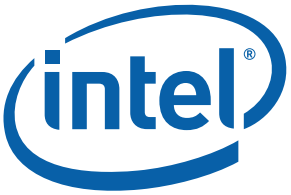


Intel® 64 and IA-32 Architectures Software Developer's Manual

Documentation Changes

July 2017

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

Document Number: 252046-055

Contents

Revision History

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

Revision	Description	Date
-024	• Removed Documentation Changes 1-21 • Added Documentation Changes 1-16	June 2009
-025	• Removed Documentation Changes 1-16 • Added Documentation Changes 1-18	September 2009
-026	• Removed Documentation Changes 1-18 • Added Documentation Changes 1-15	December 2009
-027	• Removed Documentation Changes 1-15 • Added Documentation Changes 1-24	March 2010
-028	• Removed Documentation Changes 1-24 • Added Documentation Changes 1-29	June 2010
-029	• Removed Documentation Changes 1-29 • Added Documentation Changes 1-29	September 2010
-030	• Removed Documentation Changes 1-29 • Added Documentation Changes 1-29	January 2011
-031	• Removed Documentation Changes 1-29 • Added Documentation Changes 1-29	April 2011
-032	• Removed Documentation Changes 1-29 • Added Documentation Changes 1-14	May 2011
-033	• Removed Documentation Changes 1-14 • Added Documentation Changes 1-38	October 2011
-034	• Removed Documentation Changes 1-38 • Added Documentation Changes 1-16	December 2011
-035	• Removed Documentation Changes 1-16 • Added Documentation Changes 1-18	March 2012
-036	• Removed Documentation Changes 1-18 • Added Documentation Changes 1-17	May 2012
-037	• Removed Documentation Changes 1-17 • Added Documentation Changes 1-28	August 2012
-038	• Removed Documentation Changes 1-28 • Add Documentation Changes 1-22	January 2013
-039	• Removed Documentation Changes 1-22 • Add Documentation Changes 1-17	June 2013
-040	• Removed Documentation Changes 1-17 • Add Documentation Changes 1-24	September 2013
-041	• Removed Documentation Changes 1-24 • Add Documentation Changes 1-20	February 2014
-042	• Removed Documentation Changes 1-20 • Add Documentation Changes 1-8	February 2014
-043	• Removed Documentation Changes 1-8 • Add Documentation Changes 1-43	June 2014
-044	• Removed Documentation Changes 1-43 • Add Documentation Changes 1-12	September 2014
-045	• Removed Documentation Changes 1-12 • Add Documentation Changes 1-22	January 2015
-046	• Removed Documentation Changes 1-22 • Add Documentation Changes 1-25	April 2015
-047	• Removed Documentation Changes 1-25 • Add Documentation Changes 1-19	June 2015

Revision	Description	Date
-048	• Removed Documentation Changes 1-19 • Add Documentation Changes 1-33	September 2015
-049	• Removed Documentation Changes 1-33 • Add Documentation Changes 1-33	December 2015
-050	• Removed Documentation Changes 1-33 • Add Documentation Changes 1-9	April 2016
-051	• Removed Documentation Changes 1-9 • Add Documentation Changes 1-20	June 2016
-052	• Removed Documentation Changes 1-20 • Add Documentation Changes 1-22	September 2016
-053	• Removed Documentation Changes 1-22 • Add Documentation Changes 1-26	December 2016
-054	• Removed Documentation Changes 1-26 • Add Documentation Changes 1-20	March 2017
-055	• Removed Documentation Changes 1-20 • Add Documentation Changes 1-28	July 2017

§

Preface

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

Affected Documents

Document Title	Document Number / Location
Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1: Basic Architecture	253665
Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A: Instruction Set Reference, A-L	253666
Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2B: Instruction Set Reference, M-U	253667
Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2C: Instruction Set Reference, V-Z	326018
Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2D: Instruction Set Reference	334569
Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A: System Programming Guide, Part 1	253668
Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3B: System Programming Guide, Part 2	253669
Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3C: System Programming Guide, Part 3	326019
Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3D: System Programming Guide, Part 4	332831
Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 4: Model Specific Registers	335592

Nomenclature

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

Summary Tables of Changes

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

Codes Used in Summary Tables

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

Documentation Changes(Sheet 1 of 2)

No.	DOCUMENTATION CHANGES
1	Updates to Chapter 1, Volume 1
2	Updates to Chapter 5, Volume 1
3	Updates to Chapter 13, Volume 1
4	Updates to Chapter 16, Volume 1
5	Updates to Chapter 1, Volume 2A
6	Updates to Chapter 2, Volume 2A
7	Updates to Chapter 3, Volume 2A
8	Updates to Chapter 4, Volume 2B
9	Updates to Chapter 5, Volume 2C
10	Updates to Chapter 1, Volume 3A
11	Updates to Chapter 7, Volume 3A
12	Updates to Chapter 9, Volume 3A
13	Updates to Chapter 17, Volume 3B
14	Updates to Chapter 18, Volume 3B
15	Updates to Chapter 19, Volume 3B
16	Updates to Chapter 20, Volume 3B
17	Updates to Chapter 23, Volume 3B
18	Updates to Chapter 25, Volume 3C
19	Updates to Chapter 27, Volume 3C
20	Updates to Chapter 35, Volume 3C
21	Updates to Chapter 36, Volume 3D
22	Updates to Chapter 37, Volume 3D
23	Updates to Chapter 38, Volume 3D
24	Updates to Chapter 39, Volume 3D
25	Updates to Chapter 40, Volume 3D

Documentation Changes(Sheet 2 of 2)

No.	DOCUMENTATION CHANGES
26	Updates to Appendix A, Volume 3D
27	Updates to Chapter 1, Volume 4
28	Updates to Chapter 2, Volume 4

Documentation Changes

Changes to the Intel® 64 and IA-32 Architectures Software Developer's Manual volumes follow, and are listed by chapter. Only chapters with changes are included in this document.

1. Updates to Chapter 1, Volume 1

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

Change to this chapter: Updates to list of processors supported and minor typo correction in Figure 1-2 "Syntax for CPUID, CR, and MSR Data Presentation".

CHAPTER 1
ABOUT THIS MANUAL

The *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1: Basic Architecture* (order number 253665) is part of a set that describes the architecture and programming environment of Intel® 64 and IA-32 architecture processors. Other volumes in this set are:

- The *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volumes 2A, 2B, 2C & 2D*: *Instruction Set Reference* (order numbers 253666, 253667, 326018 and 334569).
- The *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volumes 3A, 3B, 3C & 3D: System Programming Guide* (order numbers 253668, 253669, 326019 and 332831).
- The *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 4: Model-Specific Registers* (order number 335592).

The *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*, describes the basic architecture and programming environment of Intel 64 and IA-32 processors. The *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volumes 2A, 2B, 2C & 2D*, describe the instruction set of the processor and the opcode structure. These volumes apply to application programmers and to programmers who write operating systems or executives. The *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volumes 3A, 3B, 3C & 3D*, describe the operating-system support environment of Intel 64 and IA-32 processors. These volumes target operating-system and BIOS designers. In addition, the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3B*, addresses the programming environment for classes of software that host operating systems. The *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 4*, describes the model-specific registers of Intel 64 and IA-32 processors.

