2
mem to mmreg	0000 1111:1110 1110: mod mmreg r/m
PMAXUB—Maximum of Packed Unsigned Byte Integers	

Instruction and Format	Encoding
mmreg2 to mmreg1	0000 1111:1101 1110:11 mmreg1 mmreg2
mem to mmreg	0000 1111:1101 1110: mod mmreg r/m
PMINSW—Minimum of Packed Signed Word Integers	
mmreg2 to mmreg1	0000 1111:1110 1010:11 mmreg1 mmreg2
mem to mmreg	0000 1111:1110 1010: mod mmreg r/m
PMINUB—Minimum of Packed Unsigned Byte Integers	
mmreg2 to mmreg1	0000 1111:1101 1010:11 mmreg1 mmreg2
mem to mmreg	0000 1111:1101 1010: mod mmreg r/m
PMOVMSKB—Move Byte Mask To Integer	
mmreg to reg32	0000 1111:1101 0111:11 r32 mmreg
PMULHUW—Multiply Packed Unsigned Integers and Store High Result	
mmreg2 to mmreg1	0000 1111:1110 0100:11 mmreg1 mmreg2
mem to mmreg	0000 1111:1110 0100: mod mmreg r/m
PSADBW—Compute Sum of Absolute Differences	
mmreg2 to mmreg1	0000 1111:1111 0110:11 mmreg1 mmreg2
mem to mmreg	0000 1111:1111 0110: mod mmreg r/m
PSHUFW—Shuffle Packed Words	
mmreg2 to mmreg1, imm8	0000 1111:0111 0000:11 mmreg1 mmreg2: imm8
mem to mmreg, imm8	0000 1111:0111 0000: mod mmreg r/m: imm8

Table B-24 Format and Encoding of SSE Cacheability & Memory Ordering Instructions

Instruction and Format	Encoding
MASKMOVQ—Store Selected Bytes of Quadword	
mmreg2 to mmreg1	0000 1111:1111 0111:11 mmreg1 mmreg2
MOVNTPS—Store Packed Single-Precision Floating-Point Values Using Non-Temporal Hint	
xmmreg to mem	0000 1111:0010 1011: mod xmmreg r/m
MOVNTQ—Store Quadword Using Non-Temporal Hint	
mmreg to mem	0000 1111:1110 0111: mod mmreg r/m

Instruction and Format	Encoding
PREFETCHT0—Prefetch Temporal to All Cache Levels	0000 1111:0001 1000:mod^A 001 mem
PREFETCHT1—Prefetch Temporal to First Level Cache	0000 1111:0001 1000:mod^A 010 mem
PREFETCHT2—Prefetch Temporal to Second Level Cache	0000 1111:0001 1000:mod^A 011 mem
PREFETCHNTA—Prefetch Non-Temporal to All Cache Levels	0000 1111:0001 1000:mod^A 000 mem
SFENCE—Store Fence	0000 1111:1010 1110:11 111 000

...

B.9.1 Granularity Field (gg)

The granularity field (gg) indicates the size of the packed operands that the instruction is operating on. When this field is used, it is located in bits 1 and 0 of the second opcode byte. Table B-25 shows the encoding of this gg field.

Table B-25 Encoding of Granularity of Data Field (gg)

gg	Granularity of Data
00	Packed Bytes
01	Packed Words
10	Packed Doublewords
11	Quadword

Table B-26 Formats and Encodings of SSE2 Floating-Point Instructions

Instruction and Format	Encoding
ADDPD—Add Packed Double-Precision Floating-Point Values	
xmmreg2 to xmmreg1	0110 0110:0000 1111:0101 1000:11 xmmreg1 xmmreg2
mem to xmmreg	0110 0110:0000 1111:0101 1000: mod xmmreg r/m
ADDSD—Add Scalar Double-Precision Floating-Point Values	
xmmreg2 to xmmreg1	1111 0010:0000 1111:0101 1000:11 xmmreg1 xmmreg2
mem to xmmreg	1111 0010:0000 1111:0101 1000: mod xmmreg r/m
ANDNPD—Bitwise Logical AND NOT of Packed Double-Precision Floating-Point Values	
xmmreg2 to xmmreg1	0110 0110:0000 1111:0101 0101:11 xmmreg1 xmmreg2
mem to xmmreg	0110 0110:0000 1111:0101 0101: mod xmmreg r/m

Table B-26 Formats and Encodings of SSE2 Floating-Point Instructions (Continued)

Instruction and Format	Encoding
ANDPD—Bitwise Logical AND of Packed Double-Precision Floating-Point Values	
xmmreg2 to xmmreg1	0110 0110:0000 1111:0101 0100:11 xmmreg1 xmmreg2
mem to xmmreg	0110 0110:0000 1111:0101 0100: mod xmmreg r/m
CMPPD—Compare Packed Double-Precision Floating-Point Values	
xmmreg2 to xmmreg1, imm8	0110 0110:0000 1111:1100 0010:11 xmmreg1 xmmreg2: imm8
mem to xmmreg, imm8	0110 0110:0000 1111:1100 0010: mod xmmreg r/m: imm8
CMPSD—Compare Scalar Double-Precision Floating-Point Values	
xmmreg2 to xmmreg1, imm8	1111 0010:0000 1111:1100 0010:11 xmmreg1 xmmreg2: imm8
mem to xmmreg, imm8	11110 010:0000 1111:1100 0010: mod xmmreg r/m: imm8
COMISD—Compare Scalar Ordered Double-Precision Floating-Point Values and Set EFLAGS	
xmmreg2 to xmmreg1	0110 0110:0000 1111:0010 1111:11 xmmreg1 xmmreg2
mem to xmmreg	0110 0110:0000 1111:0010 1111: mod xmmreg r/m
CVTPI2PD—Convert Packed Doubleword Integers to Packed Double-Precision Floating-Point Values	
mmreg to xmmreg	0110 0110:0000 1111:0010 1010:11 xmmreg1 mmreg1
mem to xmmreg	0110 0110:0000 1111:0010 1010: mod xmmreg r/m
CVTPD2PI—Convert Packed Double-Precision Floating-Point Values to Packed Doubleword Integers	
xmmreg to mmreg	0110 0110:0000 1111:0010 1101:11 mmreg1 xmmreg1
mem to mmreg	0110 0110:0000 1111:0010 1101: mod mmreg r/m
CVTSI2SD—Convert Doubleword Integer to Scalar Double-Precision Floating-Point Value	
r32 to xmmreg1	1111 0010:0000 1111:0010 1010:11 xmmreg r32
mem to xmmreg	1111 0010:0000 1111:0010 1010: mod xmmreg r/m

Table B-26 Formats and Encodings of SSE2 Floating-Point Instructions (Continued)

Instruction and Format	Encoding
CVTSD2SI—Convert Scalar Double-Precision Floating-Point Value to Doubleword Integer	
xmmreg to r32	1111 0010:0000 1111:0010 1101:11 r32 xmmreg
mem to r32	1111 0010:0000 1111:0010 1101: mod r32 r/m
CVTTPD2PI—Convert with Truncation Packed Double-Precision Floating-Point Values to Packed Doubleword Integers	
xmmreg to mmreg	0110 0110:0000 1111:0010 1100:11 mmreg xmmreg
mem to mmreg	0110 0110:0000 1111:0010 1100: mod mmreg r/m
CVTTSD2SI—Convert with Truncation Scalar Double-Precision Floating-Point Value to Doubleword Integer	
xmmreg to r32	1111 0010:0000 1111:0010 1100:11 r32 xmmreg
mem to r32	1111 0010:0000 1111:0010 1100: mod r32 r/m
CVTPD2PS—Covert Packed Double-Precision Floating-Point Values to Packed Single-Precision Floating-Point Values	
xmmreg2 to xmmreg1	0110 0110:0000 1111:0101 1010:11 xmmreg1 xmmreg2
mem to xmmreg	0110 0110:0000 1111:0101 1010: mod xmmreg r/m
CVTPS2PD—Covert Packed Single-Precision Floating-Point Values to Packed Double-Precision Floating-Point Values	
xmmreg2 to xmmreg1	0000 1111:0101 1010:11 xmmreg1 xmmreg2
mem to xmmreg	0000 1111:0101 1010: mod xmmreg r/m
CVTSD2SS—Covert Scalar Double-Precision Floating-Point Value to Scalar Single-Precision Floating-Point Value	
xmmreg2 to xmmreg1	1111 0010:0000 1111:0101 1010:11 xmmreg1 xmmreg2
mem to xmmreg	1111 0010:0000 1111:0101 1010: mod xmmreg r/m
CVTSS2SD—Covert Scalar Single-Precision Floating-Point Value to Scalar Double-Precision Floating-Point Value	

Table B-26 Formats and Encodings of SSE2 Floating-Point Instructions (Continued)

Instruction and Format	Encoding
xmmreg2 to xmmreg1	1111 0011:0000 1111:0101 1010:11 xmmreg1 xmmreg2
mem to xmmreg	1111 0011:00001 111:0101 1010: mod xmmreg r/m
CVTPD2DQ—Convert Packed Double-Precision Floating-Point Values to Packed Doubleword Integers	
xmmreg2 to xmmreg1	1111 0010:0000 1111:1110 0110:11 xmmreg1 xmmreg2
mem to xmmreg	1111 0010:0000 1111:1110 0110: mod xmmreg r/m
CVTTPD2DQ—Convert With Truncation Packed Double-Precision Floating-Point Values to Packed Doubleword Integers	
xmmreg2 to xmmreg1	0110 0110:0000 1111:1110 0110:11 xmmreg1 xmmreg2
mem to xmmreg	0110 0110:0000 1111:1110 0110: mod xmmreg r/m
CVTDQ2PD—Convert Packed Doubleword Integers to Packed Single-Precision Floating-Point Values	
xmmreg2 to xmmreg1	1111 0011:0000 1111:1110 0110:11 xmmreg1 xmmreg2
mem to xmmreg	1111 0011:0000 1111:1110 0110: mod xmmreg r/m
CVTPS2DQ—Convert Packed Single-Precision Floating-Point Values to Packed Doubleword Integers	
xmmreg2 to xmmreg1	0110 0110:0000 1111:0101 1011:11 xmmreg1 xmmreg2
mem to xmmreg	0110 0110:0000 1111:0101 1011: mod xmmreg r/m
CVTTPS2DQ—Convert With Truncation Packed Single-Precision Floating-Point Values to Packed Doubleword Integers	
xmmreg2 to xmmreg1	1111 0011:0000 1111:0101 1011:11 xmmreg1 xmmreg2
mem to xmmreg	1111 0011:0000 1111:0101 1011: mod xmmreg r/m
CVTDQ2PS—Convert Packed Doubleword Integers to Packed Double-Precision Floating-Point Values	
xmmreg2 to xmmreg1	0000 1111:0101 1011:11 xmmreg1 xmmreg2
mem to xmmreg	0000 1111:0101 1011: mod xmmreg r/m
DIVPD—Divide Packed Double-Precision Floating-Point Values	

Table B-26 Formats and Encodings of SSE2 Floating-Point Instructions (Continued)

Instruction and Format	Encoding
xmmreg2 to xmmreg1	0110 0110:0000 1111:0101 1110:11 xmmreg1 xmmreg2
mem to xmmreg	0110 0110:0000 1111:0101 1110: mod xmmreg r/m
DIVSD—Divide Scalar Double-Precision Floating-Point Values	
xmmreg2 to xmmreg1	1111 0010:0000 1111:0101 1110:11 xmmreg1 xmmreg2
mem to xmmreg	1111 0010:0000 1111:0101 1110: mod xmmreg r/m
MAXPD—Return Maximum Packed Double-Precision Floating-Point Values	
xmmreg2 to xmmreg1	0110 0110:0000 1111:0101 1111:11 xmmreg1 xmmreg2
mem to xmmreg	0110 0110:0000 1111:0101 1111: mod xmmreg r/m
MAXSD—Return Maximum Scalar Double-Precision Floating-Point Value	
xmmreg2 to xmmreg1	1111 0010:0000 1111:0101 1111:11 xmmreg1 xmmreg2
mem to xmmreg	1111 0010:0000 1111:0101 1111: mod xmmreg r/m
MINPD—Return Minimum Packed Double-Precision Floating-Point Values	
xmmreg2 to xmmreg1	0110 0110:0000 1111:0101 1101:11 xmmreg1 xmmreg2
mem to xmmreg	0110 0110:0000 1111:0101 1101: mod xmmreg r/m
MINSD—Return Minimum Scalar Double-Precision Floating-Point Value	
xmmreg2 to xmmreg1	1111 0010:0000 1111:0101 1101:11 xmmreg1 xmmreg2
mem to xmmreg	1111 0010:0000 1111:0101 1101: mod xmmreg r/m
MOVAPD—Move Aligned Packed Double-Precision Floating-Point Values	
xmmreg1 to xmmreg2	0110 0110:0000 1111:0010 1001:11 xmmreg2 xmmreg1
xmmreg1 to mem	0110 0110:0000 1111:0010 1001: mod xmmreg r/m
xmmreg2 to xmmreg1	0110 0110:0000 1111:0010 1000:11 xmmreg1 xmmreg2
mem to xmmreg1	0110 0110:0000 1111:0010 1000: mod xmmreg r/m
MOVHPD—Move High Packed Double-Precision Floating-Point Values	
xmmreg to mem	0110 0110:0000 1111:0001 0111: mod xmmreg r/m

Table B-26 Formats and Encodings of SSE2 Floating-Point Instructions (Continued)

Instruction and Format	Encoding
mem to xmmreg	0110 0110:0000 1111:0001 0110: mod xmmreg r/m
MOVLPD—Move Low Packed Double-Precision Floating-Point Values	
xmmreg to mem	0110 0110:0000 1111:0001 0011: mod xmmreg r/m
mem to xmmreg	0110 0110:0000 1111:0001 0010: mod xmmreg r/m
MOVMSKPD—Extract Packed Double-Precision Floating-Point Sign Mask	
xmmreg to r32	0110 0110:0000 1111:0101 0000:11 r32 xmmreg
MOVSD—Move Scalar Double-Precision Floating-Point Values	
xmmreg1 to xmmreg2	1111 0010:0000 1111:0001 0001:11 xmmreg2 xmmreg1
xmmreg1 to mem	1111 0010:0000 1111:0001 0001: mod xmmreg r/m
xmmreg2 to xmmreg1	1111 0010:0000 1111:0001 0000:11 xmmreg1 xmmreg2
mem to xmmreg1	1111 0010:0000 1111:0001 0000: mod xmmreg r/m
MOVUPD—Move Unaligned Packed Double-Precision Floating-Point Values	
xmmreg2 to xmmreg1	0110 0110:0000 1111:0001 0001:11 xmmreg2 xmmreg1
mem to xmmreg1	0110 0110:0000 1111:0001 0001: mod xmmreg r/m
xmmreg1 to xmmreg2	0110 0110:0000 1111:0001 0000:11 xmmreg1 xmmreg2
xmmreg1 to mem	0110 0110:0000 1111:0001 0000: mod xmmreg r/m
MULPD—Multiply Packed Double-Precision Floating-Point Values	
xmmreg2 to xmmreg1	0110 0110:0000 1111:0101 1001:11 xmmreg1 xmmreg2
mem to xmmreg	0110 0110:0000 1111:0101 1001: mod xmmreg r/m
MULSD—Multiply Scalar Double-Precision Floating-Point Values	
xmmreg2 to xmmreg1	1111 0010:00001111:01011001:11 xmmreg1 xmmreg2
mem to xmmreg	1111 0010:00001111:01011001: mod xmmreg r/m
ORPD—Bitwise Logical OR of Double-Precision Floating-Point Values	
xmmreg2 to xmmreg1	0110 0110:0000 1111:0101 0110:11 xmmreg1 xmmreg2

Table B-26 Formats and Encodings of SSE2 Floating-Point Instructions (Continued)

Instruction and Format	Encoding
mem to xmmreg	0110 0110:0000 1111:0101 0110: mod xmmreg r/m
SHUFPD—Shuffle Packed Double-Precision Floating-Point Values	
xmmreg2 to xmmreg1, imm8	0110 0110:0000 1111:1100 0110:11 xmmreg1 xmmreg2: imm8
mem to xmmreg, imm8	0110 0110:0000 1111:1100 0110: mod xmmreg r/m: imm8
SQRTPD—Compute Square Roots of Packed Double-Precision Floating-Point Values	
xmmreg2 to xmmreg1	0110 0110:0000 1111:0101 0001:11 xmmreg1 xmmreg2
mem to xmmreg	0110 0110:0000 1111:0101 0001: mod xmmreg r/m
SQRTSD—Compute Square Root of Scalar Double-Precision Floating-Point Value	
xmmreg2 to xmmreg1	1111 0010:0000 1111:0101 0001:11 xmmreg1 xmmreg2
mem to xmmreg	1111 0010:0000 1111:0101 0001: mod xmmreg r/m
SUBPD—Subtract Packed Double-Precision Floating-Point Values	
xmmreg2 to xmmreg1	0110 0110:0000 1111:0101 1100:11 xmmreg1 xmmreg2
mem to xmmreg	0110 0110:0000 1111:0101 1100: mod xmmreg r/m
SUBSD—Subtract Scalar Double-Precision Floating-Point Values	
xmmreg2 to xmmreg1	1111 0010:0000 1111:0101 1100:11 xmmreg1 xmmreg2
mem to xmmreg	1111 0010:0000 1111:0101 1100: mod xmmreg r/m
UCOMISD—Unordered Compare Scalar Ordered Double-Precision Floating-Point Values and Set EFLAGS	
xmmreg2 to xmmreg1	0110 0110:0000 1111:0010 1110:11 xmmreg1 xmmreg2
mem to xmmreg	0110 0110:0000 1111:0010 1110: mod xmmreg r/m
UNPCKHPD—Unpack and Interleave High Packed Double-Precision Floating-Point Values	
xmmreg2 to xmmreg1	0110 0110:0000 1111:0001 0101:11 xmmreg1 xmmreg2

Table B-26 Formats and Encodings of SSE2 Floating-Point Instructions (Continued)

Instruction and Format	Encoding
mem to xmmreg	0110 0110:0000 1111:0001 0101: mod xmmreg r/m
UNPCKLPD—Unpack and Interleave Low Packed Double-Precision Floating-Point Values	
xmmreg2 to xmmreg1	0110 0110:0000 1111:0001 0100:11 xmmreg1 xmmreg2
mem to xmmreg	0110 0110:0000 1111:0001 0100: mod xmmreg r/m
XORPD—Bitwise Logical OR of Double-Precision Floating-Point Values	
xmmreg2 to xmmreg1	0110 0110:0000 1111:0101 0111:11 xmmreg1 xmmreg2
mem to xmmreg	0110 0110:0000 1111:0101 0111: mod xmmreg r/m

Table B-27 Formats and Encodings of SSE2 Integer Instructions

Instruction and Format	Encoding
MOVD—Move Doubleword	
reg to xmmreg	0110 0110:0000 1111:0110 1110: 11 xmmreg reg
reg from xmmreg	0110 0110:0000 1111:0111 1110: 11 xmmreg reg
mem to xmmreg	0110 0110:0000 1111:0110 1110: mod xmmreg r/m
mem from xmmreg	0110 0110:0000 1111:0111 1110: mod xmmreg r/m
MOVDQA—Move Aligned Double Quadword	
xmmreg2 to xmmreg1	0110 0110:0000 1111:0110 1111:11 xmmreg1 xmmreg2
xmmreg2 from xmmreg1	0110 0110:0000 1111:0111 1111:11 xmmreg1 xmmreg2
mem to xmmreg	0110 0110:0000 1111:0110 1111: mod xmmreg r/m
mem from xmmreg	0110 0110:0000 1111:0111 1111: mod xmmreg r/m
MOVDQU—Move Unaligned Double Quadword	
xmmreg2 to xmmreg1	1111 0011:0000 1111:0110 1111:11 xmmreg1 xmmreg2
xmmreg2 from xmmreg1	1111 0011:0000 1111:0111 1111:11 xmmreg1 xmmreg2
mem to xmmreg	1111 0011:0000 1111:0110 1111: mod xmmreg r/m
mem from xmmreg	1111 0011:0000 1111:0111 1111: mod xmmreg r/m
MOVQ2DQ—Move Quadword from MMX to XMM Register	
mmreg to xmmreg	1111 0011:0000 1111:1101 0110:11 mmreg1 mmreg2
MOVDQ2Q—Move Quadword from XMM to MMX Register	
xmmreg to mmreg	1111 0010:0000 1111:1101 0110:11 mmreg1 mmreg2
MOVQ—Move Quadword	
xmmreg2 to xmmreg1	1111 0011:0000 1111:0111 1110: 11 xmmreg1 xmmreg2
xmmreg2 from xmmreg1	0110 0110:0000 1111:1101 0110: 11 xmmreg1 xmmreg2
mem to xmmreg	1111 0011:0000 1111:0111 1110: mod xmmreg r/m
mem from xmmreg	0110 0110:0000 1111:1101 0110: mod xmmreg r/m
PACKSSDW[1]—Pack Dword To Word Data (signed with saturation)	
xmmreg2 to xmmreg1	0110 0110:0000 1111:0110 1011: 11 xmmreg1 xmmreg2