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

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

- Pentium® processors
- P6 family processors
- Pentium® 4 processors
- Pentium® M processors
- Intel® Xeon® processors
- Pentium® D processors
- Pentium® processor Extreme Editions
- 64-bit Intel® Xeon® processors
- Intel® Core™ Duo processor
- Intel® Core™ Solo processor
- Dual-Core Intel® Xeon® processor LV
- Intel® Core™2 Duo processor
- Intel® Core™2 Quad processor Q6000 series
- Intel® Xeon® processor 3000, 3200 series
- Intel® Xeon® processor 5000 series
- Intel® Xeon® processor 5100, 5300 series
- Intel® Core™2 Extreme processor X7000 and X6800 series
- Intel® Core™2 Extreme processor QX6000 series
- Intel® Xeon® processor 7100 series

- Intel® Pentium® Dual-Core processor
- Intel® Xeon® processor 7200, 7300 series
- Intel® Xeon® processor 5200, 5400, 7400 series
- Intel® Core™2 Extreme processor QX9000 and X9000 series
- Intel® Core™2 Quad processor Q9000 series
- Intel® Core™2 Duo processor E8000, T9000 series
- Intel® Atom™ processor family
- Intel® Atom™ processors 200, 300, D400, D500, D2000, N200, N400, N2000, E2000, Z500, Z600, Z2000, C1000 series are built from 45 nm and 32 nm processes
- Intel® Core™ i7 processor
- Intel® Core™ i5 processor
- Intel® Xeon® processor E7-8800/4800/2800 product families
- Intel® Core™ i7-3930K processor
- 2nd generation Intel® Core™ i7-2xxx, Intel® Core™ i5-2xxx, Intel® Core™ i3-2xxx processor series
- Intel® Xeon® processor E3-1200 product family
- Intel® Xeon® processor E5-2400/1400 product family
- Intel® Xeon® processor E5-4600/2600/1600 product family
- 3rd generation Intel® Core™ processors
- Intel® Xeon® processor E3-1200 v2 product family
- Intel® Xeon® processor E5-2400/1400 v2 product families
- Intel® Xeon® processor E5-4600/2600/1600 v2 product families
- Intel® Xeon® processor E7-8800/4800/2800 v2 product families
- 4th generation Intel® Core™ processors
- The Intel® Core™ M processor family
- Intel® Core™ i7-59xx Processor Extreme Edition
- Intel® Core™ i7-49xx Processor Extreme Edition
- Intel® Xeon® processor E3-1200 v3 product family
- Intel® Xeon® processor E5-2600/1600 v3 product families
- 5th generation Intel® Core™ processors
- Intel® Xeon® processor D-1500 product family
- Intel® Xeon® processor E5 v4 family
- Intel® Atom™ processor X7-Z8000 and X5-Z8000 series
- Intel® Atom™ processor Z3400 series
- Intel® Atom™ processor Z3500 series
- 6th generation Intel® Core™ processors
- Intel® Xeon® processor E3-1500m v5 product family
- 7th generation Intel® Core™ processors
- Intel® Xeon Phi™ Processor 3200, 5200, 7200 Series
- Intel® Xeon® Processor Scalable Family

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

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

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

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

The Intel® Atom™ processors 200, 300, D400, D500, D2000, N200, N400, N2000, E2000, Z500, Z600, Z2000, C1000 series are based on the Intel® Atom™ microarchitecture and supports Intel 64 architecture.

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

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

The Intel® Core™ i7 processor and Intel® Xeon® processor 3400, 5500, 7500 series are based on 45 nm Intel® microarchitecture code name Nehalem. Intel® microarchitecture code name Westmere is a 32 nm version of Intel® microarchitecture code name Nehalem. Intel® Xeon® processor 5600 series, Intel Xeon processor E7 and various Intel Core i7, i5, i3 processors are based on Intel® microarchitecture code name Westmere. These processors support Intel 64 architecture.

The Intel® Xeon® processor E5 family, Intel® Xeon® processor E3-1200 family, Intel® Xeon® processor E7-8800/4800/2800 product families, Intel® Core™ i7-3930K processor, and 2nd generation Intel® Core™ i7-2xxx, Intel® Core™ i5-2xxx, Intel® Core™ i3-2xxx processor series are based on the Intel® microarchitecture code name Sandy Bridge and support Intel 64 architecture.

The Intel® Xeon® processor E7-8800/4800/2800 v2 product families, Intel® Xeon® processor E3-1200 v2 product family and 3rd generation Intel® Core™ processors are based on the Intel® microarchitecture code name Ivy Bridge and support Intel 64 architecture.

The Intel® Xeon® processor E5-4600/2600/1600 v2 product families, Intel® Xeon® processor E5-2400/1400 v2 product families and Intel® Core™ i7-49xx Processor Extreme Edition are based on the Intel® microarchitecture code name Ivy Bridge-E and support Intel 64 architecture.

The Intel® Xeon® processor E3-1200 v3 product family and 4th Generation Intel® Core™ processors are based on the Intel® microarchitecture code name Haswell and support Intel 64 architecture.

The Intel® Xeon® processor E5-2600/1600 v3 product families and the Intel® Core™ i7-59xx Processor Extreme Edition are based on the Intel® microarchitecture code name Haswell-E and support Intel 64 architecture.

The Intel® Atom™ processor Z8000 series is based on the Intel microarchitecture code name Airmont.

The Intel® Atom™ processor Z3400 series and the Intel® Atom™ processor Z3500 series are based on the Intel microarchitecture code name Silvermont.

The Intel® Core™ M processor family, 5th generation Intel® Core™ processors, Intel® Xeon® processor D-1500 product family and the Intel® Xeon® processor E5 v4 family are based on the Intel® microarchitecture code name Broadwell and support Intel 64 architecture.

The Intel® Xeon® Processor Scalable Family, Intel® Xeon® processor E3-1500m v5 product family and 6th generation Intel® Core™ processors are based on the Intel® microarchitecture code name Skylake and support Intel 64 architecture.

The 7th generation Intel® Core™ processors are based on the Intel® microarchitecture code name Kaby Lake and support Intel 64 architecture.

The Intel® Xeon Phi™ Processor 3200, 5200, 7200 Series is based on the Intel® microarchitecture code name Knights Landing and supports Intel 64 architecture.

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

1.2 OVERVIEW OF VOLUME 1: BASIC ARCHITECTURE

A description of this manual's content follows:

Chapter 1 — About This Manual. Gives an overview of all five volumes of the *Intel® 64 and IA-32 Architectures Software Developer's Manual.* It also describes the notational conventions in these manuals and lists related Intel manuals and documentation of interest to programmers and hardware designers.

Chapter 2 — Intel® 64 and IA-32 Architectures. Introduces the Intel 64 and IA-32 architectures along with the families of Intel processors that are based on these architectures. It also gives an overview of the common features found in these processors and brief history of the Intel 64 and IA-32 architectures.

Chapter 3 — Basic Execution Environment. Introduces the models of memory organization and describes the register set used by applications.

Chapter 4 — Data Types. Describes the data types and addressing modes recognized by the processor; provides an overview of real numbers and floating-point formats and of floating-point exceptions.

Chapter 5 — Instruction Set Summary. Lists all Intel 64 and IA-32 instructions, divided into technology groups.

Chapter 6 — Procedure Calls, Interrupts, and Exceptions. Describes the procedure stack and mechanisms provided for making procedure calls and for servicing interrupts and exceptions.

Chapter 7 — Programming with General-Purpose Instructions. Describes basic load and store, program control, arithmetic, and string instructions that operate on basic data types, general-purpose and segment registers; also describes system instructions that are executed in protected mode.

Chapter 8 — Programming with the x87 FPU. Describes the x87 floating-point unit (FPU), including floating-point registers and data types; gives an overview of the floating-point instruction set and describes the processor's floating-point exception conditions.

Chapter 9 — Programming with Intel® MMX™ Technology. Describes Intel MMX technology, including MMX registers and data types; also provides an overview of the MMX instruction set.

Chapter 10 — Programming with Intel® Streaming SIMD Extensions (Intel® SSE). Describes SSE extensions, including XMM registers, the MXCSR register, and packed single-precision floating-point data types; provides an overview of the SSE instruction set and gives guidelines for writing code that accesses the SSE extensions.

Chapter 11 — Programming with Intel® Streaming SIMD Extensions 2 (Intel® SSE2). Describes SSE2 extensions, including XMM registers and packed double-precision floating-point data types; provides an overview of the SSE2 instruction set and gives guidelines for writing code that accesses SSE2 extensions. This chapter also describes SIMD floating-point exceptions that can be generated with SSE and SSE2 instructions. It also provides general guidelines for incorporating support for SSE and SSE2 extensions into operating system and applications code.

Chapter 12 — Programming with Intel® Streaming SIMD Extensions 3 (Intel® SSE3), Supplemental Streaming SIMD Extensions 3 (SSSE3), Intel® Streaming SIMD Extensions 4 (Intel® SSE4) and Intel® AES New Instructions (Intel® AESNI). Provides an overview of the SSE3 instruction set, Supplemental SSE3, SSE4, AESNI instructions, and guidelines for writing code that accesses these extensions.

Chapter 13 — Managing State Using the XSAVE Feature Set. Describes the XSAVE feature set instructions and explains how software can enable the XSAVE feature set and XSAVE-enabled features.

Chapter 14 — Programming with AVX, FMA and AVX2. Provides an overview of the Intel® AVX instruction set, FMA and Intel AVX2 extensions and gives guidelines for writing code that accesses these extensions.

Chapter 15 — Programming with Intel Transactional Synchronization Extensions. Describes the instruction extensions that support lock elision techniques to improve the performance of multi-threaded software with contended locks.

Chapter 16 — Input/Output. Describes the processor's I/O mechanism, including I/O port addressing, I/O instructions, and I/O protection mechanisms.

Chapter 17 — Processor Identification and Feature Determination. Describes how to determine the CPU type and features available in the processor.

Appendix A — EFLAGS Cross-Reference. Summarizes how the IA-32 instructions affect the flags in the EFLAGS register.

Appendix B — EFLAGS Condition Codes. Summarizes how conditional jump, move, and 'byte set on condition code' instructions use condition code flags (OF, CF, ZF, SF, and PF) in the EFLAGS register.

Appendix C — Floating-Point Exceptions Summary. Summarizes exceptions raised by the x87 FPU floating-point and SSE/SSE2/SSE3 floating-point instructions.

Appendix D — Guidelines for Writing x87 FPU Exception Handlers. Describes how to design and write MS-DOS* compatible exception handling facilities for FPU exceptions (includes software and hardware requirements and assembly-language code examples). This appendix also describes general techniques for writing robust FPU exception handlers.

Appendix E — Guidelines for Writing SIMD Floating-Point Exception Handlers. Gives guidelines for writing exception handlers for exceptions generated by SSE/SSE2/SSE3 floating-point instructions.

1.3 NOTATIONAL CONVENTIONS

This manual uses specific notation for data-structure formats, for symbolic representation of instructions, and for hexadecimal and binary numbers. This notation is described below.

1.3.1 Bit and Byte Order

In illustrations of data structures in memory, smaller addresses appear toward the bottom of the figure; addresses increase toward the top. Bit positions are numbered from right to left. The numerical value of a set bit is equal to two raised to the power of the bit position. Intel 64 and IA-32 processors are "little endian" machines; this means the bytes of a word are numbered starting from the least significant byte. See Figure 1-1.

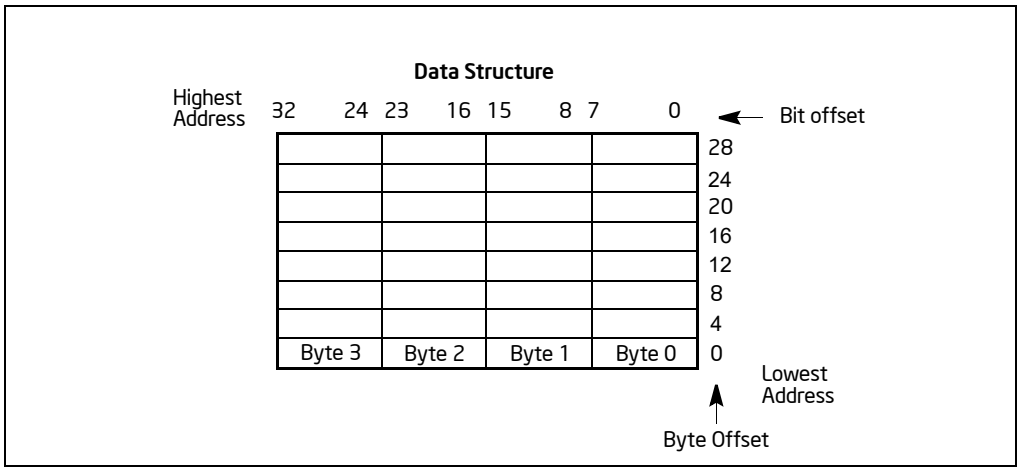


Figure 1-1. Bit and Byte Order

1.3.2 Reserved Bits and Software Compatibility

In many register and memory layout descriptions, certain bits are marked as **reserved**. When bits are marked as reserved, it is essential for compatibility with future processors that software treat these bits as having a future, though unknown, effect. The behavior of reserved bits should be regarded as not only undefined, but unpredictable.

Software should follow these guidelines in dealing with reserved bits:

- Do not depend on the states of any reserved bits when testing the values of registers that contain such bits. Mask out the reserved bits before testing.
- Do not depend on the states of any reserved bits when storing to memory or to a register.
- Do not depend on the ability to retain information written into any reserved bits.

- When loading a register, always load the reserved bits with the values indicated in the documentation, if any, or reload them with values previously read from the same register.

NOTE

Avoid any software dependence upon the state of reserved bits in Intel 64 and IA-32 registers. Depending upon the values of reserved register bits will make software dependent upon the unspecified manner in which the processor handles these bits. Programs that depend upon reserved values risk incompatibility with future processors.

1.3.2.1 Instruction Operands

When instructions are represented symbolically, a subset of the IA-32 assembly language is used. In this subset, an instruction has the following format:

```
label: mnemonic argument1, argument2, argument3
```

where:

- A **label** is an identifier which is followed by a colon.
- A **mnemonic** is a reserved name for a class of instruction opcodes which have the same function.
- The operands **argument1**, **argument2**, and **argument3** are optional. There may be from zero to three operands, depending on the opcode. When present, they take the form of either literals or identifiers for data items. Operand identifiers are either reserved names of registers or are assumed to be assigned to data items declared in another part of the program (which may not be shown in the example).

When two operands are present in an arithmetic or logical instruction, the right operand is the source and the left operand is the destination.

For example:

```
LOADREG: MOV EAX, SUBTOTAL
```

In this example, LOADREG is a label, MOV is the mnemonic identifier of an opcode, EAX is the destination operand, and SUBTOTAL is the source operand. Some assembly languages put the source and destination in reverse order.

1.3.3 Hexadecimal and Binary Numbers

Base 16 (hexadecimal) numbers are represented by a string of hexadecimal digits followed by the character H (for example, 0F82EH). A hexadecimal digit is a character from the following set: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, and F.