Table B-27 Formats and Encodings of SSE2 Integer Instructions (Continued)

Instruction and Format	Encoding
memory to xmmreg	0110 0110:0000 1111:0110 1011: mod xmmreg r/m
PACKSSWB—Pack Word To Byte Data (signed with saturation)	
xmmreg2 to xmmreg1	0110 0110:0000 1111:0110 0011: 11 xmmreg1 xmmreg2
memory to xmmreg	0110 0110:0000 1111:0110 0011: mod xmmreg r/m
PACKUSWB—Pack Word To Byte Data (unsigned with saturation)	
xmmreg2 to xmmreg1	0110 0110:0000 1111:0110 0111: 11 xmmreg1 xmmreg2
memory to xmmreg	0110 0110:0000 1111:0110 0111: mod xmmreg r/m
PADDQ—Add Packed Quadword Integers	
mmreg2 to mmreg1	0000 1111:1101 0100:11 mmreg1 mmreg2
mem to mmreg	0000 1111:1101 0100: mod mmreg r/m
xmmreg2 to xmmreg1	0110 0110:0000 1111:1101 0100:11 xmmreg1 xmmreg2
mem to xmmreg	0110 0110:0000 1111:1101 0100: mod xmmreg r/m
PADD—Add With Wrap-around	
xmmreg2 to xmmreg1	0110 0110:0000 1111: 1111 11gg: 11 xmmreg1 xmmreg2
memory to xmmreg	0110 0110:0000 1111: 1111 11gg: mod xmmreg r/m
PADDS—Add Signed With Saturation	
xmmreg2 to xmmreg1	0110 0110:0000 1111: 1110 11gg: 11 xmmreg1 xmmreg2
memory to xmmreg	0110 0110:0000 1111: 1110 11gg: mod xmmreg r/m
PADDUS—Add Unsigned With Saturation	
xmmreg2 to xmmreg1	0110 0110:0000 1111: 1101 11gg: 11 xmmreg1 xmmreg2
memory to xmmreg	0110 0110:0000 1111: 1101 11gg: mod xmmreg r/m
PAND—Bitwise And	
xmmreg2 to xmmreg1	0110 0110:0000 1111:1101 1011: 11 xmmreg1 xmmreg2
memory to xmmreg	0110 0110:0000 1111:1101 1011: mod xmmreg r/m
PANDN—Bitwise AndNot	
xmmreg2 to xmmreg1	0110 0110:0000 1111:1101 1111: 11 xmmreg1 xmmreg2
memory to xmmreg	0110 0110:0000 1111:1101 1111: mod xmmreg r/m
PAVGB—Average Packed Integers	

Table B-27 Formats and Encodings of SSE2 Integer Instructions (Continued)

Instruction and Format	Encoding
xmmreg2 to xmmreg1	0110 0110:0000 1111:11100 000:11 xmmreg1 xmmreg2
mem to xmmreg	01100110:00001111:11100000 mod xmmreg r/m
PAVGW—Average Packed Integers	
xmmreg2 to xmmreg1	0110 0110:0000 1111:1110 0011:11 xmmreg1 xmmreg2
mem to xmmreg	0110 0110:0000 1111:1110 0011 mod xmmreg r/m
PCMPEQ—Packed Compare For Equality	
xmmreg1 with xmmreg2	0110 0110:0000 1111:0111 01gg: 11 xmmreg1 xmmreg2
xmmreg with memory	0110 0110:0000 1111:0111 01gg: mod xmmreg r/m
PCMPGT—Packed Compare Greater (signed)	
xmmreg1 with xmmreg2	0110 0110:0000 1111:0110 01gg: 11 xmmreg1 xmmreg2
xmmreg with memory	0110 0110:0000 1111:0110 01gg: mod xmmreg r/m
PEXTRW—Extract Word	
xmmreg to reg32, imm8	0110 0110:0000 1111:1100 0101:11 r32 xmmreg: imm8
PINSRW—Insert Word	
reg32 to xmmreg, imm8	0110 0110:0000 1111:1100 0100:11 xmmreg r32: imm8
m16 to xmmreg, imm8	0110 0110:0000 1111:1100 0100: mod xmmreg r/m: imm8
PMADDWD—Packed Multiply Add	
xmmreg2 to xmmreg1	0110 0110:0000 1111:1111 0101: 11 xmmreg1 xmmreg2
memory to xmmreg	0110 0110:0000 1111:1111 0101: mod xmmreg r/m
PMAXSW—Maximum of Packed Signed Word Integers	
xmmreg2 to xmmreg1	0110 0110:0000 1111:1110 1110:11 xmmreg1 xmmreg2
mem to xmmreg	01100110:00001111:11101110: mod xmmreg r/m
PMAXUB—Maximum of Packed Unsigned Byte Integers	
xmmreg2 to xmmreg1	0110 0110:0000 1111:1101 1110:11 xmmreg1 xmmreg2
mem to xmmreg	0110 0110:0000 1111:1101 1110: mod xmmreg r/m

Table B-27 Formats and Encodings of SSE2 Integer Instructions (Continued)

Instruction and Format	Encoding
PMINSW—Minimum of Packed Signed Word Integers	
xmmreg2 to xmmreg1	0110 0110:0000 1111:1110 1010:11 xmmreg1 xmmreg2
mem to xmmreg	0110 0110:0000 1111:1110 1010: mod xmmreg r/m
PMINUB—Minimum of Packed Unsigned Byte Integers	
xmmreg2 to xmmreg1	0110 0110:0000 1111:1101 1010:11 xmmreg1 xmmreg2
mem to xmmreg	0110 0110:0000 1111:1101 1010 mod xmmreg r/m
PMOVMSKB—Move Byte Mask To Integer	
xmmreg to reg32	0110 0110:0000 1111:1101 0111:11 r32 xmmreg
PMULHUW—Packed multiplication, store high word (unsigned)	
xmmreg2 to xmmreg1	0110 0110:0000 1111:1110 0100: 11 xmmreg1 xmmreg2
memory to xmmreg	0110 0110:0000 1111:1110 0100: mod xmmreg r/m
PMULHW—Packed Multiplication, store high word	
xmmreg2 to xmmreg1	0110 0110:0000 1111:1110 0101: 11 xmmreg1 xmmreg2
memory to xmmreg	0110 0110:0000 1111:1110 0101: mod xmmreg r/m
PMULLW—Packed Multiplication, store low word	
xmmreg2 to xmmreg1	0110 0110:0000 1111:1101 0101: 11 xmmreg1 xmmreg2
memory to xmmreg	0110 0110:0000 1111:1101 0101: mod xmmreg r/m
PMULUDQ—Multiply Packed Unsigned Doubleword Integers	
mmreg2 to mmreg1	0000 1111:1111 0100:11 mmreg1 mmreg2
mem to mmreg	0000 1111:1111 0100: mod mmreg r/m
xmmreg2 to xmmreg1	0110 0110:00001111:1111 0100:11 xmmreg1 xmmreg2
mem to xmmreg	0110 0110:00001111:1111 0100: mod xmmreg r/m
POR—Bitwise Or	
xmmreg2 to xmmreg1	0110 0110:0000 1111:1110 1011: 11 xmmreg1 xmmreg2
memory to xmmreg	0110 0110:0000 1111:1110 1011: mod xmmreg r/m

Table B-27 Formats and Encodings of SSE2 Integer Instructions (Continued)

Instruction and Format	Encoding
PSADBW—Compute Sum of Absolute Differences	
xmmreg2 to xmmreg1	0110 0110:0000 1111:1111 0110:11 xmmreg1 xmmreg2
mem to xmmreg	0110 0110:0000 1111:1111 0110: mod xmmreg r/m
PSHUFLW—Shuffle Packed Low Words	
xmmreg2 to xmmreg1, imm8	1111 0010:0000 1111:0111 0000:11 xmmreg1 xmmreg2: imm8
mem to xmmreg, imm8	1111 0010:0000 1111:0111 0000:11 mod xmmreg r/m: imm8
PSHUFHW—Shuffle Packed High Words	
xmmreg2 to xmmreg1, imm8	1111 0011:0000 1111:0111 0000:11 xmmreg1 xmmreg2: imm8
mem to xmmreg, imm8	1111 0011:0000 1111:0111 0000: mod xmmreg r/m: imm8
PSHUFD—Shuffle Packed Doublewords	
xmmreg2 to xmmreg1, imm8	0110 0110:0000 1111:0111 0000:11 xmmreg1 xmmreg2: imm8
mem to xmmreg, imm8	0110 0110:0000 1111:0111 0000: mod xmmreg r/m: imm8
PSLLDQ—Shift Double Quadword Left Logical	
xmmreg, imm8	0110 0110:0000 1111:0111 0011:11 111 xmmreg: imm8
PSLL—Packed Shift Left Logical	
xmmreg1 by xmmreg2	0110 0110:0000 1111:1111 00gg: 11 xmmreg1 xmmreg2
xmmreg by memory	0110 0110:0000 1111:1111 00gg: mod xmmreg r/m
xmmreg by immediate	0110 0110:0000 1111:0111 00gg: 11 110 xmmreg: imm8
PSRA—Packed Shift Right Arithmetic	
xmmreg1 by xmmreg2	0110 0110:0000 1111:1110 00gg: 11 xmmreg1 xmmreg2
xmmreg by memory	0110 0110:0000 1111:1110 00gg: mod xmmreg r/m
xmmreg by immediate	0110 0110:0000 1111:0111 00gg: 11 100 xmmreg: imm8
PSRLDQ—Shift Double Quadword Right Logical	

Table B-27 Formats and Encodings of SSE2 Integer Instructions (Continued)

Instruction and Format	Encoding
xmmreg, imm8	0110 0110:00001111:01110011:11 011 xmmreg: imm8
PSRL—Packed Shift Right Logical	
xmmreg1 by xmmreg2	0110 0110:0000 1111:1101 00gg: 11 xmmreg1 xmmreg2
xmmreg by memory	0110 0110:0000 1111:1101 00gg: mod xmmreg r/m
xmmreg by immediate	0110 0110:0000 1111:0111 00gg: 11 010 xmmreg: imm8
PSUBQ—Subtract Packed Quadword Integers	
mmreg2 to mmreg1	0000 1111:11111 011:11 mmreg1 mmreg2
mem to mmreg	0000 1111:1111 1011: mod mmreg r/m
xmmreg2 to xmmreg1	0110 0110:0000 1111:1111 1011:11 xmmreg1 xmmreg2
mem to xmmreg	0110 0110:0000 1111:1111 1011: mod xmmreg r/m
PSUB—Subtract With Wrap-around	
xmmreg2 from xmmreg1	0110 0110:0000 1111:1111 10gg: 11 xmmreg1 xmmreg2
memory from xmmreg	0110 0110:0000 1111:1111 10gg: mod xmmreg r/m
PSUBS—Subtract Signed With Saturation	
xmmreg2 from xmmreg1	0110 0110:0000 1111:1110 10gg: 11 xmmreg1 xmmreg2
memory from xmmreg	0110 0110:0000 1111:1110 10gg: mod xmmreg r/m
PSUBUS—Subtract Unsigned With Saturation	
xmmreg2 from xmmreg1	0000 1111:1101 10gg: 11 xmmreg1 xmmreg2
memory from xmmreg	0000 1111:1101 10gg: mod xmmreg r/m
PUNPCKH—Unpack High Data To Next Larger Type	
xmmreg2 to xmmreg1	0110 0110:0000 1111:0110 10gg:11 xmmreg1 xmmreg2
mem to xmmreg	0110 0110:0000 1111:0110 10gg: mod xmmreg r/m
PUNPCKHQDQ—Unpack High Data	
xmmreg2 to xmmreg1	0110 0110:0000 1111:0110 1101:11 xmmreg1 xmmreg2
mem to xmmreg	0110 0110:0000 1111:0110 1101: mod xmmreg r/m
PUNPCKL—Unpack Low Data To Next Larger Type	

Table B-27 Formats and Encodings of SSE2 Integer Instructions (Continued)

Instruction and Format	Encoding
xmmreg2 to xmmreg1	0110 0110:0000 1111:0110 00gg:11 xmmreg1 xmmreg2
mem to xmmreg	0110 0110:0000 1111:0110 00gg: mod xmmreg r/m
PUNPCKLQDQ—Unpack Low Data	
xmmreg2 to xmmreg1	0110 0110:0000 1111:0110 1100:11 xmmreg1 xmmreg2
mem to xmmreg	0110 0110:0000 1111:0110 1100: mod xmmreg r/m
PXOR—Bitwise Xor	
xmmreg2 to xmmreg1	0110 0110:0000 1111:1110 1111: 11 xmmreg1 xmmreg2
memory to xmmreg	0110 0110:0000 1111:1110 1111: mod xmmreg r/m

Table B-28 Format and Encoding of SSE2 Cacheability Instructions

Instruction and Format	Encoding
MASKMOVDQU—Store Selected Bytes of Double Quadword	
xmmreg2 to xmmreg1	0110 0110:0000 1111:1111 0111:11 xmmreg1 xmmreg2
CLFLUSH—Flush Cache Line	
mem	0000 1111:1010 1110: mod 111 r/m
MOVNTPD—Store Packed Double-Precision Floating-Point Values Using Non-Temporal Hint	
xmmreg to mem	0110 0110:0000 1111:0010 1011: mod xmmreg r/m
MOVNTDQ—Store Double Quadword Using Non-Temporal Hint	
xmmreg to mem	0110 0110:0000 1111:1110 0111: mod xmmreg r/m
MOVNTI—Store Doubleword Using Non-Temporal Hint	
reg to mem	0000 1111:1100 0011: mod reg r/m
PAUSE—Spin Loop Hint	1111 0011:1001 0000
LFENCE—Load Fence	0000 1111:1010 1110: 11 101 000
MFENCE—Memory Fence	0000 1111:1010 1110: 11 110 000

...

B.11 SSSE3 FORMATS AND ENCODING TABLE

The tables in this section provide SSSE3 formats and encodings. Some SSSE3 instructions require a mandatory prefix (66H) as part of the three-byte opcode. These prefixes are included in the table below.

Table B-32 Formats and Encodings for SSSE3 Instructions

Instruction and Format	Encoding
PABSB—Packed Absolute Value Bytes	
mmreg2 to mmreg1	0000 1111:0011 1000: 0001 1100:11 mmreg1 mmreg2
mem to mmreg	0000 1111:0011 1000: 0001 1100: mod mmreg r/m
xmmreg2 to xmmreg1	0110 0110:0000 1111:0011 1000: 0001 1100:11 xmmreg1 xmmreg2
mem to xmmreg	0110 0110:0000 1111:0011 1000: 0001 1100: mod xmmreg r/m
PABSD—Packed Absolute Value Double Words	
mmreg2 to mmreg1	0000 1111:0011 1000: 0001 1110:11 mmreg1 mmreg2
mem to mmreg	0000 1111:0011 1000: 0001 1110: mod mmreg r/m
xmmreg2 to xmmreg1	0110 0110:0000 1111:0011 1000: 0001 1110:11 xmmreg1 xmmreg2
mem to xmmreg	0110 0110:0000 1111:0011 1000: 0001 1110: mod xmmreg r/m
PABSW—Packed Absolute Value Words	
mmreg2 to mmreg1	0000 1111:0011 1000: 0001 1101:11 mmreg1 mmreg2
mem to mmreg	0000 1111:0011 1000: 0001 1101: mod mmreg r/m
xmmreg2 to xmmreg1	0110 0110:0000 1111:0011 1000: 0001 1101:11 xmmreg1 xmmreg2
mem to xmmreg	0110 0110:0000 1111:0011 1000: 0001 1101: mod xmmreg r/m
PALIGNR—Packed Align Right	
mmreg2 to mmreg1, imm8	0000 1111:0011 1010: 0000 1111:11 mmreg1 mmreg2: imm8
mem to mmreg, imm8	0000 1111:0011 1010: 0000 1111: mod mmreg r/m: imm8
xmmreg2 to xmmreg1, imm8	0110 0110:0000 1111:0011 1010: 0000 1111:11 xmmreg1 xmmreg2: imm8
mem to xmmreg, imm8	0110 0110:0000 1111:0011 1010: 0000 1111: mod xmmreg r/m: imm8
PHADDD—Packed Horizontal Add Double Words	
mmreg2 to mmreg1	0000 1111:0011 1000: 0000 0010:11 mmreg1 mmreg2
mem to mmreg	0000 1111:0011 1000: 0000 0010: mod mmreg r/m

Table B-32 Formats and Encodings for SSSE3 Instructions (Continued)

Instruction and Format	Encoding
xmmreg2 to xmmreg1	0110 0110:0000 1111:0011 1000: 0000 0010:11 xmmreg1 xmmreg2
mem to xmmreg	0110 0110:0000 1111:0011 1000: 0000 0010: mod xmmreg r/m
PHADDSW—Packed Horizontal Add and Saturate	
mmreg2 to mmreg1	0000 1111:0011 1000: 0000 0011:11 mmreg1 mmreg2
mem to mmreg	0000 1111:0011 1000: 0000 0011: mod mmreg r/m
xmmreg2 to xmmreg1	0110 0110:0000 1111:0011 1000: 0000 0011:11 xmmreg1 xmmreg2
mem to xmmreg	0110 0110:0000 1111:0011 1000: 0000 0011: mod xmmreg r/m
PHADDW—Packed Horizontal Add Words	
mmreg2 to mmreg1	0000 1111:0011 1000: 0000 0001:11 mmreg1 mmreg2
mem to mmreg	0000 1111:0011 1000: 0000 0001: mod mmreg r/m
xmmreg2 to xmmreg1	0110 0110:0000 1111:0011 1000: 0000 0001:11 xmmreg1 xmmreg2
mem to xmmreg	0110 0110:0000 1111:0011 1000: 0000 0001: mod xmmreg r/m
PHSUBD—Packed Horizontal Subtract Double Words	
mmreg2 to mmreg1	0000 1111:0011 1000: 0000 0110:11 mmreg1 mmreg2
mem to mmreg	0000 1111:0011 1000: 0000 0110: mod mmreg r/m
xmmreg2 to xmmreg1	0110 0110:0000 1111:0011 1000: 0000 0110:11 xmmreg1 xmmreg2
mem to xmmreg	0110 0110:0000 1111:0011 1000: 0000 0110: mod xmmreg r/m
PHSUBSW—Packed Horizontal Subtract and Saturate	
mmreg2 to mmreg1	0000 1111:0011 1000: 0000 0111:11 mmreg1 mmreg2
mem to mmreg	0000 1111:0011 1000: 0000 0111: mod mmreg r/m
xmmreg2 to xmmreg1	0110 0110:0000 1111:0011 1000: 0000 0111:11 xmmreg1 xmmreg2
mem to xmmreg	0110 0110:0000 1111:0011 1000: 0000 0111: mod xmmreg r/m
PHSUBW—Packed Horizontal Subtract Words	
mmreg2 to mmreg1	0000 1111:0011 1000: 0000 0101:11 mmreg1 mmreg2
mem to mmreg	0000 1111:0011 1000: 0000 0101: mod mmreg r/m