Base 2 (binary) numbers are represented by a string of 1s and 0s, sometimes followed by the character B (for example, 1010B). The "B" designation is only used in situations where confusion as to the type of number might arise.

1.3.4 Segmented Addressing

The processor uses byte addressing. This means memory is organized and accessed as a sequence of bytes. Whether one or more bytes are being accessed, a byte address is used to locate the byte or bytes memory. The range of memory that can be addressed is called an **address space**.

The processor also supports segmented addressing. This is a form of addressing where a program may have many independent address spaces, called **segments**. For example, a program can keep its code (instructions) and stack in separate segments. Code addresses would always refer to the code space, and stack addresses would always refer to the stack space. The following notation is used to specify a byte address within a segment:

```
Segment-register:Byte-address
```

For example, the following segment address identifies the byte at address FF79H in the segment pointed by the DS register:

DS:FF79H

The following segment address identifies an instruction address in the code segment. The CS register points to the code segment and the EIP register contains the address of the instruction.

CS:EIP

1.3.5 A New Syntax for CPUID, CR, and MSR Values

Obtain feature flags, status, and system information by using the CPUID instruction, by checking control register bits, and by reading model-specific registers. We are moving toward a new syntax to represent this information. See Figure 1-2.

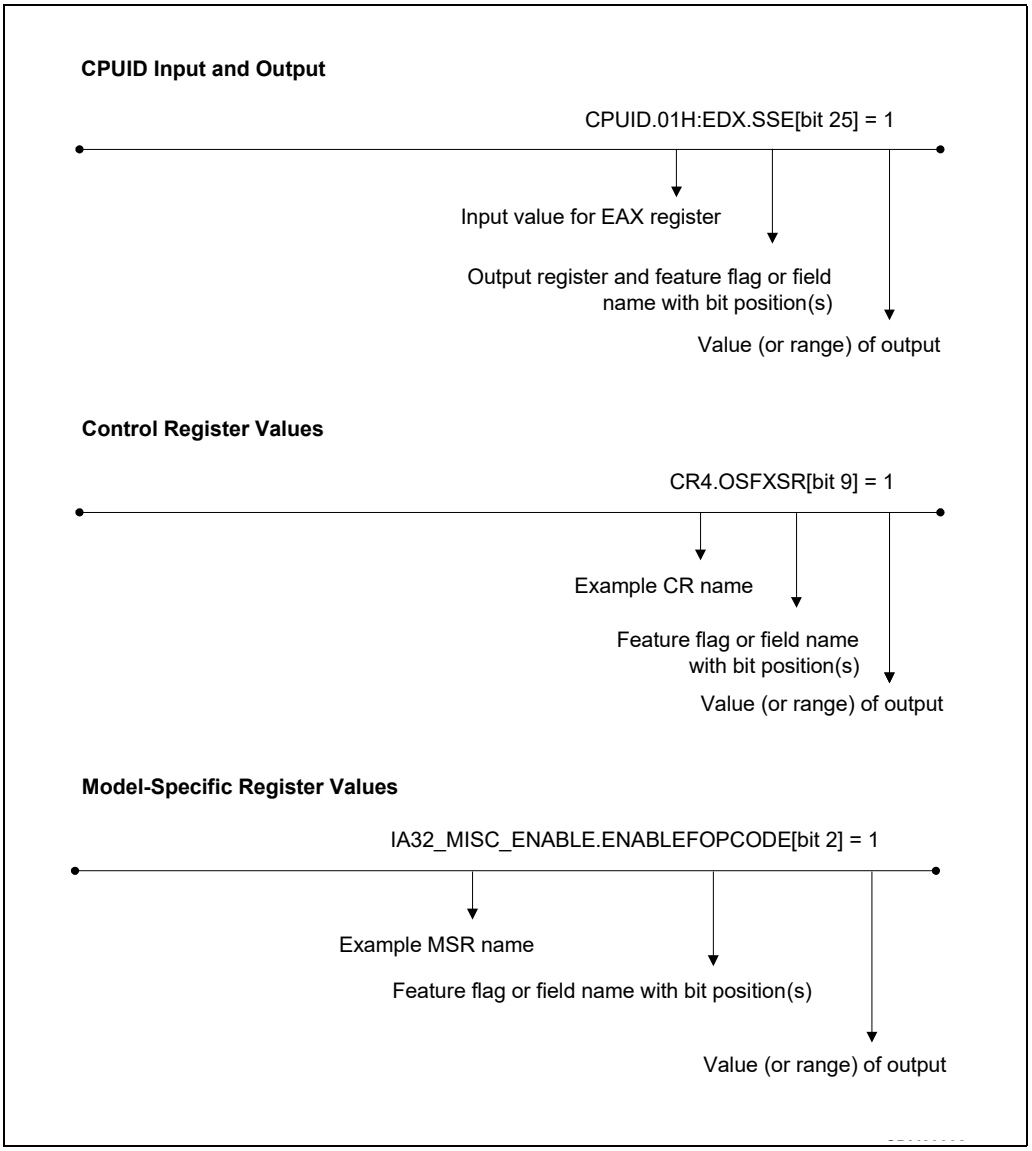


Figure 1-2. Syntax for CPUID, CR, and MSR Data Presentation

1.3.6 Exceptions

An exception is an event that typically occurs when an instruction causes an error. For example, an attempt to divide by zero generates an exception. However, some exceptions, such as breakpoints, occur under other conditions. Some types of exceptions may provide error codes. An error code reports additional information about the error. An example of the notation used to show an exception and error code is shown below:

```
#PF(fault code)
```

This example refers to a page-fault exception under conditions where an error code naming a type of fault is reported. Under some conditions, exceptions that produce error codes may not be able to report an accurate code. In this case, the error code is zero, as shown below for a general-protection exception:

```
#GP(0)
```

1.4 RELATED LITERATURE

Literature related to Intel 64 and IA-32 processors is listed and viewable on-line at:

http://www.intel.com/content/www/us/en/processors/architectures-software-developer-manuals.html

See also:

- The data sheet for a particular Intel 64 or IA-32 processor
- The specification update for a particular Intel 64 or IA-32 processor
- Intel® C++ Compiler documentation and online help:
  http://software.intel.com/en-us/articles/intel-compilers/
- Intel® Fortran Compiler documentation and online help:
  http://software.intel.com/en-us/articles/intel-compilers/
- Intel® Software Development Tools:
  https://software.intel.com/en-us/intel-sdp-home
- Intel® 64 and IA-32 Architectures Software Developer's Manual (in one, four or ten volumes):
  https://software.intel.com/en-us/articles/intel-sdm
- Intel® 64 and IA-32 Architectures Optimization Reference Manual:
  https://software.intel.com/en-us/articles/intel-sdm#optimization
- Intel 64 Architecture x2APIC Specification:

  http://www.intel.com/content/www/us/en/architecture-and-technology/64-architecture-x2apic-specification.html
- Intel® Trusted Execution Technology Measured Launched Environment Programming Guide:

  http://www.intel.com/content/www/us/en/software-developers/intel-txt-software-development-guide.html
- Developing Multi-threaded Applications: A Platform Consistent Approach:
  https://software.intel.com/sites/default/files/article/147714/51534-developing-multithreaded-applications.pdf
- Using Spin-Loops on Intel® Pentium® 4 Processor and Intel® Xeon® Processor:
  https://software.intel.com/sites/default/files/22/30/25602
- Performance Monitoring Unit Sharing Guide
  http://software.intel.com/file/30388

Literature related to selected features in future Intel processors are available at:

- Intel® Architecture Instruction Set Extensions Programming Reference
  https://software.intel.com/en-us/isa-extensions
- Intel® Software Guard Extensions (Intel® SGX) Programming Reference
  https://software.intel.com/en-us/isa-extensions/intel-sgx

More relevant links are:

- Intel® Developer Zone:
  https://software.intel.com/en-us
- Developer centers:
  http://www.intel.com/content/www/us/en/hardware-developers/developer-centers.html
- Processor support general link:
  http://www.intel.com/support/processors/
- Intel® Hyper-Threading Technology (Intel® HT Technology):
  http://www.intel.com/technology/platform-technology/hyper-threading/index.htm

2. Updates to Chapter 5, Volume 1

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

Change to this chapter: Typo correction in Section 5.6.1.2 "SSE2 Packed Arithmetic Instructions". Typo corrections in Section 5.25 “Intel® Software Guard Extensions”.

CHAPTER 5
INSTRUCTION SET SUMMARY

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

- General purpose
- x87 FPU
- x87 FPU and SIMD state management
- Intel® MMX technology
- SSE extensions
- SSE2 extensions
- SSE3 extensions
- SSSE3 extensions
- SSE4 extensions
- AESNI and PCLMULQDQ
- Intel® AVX extensions
- F16C, RDRAND, RDSEED, FS/GS base access
- FMA extensions
- Intel® AVX2 extensions
- Intel® Transactional Synchronization extensions
- System instructions
- IA-32e mode: 64-bit mode instructions
- VMX instructions
- SMX instructions
- ADCX and ADOX
- Intel® Memory Protection Extensions
- Intel® Security Guard Extensions

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

Table 5-1. Instruction Groups in Intel 64 and IA-32 Processors

Instruction Set Architecture	Intel 64 and IA-32 Processor Support
General Purpose	All Intel 64 and IA-32 processors.
x87 FPU	Intel486, Pentium, Pentium with MMX Technology, Celeron, Pentium Pro, Pentium II, Pentium II Xeon, Pentium III, Pentium III Xeon, Pentium 4, Intel Xeon processors, Pentium M, Intel Core Solo, Intel Core Duo, Intel Core 2 Duo processors, Intel Atom processors.
x87 FPU and SIMD State Management	Pentium II, Pentium II Xeon, Pentium III, Pentium III Xeon, Pentium 4, Intel Xeon processors, Pentium M, Intel Core Solo, Intel Core Duo, Intel Core 2 Duo processors, Intel Atom processors.
MMX Technology	Pentium with MMX Technology, Celeron, Pentium II, Pentium II Xeon, Pentium III, Pentium III Xeon, Pentium 4, Intel Xeon processors, Pentium M, Intel Core Solo, Intel Core Duo, Intel Core 2 Duo processors, Intel Atom processors.
SSE Extensions	Pentium III, Pentium III Xeon, Pentium 4, Intel Xeon processors, Pentium M, Intel Core Solo, Intel Core Duo, Intel Core 2 Duo processors, Intel Atom processors.

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

Instruction Set Architecture	Intel 64 and IA-32 Processor Support
SSE2 Extensions	Pentium 4, Intel Xeon processors, Pentium M, Intel Core Solo, Intel Core Duo, Intel Core 2 Duo processors, Intel Atom processors.
SSE3 Extensions	Pentium 4 supporting HT Technology (built on 90nm process technology), Intel Core Solo, Intel Core Duo, Intel Core 2 Duo processors, Intel Xeon processor 3xxxx, 5xxx, 7xxx Series, Intel Atom processors.
SSSE3 Extensions	Intel Xeon processor 3xxx, 5100, 5200, 5300, 5400, 5500, 5600, 7300, 7400, 7500 series, Intel Core 2 Extreme processors QX6000 series, Intel Core 2 Duo, Intel Core 2 Quad processors, Intel Pentium Dual-Core processors, Intel Atom processors.
IA-32e mode: 64-bit mode instructions	Intel 64 processors.
System Instructions	Intel 64 and IA-32 processors.
VMX Instructions	Intel 64 and IA-32 processors supporting Intel Virtualization Technology.
SMX Instructions	Intel Core 2 Duo processor E6x50, E8xxx; Intel Core 2 Quad processor Q9xxx.

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

Instruction Set Architecture	Processor Generation Introduction
SSE4.1 Extensions	Intel Xeon processor 3100, 3300, 5200, 5400, 7400, 7500 series, Intel Core 2 Extreme processors QX9000 series, Intel Core 2 Quad processor Q9000 series, Intel Core 2 Duo processors 8000 series, T9000 series.
SSE4.2 Extensions, CRC32, POPCNT	Intel Core i7 965 processor, Intel Xeon processors X3400, X3500, X5500, X6500, X7500 series.
AESNI, PCLMULQDQ	Intel Xeon processor E7 series, Intel Xeon processors X3600, X5600, Intel Core i7 980X processor; Use CPUID to verify presence of AESNI and PCLMULQDQ across Intel Core processor families.
Intel AVX	Intel Xeon processor E3 and E5 families; 2nd Generation Intel Core i7, i5, i3 processor 2xxx families.
F16C, RDRAND, FS/GS base access	3rd Generation Intel Core processors, Intel Xeon processor E3-1200 v2 product family, Next Generation Intel Xeon processors, Intel Xeon processor E5 v2 and E7 v2 families.
FMA, AVX2, BMI1, BMI2, INVPCID	Intel Xeon processor E3-1200 v3 product family; 4th Generation Intel Core processor family.
TSX	Intel Xeon processor E7 v3 product family.
ADX, RDSEED, CLAC, STAC	Intel Core M processor family; 5th Generation Intel Core processor family.
CLFLUSHOPT, XSAVEC, XSAVES, MPX, SGX1	6th Generation Intel Core processor family.

The following sections list instructions in each major group and subgroup. Given for each instruction is its mnemonic and descriptive names. When two or more mnemonics are given (for example, CMOVA/CMOVNBE), they represent different mnemonics for the same instruction opcode. Assemblers support redundant mnemonics for some instructions to make it easier to read code listings. For instance, CMOVA (Conditional move if above) and CMOVNBE (Conditional move if not below or equal) represent the same condition. For detailed information about specific instructions, see the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volumes 2A, 2B, 2C & 2D.*

5.1 GENERAL-PURPOSE INSTRUCTIONS

The general-purpose instructions preform basic data movement, arithmetic, logic, program flow, and string operations that programmers commonly use to write application and system software to run on Intel 64 and IA-32 processors. They operate on data contained in memory, in the general-purpose registers (EAX, EBX, ECX, EDX, EDI, ESI, EBP, and ESP) and in the EFLAGS register. They also operate on address information contained in memory, the general-purpose registers, and the segment registers (CS, DS, SS, ES, FS, and GS).

This group of instructions includes the data transfer, binary integer arithmetic, decimal arithmetic, logic operations, shift and rotate, bit and byte operations, program control, string, flag control, segment register operations, and miscellaneous subgroups. The sections that following introduce each subgroup.

For more detailed information on general purpose-instructions, see Chapter 7, "Programming With General-Purpose Instructions."

5.1.1 Data Transfer Instructions

The data transfer instructions move data between memory and the general-purpose and segment registers. They also perform specific operations such as conditional moves, stack access, and data conversion.