Table B-32 Formats and Encodings for SSSE3 Instructions (Continued)

Instruction and Format	Encoding
xmmreg2 to xmmreg1	0110 0110:0000 1111:0011 1000: 0000 0101:11 xmmreg1 xmmreg2
mem to xmmreg	0110 0110:0000 1111:0011 1000: 0000 0101: mod xmmreg r/m
PMADDUBSW—Multiply and Add Packed Signed and Unsigned Bytes	
mmreg2 to mmreg1	0000 1111:0011 1000: 0000 0100:11 mmreg1 mmreg2
mem to mmreg	0000 1111:0011 1000: 0000 0100: mod mmreg r/m
xmmreg2 to xmmreg1	0110 0110:0000 1111:0011 1000: 0000 0100:11 xmmreg1 xmmreg2
mem to xmmreg	0110 0110:0000 1111:0011 1000: 0000 0100: mod xmmreg r/m
PMULHRSW—Packed Multiply HIgn with Round and Scale	
mmreg2 to mmreg1	0000 1111:0011 1000: 0000 1011:11 mmreg1 mmreg2
mem to mmreg	0000 1111:0011 1000: 0000 1011: mod mmreg r/m
xmmreg2 to xmmreg1	0110 0110:0000 1111:0011 1000: 0000 1011:11 xmmreg1 xmmreg2
mem to xmmreg	0110 0110:0000 1111:0011 1000: 0000 1011: mod xmmreg r/m
PSHUFB—Packed Shuffle Bytes	
mmreg2 to mmreg1	0000 1111:0011 1000: 0000 0000:11 mmreg1 mmreg2
mem to mmreg	0000 1111:0011 1000: 0000 0000: mod mmreg r/m
xmmreg2 to xmmreg1	0110 0110:0000 1111:0011 1000: 0000 0000:11 xmmreg1 xmmreg2
mem to xmmreg	0110 0110:0000 1111:0011 1000: 0000 0000: mod xmmreg r/m
PSIGNB—Packed Sign Bytes	
mmreg2 to mmreg1	0000 1111:0011 1000: 0000 1000:11 mmreg1 mmreg2
mem to mmreg	0000 1111:0011 1000: 0000 1000: mod mmreg r/m
xmmreg2 to xmmreg1	0110 0110:0000 1111:0011 1000: 0000 1000:11 xmmreg1 xmmreg2
mem to xmmreg	0110 0110:0000 1111:0011 1000: 0000 1000: mod xmmreg r/m
PSIGND—Packed Sign Double Words	
mmreg2 to mmreg1	0000 1111:0011 1000: 0000 1010:11 mmreg1 mmreg2
mem to mmreg	0000 1111:0011 1000: 0000 1010: mod mmreg r/m
xmmreg2 to xmmreg1	0110 0110:0000 1111:0011 1000: 0000 1010:11 xmmreg1 xmmreg2
mem to xmmreg	0110 0110:0000 1111:0011 1000: 0000 1010: mod xmmreg r/m

Table B-32 Formats and Encodings for SSSE3 Instructions (Continued)

Instruction and Format	Encoding
PSIGNW—Packed Sign Words	
mmreg2 to mmreg1	0000 1111:0011 1000: 0000 1001:11 mmreg1 mmreg2
mem to mmreg	0000 1111:0011 1000: 0000 1001: mod mmreg r/m
xmmreg2 to xmmreg1	0110 0110:0000 1111:0011 1000: 0000 1001:11 xmmreg1 xmmreg2
mem to xmmreg	0110 0110:0000 1111:0011 1000: 0000 1001: mod xmmreg r/m

...

B.14 SSE4.1 FORMATS AND ENCODING TABLE

The tables in this section provide SSE4.1 formats and encodings. Some SSE4.1 instructions require a mandatory prefix (66H, F2H, F3H) as part of the three-byte opcode. These prefixes are included in the tables.

In 64-bit mode, some instructions requires REX.W, the byte sequence of REX.W prefix in the opcode sequence is shown.

Table B-35 Encodings of SSE4.1 instructions

Instruction and Format	Encoding
BLENDPD — Blend Packed Double-Precision Floats	
xmmreg2 to xmmreg1	0110 0110:0000 1111:0011 1010: 0000 1101:11 xmmreg1 xmmreg2
mem to xmmreg	0110 0110:0000 1111:0011 1010: 0000 1101: mod xmmreg r/m
BLENDPS — Blend Packed Single-Precision Floats	
xmmreg2 to xmmreg1	0110 0110:0000 1111:0011 1010: 0000 1100:11 xmmreg1 xmmreg2
mem to xmmreg	0110 0110:0000 1111:0011 1010: 0000 1100: mod xmmreg r/m
BLENDVPD — Variable Blend Packed Double-Precision Floats	
xmmreg2 to xmmreg1 <xmm0>	0110 0110:0000 1111:0011 1000: 0001 0101:11 xmmreg1 xmmreg2
mem to xmmreg <xmm0>	0110 0110:0000 1111:0011 1000: 0001 0101: mod xmmreg r/m
BLENDVPS — Variable Blend Packed Single-Precision Floats	
xmmreg2 to xmmreg1 <xmm0>	0110 0110:0000 1111:0011 1000: 0001 0100:11 xmmreg1 xmmreg2

Table B-35 Encodings of SSE4.1 instructions

Instruction and Format	Encoding
mem to xmmreg <xmm0>	0110 0110:0000 1111:0011 1000: 0001 0100: mod xmmreg r/m
DPPD — Packed Double-Precision Dot Products	
xmmreg2 to xmmreg1, imm8	0110 0110:0000 1111:0011 1010: 0100 0001:11 xmmreg1 xmmreg2: imm8
mem to xmmreg, imm8	0110 0110:0000 1111:0011 1010: 0100 0001: mod xmmreg r/m: imm8
DPPS — Packed Single-Precision Dot Products	
xmmreg2 to xmmreg1, imm8	0110 0110:0000 1111:0011 1010: 0100 0000:11 xmmreg1 xmmreg2: imm8
mem to xmmreg, imm8	0110 0110:0000 1111:0011 1010: 0100 0000: mod xmmreg r/m: imm8
EXTRACTPS — Extract From Packed Single-Precision Floats	
reg from xmmreg , imm8	0110 0110:0000 1111:0011 1010: 0001 0111:11 xmmreg reg: imm8
mem from xmmreg , imm8	0110 0110:0000 1111:0011 1010: 0001 0111: mod xmmreg r/m: imm8
INSERTPS — Insert Into Packed Single- Precision Floats	
xmmreg2 to xmmreg1, imm8	0110 0110:0000 1111:0011 1010: 0010 0001:11 xmmreg1 xmmreg2: imm8
mem to xmmreg, imm8	0110 0110:0000 1111:0011 1010: 0010 0001: mod xmmreg r/m: imm8
MOVNTDQA — Load Double Quadword Non-temporal Aligned	
m128 to xmmreg	0110 0110:0000 1111:0011 1000: 0010 1010:11 r/m xmmreg2
MPSADBW — Multiple Packed Sums of Absolute Difference	
xmmreg2 to xmmreg1, imm8	0110 0110:0000 1111:0011 1010: 0100 0010:11 xmmreg1 xmmreg2: imm8
mem to xmmreg, imm8	0110 0110:0000 1111:0011 1010: 0100 0010: mod xmmreg r/m: imm8
PACKUSDW — Pack with Unsigned Saturation	
xmmreg2 to xmmreg1	0110 0110:0000 1111:0011 1000: 0010 1011:11 xmmreg1 xmmreg2
mem to xmmreg	0110 0110:0000 1111:0011 1000: 0010 1011: mod xmmreg r/m

Table B-35 Encodings of SSE4.1 instructions

Instruction and Format	Encoding
PBLENDVB — Variable Blend Packed Bytes	
xmmreg2 to xmmreg1 <xmm0>	0110 0110:0000 1111:0011 1000: 0001 0000:11 xmmreg1 xmmreg2
mem to xmmreg <xmm0>	0110 0110:0000 1111:0011 1000: 0001 0000: mod xmmreg r/m
PBLENDW — Blend Packed Words	
xmmreg2 to xmmreg1, imm8	0110 0110:0000 1111:0011 1010: 0001 1110:11 xmmreg1 xmmreg2: imm8
mem to xmmreg, imm8	0110 0110:0000 1111:0011 1010: 0000 1110: mod xmmreg r/m: imm8
PCMPEQQ — Compare Packed Qword Data of Equal	
xmmreg2 to xmmreg1	0110 0110:0000 1111:0011 1000: 0010 1001:11 xmmreg1 xmmreg2
mem to xmmreg	0110 0110:0000 1111:0011 1000: 0010 1001: mod xmmreg r/m
PEXTRB — Extract Byte	
reg from xmmreg , imm8	0110 0110:0000 1111:0011 1010: 0001 0100:11 xmmreg reg: imm8
xmmreg to mem, imm8	0110 0110:0000 1111:0011 1010: 0001 0100: mod xmmreg r/m: imm8
PEXTRD — Extract DWord	
reg from xmmreg, imm8	0110 0110:0000 1111:0011 1010: 0001 0110:11 xmmreg reg: imm8
xmmreg to mem, imm8	0110 0110:0000 1111:0011 1010: 0001 0110: mod xmmreg r/m: imm8
PEXTRQ — Extract QWord	
r64 from xmmreg, imm8	0110 0110:REX.W:0000 1111:0011 1010: 0001 0110:11 xmmreg reg: imm8
m64 from xmmreg, imm8	0110 0110:REX.W:0000 1111:0011 1010: 0001 0110: mod xmmreg r/m: imm8
PEXTRW — Extract Word	
reg from xmmreg, imm8	0110 0110:0000 1111:0011 1010: 0001 0101:11 reg xmmreg: imm8
mem from xmmreg, imm8	0110 0110:0000 1111:0011 1010: 0001 0101: mod xmmreg r/m: imm8
PHMINPOSUW — Packed Horizontal Word Minimum	
xmmreg2 to xmmreg1	0110 0110:0000 1111:0011 1000: 0100 0001:11 xmmreg1 xmmreg2

Table B-35 Encodings of SSE4.1 instructions

Instruction and Format	Encoding
mem to xmmreg	0110 0110:0000 1111:0011 1000: 0100 0001: mod xmmreg r/m
PINSRB — Extract Byte	
reg to xmmreg, imm8	0110 0110:0000 1111:0011 1010: 0010 0000:11 xmmreg reg: imm8
mem to xmmreg, imm8	0110 0110:0000 1111:0011 1010: 0010 0000: mod xmmreg r/m: imm8
PINSRD — Extract DWord	
reg to xmmreg, imm8	0110 0110:0000 1111:0011 1010: 0010 0010:11 xmmreg reg: imm8
mem to xmmreg, imm8	0110 0110:0000 1111:0011 1010: 0010 0010: mod xmmreg r/m: imm8
PINSRQ — Extract QWord	
r64 to xmmreg, imm8	0110 0110:REX.W:0000 1111:0011 1010: 0010 0010:11 xmmreg reg: imm8
m64 to xmmreg, imm8	0110 0110:REX.W:0000 1111:0011 1010: 0010 0010: mod xmmreg r/m: imm8
PMAXSB — Maximum of Packed Signed Byte Integers	
xmmreg2 to xmmreg1	0110 0110:0000 1111:0011 1000: 0011 1100:11 xmmreg1 xmmreg2
mem to xmmreg	0110 0110:0000 1111:0011 1000: 0011 1100: mod xmmreg r/m
PMAXSD — Maximum of Packed Signed Dword Integers	
xmmreg2 to xmmreg1	0110 0110:0000 1111:0011 1000: 0011 1101:11 xmmreg1 xmmreg2
mem to xmmreg	0110 0110:0000 1111:0011 1000: 0011 1101: mod xmmreg r/m
PMAXUD — Maximum of Packed Unsigned Dword Integers	
xmmreg2 to xmmreg1	0110 0110:0000 1111:0011 1000: 0011 1111:11 xmmreg1 xmmreg2
mem to xmmreg	0110 0110:0000 1111:0011 1000: 0011 1111: mod xmmreg r/m
PMAXUW — Maximum of Packed Unsigned Word Integers	
xmmreg2 to xmmreg1	0110 0110:0000 1111:0011 1000: 0011 1110:11 xmmreg1 xmmreg2
mem to xmmreg	0110 0110:0000 1111:0011 1000: 0011 1110: mod xmmreg r/m

Table B-35 Encodings of SSE4.1 instructions

Instruction and Format	Encoding
PMINSB — Minimum of Packed Signed Byte Integers	
xmmreg2 to xmmreg1	0110 0110:0000 1111:0011 1000: 0011 1000:11 xmmreg1 xmmreg2
mem to xmmreg	0110 0110:0000 1111:0011 1000: 0011 1000: mod xmmreg r/m
PMINSD — Minimum of Packed Signed Dword Integers	
xmmreg2 to xmmreg1	0110 0110:0000 1111:0011 1000: 0011 1001:11 xmmreg1 xmmreg2
mem to xmmreg	0110 0110:0000 1111:0011 1000: 0011 1001: mod xmmreg r/m
PMINUD — Minimum of Packed Unsigned Dword Integers	
xmmreg2 to xmmreg1	0110 0110:0000 1111:0011 1000: 0011 1011:11 xmmreg1 xmmreg2
mem to xmmreg	0110 0110:0000 1111:0011 1000: 0011 1011: mod xmmreg r/m
PMINUW — Minimum of Packed Unsigned Word Integers	
xmmreg2 to xmmreg1	0110 0110:0000 1111:0011 1000: 0011 1010:11 xmmreg1 xmmreg2
mem to xmmreg	0110 0110:0000 1111:0011 1000: 0011 1010: mod xmmreg r/m
PMOVSXBD — Packed Move Sign Extend - Byte to Dword	
xmmreg2 to xmmreg1	0110 0110:0000 1111:0011 1000: 0010 0001:11 xmmreg1 xmmreg2
mem to xmmreg	0110 0110:0000 1111:0011 1000: 0010 0001: mod xmmreg r/m
PMOVSXBQ — Packed Move Sign Extend - Byte to Qword	
xmmreg2 to xmmreg1	0110 0110:0000 1111:0011 1000: 0010 0010:11 xmmreg1 xmmreg2
mem to xmmreg	0110 0110:0000 1111:0011 1000: 0010 0010: mod xmmreg r/m
PMOVSXBW — Packed Move Sign Extend - Byte to Word	
xmmreg2 to xmmreg1	0110 0110:0000 1111:0011 1000: 0010 0000:11 xmmreg1 xmmreg2
mem to xmmreg	0110 0110:0000 1111:0011 1000: 0010 0000: mod xmmreg r/m

Table B-35 Encodings of SSE4.1 instructions

Instruction and Format	Encoding
PMOVSXWD — Packed Move Sign Extend - Word to Dword	
xmmreg2 to xmmreg1	0110 0110:0000 1111:0011 1000: 0010 0011:11 xmmreg1 xmmreg2
mem to xmmreg	0110 0110:0000 1111:0011 1000: 0010 0011: mod xmmreg r/m
PMOVSXWQ — Packed Move Sign Extend - Word to Qword	
xmmreg2 to xmmreg1	0110 0110:0000 1111:0011 1000: 0010 0100:11 xmmreg1 xmmreg2
mem to xmmreg	0110 0110:0000 1111:0011 1000: 0010 0100: mod xmmreg r/m
PMOVSXDQ — Packed Move Sign Extend - Dword to Qword	
xmmreg2 to xmmreg1	0110 0110:0000 1111:0011 1000: 0010 0101:11 xmmreg1 xmmreg2
mem to xmmreg	0110 0110:0000 1111:0011 1000: 0010 0101: mod xmmreg r/m
PMOVZXBD — Packed Move Zero Extend - Byte to Dword	
xmmreg2 to xmmreg1	0110 0110:0000 1111:0011 1000: 0011 0001:11 xmmreg1 xmmreg2
mem to xmmreg	0110 0110:0000 1111:0011 1000: 0011 0001: mod xmmreg r/m
PMOVZXBQ — Packed Move Zero Extend - Byte to Qword	
xmmreg2 to xmmreg1	0110 0110:0000 1111:0011 1000: 0011 0010:11 xmmreg1 xmmreg2
mem to xmmreg	0110 0110:0000 1111:0011 1000: 0011 0010: mod xmmreg r/m
PMOVZXBW — Packed Move Zero Extend - Byte to Word	
xmmreg2 to xmmreg1	0110 0110:0000 1111:0011 1000: 0011 0000:11 xmmreg1 xmmreg2
mem to xmmreg	0110 0110:0000 1111:0011 1000: 0011 0000: mod xmmreg r/m
PMOVZXWD — Packed Move Zero Extend - Word to Dword	
xmmreg2 to xmmreg1	0110 0110:0000 1111:0011 1000: 0011 0011:11 xmmreg1 xmmreg2
mem to xmmreg	0110 0110:0000 1111:0011 1000: 0011 0011: mod xmmreg r/m

Table B-35 Encodings of SSE4.1 instructions

Instruction and Format	Encoding
PMOVZXWQ — Packed Move Zero Extend - Word to Qword	
xmmreg2 to xmmreg1	0110 0110:0000 1111:0011 1000: 0011 0100:11 xmmreg1 xmmreg2
mem to xmmreg	0110 0110:0000 1111:0011 1000: 0011 0100: mod xmmreg r/m
PMOVZXDQ — Packed Move Zero Extend - Dword to Qword	
xmmreg2 to xmmreg1	0110 0110:0000 1111:0011 1000: 0011 0101:11 xmmreg1 xmmreg2
mem to xmmreg	0110 0110:0000 1111:0011 1000: 0011 0101: mod xmmreg r/m
PMULDQ — Multiply Packed Signed Dword Integers	
xmmreg2 to xmmreg1	0110 0110:0000 1111:0011 1000: 0010 1000:11 xmmreg1 xmmreg2
mem to xmmreg	0110 0110:0000 1111:0011 1000: 0010 1000: mod xmmreg r/m
PMULLD — Multiply Packed Signed Dword Integers, Store low Result	
xmmreg2 to xmmreg1	0110 0110:0000 1111:0011 1000: 0100 0000:11 xmmreg1 xmmreg2
mem to xmmreg	0110 0110:0000 1111:0011 1000: 0100 0000: mod xmmreg r/m
PTEST — Logical Compare	
xmmreg2 to xmmreg1	0110 0110:0000 1111:0011 1000: 0001 0111:11 xmmreg1 xmmreg2
mem to xmmreg	0110 0110:0000 1111:0011 1000: 0001 0111: mod xmmreg r/m
ROUNDPD — Round Packed Double-Precision Values	
xmmreg2 to xmmreg1, imm8	0110 0110:0000 1111:0011 1010: 0000 1001:11 xmmreg1 xmmreg2: imm8
mem to xmmreg, imm8	0110 0110:0000 1111:0011 1010: 0000 1001: mod xmmreg r/m: imm8
ROUNDPS — Round Packed Single-Precision Values	
xmmreg2 to xmmreg1, imm8	0110 0110:0000 1111:0011 1010: 0000 1000:11 xmmreg1 xmmreg2: imm8
mem to xmmreg, imm8	0110 0110:0000 1111:0011 1010: 0000 1000: mod xmmreg r/m: imm8
ROUNDSD — Round Scalar Double-Precision Value	

Table B-35 Encodings of SSE4.1 instructions

Instruction and Format	Encoding
xmmreg2 to xmmreg1, imm8	0110 0110:0000 1111:0011 1010: 0000 1011:11 xmmreg1 xmmreg2: imm8
mem to xmmreg, imm8	0110 0110:0000 1111:0011 1010: 0000 1011: mod xmmreg r/m: imm8
ROUNDSS — Round Scalar Single-Precision Value	
xmmreg2 to xmmreg1, imm8	0110 0110:0000 1111:0011 1010: 0000 1010:11 xmmreg1 xmmreg2: imm8
mem to xmmreg, imm8	0110 0110:0000 1111:0011 1010: 0000 1010: mod xmmreg r/m: imm8

...