MOV	Move data between general-purpose registers; move data between memory and general-purpose or segment registers; move immediates to general-purpose registers.
CMOVE/CMOVZ	Conditional move if equal/Conditional move if zero.
CMOVNE/CMOVNZ	Conditional move if not equal/Conditional move if not zero.
CMOVA/CMOVNBE	Conditional move if above/Conditional move if not below or equal.
CMOVAE/CMOVNB	Conditional move if above or equal/Conditional move if not below.
CMOVB/CMOVNAE	Conditional move if below/Conditional move if not above or equal.
CMOVBE/CMOVNA	Conditional move if below or equal/Conditional move if not above.
CMOVG/CMOVNLE	Conditional move if greater/Conditional move if not less or equal.
CMOVGE/CMOVNL	Conditional move if greater or equal/Conditional move if not less.
CMOVL/CMOVNGE	Conditional move if less/Conditional move if not greater or equal.
CMOVLE/CMOVNG	Conditional move if less or equal/Conditional move if not greater.
CMOVC	Conditional move if carry.
CMOVNC	Conditional move if not carry.
CMOVO	Conditional move if overflow.
CMOVNO	Conditional move if not overflow.
CMOVS	Conditional move if sign (negative).
CMOVNS	Conditional move if not sign (non-negative).
CMOVP/CMOVPE	Conditional move if parity/Conditional move if parity even.
CMOVNP/CMOVPO	Conditional move if not parity/Conditional move if parity odd.
XCHG	Exchange.
BSWAP	Byte swap.
XADD	Exchange and add.
CMPXCHG	Compare and exchange.
CMPXCHG8B	Compare and exchange 8 bytes.
PUSH	Push onto stack.
POP	Pop off of stack.
PUSHA/PUSHAD	Push general-purpose registers onto stack.
POPA/POPAD	Pop general-purpose registers from stack.
CWD/CDQ	Convert word to doubleword/Convert doubleword to quadword.
CBW/CWDE	Convert byte to word/Convert word to doubleword in EAX register.
MOVSX	Move and sign extend.

MOVZX	Move and zero extend.

5.1.2 Binary Arithmetic Instructions

The binary arithmetic instructions perform basic binary integer computations on byte, word, and doubleword integers located in memory and/or the general purpose registers.

ADCX	Unsigned integer add with carry.
ADOX	Unsigned integer add with overflow.
ADD	Integer add.
ADC	Add with carry.
SUB	Subtract.
SBB	Subtract with borrow.
IMUL	Signed multiply.
MUL	Unsigned multiply.
IDIV	Signed divide.
DIV	Unsigned divide.
INC	Increment.
DEC	Decrement.
NEG	Negate.
CMP	Compare.

5.1.3 Decimal Arithmetic Instructions

The decimal arithmetic instructions perform decimal arithmetic on binary coded decimal (BCD) data.

DAA	Decimal adjust after addition.
DAS	Decimal adjust after subtraction.
AAA	ASCII adjust after addition.
AAS	ASCII adjust after subtraction.
AAM	ASCII adjust after multiplication.
AAD	ASCII adjust before division.

5.1.4 Logical Instructions

The logical instructions perform basic AND, OR, XOR, and NOT logical operations on byte, word, and doubleword values.

AND	Perform bitwise logical AND.
OR	Perform bitwise logical OR.
XOR	Perform bitwise logical exclusive OR.
NOT	Perform bitwise logical NOT.

5.1.5 Shift and Rotate Instructions

The shift and rotate instructions shift and rotate the bits in word and doubleword operands.

SAR	Shift arithmetic right.
SHR	Shift logical right.
SAL/SHL	Shift arithmetic left/Shift logical left.
SHRD	Shift right double.

SHLD	Shift left double.
ROR	Rotate right.
ROL	Rotate left.
RCR	Rotate through carry right.
RCL	Rotate through carry left.

5.1.6 Bit and Byte Instructions

Bit instructions test and modify individual bits in word and doubleword operands. Byte instructions set the value of a byte operand to indicate the status of flags in the EFLAGS register.

BT	Bit test.
BTS	Bit test and set.
BTR	Bit test and reset.
BTC	Bit test and complement.
BSF	Bit scan forward.
BSR	Bit scan reverse.
SETE/SETZ	Set byte if equal/Set byte if zero.
SETNE/SETNZ	Set byte if not equal/Set byte if not zero.
SETA/SETNBE	Set byte if above/Set byte if not below or equal.
SETAE/SETNB/SETNC	Set byte if above or equal/Set byte if not below/Set byte if not carry.
SETB/SETNAE/SETC	Set byte if below/Set byte if not above or equal/Set byte if carry.
SETBE/SETNA	Set byte if below or equal/Set byte if not above.
SETG/SETNLE	Set byte if greater/Set byte if not less or equal.
SETGE/SETNL	Set byte if greater or equal/Set byte if not less.
SETL/SETNGE	Set byte if less/Set byte if not greater or equal.
SETLE/SETNG	Set byte if less or equal/Set byte if not greater.
SETS	Set byte if sign (negative).
SETNS	Set byte if not sign (non-negative).
SETO	Set byte if overflow.
SETNO	Set byte if not overflow.
SETPE/SETP	Set byte if parity even/Set byte if parity.
SETPO/SETNP	Set byte if parity odd/Set byte if not parity.
TEST	Logical compare.
CRC32[1]	Provides hardware acceleration to calculate cyclic redundancy checks for fast and efficient implementation of data integrity protocols.
POPCNT[2]	This instruction calculates of number of bits set to 1 in the second operand (source) and returns the count in the first operand (a destination register).

5.1.7 Control Transfer Instructions

The control transfer instructions provide jump, conditional jump, loop, and call and return operations to control program flow.

JMP	Jump.
JE/JZ	Jump if equal/Jump if zero.
JNE/JNZ	Jump if not equal/Jump if not zero.

1. Processor support of CRC32 is enumerated by CPUID.01:ECX[SSE4.2] = 1
2. Processor support of POPCNT is enumerated by CPUID.01:ECX[POPCNT] = 1

JA/JNBE	Jump if above/Jump if not below or equal.
JAE/JNB	Jump if above or equal/Jump if not below.
JB/JNAE	Jump if below/Jump if not above or equal.
JBE/JNA	Jump if below or equal/Jump if not above.
JG/JNLE	Jump if greater/Jump if not less or equal.
JGE/JNL	Jump if greater or equal/Jump if not less.
JL/JNGE	Jump if less/Jump if not greater or equal.
JLE/JNG	Jump if less or equal/Jump if not greater.
JC	Jump if carry.
JNC	Jump if not carry.
JO	Jump if overflow.
JNO	Jump if not overflow.
JS	Jump if sign (negative).
JNS	Jump if not sign (non-negative).
JPO/JNP	Jump if parity odd/Jump if not parity.
JPE/JP	Jump if parity even/Jump if parity.
JCXZ/JECXZ	Jump register CX zero/Jump register ECX zero.
LOOP	Loop with ECX counter.
LOOPZ/LOOPE	Loop with ECX and zero/Loop with ECX and equal.
LOOPNZ/LOOPNE	Loop with ECX and not zero/Loop with ECX and not equal.
CALL	Call procedure.
RET	Return.
IRET	Return from interrupt.
INT	Software interrupt.
INTO	Interrupt on overflow.
BOUND	Detect value out of range.
ENTER	High-level procedure entry.
LEAVE	High-level procedure exit.

5.1.8 String Instructions

The string instructions operate on strings of bytes, allowing them to be moved to and from memory.