13. Updates to Appendix C, Volume 2B

Change bars show changes to Appendix C of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2B:* Instruction Set Reference, N-Z.

---

...

Table C-1 Simple Intrinsics

Mnemonic	Intrinsic
...	
CVTSI2SD	__m128d _mm_cvtsi32_sd(__m128d a, int b)
CVTSI2SS	__m128 _mm_cvt_si2ss(__m128 a, int b) __m128 _mm_cvtsi32_ss(__m128 a, int b) __m128 _mm_cvtsi64_ss(__m128 a, __int64 b)
CVTSS2SD	__m128d _mm_cvtss_sd(__m128d a, __m128 b)
...	

...

14. Updates to Chapter 1, Volume 3A

Change bars show changes to Chapter 1 of the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A:* System Programming Guide, Part 1.

...

1.1 PROCESSORS COVERED IN THIS MANUAL

This manual set includes information pertaining primarily to the most recent Intel® 64 and IA-32 processors, which include:

- Pentium® processors
- P6 family processors
- Pentium® 4 processors
- Pentium® M processors
- Intel® Xeon® processors
- Pentium® D processors
- Pentium® processor Extreme Editions
- 64-bit Intel® Xeon® processors
- Intel® Core™ Duo processor
- Intel® Core™ Solo processor
- Dual-Core Intel® Xeon® processor LV
- Intel® Core™2 Duo processor
- Intel® Core™2 Quad processor Q6000 series
- Intel® Xeon® processor 3000, 3200 series
- Intel® Xeon® processor 5000 series
- Intel® Xeon® processor 5100, 5300 series
- Intel® Core™2 Extreme processor X7000 and X6800 series
- Intel® Core™2 Extreme QX6000 series
- Intel® Xeon® processor 7100 series
- Intel® Pentium® Dual-Core processor
- Intel® Xeon® processor 7200, 7300 series
- Intel® Core™2 Extreme QX9000 series
- Intel® Xeon® processor 5200, 5400, 7400 series
- Intel® CoreTM2 Extreme processor QX9000 and X9000 series
- Intel® CoreTM2 Quad processor Q9000 series
- Intel® CoreTM2 Duo processor E8000, T9000 series
- Intel® AtomTM processor family
- Intel® CoreTM i7 processor
- Intel® CoreTM i5 processor
- Intel® Xeon® processor E7-8800/4800/2800 product families

P6 family processors are IA-32 processors based on the P6 family microarchitecture. This includes the Pentium® Pro, Pentium® II, Pentium® III, and Pentium® III Xeon® processors.

The Pentium® 4, Pentium® D, and Pentium® processor Extreme Editions are based on the Intel NetBurst® microarchitecture. Most early Intel® Xeon® processors are based on the Intel NetBurst® microarchitecture. Intel Xeon processor 5000, 7100 series are based on the Intel NetBurst® microarchitecture.

The Intel® Core™ Duo, Intel® Core™ Solo and dual-core Intel® Xeon® processor LV are based on an improved Pentium® M processor microarchitecture.

The Intel® Xeon® processor 3000, 3200, 5100, 5300, 7200, and 7300 series, Intel® Pentium® dual-core, Intel® Core™2 Duo, Intel® Core™2 Quad and Intel® Core™2 Extreme processors are based on Intel® Core™ microarchitecture.

The Intel® Xeon® processor 5200, 5400, 7400 series, Intel® Core™2 Quad processor Q9000 series, and Intel® Core™2 Extreme processors QX9000, X9000 series, Intel® Core™2 processor E8000 series are based on Enhanced Intel® Core™ microarchitecture.

The Intel® Atom™ processor family is based on the Intel® Atom™ microarchitecture and supports Intel 64 architecture.

The Intel® Core™i7 processor and the Intel® Core™i5 processor are based on the Intel® microarchitecture code name Nehalem and support Intel 64 architecture.

Processors based on the Intel® microarchitecture code name Westmere support Intel 64 architecture.

P6 family, Pentium® M, Intel® Core™ Solo, Intel® Core™ Duo processors, dual-core Intel® Xeon® processor LV, and early generations of Pentium 4 and Intel Xeon processors support IA-32 architecture. The Intel® Atom™ processor Z5xx series support IA-32 architecture.

The Intel® Xeon® processor E7-8800/4800/2800 product families, Intel® Xeon® processor 3000, 3200, 5000, 5100, 5200, 5300, 5400, 7100, 7200, 7300, 7400 series, Intel® Core™2 Duo, Intel® Core™2 Extreme processors, Intel Core 2 Quad processors, Pentium® D processors, Pentium® Dual-Core processor, newer generations of Pentium 4 and Intel Xeon processor family support Intel® 64 architecture.

IA-32 architecture is the instruction set architecture and programming environment for Intel's 32-bit microprocessors. Intel® 64 architecture is the instruction set architecture and programming environment which is a superset of and compatible with IA-32 architecture.

...

15. 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.

...

2.6 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 state, SSE state, AVX state, and other processor extended states that Intel 64 architecture may introduce in the future. The OS programs XCR0 to reflect the features it supports.

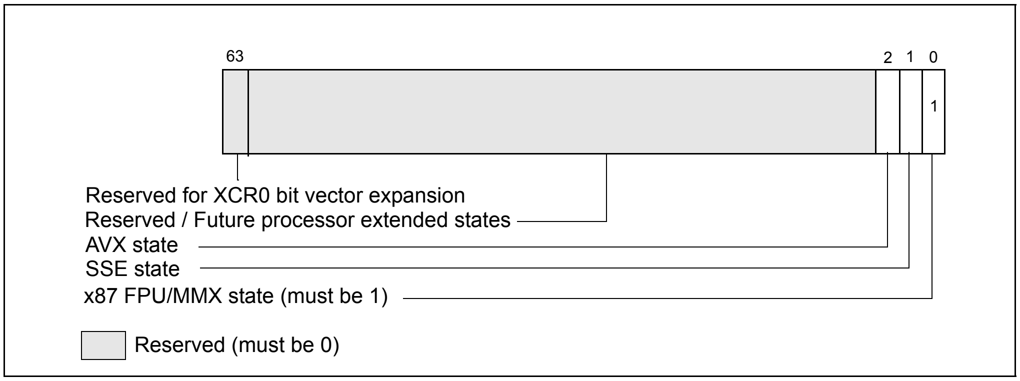
63
2
1
0
1
Reserved for XCR0 bit vector expansion
Reserved / Future processor extended states
AVX state
SSE state
x87 FPU/MMX state (must be 1)
Reserved (must be 0)

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, AVX 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 three processor states defined, with up to 61 bits reserved for future processor extended states:

- XCR0.X87 (bit 0): This bit 0 must be 1. An attempt to write 0 to this bit causes a #GP exception.
- XCR0.SSE (bit 1): If 1, XSAVE, XSAVEOPT, and XRSTOR can be used to manage MXCSR and XMM registers (XMM0-XMM15 in 64-bit mode; otherwise XMM0-XMM7).
- XCR0.AVX (bit 2): If 1, AVX instructions can be executed and XSAVE, XSAVEOPT, and XRSTOR can be used to manage the upper halves of the YMM registers (YMM0-YMM15 in 64-bit mode; otherwise YMM0-YMM7).

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. An attempt to write 0 to XCR0.SSE (bit 1) and 1 to XCR0.AVX (bit 2) also results in a #GP exception.

If a bit in XCR0 is 1, software can use the XSAVE instruction to save the corresponding processor state 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.

...

16. 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.

...

Table 4-19 Format of an IA-32e Page-Table Entry that Maps a 4-KByte Page

Bit Position(s)	Contents
0 (P)	Present; must be 1 to map a 4-KByte page
1 (R/W)	Read/write; if 0, writes may not be allowed to the 4-KByte page referenced by this entry (depends on CPL and CR0.WP; see Section 4.6)
2 (U/S)	User/supervisor; if 0, accesses with CPL=3 are not allowed to the 4-KByte page referenced by this entry (see Section 4.6)
3 (PWT)	Page-level write-through; indirectly determines the memory type used to access the 4-KByte page referenced by this entry (see Section 4.9.2)
4 (PCD)	Page-level cache disable; indirectly determines the memory type used to access the 4-KByte page referenced by this entry (see Section 4.9.2)
5 (A)	Accessed; indicates whether software has accessed the 4-KByte page referenced by this entry (see Section 4.8)
6 (D)	Dirty; indicates whether software has written to the 4-KByte page referenced by this entry (see Section 4.8)
7 (PAT)	Indirectly determines the memory type used to access the 4-KByte page referenced by this entry (see Section 4.9.2)
8 (G)	Global; if CR4.PGE = 1, determines whether the translation is global (see Section 4.10); ignored otherwise
11:9	Ignored
(M–1):12	Physical address of the 4-KByte page referenced by this entry
51:M	Reserved (must be 0)
62:52	Ignored
63 (XD)	If IA32_EFER.NXE = 1, execute-disable (if 1, instruction fetches are not allowed from the 4-KByte page controlled by this entry; see Section 4.6); otherwise, reserved (must be 0)

...

17. 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.

...

7.2.4 Task Register

The task register holds the 16-bit segment selector and the entire segment descriptor (32-bit base address (64 bits in IA-32e mode), 16-bit segment limit, and descriptor attributes) for the TSS of the current task (see Figure 2-5). This information is copied from the TSS descriptor in the GDT for the current task. Figure 7-5 shows the path the processor uses to access the TSS (using the information in the task register).

The task register has a visible part (that can be read and changed by software) and an invisible part (maintained by the processor and is inaccessible by software). The segment selector in the visible portion points to a TSS descriptor in the GDT. The processor uses the invisible portion of the task register to cache the segment descriptor for the TSS. Caching these values in a register makes execution of the task more efficient. The LTR (load task register) and STR (store task register) instructions load and read the visible portion of the task register:

The LTR instruction loads a segment selector (source operand) into the task register that points to a TSS descriptor in the GDT. It then loads the invisible portion of the task register with information from the TSS descriptor. LTR is a privileged instruction that may be executed only when the CPL is 0. It's used during system initialization to put an initial value in the task register. Afterwards, the contents of the task register are changed implicitly when a task switch occurs.

The STR (store task register) instruction stores the visible portion of the task register in a general-purpose register or memory. This instruction can be executed by code running at any privilege level in order to identify the currently running task. However, it is normally used only by operating system software.

On power up or reset of the processor, segment selector and base address are set to the default value of 0; the limit is set to FFFFH.

...

18. 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.

...

10.5.4.1 TSC-Deadline Mode

The mode of operation of the local-APIC timer is determined by the LVT Timer Register. Specifically, if CPUID.01H:ECX.TSC_Deadline[bit 24] = 0, the mode is determined by bit

17 of the register; if CPUID.01H:ECX.TSC_Deadline[bit 24] = 1, the mode is determined by bits 18:17. See Figure 10-8. (If CPUID.01H:ECX.TSC_Deadline[bit 24] = 0, bit 18 of the register is reserved.) A write to the LVT Timer Register that changes the timer mode disarms the local APIC timer. The supported timer modes are given in Table 10-2. The three modes of the local APIC timer are mutually exclusive.

Table 10-2 Local APIC Timer Modes

LVT Bits [18:17]	Timer Mode
00b	One-shot mode, program count-down value in an initial-count register. See Section 10.5.4
01b	Periodic mode, program interval value in an initial-count register. See Section 10.5.4
10b	TSC-Deadline mode, program target value in IA32_TSC_DEADLINE MSR.
11b	Reserved

The TSC-deadline mode allows software to use local APIC timer to single interrupt at an absolute time. In TSC-deadline mode, writes to the initial-count register are ignored; and current-count register always reads 0. Instead, timer behavior is controlled using the IA32_TSC_DEADLINE MSR.

The IA32_TSC_DEADLINE MSR (MSR address 6E0H) is a per-logical processor MSR that specifies the time at which a timer interrupt should occur. Writing a non-zero 64-bit value into IA32_TSC_DEADLINE arms the timer. An interrupt is generated when the logical processor's time-stamp counter equals or exceeds the target value in the IA32_TSC_DEADLINE MSR.[1] When the timer generates an interrupt, it disarms itself and clears the IA32_TSC_DEADLINE MSR. Thus, each write to the IA32_TSC_DEADLINE MSR generates at most one timer interrupt.

...

10.11 MESSAGE SIGNALLED INTERRUPTS

The *PCI Local Bus Specification, Rev 2.2* (www.pcisig.com) introduces the concept of message signalled interrupts. As the specification indicates:

> "Message signalled interrupts (MSI) is an optional feature that enables PCI devices to request service by writing a system-specified message to a system-specified address (PCI DWORD memory write transaction). The transaction address specifies the message destination while the transaction data specifies the message. System software is expected to initialize the message destination and message during device configuration, allocating one or more non-shared messages to each MSI capable function."

The capabilities mechanism provided by the *PCI Local Bus Specification* is used to identify and configure MSI capable PCI devices. Among other fields, this structure contains a

[1] If the logical processor is in VMX non-root operation, a read of the time-stamp counter (using either RDMSR, RDTSC, or RDTSCP) may not return the actual value of the time-stamp counter; see Chapter 22 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3B*. It is the responsibility of software operating in VMX root operation to coordinate the virtualization of the time-stamp counter and the IA32_TSC_DEADLINE MSR.

Message Data Register and a Message Address Register. To request service, the PCI device function writes the contents of the Message Data Register to the address contained in the Message Address Register (and the Message Upper Address register for 64-bit message addresses).

Section 10.11.1 and Section 10.11.2 provide layout details for the Message Address Register and the Message Data Register. The operation issued by the device is a PCI write command to the Message Address Register with the Message Data Register contents. The operation follows semantic rules as defined for PCI write operations and is a DWORD operation.

...

10.12.6 Routing of Device Interrupts in x2APIC Mode

The x2APIC architecture is intended to work with all existing IOxAPIC units as well as all PCI and PCI Express (PCIe) devices that support the capability for message-signaled interrupts (MSI). Support for x2APIC modifies only the following:

- the local APIC units;
- the interconnects joining IOxAPIC units to the local APIC units; and
- the interconnects joining MSI-capable PCI and PCIe devices to the local APIC units.

No modifications are required to MSI-capable PCI and PCIe devices. Similarly, no modifications are required to IOxAPIC units. This made possible through use of the interrupt-remapping architecture specified in the *Intel® Virtualization Technology for Directed I/O*, Revision 1.3 for the routing of interrupts from MSI-capable devices to local APIC units operating in x2APIC mode.

10.12.7 Initialization by System Software

Routing of device interrupts to local APIC units operating in x2APIC mode requires use of the interrupt-remapping architecture specified in the *Intel® Virtualization Technology for Directed I/O*, Revision 1.3. Because of this, BIOS must enumerate support for and software must enable this interrupt remapping with Extended Interrupt Mode Enabled before it enabling x2APIC mode in the local APIC units.

The ACPI interfaces for the x2APIC are described in Section 5.2, "ACPI System Description Tables," of the *Advanced Configuration and Power Interface Specification*, Revision 4.0a (http://www.acpi.info/spec.htm). The default behavior for BIOS is to pass the control to the operating system with the local x2APICs in xAPIC mode if all APIC IDs reported by CPUID.0BH:EDX are less than 255, and in x2APIC mode if there are any logical processor reporting an APIC ID of 255 or greater.

...

19. 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.

...

13.1.5 Providing Non-Numeric Exception Handlers for Exceptions Generated by the SSE/SSE2/SSE3/SSSE3/SSE4 Instructions

SSE/SSE2/SSE3/SSSE3/SSE4 instructions can generate the same type of memory access exceptions (such as, page fault, segment not present, and limit violations) and other non-numeric exceptions as other Intel 64 and IA-32 architecture instructions generate.

Ordinarily, existing exception handlers can handle these and other non-numeric exceptions without code modification. However, depending on the mechanisms used in existing exception handlers, some modifications might need to be made.

The SSE/SSE2/SSE3/SSSE3/SSE4 extensions can generate the non-numeric exceptions listed below:

- Memory Access Exceptions:
  - Invalid opcode (#UD).
  - Stack-segment fault (#SS).
  - General protection (#GP). Executing most SSE/SSE2/SSE3 instructions with an unaligned 128-bit memory reference generates a general-protection exception. (The MOVUPS and MOVUPD instructions allow unaligned a loads or stores of 128-bit memory locations, without generating a general-protection exception.) A 128-bit reference within the stack segment that is not aligned to a 16-byte boundary will also generate a general-protection exception, instead a stack-segment fault exception (#SS).
  - Page fault (#PF).
  - Alignment check (#AC). When enabled, this type of alignment check operates on operands that are less than 128-bits in size: 16-bit, 32-bit, and 64-bit. To enable the generation of alignment check exceptions, do the following:
    - Set the AM flag (bit 18 of control register CR0)
    - Set the AC flag (bit 18 of the EFLAGS register)
    - CPL must be 3

    If alignment check exceptions are enabled, 16-bit, 32-bit, and 64-bit misalignment will be detected for the MOVUPD and MOVUPS instructions; detection of 128-bit misalignment is not guaranteed and may vary with implementation.
- System Exceptions:
  - Invalid-opcode exception (#UD). This exception is generated when executing SSE/SSE2/SSE3/SSSE3 instructions under the following conditions:
    - SSE/SSE2/SSE3/SSSE3/SSE4_1/SSE4_2 feature flags returned by CPUID are set to 0. This condition does not affect the CLFLUSH instruction, nor POPCNT.
    - The CLFSH feature flag returned by the CPUID instruction is set to 0. This exception condition only pertains to the execution of the CLFLUSH instruction.
    - The POPCNT feature flag returned by the CPUID instruction is set to 0. This exception condition only pertains to the execution of the POPCNT instruction.
    - The EM flag (bit 2) in control register CR0 is set to 1, regardless of the value of TS flag (bit 3) of CR0. This condition does not affect the PAUSE, PREFETCH*h*, MOVNTI, SFENCE, LFENCE, MFENCE, CLFLUSH, CRC32 and POPCNT instructions.

- The OSFXSR flag (bit 9) in control register CR4 is set to 0. This condition does not affect the PSHUFW, MOVNTQ, MOVNTI, PAUSE, PREFETCH*h*, SFENCE, LFENCE, MFENCE, CLFLUSH, CRC32 and POPCNT instructions.
- Executing a instruction that causes a SIMD floating-point exception when the OSXMMEXCPT flag (bit 10) in control register CR4 is set to 0. See Section 13.5.1, "Using the TS Flag to Control the Saving of the x87 FPU, MMX, SSE, SSE2, SSE3 SSSE3 and SSE4 State."

— Device not available (#NM). This exception is generated by executing a SSE/SSE2/SSE3/SSSE3/SSE4 instruction when the TS flag (bit 3) of CR0 is set to 1.

Other exceptions can occur indirectly due to faulty execution of the above exceptions.

...

13.10.6.1 XSAVEOPT Usage Guidelines

When using the XSAVEOPT facility, software must be aware of the guidelines outlined in Chapter 4, "XSAVEOPT—Save Processor Extended States Optimized" in *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2B*.

...

20. 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.

...

Example 15-5 Corrected Error Handler Pseudocode with UCR Support

```
Corrected Error HANDLER:  (* Called from CMCI handler or OS CMC Polling Dispatcher*)
IF CPU supports MCA
    THEN
        FOR each bank of machine-check registers
            DO
                READ IA32_MCi_STATUS;
                IF VAL flag in IA32_MCi_STATUS = 1
                    THEN
                        IF UC Flag in IA32_MCi_STATUS = 0 (* It is a corrected error *)
                            THEN
                                GOTO LOG CMC ERROR;
                            ELSE
                                IF Bit 24 in IA32_MCG_CAP = 0
                                    THEN
                                        GOTO CONTINUE;
                                FI;
                                IF S Flag in IA32_MCi_STATUS = 0 AND AR Flag in IA32_MCi_STATUS = 0
                                    THEN (* It is a uncorrected no action required error *)
                                        GOTO LOG CMC ERROR
                                FI
                                IF EN Flag in IA32_MCi_STATUS = 0
                                    THEN (* It is a spurious MCA error *)
                                        GOTO LOG CMC ERROR
                                FI;
                        FI;
                FI;
                GOTO CONTINUE;
            LOG CMC ERROR:
                SAVE IA32_MCi_STATUS;
                If MISCV Flag in IA32_MCi_STATUS
                    THEN
                        SAVE IA32_MCi_MISC;
                        SET all 0 to IA32_MCi_MISC;
                FI;
                IF ADDRV Flag in IA32_MCi_STATUS
                    THEN
                        SAVE IA32_MCi_ADDR;
                        SET all 0 to IA32_MCi_ADDR
                FI;
                SET all 0 to IA32_MCi_STATUS;
                CONTINUE:
            OD;
    ( *END FOR *)
FI;
```

...