MOVS/MOVSB	Move string/Move byte string.
MOVS/MOVSW	Move string/Move word string.
MOVS/MOVSD	Move string/Move doubleword string.
CMPS/CMPSB	Compare string/Compare byte string.
CMPS/CMPSW	Compare string/Compare word string.
CMPS/CMPSD	Compare string/Compare doubleword string.
SCAS/SCASB	Scan string/Scan byte string.
SCAS/SCASW	Scan string/Scan word string.
SCAS/SCASD	Scan string/Scan doubleword string.
LODS/LODSB	Load string/Load byte string.
LODS/LODSW	Load string/Load word string.
LODS/LODSD	Load string/Load doubleword string.
STOS/STOSB	Store string/Store byte string.
STOS/STOSW	Store string/Store word string.

STOS/STOSD	Store string/Store doubleword string.
REP	Repeat while ECX not zero.
REPE/REPZ	Repeat while equal/Repeat while zero.
REPNE/REPNZ	Repeat while not equal/Repeat while not zero.

5.1.9 I/O Instructions

These instructions move data between the processor's I/O ports and a register or memory.

IN	Read from a port.
OUT	Write to a port.
INS/INSB	Input string from port/Input byte string from port.
INS/INSW	Input string from port/Input word string from port.
INS/INSD	Input string from port/Input doubleword string from port.
OUTS/OUTSB	Output string to port/Output byte string to port.
OUTS/OUTSW	Output string to port/Output word string to port.
OUTS/OUTSD	Output string to port/Output doubleword string to port.

5.1.10 Enter and Leave Instructions

These instructions provide machine-language support for procedure calls in block-structured languages.

ENTER	High-level procedure entry.
LEAVE	High-level procedure exit.

5.1.11 Flag Control (EFLAG) Instructions

The flag control instructions operate on the flags in the EFLAGS register.

STC	Set carry flag.
CLC	Clear the carry flag.
CMC	Complement the carry flag.
CLD	Clear the direction flag.
STD	Set direction flag.
LAHF	Load flags into AH register.
SAHF	Store AH register into flags.
PUSHF/PUSHFD	Push EFLAGS onto stack.
POPF/POPFD	Pop EFLAGS from stack.
STI	Set interrupt flag.
CLI	Clear the interrupt flag.

5.1.12 Segment Register Instructions

The segment register instructions allow far pointers (segment addresses) to be loaded into the segment registers.

LDS	Load far pointer using DS.
LES	Load far pointer using ES.
LFS	Load far pointer using FS.
LGS	Load far pointer using GS.
LSS	Load far pointer using SS.

5.1.13 Miscellaneous Instructions

The miscellaneous instructions provide such functions as loading an effective address, executing a "no-operation," and retrieving processor identification information.

LEA	Load effective address.
NOP	No operation.
UD	Undefined instruction.
XLAT/XLATB	Table lookup translation.
CPUID	Processor identification.
MOVBE[1]	Move data after swapping data bytes.
PREFETCHW	Prefetch data into cache in anticipation of write.
PREFETCHWT1	Prefetch hint T1 with intent to write.
CLFLUSH	Flushes and invalidates a memory operand and its associated cache line from all levels of the processor's cache hierarchy.
CLFLUSHOPT	Flushes and invalidates a memory operand and its associated cache line from all levels of the processor's cache hierarchy with optimized memory system throughput.

5.1.14 User Mode Extended Sate Save/Restore Instructions

XSAVE	Save processor extended states to memory.
XSAVEC	Save processor extended states with compaction to memory.
XSAVEOPT	Save processor extended states to memory, optimized.
XRSTOR	Restore processor extended states from memory.
XGETBV	Reads the state of an extended control register.

5.1.15 Random Number Generator Instructions

RDRAND	Retrieves a random number generated from hardware.
RDSEED	Retrieves a random number generated from hardware.

5.1.16 BMI1, BMI2

ANDN	Bitwise AND of first source with inverted 2nd source operands.
BEXTR	Contiguous bitwise extract.
BLSI	Extract lowest set bit.
BLSMSK	Set all lower bits below first set bit to 1.
BLSR	Reset lowest set bit.
BZHI	Zero high bits starting from specified bit position.
LZCNT	Count the number leading zero bits.
MULX	Unsigned multiply without affecting arithmetic flags.
PDEP	Parallel deposit of bits using a mask.
PEXT	Parallel extraction of bits using a mask.
RORX	Rotate right without affecting arithmetic flags.
SARX	Shift arithmetic right.
SHLX	Shift logic left.
SHRX	Shift logic right.
TZCNT	Count the number trailing zero bits.

1. Processor support of MOVBE is enumerated by CPUID.01:ECX.MOVBE[bit 22] = 1.

5.1.16.1 Detection of VEX-encoded GPR Instructions, LZCNT and TZCNT, PREFETCHW

VEX-encoded general-purpose instructions do not operate on any vector registers.

There are separate feature flags for the following subsets of instructions that operate on general purpose registers, and the detection requirements for hardware support are:

CPUID.(EAX=07H, ECX=0H):EBX.BMI1[bit 3]: if 1 indicates the processor supports the first group of advanced bit manipulation extensions (ANDN, BEXTR, BLSI, BLSMSK, BLSR, TZCNT);

CPUID.(EAX=07H, ECX=0H):EBX.BMI2[bit 8]: if 1 indicates the processor supports the second group of advanced bit manipulation extensions (BZHI, MULX, PDEP, PEXT, RORX, SARX, SHLX, SHRX);

CPUID.EAX=80000001H:ECX.LZCNT[bit 5]: if 1 indicates the processor supports the LZCNT instruction.

CPUID.EAX=80000001H:ECX.PREFTEHCHW[bit 8]: if 1 indicates the processor supports the PREFTEHCHW instruction. CPUID.(EAX=07H, ECX=0H):ECX.PREFTEHCHWT1[bit 0]: if 1 indicates the processor supports the PREFTEHCHWT1 instruction.

5.2 X87 FPU INSTRUCTIONS

The x87 FPU instructions are executed by the processor's x87 FPU. These instructions operate on floating-point, integer, and binary-coded decimal (BCD) operands. For more detail on x87 FPU instructions, see Chapter 8, "Programming with the x87 FPU."

These instructions are divided into the following subgroups: data transfer, load constants, and FPU control instructions. The sections that follow introduce each subgroup.

5.2.1 x87 FPU Data Transfer Instructions

The data transfer instructions move floating-point, integer, and BCD values between memory and the x87 FPU registers. They also perform conditional move operations on floating-point operands.

FLD	Load floating-point value.
FST	Store floating-point value.
FSTP	Store floating-point value and pop.
FILD	Load integer.
FIST	Store integer.
FISTP[1]	Store integer and pop.
FBLD	Load BCD.
FBSTP	Store BCD and pop.
FXCH	Exchange registers.
FCMOVE	Floating-point conditional move if equal.
FCMOVNE	Floating-point conditional move if not equal.
FCMOVB	Floating-point conditional move if below.
FCMOVBE	Floating-point conditional move if below or equal.
FCMOVNB	Floating-point conditional move if not below.
FCMOVNBE	Floating-point conditional move if not below or equal.
FCMOVU	Floating-point conditional move if unordered.
FCMOVNU	Floating-point conditional move if not unordered.

5.2.2 x87 FPU Basic Arithmetic Instructions

The basic arithmetic instructions perform basic arithmetic operations on floating-point and integer operands.