21. 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.

...

Intel 64 and IA-32 architectures provide debug facilities for use in debugging code and monitoring performance. These facilities are valuable for debugging application software, system software, and multitasking operating systems. Debug support is accessed using debug registers (DR0 through DR7) and model-specific registers (MSRs):

- Debug registers hold the addresses of memory and I/O locations called breakpoints. Breakpoints are user-selected locations in a program, a data-storage area in memory, or specific I/O ports. They are set where a programmer or system designer wishes to halt execution of a program and examine the state of the processor by invoking debugger software. A debug exception (#DB) is generated when a memory or I/O access is made to a breakpoint address.
- MSRs monitor branches, interrupts, and exceptions; they record addresses of the last branch, interrupt or exception taken and the last branch taken before an interrupt or exception.

...

16.3.1.1 Instruction-Breakpoint Exception Condition

The processor reports an instruction breakpoint when it attempts to execute an instruction at an address specified in a breakpoint-address register (DR0 through DR3) that has been set up to detect instruction execution (R/W flag is set to 0). Upon reporting the instruction breakpoint, the processor generates a fault-class, debug exception (#DB) before it executes the target instruction for the breakpoint.

Instruction breakpoints are the highest priority debug exceptions. They are serviced before any other exceptions detected during the decoding or execution of an instruction.

However, if a code instruction breakpoint is placed on an instruction located immediately after a POP SS/MOV SS instruction, the breakpoint may not be triggered. In most situations, POP SS/MOV SS will inhibit such interrupts (see “MOV—Move” and “POP—Pop a Value from the Stack” in Chapters 3 and 4 of the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volumes 2A & 2B*).

...

16.3.1.2 Data Memory and I/O Breakpoint Exception Conditions

Data memory and I/O breakpoints are reported when the processor attempts to access a memory or I/O address specified in a breakpoint-address register (DR0 through DR3) that has been set up to detect data or I/O accesses (R/W flag is set to 1, 2, or 3). The processor generates the exception after it executes the instruction that made the access, so these breakpoint condition causes a trap-class exception to be generated.

Because data breakpoints are traps, the original data is overwritten before the trap exception is generated. If a debugger needs to save the contents of a write breakpoint location, it should save the original contents before setting the breakpoint. The handler can report the saved value after the breakpoint is triggered. The address in the debug registers can be used to locate the new value stored by the instruction that triggered the breakpoint.

Intel486 and later processors ignore the GE and LE flags in DR7. In Intel386 processors, exact data breakpoint matching does not occur unless it is enabled by setting the LE and/or the GE flags.

P6 family processors are unable to report data breakpoints exactly for the REP MOVS and REP STOS instructions until the completion of the iteration after the iteration in which the breakpoint occurred.

For repeated INS and OUTS instructions that generate an I/O-breakpoint debug exception, the processor generates the exception after the completion of the first iteration. Repeated INS and OUTS instructions generate a memory-breakpoint debug exception after the iteration in which the memory address breakpoint location is accessed.

...

16.4.4 Branch Trace Messages

Setting the TR flag (bit 6) in the IA32_DEBUGCTL MSR enables branch trace messages (BTMs). Thereafter, when the processor detects a branch, exception, or interrupt, it sends a branch record out on the system bus as a BTM. A debugging device that is monitoring the system bus can read these messages and synchronize operations with taken branch, interrupt, and exception events.

When interrupts or exceptions occur in conjunction with a taken branch, additional BTMs are sent out on the bus, as described in Section 16.4.2, “Monitoring Branches, Exceptions, and Interrupts.”

For P6 processor family, Pentium M processor family, processors based on Intel Core microarchitecture, TR and LBR bits can not be set at the same time due to hardware limitation. The content of LBR stack is undefined when TR is set.

For IA processor families based on Intel NetBurst microarchitecture, Intel microarchitecture code name Nehalem and Intel Atom processor family, the processor can collect branch records in the LBR stack and at the same time send/store BTMs when both the TR and LBR flags are set in the IA32_DEBUGCTL MSR (or the equivalent MSR_DEBUGCTLA, MSR_DEBUGCTLB).

The following exception applies:

- BTM may not be observable on Intel Atom processor family processors that do not provide an externally visible system bus.

...

16.6.1 LBR Stack

Processors based on Intel microarchitecture code name Nehalem provide 16 pairs of MSR to record last branch record information. The layout of each MSR pair is shown in Table 16-6 and Table 16-7.

Table 16-6 IA32_LASTBRANCH_x_FROM_IP

Bit Field	Bit Offset	Access	Description
Data	47:0	R/O	The linear address of the branch instruction itself, This is the "branch from" address
SIGN_EXt	62:48	R/O	Signed extension of bit 47 of this register
MISPRED	63	R/O	When set, indicates the branch was predicted; otherwise, the branch was mispredicted.

Table 16-7 IA32_LASTBRANCH_x_TO_IP

Bit Field	Bit Offset	Access	Description
Data	47:0	R/O	The linear address of the target of the branch instruction itself, This is the "branch to" address
SIGN_EXt	63:48	R/O	Signed extension of bit 47 of this register

...

22. Updates to Chapter 19, Volume 3A

Change bars show changes to Chapter 19 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A:* System Programming Guide, Part 1.

...

19.25.3 Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers

MMX instructions and a subset of SSE, SSE2, SSSE3 instructions operate on MMX registers. The exception conditions of these instructions are described in the following tables.

Table 19-4 Exception Conditions for Legacy SIMD/MMX Instructions with FP Exception and 16-Byte Alignment

Exception	Real	Virtual 8086	Protected and Compatibility	64-bit	Cause of Exception
Invalid Opcode, #UD	X	X	X	X	If an unmasked SIMD floating-point exception and CR4.OSXMMEXCPT[bit 10] = 0.
	X	X	X	X	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	X	X	If any corresponding CPUID feature flag is ‘0’
#MF	X	X	X	X	If there is a pending X87 FPU exception
#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
#PF(fault-code)		X	X	X	For a page fault
#XM	X	X	X	X	If an unmasked SIMD floating-point exception and CR4.OSXMMEXCPT[bit 10] = 1
Applicable Instructions	CVTPD2PI, CVTTPD2PI				

Table 19-5 Exception Conditions for Legacy SIMD/MMX Instructions with XMM and FP Exception

Exception	Real	Virtual 80x86	Protected and Compatibility	64-bit	Cause of Exception
Invalid Opcode, #UD	X	X	X	X	If an unmasked SIMD floating-point exception and CR4.OSXMMEXCPT[bit 10] = 0.
	X	X	X	X	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	X	X	If any corresponding CPUID feature flag is '0'
#MF	X	X	X	X	If there is a pending X87 FPU exception
#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
#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
Applicable Instructions	CVTPI2PS, CVTPS2PI, CVTTPS2PI				

Table 19-6 Exception Conditions for Legacy SIMD/MMX Instructions with XMM and without FP Exception

Exception	Real	Virtual 80x86	Protected and Compatibility	64-bit	Cause of Exception
Invalid Opcode, #UD	X	X	X	X	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	X	X	If any corresponding CPUID feature flag is '0'
#MF[1]	X	X	X	X	If there is a pending X87 FPU exception
#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
#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.
Applicable Instructions	CVTPI2PD				

NOTES:

[1] Applies to "CVTPI2PD xmm, mm" but not "CVTPI2PD xmm, m64".

Table 19-7 Exception Conditions for SIMD/MMX Instructions with Memory Reference

Exception	Real	Virtual 80x86	Protected and Compatibility	64-bit	Cause of Exception
Invalid Opcode, #UD	X	X	X	X	If CR0.EM[bit 2] = 1.
	X	X	X	X	If preceded by a LOCK prefix (F0H)
	X	X	X	X	If any corresponding CPUID feature flag is ‘0’
#MF	X	X	X	X	If there is a pending X87 FPU exception
#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
#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.
Applicable Instructions	PABSB, PABSD, PABSW, PACKSSWB, PACKSSDW, PACKUSWB, PADDB, PADDD, PADDQ, PADDW, PADDSB, PADDSW, PADDUSB, PADDUSW, PALIGNR, PAND, PANDN, PAVGB, PAVGW, PCMPEQB, PCMPEQD, PCMPEQW, PCMPGTB, PCMPGTD, PCMPGTW, PHADDD, PHADDW, PHADDSW, PHSUBD, PHSUBW, PHSUBSW, PINSRW, PMADDUBSW, PMADDWD, PMAXSW, PMAXUB, PMINSW, PMINUB, PMULHRSW, PMULHUW, PMULHW, PMULLW, PMULUDQ, PSADBW, PSHUFB, PSHUFW, PSIGNB PSIGND PSIGNW, PSLLW, PSLLD, PSLLQ, PSRAD, PSRAW, PSRLW, PSRLD, PSRLQ, PSUBB, PSUBD, PSUBQ, PSUBW, PSUBSB, PSUBSW, PSUBUSB, PSUBUSW, PUNPCKHBW, PUNPCKHWD, PUNPCKHDQ, PUNPCKLBW, PUNPCKLWD, PUNPCKLDQ, PXOR				

Table 19-8 Exception Conditions for Legacy SIMD/MMX Instructions without FP Exception

Exception	Real	Virtual 80x86	Protected and Compatibility	64-bit	Cause of Exception
Invalid Opcode, #UD	X	X	X	X	If CR0.EM[bit 2] = 1. If ModR/M.mod != 11b[1]
	X	X	X	X	If preceded by a LOCK prefix (F0H)
	X	X	X	X	If any corresponding CPUID feature flag is '0'
#MF	X	X	X	X	If there is a pending X87 FPU exception
#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
#GP(0)			X		For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments. If the destination operand is in a non-writable segment.[2] If the DS, ES, FS, or GS register contains a NULL segment selector.[3]
				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
#PF(fault-code)		X	X	X	For a page fault
#AC(0)		X	X	X	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
Applicable Instructions	MASKMOVQ, MOVNTQ, "MOVQ (mmreg)"				

NOTES:

1. Applies to MASKMOVQ only.
2. Applies to MASKMOVQ and MOVQ (mmreg) only.
3. Applies to MASKMOVQ only.

Table 19-9 Exception Conditions for Legacy SIMD/MMX Instructions without Memory Reference

Exception	Real	Virtual 80x86	Protected and Compatibility	64-bit	Cause of Exception
Invalid Opcode, #UD	X	X	X	X	If CR0.EM[bit 2] = 1.
	X	X	X	X	If preceded by a LOCK prefix (F0H)
	X	X	X	X	If any corresponding CPUID feature flag is '0'
#MF	X	X	X	X	If there is a pending X87 FPU exception
#NM			X	X	If CR0.TS[bit 3]=1
Applicable Instructions	PEXTRW, PMOVMSKB				

...

23. 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.

...

22.3 OTHER CAUSES OF VM EXITS

In addition to VM exits caused by instruction execution, the following events can cause VM exits:

- **Exceptions.** Exceptions (faults, traps, and aborts) cause VM exits based on the exception bitmap (see Section 21.6.3). If an exception occurs, its vector (in the range 0–31) is used to select a bit in the exception bitmap. If the bit is 1, a VM exit occurs; if the bit is 0, the exception is delivered normally through the guest IDT. This use of the exception bitmap applies also to exceptions generated by the instructions INT3, INTO, BOUND, and UD2.

  Page faults (exceptions with vector 14) are specially treated. When a page fault occurs, a logical processor consults (1) bit 14 of the exception bitmap; (2) the error code produced with the page fault [PFEC]; (3) the page-fault error-code mask field [PFEC_MASK]; and (4) the page-fault error-code match field [PFEC_MATCH]. It checks if PFEC & PFEC_MASK = PFEC_MATCH. If there is equality, the specification of bit 14 in the exception bitmap is followed (for example, a VM exit occurs if that bit is set). If there is inequality, the meaning of that bit is reversed (for example, a VM exit occurs if that bit is clear).

Thus, if software desires VM exits on all page faults, it can set bit 14 in the exception bitmap to 1 and set the page-fault error-code mask and match fields each to 00000000H. If software desires VM exits on no page faults, it can set bit 14 in the exception bitmap to 1, the page-fault error-code mask field to 00000000H, and the page-fault error-code match field to FFFFFFFFH.

- **Triple fault.** A VM exit occurs if the logical processor encounters an exception while attempting to call the double-fault handler and that exception itself does not cause a VM exit due to the exception bitmap. This applies to the case in which the double-fault exception was generated within VMX non-root operation, the case in which the double-fault exception was generated during event injection by VM entry, and to the case in which VM entry is injecting a double-fault exception.
- **External interrupts.** An external interrupt causes a VM exit if the "external-interrupt exiting" VM-execution control is 1. Otherwise, the interrupt is delivered normally through the IDT. (If a logical processor is in the shutdown state or the wait-for-SIPI state, external interrupts are blocked. The interrupt is not delivered through the IDT and no VM exit occurs.)
- **Non-maskable interrupts (NMIs).** An NMI causes a VM exit if the "NMI exiting" VM-execution control is 1. Otherwise, it is delivered using descriptor 2 of the IDT. (If a logical processor is in the wait-for-SIPI state, NMIs are blocked. The NMI is not delivered through the IDT and no VM exit occurs.)
- **INIT signals.** INIT signals cause VM exits. A logical processor performs none of the operations normally associated with these events. Such exits do not modify register state or clear pending events as they would outside of VMX operation. (If a logical processor is in the wait-for-SIPI state, INIT signals are blocked. They do not cause VM exits in this case.)
- **Start-up IPIs (SIPIs). SIPIs cause VM exits.** If a logical processor is not in the wait-for-SIPI activity state when a SIPI arrives, no VM exit occurs and the SIPI is discarded. VM exits due to SIPIs do not perform any of the normal operations associated with those events: they do not modify register state as they would outside of VMX operation. (If a logical processor is not in the wait-for-SIPI state, SIPIs are blocked. They do not cause VM exits in this case.)
- **Task switches.** Task switches are not allowed in VMX non-root operation. Any attempt to effect a task switch in VMX non-root operation causes a VM exit. See Section 22.6.2.
- **System-management interrupts (SMIs).** If the logical processor is using the dual-monitor treatment of SMIs and system-management mode (SMM), SMIs cause SMM VM exits. See Section 26.15.2.[1]
- **VMX-preemption timer.** A VM exit occurs when the timer counts down to zero. See Section 22.7.1 for details of operation of the VMX-preemption timer. As noted in that section, the timer does not cause VM exits if the logical processor is outside the C-states C0, C1, and C2.

  Debug-trap exceptions and higher priority events take priority over VM exits caused by the VMX-preemption timer. VM exits caused by the VMX-preemption timer take priority over VM exits caused by the "NMI-window exiting" VM-execution control and lower priority events.

  These VM exits wake a logical processor from the same inactive states as would a non-maskable interrupt. Specifically, they wake a logical processor from the

[1] Under the dual-monitor treatment of SMIs and SMM, SMIs also cause SMM VM exits if they occur in VMX root operation outside SMM. If the processor is using the default treatment of SMIs and SMM, SMIs are delivered as described in Section 26.14.1.

shutdown state and from the states entered using the HLT and MWAIT instructions. These VM exits do not occur if the logical processor is in the wait-for-SIPI state.

...

22.7.1 VMX-Preemption Timer

If the last VM entry was performed with the 1-setting of “activate VMX-preemption timer” VM-execution control, the **VMX-preemption timer** counts down (from the value loaded by VM entry; see Section 23.6.4) in VMX non-root operation. When the timer counts down to zero, it stops counting down and a VM exit occurs (see Section 22.3).

The VMX-preemption timer counts down at rate proportional to that of the timestamp counter (TSC). Specifically, the timer counts down by 1 every time bit X in the TSC changes due to a TSC increment. The value of X is in the range 0–31 and can be determined by consulting the VMX capability MSR IA32_VMX_MISC (see Appendix G.6).

The VMX-preemption timer operates in the C-states C0, C1, and C2; it also operates in the shutdown and wait-for-SIPI states. If the timer counts down to zero in C1, C2, or shutdown, the logical processor transitions to the C0 C-state and causes a VM exit. (The timer does not cause a VM exit if it counts down to zero in the wait-for-SIPI state.) The timer is not decremented and does not cause VM exits in C-states deeper than C2.

...

24. 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.

...

23.6.4 VMX-Preemption Timer

If the “activate VMX-preemption timer” VM-execution control is 1, VM entry starts the VMX-preemption timer with the unsigned value in the VMX-preemption timer-value field.

It is possible for the VMX-preemption timer to expire during VM entry (e.g., if the value in the VMX-preemption timer-value field is zero). If this happens (and if the VM entry was not to the wait-for-SIPI state), a VM exit occurs with its normal priority after any event injection and before execution of any instruction following VM entry. For example, any pending debug exceptions established by VM entry (see Section 23.6.3) take priority over a timer-induced VM exit. (The timer-induced VM exit will occur after delivery of the debug exception, unless that exception or its delivery causes a different VM exit.)

See Section 22.7.1 for details of the operation of the VMX-preemption timer in VMX non-root operation, including the blocking and priority of the VM exits that it causes.

...

25. Updates to Chapter 25, Volume 3B

Change bars show changes to Chapter 25 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3B:* System Programming Guide, Part 2.

...

25.2.3.3 Prioritization of EPT-Induced VM Exits

The translation of a linear address to a physical address requires one or more translations of guest-physical addresses using EPT (see Section 25.2.1). This section specifies the relative priority of EPT-induced VM exits with respect to each other and to other events that may be encountered when accessing memory using a linear address.

For an access to a guest-physical address, determination of whether an EPT misconfiguration or an EPT violation occurs is based on an iterative process:[1]

1. An EPT paging-structure entry is read (initially, this is an EPT PML4 entry):
   a. If the entry is not present (bits 2:0 are all 0), an EPT violation occurs.
   b. If the entry is present but its contents are not configured properly (see Section 25.2.3.1), an EPT misconfiguration occurs.
   c. If the entry is present and its contents are configured properly, operation depends on whether the entry references another EPT paging structure (whether it is an EPT PDE with bit 7 set to 1 or an EPT PTE):
      i) If the entry does reference another EPT paging structure, an entry from that structure is accessed; step 1 is executed for that other entry.
      ii) Otherwise, the entry is used to produce the ultimate physical address (the translation of the original guest-physical address); step 2 is executed.
2. Once the ultimate physical address is determined, the privileges determined by the EPT paging-structure entries are evaluated:
   a. If the access to the guest-physical address is not allowed by these privileges (see Section 25.2.3.2), an EPT violation occurs.
   b. If the access to the guest-physical address is allowed by these privileges, memory is accessed using the ultimate physical address.

If CR0.PG = 1, the translation of a linear address is also an iterative process, with the processor first accessing an entry in the guest paging structure referenced by the guest-physical address in CR3 (or, if PAE paging is in use, the guest-physical address in the appropriate PDPTE register), then accessing an entry in another guest paging structure referenced by the guest-physical address in the first guest paging-structure entry, etc. Each guest-physical address is itself translated using EPT and may cause an EPT-induced VM exit. The following items detail how page faults and EPT-induced VM exits are recognized during this iterative process:

1. An attempt is made to access a guest paging-structure entry with a guest-physical address (initially, the address in CR3 or PDPTE register).
   a. If the access fails because of an EPT misconfiguration or an EPT violation (see above), an EPT-induced VM exit occurs.

[1] This is a simplification of the more detailed description given in Section 25.2.2.

b. If the access does not cause an EPT-induced VM exit, bit 0 (the present flag) of the entry is consulted:

i) If the present flag is 0 or any reserved bit is set, a page fault occurs.

ii) If the present flag is 1, no reserved bit is set, operation depends on whether the entry references another guest paging structure (whether it is a guest PDE with PS = 1 or a guest PTE):

- If the entry does reference another guest paging structure, an entry from that structure is accessed; step 1 is executed for that other entry.
- Otherwise, the entry is used to produce the ultimate guest-physical address (the translation of the original linear address); step 2 is executed.

2. Once the ultimate guest-physical address is determined, the privileges determined by the guest paging-structure entries are evaluated:

a. If the access to the linear address is not allowed by these privileges (e.g., it was a write to a read-only page), a page fault occurs.

b. If the access to the linear address is allowed by these privileges, an attempt is made to access memory at the ultimate guest-physical address:

i) If the access fails because of an EPT misconfiguration or an EPT violation (see above), an EPT-induced VM exit occurs.

ii) If the access does not cause an EPT-induced VM exit, memory is accessed using the ultimate physical address (the translation, using EPT, of the ultimate guest-physical address).

If CR0.PG = 0, a linear address is treated as a guest-physical address and is translated using EPT (see above). This process, if it completes without an EPT violation or EPT misconfiguration, produces a physical address and determines the privileges allowed by the EPT paging-structure entries. If these privileges do not allow the access to the physical address (see Section 25.2.3.2), an EPT violation occurs. Otherwise, memory is accessed using the physical address.

...

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.

...

30.1 PERFORMANCE MONITORING OVERVIEW

Performance monitoring was introduced in the Pentium processor with a set of model-specific performance-monitoring counter MSRs. These counters permit selection of processor performance parameters to be monitored and measured. The information obtained from these counters can be used for tuning system and compiler performance.

In Intel P6 family of processors, the performance monitoring mechanism was enhanced to permit a wider selection of events to be monitored and to allow greater control events to be monitored. Next, Pentium 4 and Intel Xeon processors introduced a new performance monitoring mechanism and new set of performance events.

The performance monitoring mechanisms and performance events defined for the Pentium, P6 family, Pentium 4, and Intel Xeon processors are not architectural. They are all model specific (not compatible among processor families). Intel Core Solo and Intel Core Duo processors support a set of architectural performance events and a set of non-architectural performance events. Processors based on Intel Core microarchitecture and Intel® Atom™ microarchitecture support enhanced architectural performance events and non-architectural performance events.

Starting with Intel Core Solo and Intel Core Duo processors, there are two classes of performance monitoring capabilities. The first class supports events for monitoring performance using counting or sampling usage. These events are non-architectural and vary from one processor model to another. They are similar to those available in Pentium M processors. These non-architectural performance monitoring events are specific to the microarchitecture and may change with enhancements. They are discussed in Section 30.3, "Performance Monitoring (Intel® Core™ Solo and Intel® Core™ Duo Processors)." Non-architectural events for a given microarchitecture can not be enumerated using CPUID; and they are listed in Appendix A, "Performance-Monitoring Events."

The second class of performance monitoring capabilities is referred to as architectural performance monitoring. This class supports the same counting and sampling usages, with a smaller set of available events. The visible behavior of architectural performance events is consistent across processor implementations. Availability of architectural performance monitoring capabilities is enumerated using the CPUID.0AH. These events are discussed in Section 30.2.

See also:

- — Section 30.2, "Architectural Performance Monitoring"
- — Section 30.3, "Performance Monitoring (Intel® Core™ Solo and Intel® Core™ Duo Processors)"
- — Section 30.4, "Performance Monitoring (Processors based on Intel® Core™ Microarchitecture)"
- — Section 30.5, "Performance Monitoring (Processors based on Intel® Atom™ Microarchitecture)"
- — Section 30.6, "Performance Monitoring for Processors based on Intel® Microarchitecture code name Nehalem"
- — Section 30.7, "Performance Monitoring for Processors Based on Intel® Microarchitecture Code Name Westmere"
- — Section 30.8, "Performance Monitoring for Processors based on Intel® Microarchitecture code name Sandy Bridge"
- — Section 30.9, "Performance Monitoring (Processors Based on Intel NetBurst® microarchitecture)"
- — Section 30.10, "Performance Monitoring and Intel Hyper-Threading Technology in Processors Based on Intel NetBurst® Microarchitecture"
- — Section 30.13, "Performance Monitoring and Dual-Core Technology"
- — Section 30.14, "Performance Monitoring on 64-bit Intel Xeon Processor MP with Up to 8-MByte L3 Cache"
- — Section 30.16, "Performance Monitoring (P6 Family Processor)"
- — Section 30.17, "Performance Monitoring (Pentium Processors)"

...

30.2.1.1 Architectural Performance Monitoring Version 1 Facilities

Architectural performance monitoring facilities include a set of performance monitoring counters and performance event select registers. These MSRs have the following properties:

- IA32_PMCx MSRs start at address 0C1H and occupy a contiguous block of MSR address space; the number of MSRs per logical processor is reported using CPUID.0AH:EAX[15:8].
- IA32_PERFEVTSELx MSRs start at address 186H and occupy a contiguous block of MSR address space. Each performance event select register is paired with a corresponding performance counter in the 0C1H address block.
- The bit width of an IA32_PMCx MSR is reported using the CPUID.0AH:EAX[23:16]. This the number of valid bits for read operation. On write operations, the lower-order 32 bits of the MSR may be written with any value, and the high-order bits are sign-extended from the value of bit 31.
- Bit field layout of IA32_PERFEVTSELx MSRs is defined architecturally.

...

30.2.2 Architectural Performance Monitoring Version 3 Facilities

The facilities provided by architectural performance monitoring version 1 and 2 are also supported by architectural performance monitoring version 3. Additionally version 3 provides enhancements to support a processor core comprising of more than one logical processor, i.e. a processor core supporting Intel Hyper-Threading Technology or simultaneous multi-threading capability. Specifically,

- CPUID leaf 0AH provides enumeration mechanisms to query:
  - — The number of general-purpose performance counters (IA32_PMCx) is reported in CPUID.0AH:EAX[15:8], the bit width of general-purpose performance counters (see also Section 30.2.1.1) is reported in CPUID.0AH:EAX[23:16].
  - — The bit vector representing the set of architectural performance monitoring events supported (see Section 30.2.3)
  - — The number of fixed-function performance counters, the bit width of fixed-function performance counters (see also Section 30.2.2.1).
- Each general-purpose performance counter IA32_PMCx (starting at MSR address 0C1H) is associated with a corresponding IA32_PERFEVTSELx MSR (starting at MSR address 186H). The Bit field layout of IA32_PERFEVTSELx MSRs is defined architecturally in Figure 30-6.

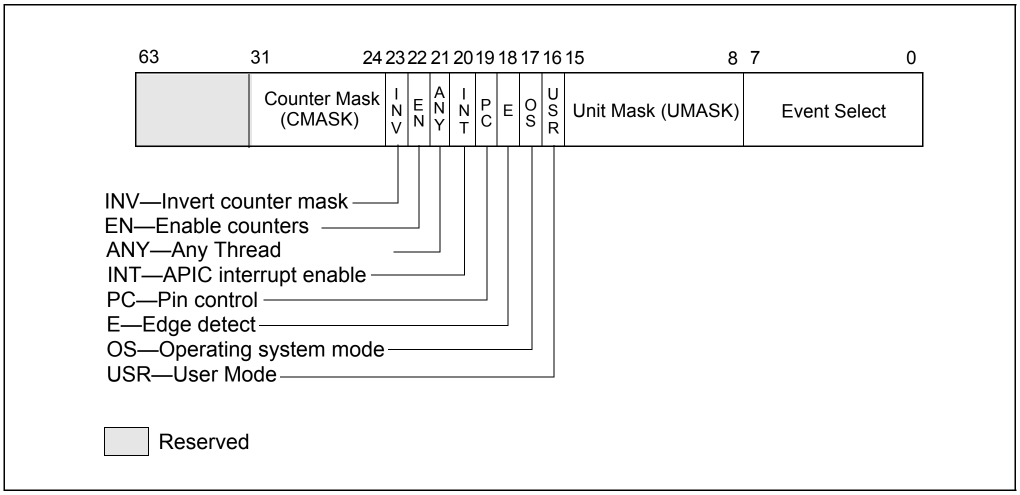

Figure 30-6 Layout of IA32_PERFEVTSELx MSRs Supporting Architectural Performance Monitoring Version 3

Bit 21 (AnyThread) of IA32_PERFEVTSELx is supported in architectural performance monitoring version 3. When set to 1, it enables counting the associated event conditions (including matching the thread's CPL with the OS/USR setting of IA32_PERFEVTSELx) occurring across all logical processors sharing a processor core. When bit 21 is 0, the counter only increments the associated event conditions (including matching the thread's CPL with the OS/USR setting of IA32_PERFEVTSELx) occurring in the logical processor which programmed the IA32_PERFEVTSELx MSR.

- Each fixed-function performance counter IA32_FIXED_CTRx (starting at MSR address 309H) is configured by a 4-bit control block in the IA32_PERF_FIXED_CTR_CTRL MSR. The control block also allow thread-specificity configuration using an AnyThread bit. The layout of IA32_PERF_FIXED_CTR_CTRL MSR is shown.

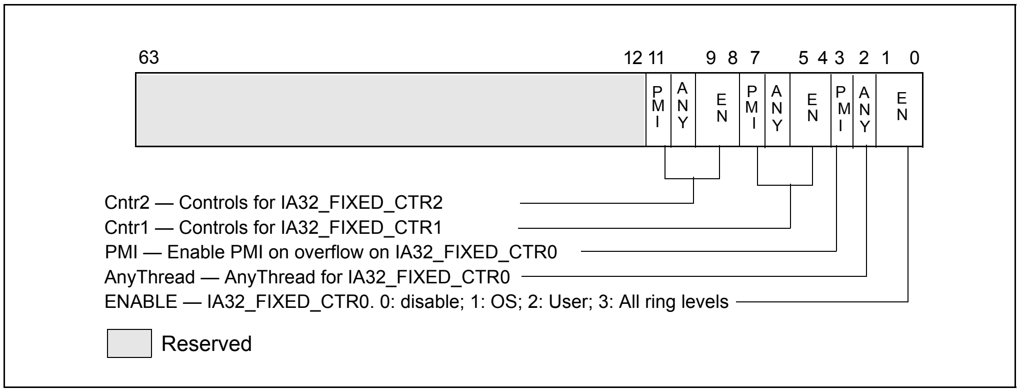
63
12 11
9 8 7
5 4 3 2 1 0
PMI
ANY
EN
PMI
ANY
EN
PMI
ANY
EN
Cntr2 — Controls for IA32_FIXED_CTR2
Cntr1 — Controls for IA32_FIXED_CTR1
PMI — Enable PMI on overflow on IA32_FIXED_CTR0
AnyThread — AnyThread for IA32_FIXED_CTR0
ENABLE — IA32_FIXED_CTR0. 0: disable; 1: OS; 2: User; 3: All ring levels
Reserved

Figure 30-7 Layout of IA32_FIXED_CTR_CTRL MSR Supporting Architectural Performance Monitoring Version 3

Each control block for a fixed-function performance counter provides a **AnyThread** (bit position 2 + 4*N, N= 0, 1, etc.) bit. When set to 1, it enables counting the associated event conditions (including matching the thread's CPL with the ENABLE setting of the corresponding control block of IA32_PERF_FIXED_CTR_CTRL) occurring across all logical processors sharing a processor core. When an **AnyThread** bit is 0 in IA32_PERF_FIXED_CTR_CTRL, the corresponding fixed counter only increments the associated event conditions occurring in the logical processor which programmed the IA32_PERF_FIXED_CTR_CTRL MSR.

- The IA32_PERF_GLOBAL_CTRL, IA32_PERF_GLOBAL_STATUS, IA32_PERF_GLOBAL_OVF_CTRL MSRs provide single-bit controls/status for each general-purpose and fixed-function performance counter. Figure 30-8 shows the layout of these MSR for N general-purpose performance counters (where N is reported by CPUID.0AH:EAX[15:8]) and three fixed-function counters.

...

30.2.3 Pre-defined Architectural Performance Events

Table 30-1 lists architecturally defined events.

Table 30-1 UMask and Event Select Encodings for Pre-Defined Architectural Performance Events

Bit Position CPUID.AH.EBX	Event Name	UMask	Event Select
0	UnHalted Core Cycles	00H	3CH
1	Instruction Retired	00H	C0H
2	UnHalted Reference Cycles	01H	3CH
3	LLC Reference	4FH	2EH
4	LLC Misses	41H	2EH
5	Branch Instruction Retired	00H	C4H
6	Branch Misses Retired	00H	C5H

A processor that supports architectural performance monitoring may not support all the predefined architectural performance events (Table 30-1). The non-zero bits in CPUID.0AH:EBX indicate the events that are not available.

The behavior of each architectural performance event is expected to be consistent on all processors that support that event. Minor variations between microarchitectures are noted below:

- **UnHalted Core Cycles —** Event select 3CH, Umask 00H

  This event counts core clock cycles when the clock signal on a specific core is running (not halted). The counter does not advance in the following conditions:

  — an ACPI C-state other than C0 for normal operation

  — HLT

  — STPCLK# pin asserted

  — being throttled by TM1

  — during the frequency switching phase of a performance state transition (see Chapter 14, "Power and Thermal Management")

  The performance counter for this event counts across performance state transitions using different core clock frequencies

- **Instructions Retired —** Event select C0H, Umask 00H

  This event counts the number of instructions at retirement. For instructions that consist of multiple micro-ops, this event counts the retirement of the last micro-op of the instruction. An instruction with a REP prefix counts as one instruction (not per iteration). Faults before the retirement of the last micro-op of a multi-ops instruction are not counted.

  This event does not increment under VM-exit conditions. Counters continue counting during hardware interrupts, traps, and inside interrupt handlers.

- **UnHalted Reference Cycles —** Event select 3CH, Umask 01H

  This event counts reference clock cycles while the clock signal on the core is running. The reference clock operates at a fixed frequency, irrespective of core frequency changes due to performance state transitions. Processors may implement this behavior differently. See Table A-10 and Table A-12 in Appendix A, "Performance-Monitoring Events."

- **Last Level Cache References —** Event select 2EH, Umask 4FH

  This event counts requests originating from the core that reference a cache line in the last level cache. The event count includes speculation and cache line fills due to the first-level cache hardware prefetcher, but may exclude cache line fills due to other hardware-prefetchers.

  Because cache hierarchy, cache sizes and other implementation-specific characteristics; value comparison to estimate performance differences is not recommended.

- **Last Level Cache Misses —** Event select 2EH, Umask 41H

  This event counts each cache miss condition for references to the last level cache. The event count may include speculation and cache line fills due to the first-level cache hardware prefetcher, but may exclude cache line fills due to other hardware-prefetchers.

  Because cache hierarchy, cache sizes and other implementation-specific characteristics; value comparison to estimate performance differences is not recommended.

  Because cache hierarchy, cache sizes and other implementation-specific characteristics; value comparison to estimate performance differences is not recommended.

- **Branch Instructions Retired —** Event select C4H, Umask 00H

  This event counts branch instructions at retirement. It counts the retirement of the last micro-op of a branch instruction.

- **All Branch Mispredict Retired —** Event select C5H, Umask 00H

  This event counts mispredicted branch instructions at retirement. It counts the retirement of the last micro-op of a branch instruction in the architectural path of execution and experienced misprediction in the branch prediction hardware.

  Branch prediction hardware is implementation-specific across microarchitectures; value comparison to estimate performance differences is not recommended.

NOTE

Programming decisions or software precisians on functionality should not be based on the event values or dependent on the existence of performance monitoring events.

...

30.4.4.4 Re-configuring PEBS Facilities

When software needs to reconfigure PEBS facilities, it should allow a quiescent period between stopping the prior event counting and setting up a new PEBS event. The quiescent period is to allow any latent residual PEBS records to complete its capture at their previously specified buffer address (provided by IA32_DS_AREA).

...

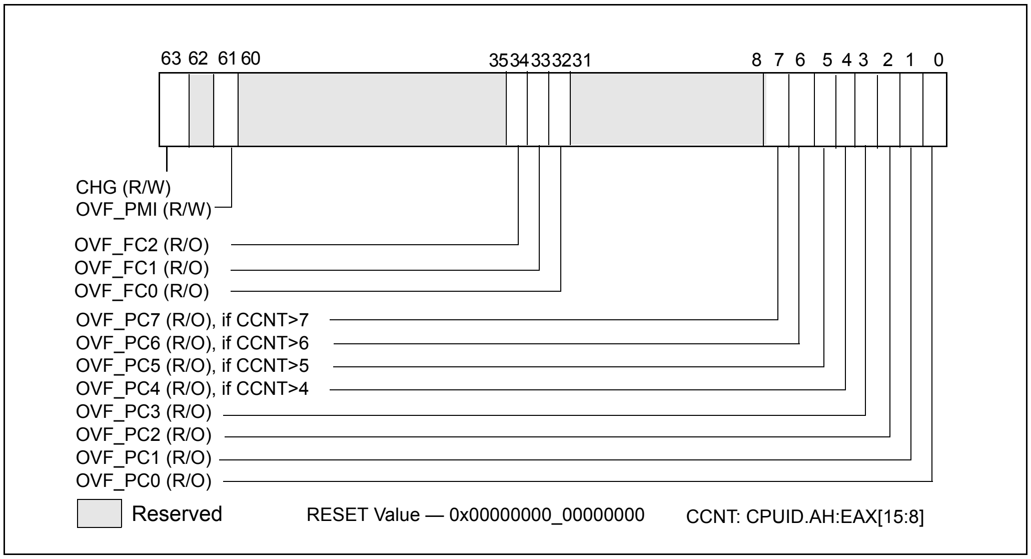

Figure 30-13 IA32_PERF_GLOBAL_STATUS MSR

...

30.7 PERFORMANCE MONITORING FOR PROCESSORS BASED ON INTEL® MICROARCHITECTURE CODE NAME WESTMERE

All of the performance monitoring programming interfaces (architectural and non-architectural core PMU facilities, and uncore PMU) described in Section 30.6 also apply to processors based on Intel® microarchitecture code name Westmere.

Table 30-14 describes a non-architectural performance monitoring event (event code 0B7H) and associated MSR_OFFCORE_RSP_0 (address 1A6H) in the core PMU. This event and a second functionally equivalent offcore response event using event code 0BBH and MSR_OFFCORE_RSP_1 (address 1A7H) are supported in processors based on Intel microarchitecture code name Westmere. The event code and event mask definitions of Non-architectural performance monitoring events are listed in Table A-11.

The load latency facility is the same as described in Section 30.6.1.2, but added enhancement to provide more information in the data source encoding field of each load latency record. The additional information relates to STLB_MISS and LOCK, see Table 30-22.

30.7.1 Intel Xeon Processor E7 Family Performance Monitoring Facility

The performance monitoring facility in the processor core of the Intel Xeon processor E7 family is the same as those supported in the Intel Xeon processor 5600 series[1]. The uncore subsystem in the Intel Xeon processor E7 family is similar to those of the Intel Xeon processor 7500 series. The high level construction of the uncore sub-system is similar to that shown in Figure 30-24, with the additional capability that up to 10 C-Box units are supported.

Table 30-18 summarizes the number MSRs for uncore PMU for each box.

Table 30-18 Uncore PMU MSR Summary for Intel Xeon Processor E7 Family

Box	# of Boxes	Counters per Box	Counter Width	General Purpose	Global Enable	Sub-control MSRs
C-Box	10	6	48	Yes	per-box	None
S-Box	2	4	48	Yes	per-box	Match/Mask
B-Box	2	4	48	Yes	per-box	Match/Mask
M-Box	2	6	48	Yes	per-box	Yes
R-Box	1	16 (2 port, 8 per port)	48	Yes	per-box	Yes
W-Box	1	4	48	Yes	per-box	None
		1	48	No	per-box	None
U-Box	1	1	48	Yes	uncore	None

...

30.8 PERFORMANCE MONITORING FOR PROCESSORS BASED ON INTEL® MICROARCHITECTURE CODE NAME SANDY BRIDGE

Intel Core i7, i5, i3 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) 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.

[1] Exceptions are indicated for event code 0FH in .Table A-6; and valid bits of data source encoding field of each load latency record is limited to bits 5:4 of Table 30-22.

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 30-19.

...

Table 30-21 PEBS Performance Events for Intel microarchitecture code name Sandy Bridge

Event Name	Event Select	Sub-event	UMask
INST_RETIRED	C0H	PREC_DIST	01H[1]
UOPS_RETIRED	C2H	All	01H
		Retire_Slots	02H
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

NOTES:

1. Only available on IA32_PMC1.

...

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 store 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 30-23 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 store 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.
Reserved	A8H	Reserved

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 "skid" 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.PREC_DIST precise event, and must use IA32_PMC1 with PerfEvtSel1 property configured and bit 1 in the IA32_PEBS_ENABLE set to 1. INST_RETIRED.PREC_DIST is a non-architectural performance event, it is not supported in prior generation microarchitectures. Additionally, current implementation of PDIR limits tool to quiesce the rest of the programmable counters in the core when PDIR is active.

30.8.5 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 30-24 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.

...

30.8.6 Uncore Performance Monitoring Facilities In Intel® Core i7, i5, i3 Processors 2xxx Series

The uncore sub-system in Intel Core i7, i5, i3 processors 2xxx Series provides a unified L3 that can support up to four processor cores. The L3 cache consists multiple slices, each slice interface with a processor via a coherence engine, referred to as a C-Box. Each C-Box provides dedicated facility of MSRs to select uncore performance monitoring events and each C-Box event select MSR is paired with a counter register, similar in style as those described in Section 30.6.2.2. The layout of the event select MSRs in the C-Boxes are shown in Figure 30-31.

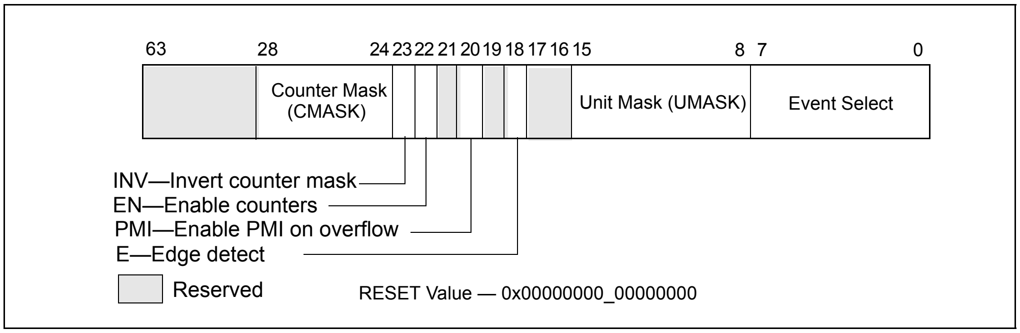

Figure 30-31 Layout of MSR_UNC_CBO_N_PERFEVTSELx MSR for C-Box N

At the uncore domain level, there is a master set of control MSRs that centrally manages all the performance monitoring facility of uncore units. Figure 30-32 shows the layout of the uncore domain global control

MSR bit 31 of MSR_UNC_PERF_GLOBAL_CTRL provides the capability to freeze all uncore counters when an overflow condition in a unit counter. When set and upon a counter overflow, the uncore PMU logic will clear the global enable bit, bit 29.

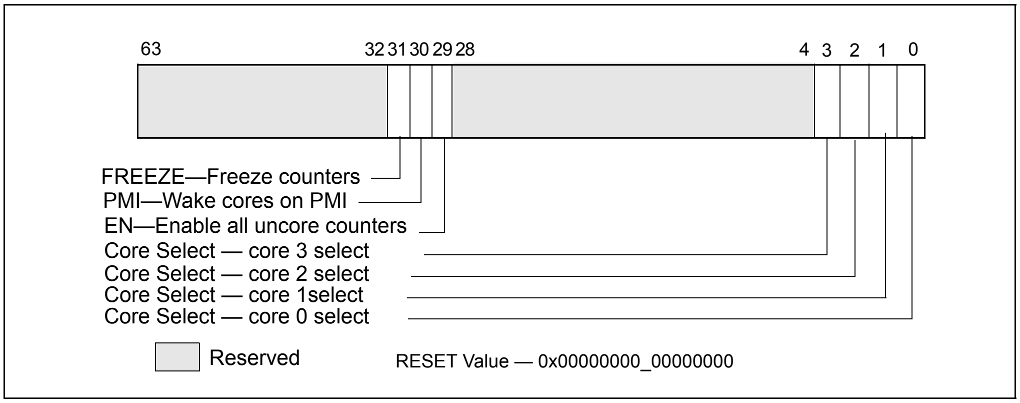
63
32
31
30
29
28
4
3
2
1
0
FREEZE—Freeze counters
PMI—Wake cores on PMI
EN—Enable all uncore counters
Core Select — core 3 select
Core Select — core 2 select
Core Select — core 1select
Core Select — core 0 select
Reserved
RESET Value — 0x00000000_00000000

Figure 30-32 Layout of MSR_UNC_PERF_GLOBAL_CTRL MSR for Uncore

Additionally, there is also a fixed counter, counting uncore clockticks, for the uncore domain. Table 30-27 summarizes the number MSRs for uncore PMU for each box.

Table 30-27 Uncore PMU MSR Summary

Box	# of Boxes	Counters per Box	Counter Width	General Purpose	Global Enable	Comment
C-Box	Up to 4	2	44	Yes	Per-box	
NCU		1	48	No	Uncore	

30.8.6.1 Uncore Performance Monitoring Events

There are certain restrictions on the uncore performance counters in each C-Box. Specifically,

- Occupancy events are supported only with counter 0 but not counter 1.

Other uncore C-Box events can be programmed with either counter 0 or 1.

The C-Box uncore performance events described in Table A-3 can collect performance characteristics of transactions initiated by processor core. In that respect, they are similar to various sub-events in the OFFCORE_RESPONSE family of performance events in the core PMU. Information such as data supplier locality (LLC HIT/MISS) and snoop responses can be collected via OFFCORE_RESPONSE and qualified on a per-thread basis.

On the other hand, uncore performance event logic can not associate its counts with the same level of per-thread qualification attributes as the core PMU events can. Therefore, whenever similar event programming capabilities are available from both core PMU and uncore PMU, the recommendation is that utilizing the core PMU events may be less affected by artifacts, complex interactions and other factors.

...

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.

...

A.1 ARCHITECTURAL PERFORMANCE-MONITORING EVENTS

Architectural performance events are introduced in Intel Core Solo and Intel Core Duo processors. They are also supported on processors based on Intel Core microarchitecture. Table A-1 lists pre-defined architectural performance events that can be configured using general-purpose performance counters and associated event-select registers.

Table A-1 Architectural Performance Events

Event Num.	Event Mask Mnemonic	Umask Value	Description	Comment
3CH	UnHalted Core Cycles	00H	Unhalted core cycles	
3CH	UnHalted Reference Cycles	01H	Unhalted reference cycles	Measures bus cycle[1]
C0H	Instruction Retired	00H	Instruction retired	
2EH	LLC Reference	4FH	Last level cache references	
2EH	LLC Misses	41H	Last level cache misses	
C4H	Branch Instruction Retired	00H	Branch instruction at retirement	
C5H	Branch Misses Retired	00H	Mispredicted Branch Instruction at retirement	

NOTES:

1. Implementation of this event in Intel Core 2 processor family, Intel Core Duo, and Intel Core Solo processors measures bus clocks.

...

Table A-2 Non-Architectural Performance Events in the Processor Core for Intel Core i7, i5, i3 Processors 2xxx Series

Event Num.	Umask Value	Event Mask Mnemonic	Description	Comment
...				
08H	01H	DTLB_LOAD_MISSES.MISS_CAUSES_A_WALK	Misses in all TLB levels that cause a page walk of any page size.	
...				
0DH	03H	INT_MISC.RECOVERY_CYCLES	Cycles waiting to recover after Machine Clears or JEClear. Set Cmask= 1.	Set Edge to count occurrences
...				
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
10H	01H	FP_COMP_OPS_EXE.X87	Counts number of X87 uops executed.	
10H	10H	FP_COMP_OPS_EXE.SSE_FP_PACKED_DOUBLE	Counts number of SSE* double precision FP packed uops executed.	
10H	20H	FP_COMP_OPS_EXE.SSE_FP_SCALAR_SINGLE	Counts number of SSE* single precision FP scalar uops executed.	
10H	40H	FP_COMP_OPS_EXE.SSE_PACKED SINGLE	Counts number of SSE* single precision FP packed uops executed.	
10H	80H	FP_COMP_OPS_EXE.SSE_SCALAR_DOUBLE	Counts number of SSE* double precision FP scalar uops executed.	
11H	01H	SIMD_FP_256.PACKED_SINGLE	Counts 256-bit packed single-precision floating-point instructions	
11H	02H	SIMD_FP_256.PACKED_DOUBLE	Counts 256-bit packed double-precision floating-point instructions	
...				
24H	01H	L2_RQSTS.DEMAND_DATA_RD_HIT	Demand Data Read requests that hit L2 cache	
24H	03H	L2_RQSTS.ALL_DEMAND_DATA_RD	Counts any demand and L1 HW prefetch 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.ALL_RFO	Counts all L2 store RFO requests.	
...				
2EH	4FH	LONGEST_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	LONGEST_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_THREAD_UNHALTED.REF_XCLK	Increments at the frequency of XCLK (100 MHz) when not halted.	see Table A-1
...				
51H	04H	L1D.EVICTION	Counts the number of modified lines evicted from the L1 data cache due to replacement.	
...				
85H	01H	ITLB_MISSES.MISS_CAUSES_A_WALK	Misses in all ITLB levels that cause page walks	
...				
88H	02H	BR_INST_EXEC.DIRECT_JMP	Qualify all unconditional near branch instructions excluding calls and indirect branches.	Must combine with umask 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 80H
88H	08H	BR_INST_EXEC.RETURN_NEAR	Qualify indirect near branches that have a return mnemonic.	Must combine with umask 80H
88H	10H	BR_INST_EXEC.DIRECT_NEAR_CALL	Qualify unconditional near call branch instructions, excluding non call branch, executed.	Must combine with umask 80H
88H	20H	BR_INST_EXEC.INDIRECT_NEAR_CALL	Qualify indirect near calls, including both register and memory indirect, executed.	Must combine with umask 80H
88H	40H	BR_INST_EXEC.NONTAKEN	Qualify non-taken near branches executed.	Applicable to umask 01H only
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 80H
89H	08H	BR_MISP_EXEC.RETURN_NEAR	Qualify mispredicted indirect near branches that have a return mnemonic.	Must combine with umask 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 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 80H
89H	40H	BR_MISP_EXEC.NONTAKEN	Qualify mispredicted non-taken near branches executed,.	Applicable to umask 01H only
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).	
...				
A2H	08H	RESOURCE_STALLS.SB	Cycles stalled due to no store buffers available. (not including draining form sync).	
...				
ABH	02H	DSB2MITE_SWITCHES.PENALTY_CYCLES	Cycles DSB to MITE switches caused delay.	
...				
B7H	01H	OFF_CORE_RESPONSE_0	see Section 30.8.5, “Off-core Response Performance Monitoring”; PMC0 only.	Requires programming MSR 01A6H
BBH	01H	OFF_CORE_RESPONSE_1	See Section 30.8.5, “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	05H	L1D_BLOCKS.BANK_CONFLICT_CYCLES	Cycles when dispatched loads are cancelled due to L1D bank conflicts with other load ports	cmask=1
C0H	00H	INST_RETIRED.ANY_P	Number of instructions at retirement	See Table A-1

C0H	01H	INST_RETIRED.PREC_DIST	Precise instruction retired event with HW to reduce effect of PEBS shadow in IP distribution	PMC1 only; Must quiesce other PMCs.
C0H	02H	INST_RETIRED.X87	X87 instruction retired event	
C1H	02H	OTHER_ASSISTS.ITLB_MISS_RETIRED	Instructions that experienced an ITLB miss.	
C1H	08H	OTHER_ASSISTS.AVX_STORE	Number of assists associated with 256-bit AVX store operations.	
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.	
...				
C4H	00H	BR_INST_RETIRED.ALL_BRANCHES	Branch instructions at retirement	See Table A-1
...				
C5H	00H	BR_MISP_RETIRED.ALL_BRANCHES	Mispredicted branch instructions at retirement	See Table A-1
...				
CDH	01H	MEM_TRANS_RETIRED.LOAD_LATENCY	Sample loads with specified latency threshold. PMC3 only.	Specify threshold in MSR 0x3F6
CDH	02H	MEM_TRANS_RETIRED.PRECISE_STORE	Sample stores and collect precise store operation via PEBS record. PMC3 only.	See Section 30.8.4.3
...				

...

Non-architectural Performance monitoring events that are located in the uncore sub-system are implementation specific between different platforms using processors based on Intel microarchitecture Sandy Bridge. Processors with CPUID signature of DisplayFamily_DisplayModel 06_2AH support performance events listed in Table A-3.

Table A-3 Non-Architectural Performance Events in the Processor Uncore for Intel Core i7, i5, i3 Processor 2xxx Series

Event Num.	Umask Value	Event Mask Mnemonic	Description	Comment
22H	01H	UNC_CBO_XSNP_RESPONSE.RSPIHITI	Snoop responses received from processor cores to requests initiated by this Cbox.	Must combine with one of the umask values of 20H, 40H, 80H
22H	02H	UNC_CBO_XSNP_RESPONSE.RSPIHITFSE		
22H	04H	UNC_CBO_XSNP_RESPONSE.RSPSHITFSE		
22H	08H	UNC_CBO_XSNP_RESPONSE.RSPSFWDM		
22H	01H	UNC_CBO_XSNP_RESPONSE.RSPIFWDM		
22H	20H	UNC_CBO_XSNP_RESPONSE.AND_EXTERNAL	Filter on cross-core snoops resulted in external snoop request. Must combine with at least one of 01H, 02H, 04H, 08H, 10H	
22H	40H	UNC_CBO_XSNP_RESPONSE.AND_XCORE	Filter on cross-core snoops resulted in core request. Must combine with at least one of 01H, 02H, 04H, 08H, 10H	
22H	80H	UNC_CBO_XSNP_RESPONSE.AND_XCORE	Filter on cross-core snoops resulted in LLC evictions. Must combine with at least one of 01H, 02H, 04H, 08H, 10H	
34H	01H	UNC_CBO_CACHE_LOOKUP.M	LLC lookup request that access cache and found line in M-state.	Must combine with one of the umask values of 10H, 20H, 40H, 80H
34H	02H	UNC_CBO_CACHE_LOOKUP.E	LLC lookup request that access cache and found line in E-state.	
34H	04H	UNC_CBO_CACHE_LOOKUP.S	LLC lookup request that access cache and found line in S-state.	
34H	08H	UNC_CBO_CACHE_LOOKUP.I	LLC lookup request that access cache and found line in I-state.	
34H	10H	UNC_CBO_CACHE_LOOKUP.AND_READ	Filter on processor core initiated cacheable read requests. Must combine with at least one of 01H, 02H, 04H, 08H	
34H	20H	UNC_CBO_CACHE_LOOKUP.AND_READ	Filter on processor core initiated cacheable write requests. Must combine with at least one of 01H, 02H, 04H, 08H	
34H	40H	UNC_CBO_CACHE_LOOKUP.AND_EXTSNP	Filter on external snoop requests. Must combine with at least one of 01H, 02H, 04H, 08H	

Table A-3 Non-Architectural Performance Events in the Processor Uncore for Intel Core i7, i5, i3 Processor 2xxx Series

Event Num.	Umask Value	Event Mask Mnemonic	Description	Comment
34H	80H	UNC_CBO_CACHE_LOOKUP.AND_ANY	Filter on any IRQ or IPQ initiated requests including uncacheable, non-coherent requests. Must combine with at least one of 01H, 02H, 04H, 08H	
80H	01H	UNC_IMPH_CBO_TRK_OCCUPANCY.ALL	Counts cycles weighted by the number of core-outgoing valid entries. Valid entries are between allocation to the first of IDIO or DRSO messages. Accounts for coherent and in-coherent traffic	Counter 0 only
81H	01H	UNC_IMPH_CBO_TRK_REQUEST.ALL	Counts the number of core-outgoing entries. Accounts for coherent and in-coherent traffic	
81H	20H	UNC_IMPH_CBO_TRK_REQUEST.WRITES	Counts the number of allocated write entries, include full, partial, and evictions.	
81H	80H	UNC_IMPH_CBO_TRK_REQUEST.EVICTIONS	Counts the number of evictions allocated.	
83H	01H	UNC_IMPH_COH_TRK_OCCUPANCY.ALL	Counts cycles weighted by the number of core-outgoing valid entries in the coherent tracker queue.	Counter 0 only
84H	01H	UNC_IMPH_COH_TRK_REQUEST.ALL	Counts the number of core-outgoing entries in the coherent tracker queue.	

...

Table A-4 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
...				
0BH	01H	MEM_INST_RETIRED.LOADS	Counts the number of instructions with an architecturally-visible load retired on the architected path.	
0BH	02H	MEM_INST_RETIRED.STORES	Counts the number of instructions with an architecturally-visible store retired on the architected path.	
...				
49H	20H	DTLB_MISSES.PDE_MISS	Number of DTLB misses caused by low part of address, includes references to 2M pages because 2M pages do not use the PDE.	
49H	80H	DTLB_MISSES.LARGE_WALK_COMPLETED	Counts number of misses in the STLB which resulted in a completed page walk for large pages.	
...				
C4H	00H	BR_INST_RETIRED.ALL_BRANCHES	Branch instructions at retirement	See Table A-1
...				
C5H	00H	BR_MISP_RETIRED.ALL_BRANCHES	Mispredicted branch instructions at retirement	See Table A-1
...				

...

A.4 PERFORMANCE MONITORING EVENTS FOR PROCESSORS BASED ON INTEL® MICROARCHITECTURE CODE NAME WESTMERE

Intel 64 processors based on Intel® microarchitecture code name Westmere support the architectural and non-architectural performance-monitoring events listed in Table A-1 and Table A-6. Table A-6 applies to processors with CPUID signature of DisplayFamily_DisplayModel encoding with the following values: 06_25H, 06_2CH. In addition, these processors (CPUID signature of DisplayFamily_DisplayModel 06_25H, 06_2CH) also support the following non-architectural, product-specific uncore performance-monitoring events listed in Table A-7. Fixed counters support the architecture events defined in Table A-9.

Table A-6 Non-Architectural Performance Events in the Processor Core for Processors Based on Intel Microarchitecture Code Name Westmere

Event Num.	Umask Value	Event Mask Mnemonic	Description	Comment
...				
0BH	01H	MEM_INST_RETIRED.LOADS	Counts the number of instructions with an architecturally-visible load retired on the architected path.	
0BH	02H	MEM_INST_RETIRED.STORES	Counts the number of instructions with an architecturally-visible store retired on the architected path.	
...				
0FH	01H	MEM_UNCORE_RETIRED.UNKNOWN_SOURCE	Load instructions retired with unknown LLC miss (Precise Event).	Applicable to one and two sockets
0FH	02H	MEM_UNCORE_RETIRED.OHTER_CORE_L2_HIT	Load instructions retired that HIT modified data in sibling core (Precise Event).	Applicable to one and two sockets
0FH	04H	MEM_UNCORE_RETIRED.REMOTE_HITM	Load instructions retired that HIT modified data in remote socket (Precise Event).	Applicable to two sockets only
0FH	08H	MEM_UNCORE_RETIRED.LOCAL_DRAM_AND_REMOTE_CACHE_HIT	Load instructions retired local dram and remote cache HIT data sources (Precise Event).	Applicable to one and two sockets
0FH	10H	MEM_UNCORE_RETIRED.REMOTE_DRAM	Load instructions retired remote DRAM and remote home-remote cache HITM (Precise Event).	Applicable to two sockets only
0FH	20H	MEM_UNCORE_RETIRED.OTHER_LLC_MISS	Load instructions retired other LLC miss (Precise Event).	Applicable to two sockets only
0FH	80H	MEM_UNCORE_RETIRED.UNCACHEABLE	Load instructions retired I/O (Precise Event).	Applicable to one and two sockets
...				
49H	20H	DTLB_MISSES.PDE_MISS	Number of DTLB misses caused by low part of address, includes references to 2M pages because 2M pages do not use the PDE.	
...				
4CH	01H	LOAD_HIT_PRE	Counts load operations sent to the L1 data cache while a previous SSE prefetch instruction to the same cache line has started prefetching but has not yet finished.	Counter 0, 1 only
4EH	01H	L1D_PREFETCH.REQUESTS	Counts number of hardware prefetch requests dispatched out of the prefetch FIFO.	Counter 0, 1 only

4EH	02H	L1D_PREFETCH.MISS	Counts number of hardware prefetch requests that miss the L1D. There are two prefetchers in the L1D. A streamer, which predicts lines sequentially after this one should be fetched, and the IP prefetcher that remembers access patterns for the current instruction. The streamer prefetcher stops on an L1D hit, while the IP prefetcher does not.	Counter 0, 1 only
4EH	04H	L1D_PREFETCH.TRIGGERS	Counts number of prefetch requests triggered by the Finite State Machine and pushed into the prefetch FIFO. Some of the prefetch requests are dropped due to overwrites or competition between the IP index prefetcher and streamer prefetcher. The prefetch FIFO contains 4 entries.	Counter 0, 1 only
...				
63H	01H	CACHE_LOCK_CYCLES.L1D_L2	Cycle count during which the L1D and L2 are locked. A lock is asserted when there is a locked memory access, due to uncacheable memory, a locked operation that spans two cache lines, or a page walk from an uncacheable page table. This event does not cause locks, it merely detects them.	Counter 0, 1 only. L1D and L2 locks have a very high performance penalty and it is highly recommended to avoid such accesses.
...				
C4H	00H	BR_INST_RETIRED.ALL_BRANCHES	Branch instructions at retirement	See Table A-1
...				
C5H	00H	BR_MISP_RETIRED.ALL_BRANCHES	Mispredicted branch instructions at retirement	See Table A-1
...				
D1H	01H	UOPS_DECODED.STALL_CYCLES	Counts the cycles of decoder stalls. INV=1, Cmask= 1	
...				

...

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_2DH	Next Generation Intel Xeon Processor
06_2FH	Intel Xeon Processors E7 Family
06_2AH	Intel Xeon processors E3 series; Second Generation Intel Core i7, i5, i3 Processors 2xxx Series
06_2EH	Intel Xeon Processor 7500, 6500 series
06_25H, 06_2CH	Intel Xeon Processors 3600, 5600 series, Intel Core i7, i5 and i3 Processors
06_1EH, 06_1FH	Intel Core i7 and i5 Processors
06_1AH	Intel Core i7 Processor, Intel Xeon Processor 3400, 3500, 5500 series
06_1DH	Intel Xeon Processor MP 7400 series
06_17H	Intel Xeon Processor 3100, 3300, 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

...

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			
...				
1D9H	473	IA32_DEBUGCTL (MSR_DEBUGCTLA, MSR_DEBUGCTLB)	Trace/Profile Resource Control (R/W)	06_0EH
		0	LBR: Setting this bit to 1 enables the processor to record a running trace of the most recent branches taken by the processor in the LBR stack.	06_01H
		1	BTF: Setting this bit to 1 enables the processor to treat EFLAGS.TF as single-step on branches instead of single-step on instructions.	06_01H
		5:2	Reserved	
		6	TR: Setting this bit to 1 enables branch trace messages to be sent.	06_0EH
		7	BTS: Setting this bit enables branch trace messages (BTMs) to be logged in a BTS buffer.	06_0EH
		8	BTINT: When clear, BTMs are logged in a BTS buffer in circular fashion. When this bit is set, an interrupt is generated by the BTS facility when the BTS buffer is full.	06_0EH
		9	1: BTS_OFF_OS: When set, BTS or BTM is skipped if CPL = 0.	06_0FH
		10	BTS_OFF_USR: When set, BTS or BTM is skipped if CPL > 0.	06_0FH
		11	FREEZE_LBRS_ON_PMI: When set, the LBR stack is frozen on a PMI request.	If CPUID.01H: ECX[15] = 1 and CPUID.0AH: EAX[7:0] > 1
		12	FREEZE_PERFMON_ON_PMI: When set, each ENABLE bit of the global counter control MSR are frozen (address 3BFH) on a PMI request	If CPUID.01H: ECX[15] = 1 and CPUID.0AH: EAX[7:0] > 1

		13	ENABLE_UNCORE_PMI: When set, enables the logical processor to receive and generate PMI on behalf of the uncore.	06_1AH
		14	FREEZE_WHILE_SMM: When set, freezes perfmon and trace messages while in SMM.	if IA32_PERF_CAPABILITIES[12] = ’1
		63:15	Reserved	
…				
38DH	909	IA32_FIXED_CTR_CTRL (MSR_PERF_FIXED_CTR_CTRL)	Fixed-Function Performance Counter Control (R/W) Counter increments while the results of ANDing respective enable bit in IA32_PERF_GLOBAL_CTRL with the corresponding OS or USR bits in this MSR is true.	If CPUID.0AH: EAX[7:0] > 1
		0	EN0_OS: Enable Fixed Counter 0 to count while CPL = 0	
		1	EN0_Usr: Enable Fixed Counter 0 to count while CPL > 0	
		2	AnyThread: When set to 1, it enables counting the associated event conditions occurring across all logical processors sharing a processor core. When set to 0, the counter only increments the associated event conditions occurring in the logical processor which programmed the MSR.	If CPUID.0AH: EAX[7:0] > 2
		3	EN0_PMI: Enable PMI when fixed counter 0 overflows	
		4	EN1_OS: Enable Fixed Counter 1to count while CPL = 0	
		5	EN1_Usr: Enable Fixed Counter 1to count while CPL > 0	

		6	AnyThread: When set to 1, it enables counting the associated event conditions occurring across all logical processors sharing a processor core. When set to 0, the counter only increments the associated event conditions occurring in the logical processor which programmed the MSR.	If CPUID.0AH: EAX[7:0] > 2
		7	EN1_PMI: Enable PMI when fixed counter 1 overflows	
		8	EN2_OS: Enable Fixed Counter 2 to count while CPL = 0	
		9	EN2_Usr: Enable Fixed Counter 2 to count while CPL > 0	
		10	AnyThread: When set to 1, it enables counting the associated event conditions occurring across all logical processors sharing a processor core. When set to 0, the counter only increments the associated event conditions occurring in the logical processor which programmed the MSR.	If CPUID.0AH: EAX[7:0] > 2
		11	EN2_PMI: Enable PMI when fixed counter 2 overflows	
		63:12	Reserved	
...				

...

B.3 MSRS IN THE INTEL® ATOM™ PROCESSOR FAMILY

Table B-4 lists model-specific registers (MSRs) for Intel Atom processor family, architectural MSR addresses are also included in Table B-4. These processors have a CPUID signature with DisplayFamily_DisplayModel of 06_1CH, see Table B-1.

The column "Shared/Unique" applies to logical processors sharing the same core in processors based on the Intel Atom microarchitecture. "Unique" means each logical processor has a separate MSR, or a bit field in an MSR governs only a logical processor. "Shared" means the MSR or the bit field in an MSR address governs the operation of both logical processors in the same core.

...

B.4 MSRS IN THE INTEL® MICROARCHITECTURE CODE NAME NEHALEM

Table B-5 lists model-specific registers (MSRs) that are common for Intel® microarchitecture code name Nehalem. These include Intel Core i7 and i5 processor family. Architectural MSR addresses are also included in Table B-5. These processors have a CPUID signature with DisplayFamily_DisplayModel of 06_1AH, 06_1EH, 06_1FH, 06_2EH, see Table B-1. Additional MSRs specific to 06_1AH, 06_1EH, 06_1FH are listed in Table B-6. Some MSRs listed in these tables are used by BIOS. More information about these MSR can be found at http://biosbits.org.

The column "Scope" represents the package/core/thread scope of individual bit field of an MSR. "Thread" means this bit field must be programmed on each logical processor independently. "Core" means the bit field must be programmed on each processor core independently, logical processors in the same core will be affected by change of this bit on the other logical processor in the same core. "Package" means the bit field must be programmed once for each physical package. Change of a bit filed with a package scope will affect all logical processors in that physical package.

Table B-5 MSRs in Processors Based on Intel Microarchitecture Code Name Nehalem

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
...				
CEH	206	MSR_PLATFORM_INFO	Package	see http://biosbits.org.
		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_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. See `http://biosbits.org`.
		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.
E4H	228	MSR_PMG_IO_CAPTURE_BASE	Core	**Power Management IO Redirection in C-state** (R/W) See http://biosbits.org.
		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.
...				
1AAH	426	MSR_MISC_PWR_MGMT		See http://biosbits.org.
		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_POWER_CURRENT_LIMIT		See http://biosbits.org.

		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
...				
1FCH	508	MSR_POWER_CTL	Core	Power Control Register. See http:// biosbits.org.
		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
...				

...

B.4.2 Additional MSRs In the Intel® Xeon® Processor 7500 Series

Intel Xeon Processor 7500 series support MSRs listed in Table B-5 (except MSR address 1ADH) and additional model-specific registers listed in Table B-7.

Table B-7 Additional MSRs in Intel Xeon Processor 7500 Series

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
1ADH	429	MSR_TURBO_RATIO_LIMIT	Package	**Reserved.** Attempt to read/write will cause #UD
...				

...

B.6 MSRS IN THE INTEL XEON PROCESSOR E7 FAMILY (INTEL® MICROARCHITECTURE CODE NAME WESTMERE)

Intel Xeon processor E7 family (Intel® microarchitecture code name Westmere) supports the MSR interfaces listed in Table B-5 (except MSR address 1ADH), Table B-6, plus additional MSR listed in Table B-9.

Table B-9 Additional MSRs Supported by Intel Xeon Processor E7 Family

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	**Reserved.** Attempt to read/write will cause #UD
1B0H	432	IA32_ENERGY_PERF_BIAS	Package	see Table B-2
F40H	3904	MSR_C8_PMON_BOX_CTRL	Package	Uncore C-box 8 perfmon local box control MSR
F41H	3905	MSR_C8_PMON_BOX_STATUS	Package	Uncore C-box 8 perfmon local box status MSR
F42H	3906	MSR_C8_PMON_BOX_OVF_CTRL	Package	Uncore C-box 8 perfmon local box overflow control MSR
F50H	3920	MSR_C8_PMON_EVNT_SEL0	Package	Uncore C-box 8 perfmon event select MSR
F51H	3921	MSR_C8_PMON_CTR0	Package	Uncore C-box 8 perfmon counter MSR
F52H	3922	MSR_C8_PMON_EVNT_SEL1	Package	Uncore C-box 8 perfmon event select MSR
F53H	3923	MSR_C8_PMON_CTR1	Package	Uncore C-box 8 perfmon counter MSR
F54H	3924	MSR_C8_PMON_EVNT_SEL2	Package	Uncore C-box 8 perfmon event select MSR
F55H	3925	MSR_C8_PMON_CTR2	Package	Uncore C-box 8 perfmon counter MSR
F56H	3926	MSR_C8_PMON_EVNT_SEL3	Package	Uncore C-box 8 perfmon event select MSR
F57H	3927	MSR_C8_PMON_CTR3	Package	Uncore C-box 8 perfmon counter MSR
F58H	3928	MSR_C8_PMON_EVNT_SEL4	Package	Uncore C-box 8 perfmon event select MSR
F59H	3929	MSR_C8_PMON_CTR4	Package	Uncore C-box 8 perfmon counter MSR

Table B-9 Additional MSRs Supported by Intel Xeon Processor E7 Family (Continued)

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
F5AH	3930	MSR_C8_PMON_EVNT_SEL5	Package	Uncore C-box 8 perfmon event select MSR
F5BH	3931	MSR_C8_PMON_CTR5	Package	Uncore C-box 8 perfmon counter MSR
FC0H	4032	MSR_C9_PMON_BOX_CTRL	Package	Uncore C-box 9 perfmon local box control MSR
FC1H	4033	MSR_C9_PMON_BOX_STATUS	Package	Uncore C-box 9 perfmon local box status MSR
FC2H	4034	MSR_C9_PMON_BOX_OVF_CTRL	Package	Uncore C-box 9 perfmon local box overflow control MSR
FD0H	4048	MSR_C9_PMON_EVNT_SEL0	Package	Uncore C-box 9 perfmon event select MSR
FD1H	4049	MSR_C9_PMON_CTR0	Package	Uncore C-box 9 perfmon counter MSR
FD2H	4050	MSR_C9_PMON_EVNT_SEL1	Package	Uncore C-box 9 perfmon event select MSR
FD3H	4051	MSR_C9_PMON_CTR1	Package	Uncore C-box 9 perfmon counter MSR
FD4H	4052	MSR_C9_PMON_EVNT_SEL2	Package	Uncore C-box 9 perfmon event select MSR
FD5H	4053	MSR_C9_PMON_CTR2	Package	Uncore C-box 9 perfmon counter MSR
FD6H	4054	MSR_C9_PMON_EVNT_SEL3	Package	Uncore C-box 9 perfmon event select MSR
FD7H	4055	MSR_C9_PMON_CTR3	Package	Uncore C-box 9 perfmon counter MSR
FD8H	4056	MSR_C9_PMON_EVNT_SEL4	Package	Uncore C-box 9 perfmon event select MSR
FD9H	4057	MSR_C9_PMON_CTR4	Package	Uncore C-box 9 perfmon counter MSR
FDAH	4058	MSR_C9_PMON_EVNT_SEL5	Package	Uncore C-box 9 perfmon event select MSR
FDBH	4059	MSR_C9_PMON_CTR5	Package	Uncore C-box 9 perfmon counter MSR

...

Table B-10 MSRs Supported by Intel Processors Based on Intel Microarchitecture Code Name Sandy Bridge

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
...				
CEH	206	MSR_PLATFORM_INFO	Package	See http://biosbits.org.
		7:0		Reserved.
...				
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. See http://biosbits.org.
		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
		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) See http://biosbits.org.
		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.
...				
1AAH	426	MSR_MISC_PWR_MGMT		See http://biosbits.org.
1ACH	428	MSR_TURBO_PWR_CURRENT_LIMIT		See http://biosbits.org.
...				
1FCH	508	MSR_POWER_CTL	Core	See http://biosbits.org.
...				
391H	913	MSR_UNC_PERF_GLOBAL_CTRL	Package	Uncore PMU global control

		0		Core 0 select
		1		Core 1 select
		2		Core 2 select
		3		Core 3 select
		18:4		Reserved
		29		Enable all uncore counters
		30		Enable PMI on overflow
		31		Enable Freezing counter when overflow
		63:32		Reserved.
392H	914	MSR_UNC_PERF_ GLOBAL_STATUS	Package	Uncore PMU main status
		0		Fixed counter overflowed
		1		CBox counter overflowed
		63:2		Reserved.
394H	916	MSR_UNC_PERF_ FIXED_CTRL	Package	Uncore fixed counter control (R/W)
		19:0		Reserved
		20		Enable overflow
		21		Reserved
		22		Enable counting
		63:23		Reserved.
395H	917	MSR_UNC_PERF_ FIXED_CTR	Package	Uncore fixed counter
		47:0		Current count
		63:48		Reserved.
...				
700H	1792	MSR_UNC_CBO_0_ PERFEVTSEL0	Package	Uncore C-Box 0, counter 0 event select MSR
701H	1793	MSR_UNC_CBO_0_ PERFEVTSEL1	Package	Uncore C-Box 0, counter 1 event select MSR
705H	1797	MSR_UNC_CBO_0_ UNIT_STATUS	Package	Uncore C-Box 0, Overflow Status
706H	1798	MSR_UNC_CBO_0_ PER_CTR0	Package	Uncore C-Box 0, performance counter 0
707H	1799	MSR_UNC_CBO_0_ PER_CTR1	Package	Uncore C-Box 0, performance counter 1
710H	1808	MSR_UNC_CBO_1_ PERFEVTSEL0	Package	Uncore C-Box 1, counter 0 event select MSR
711H	1809	MSR_UNC_CBO_1_ PERFEVTSEL1	Package	Uncore C-Box 1, counter 1 event select MSR

715H	1813	MSR_UNC_CBO_1_ UNIT_STATUS	Package	Uncore C-Box 1, Overflow Status
716H	1814	MSR_UNC_CBO_1_ PER_CTR0	Package	Uncore C-Box 1, performance counter 0
717H	1815	MSR_UNC_CBO_1_ PER_CTR1	Package	Uncore C-Box 1, performance counter 1
720H	1824	MSR_UNC_CBO_2_ PERFEVTSEL0	Package	Uncore C-Box 2, counter 0 event select MSR
721H	1824	MSR_UNC_CBO_2_ PERFEVTSEL1	Package	Uncore C-Box 2, counter 1 event select MSR
725H	1829	MSR_UNC_CBO_2_ UNIT_STATUS	Package	Uncore C-Box 2, Overflow Status
726H	1830	MSR_UNC_CBO_2_ PER_CTR0	Package	Uncore C-Box 2, performance counter 0
727H	1831	MSR_UNC_CBO_2_ PER_CTR1	Package	Uncore C-Box 2, performance counter 1
730H	1840	MSR_UNC_CBO_3_ PERFEVTSEL0	Package	Uncore C-Box 3, counter 0 event select MSR
731H	1841	MSR_UNC_CBO_3_ PERFEVTSEL1	Package	Uncore C-Box 3, counter 1 event select MSR
725H	1845	MSR_UNC_CBO_3_ UNIT_STATUS	Package	Uncore C-Box 3, Overflow Status
736H	1846	MSR_UNC_CBO_3_ PER_CTR0	Package	Uncore C-Box 3, performance counter 0
737H	1847	MSR_UNC_CBO_3_ PER_CTR1	Package	Uncore C-Box 3, performance counter 1

...

29. Updates to Appendix E, Volume 3B

Change bars show changes to Appendix E of the *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3B:* System Programming Guide, Part 2.

...

E.4.1 Internal Machine Check Errors

Table E-13 Machine Check Error Codes for IA32_MC4_STATUS

Type	Bit No.	Bit Function	Bit Description
MCA error codes[1]	0-15	MCACOD	
Model specific errors	19:16	Reserved except for the following	0000b - No Error 0001b - Non_IMem_Sel 0010b - I_Parity_Error 0011b - Bad_OpCode 0100b - I_Stack_Underflow 0101b - I_Stack_Overflow 0110b - D_Stack_Underflow 0111b - D_Stack_Overflow 1000b - Non-DMem_Sel 1001b - D_Parity_Error
	23-20	Reserved	Reserved
	31-24	Reserved except for the following	00h - No Error 0Dh - MC_IMC_FORCE_SR_S3_TIMEOUT 0Eh - MC_CPD_UNCPD_ST_TIMOUT 0Fh - MC_PKGS_SAFE_WP_TIMEOUT 43h - MC_PECI_MAILBOX_QUIESCE_TIMEOUT 5Ch - MC_MORE_THAN_ONE_LT_AGENT 60h - MC_INVALID_PKGS_REQ_PCH 61h - MC_INVALID_PKGS_REQ_QPI 62h - MC_INVALID_PKGS_RES_QPI 63h - MC_INVALID_PKGC_RES_PCH 64h - MC_INVALID_PKG_STATE_CONFIG 70h - MC_WATCHDG_TIMEOUT_PKGC_SLAVE 71h - MC_WATCHDG_TIMEOUT_PKGC_MASTER 70h - MC_WATCHDG_TIMEOUT_PKGS_MASTER 7ah - MC_HA_FAILSTS_CHANGE_DETECTED 81h - MC_RECOVERABLE_DIE_THERMAL_TOO_HOT
	56-32	Reserved	Reserved
Status register validity indicators[1]	57-63		

NOTES:

1. These fields are architecturally defined. Refer to Chapter 15, "Machine-Check Architecture," for more information.

...