1. SSE3 provides an instruction FISTTP for integer conversion.

FADD	Add floating-point
FADDP	Add floating-point and pop
FIADD	Add integer
FSUB	Subtract floating-point
FSUBP	Subtract floating-point and pop
FISUB	Subtract integer
FSUBR	Subtract floating-point reverse
FSUBRP	Subtract floating-point reverse and pop
FISUBR	Subtract integer reverse
FMUL	Multiply floating-point
FMULP	Multiply floating-point and pop
FIMUL	Multiply integer
FDIV	Divide floating-point
FDIVP	Divide floating-point and pop
FIDIV	Divide integer
FDIVR	Divide floating-point reverse
FDIVRP	Divide floating-point reverse and pop
FIDIVR	Divide integer reverse
FPREM	Partial remainder
FPREM1	IEEE Partial remainder
FABS	Absolute value
FCHS	Change sign
FRNDINT	Round to integer
FSCALE	Scale by power of two
FSQRT	Square root
FXTRACT	Extract exponent and significand

5.2.3 x87 FPU Comparison Instructions

The compare instructions examine or compare floating-point or integer operands.

FCOM	Compare floating-point.
FCOMP	Compare floating-point and pop.
FCOMPP	Compare floating-point and pop twice.
FUCOM	Unordered compare floating-point.
FUCOMP	Unordered compare floating-point and pop.
FUCOMPP	Unordered compare floating-point and pop twice.
FICOM	Compare integer.
FICOMP	Compare integer and pop.
FCOMI	Compare floating-point and set EFLAGS.
FUCOMI	Unordered compare floating-point and set EFLAGS.
FCOMIP	Compare floating-point, set EFLAGS, and pop.
FUCOMIP	Unordered compare floating-point, set EFLAGS, and pop.
FTST	Test floating-point (compare with 0.0).
FXAM	Examine floating-point.

5.2.4 x87 FPU Transcendental Instructions

The transcendental instructions perform basic trigonometric and logarithmic operations on floating-point operands.

FSIN	Sine
FCOS	Cosine
FSINCOS	Sine and cosine
FPTAN	Partial tangent
FPATAN	Partial arctangent
F2XM1	$2^x - 1$
FYL2X	$y*\log_2 x$
FYL2XP1	$y*\log_2(x+1)$

5.2.5 x87 FPU Load Constants Instructions

The load constants instructions load common constants, such as π, into the x87 floating-point registers.

FLD1	Load +1.0
FLDZ	Load +0.0
FLDPI	Load π
FLDL2E	Load $\log_2 e$
FLDLN2	Load $\log_e 2$
FLDL2T	Load $\log_2 10$
FLDLG2	Load $\log_{10} 2$

5.2.6 x87 FPU Control Instructions

The x87 FPU control instructions operate on the x87 FPU register stack and save and restore the x87 FPU state.

FINCSTP	Increment FPU register stack pointer.
FDECSTP	Decrement FPU register stack pointer.
FFREE	Free floating-point register.
FINIT	Initialize FPU after checking error conditions.
FNINIT	Initialize FPU without checking error conditions.
FCLEX	Clear floating-point exception flags after checking for error conditions.
FNCLEX	Clear floating-point exception flags without checking for error conditions.
FSTCW	Store FPU control word after checking error conditions.
FNSTCW	Store FPU control word without checking error conditions.
FLDCW	Load FPU control word.
FSTENV	Store FPU environment after checking error conditions.
FNSTENV	Store FPU environment without checking error conditions.
FLDENV	Load FPU environment.
FSAVE	Save FPU state after checking error conditions.
FNSAVE	Save FPU state without checking error conditions.
FRSTOR	Restore FPU state.
FSTSW	Store FPU status word after checking error conditions.
FNSTSW	Store FPU status word without checking error conditions.
WAIT/FWAIT	Wait for FPU.
FNOP	FPU no operation.

5.3 X87 FPU AND SIMD STATE MANAGEMENT INSTRUCTIONS

Two state management instructions were introduced into the IA-32 architecture with the Pentium II processor family:

FXSAVE	Save x87 FPU and SIMD state.
FXRSTOR	Restore x87 FPU and SIMD state.

Initially, these instructions operated only on the x87 FPU (and MMX) registers to perform a fast save and restore, respectively, of the x87 FPU and MMX state. With the introduction of SSE extensions in the Pentium III processor family, these instructions were expanded to also save and restore the state of the XMM and MXCSR registers. Intel 64 architecture also supports these instructions.

See Section 10.5, "FXSAVE and FXRSTOR Instructions," for more detail.

5.4 MMX™ INSTRUCTIONS

Four extensions have been introduced into the IA-32 architecture to permit IA-32 processors to perform single-instruction multiple-data (SIMD) operations. These extensions include the MMX technology, SSE extensions, SSE2 extensions, and SSE3 extensions. For a discussion that puts SIMD instructions in their historical context, see Section 2.2.7, "SIMD Instructions."

MMX instructions operate on packed byte, word, doubleword, or quadword integer operands contained in memory, in MMX registers, and/or in general-purpose registers. For more detail on these instructions, see Chapter 9, "Programming with Intel® MMX™ Technology."

MMX instructions can only be executed on Intel 64 and IA-32 processors that support the MMX technology. Support for these instructions can be detected with the CPUID instruction. See the description of the CPUID instruction in Chapter 3, "Instruction Set Reference, A-L," of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A.*

MMX instructions are divided into the following subgroups: data transfer, conversion, packed arithmetic, comparison, logical, shift and rotate, and state management instructions. The sections that follow introduce each subgroup.

5.4.1 MMX Data Transfer Instructions

The data transfer instructions move doubleword and quadword operands between MMX registers and between MMX registers and memory.

MOVD	Move doubleword.
MOVQ	Move quadword.

5.4.2 MMX Conversion Instructions

The conversion instructions pack and unpack bytes, words, and doublewords

PACKSSWB	Pack words into bytes with signed saturation.
PACKSSDW	Pack doublewords into words with signed saturation.
PACKUSWB	Pack words into bytes with unsigned saturation.
PUNPCKHBW	Unpack high-order bytes.
PUNPCKHWD	Unpack high-order words.
PUNPCKHDQ	Unpack high-order doublewords.
PUNPCKLBW	Unpack low-order bytes.
PUNPCKLWD	Unpack low-order words.
PUNPCKLDQ	Unpack low-order doublewords.

5.4.3 MMX Packed Arithmetic Instructions

The packed arithmetic instructions perform packed integer arithmetic on packed byte, word, and doubleword integers.

PADDB	Add packed byte integers.
PADDW	Add packed word integers.
PADDD	Add packed doubleword integers.
PADDSB	Add packed signed byte integers with signed saturation.
PADDSW	Add packed signed word integers with signed saturation.
PADDUSB	Add packed unsigned byte integers with unsigned saturation.
PADDUSW	Add packed unsigned word integers with unsigned saturation.
PSUBB	Subtract packed byte integers.
PSUBW	Subtract packed word integers.
PSUBD	Subtract packed doubleword integers.
PSUBSB	Subtract packed signed byte integers with signed saturation.
PSUBSW	Subtract packed signed word integers with signed saturation.
PSUBUSB	Subtract packed unsigned byte integers with unsigned saturation.
PSUBUSW	Subtract packed unsigned word integers with unsigned saturation.
PMULHW	Multiply packed signed word integers and store high result.
PMULLW	Multiply packed signed word integers and store low result.
PMADDWD	Multiply and add packed word integers.

5.4.4 MMX Comparison Instructions

The compare instructions compare packed bytes, words, or doublewords.

PCMPEQB	Compare packed bytes for equal.
PCMPEQW	Compare packed words for equal.
PCMPEQD	Compare packed doublewords for equal.
PCMPGTB	Compare packed signed byte integers for greater than.
PCMPGTW	Compare packed signed word integers for greater than.
PCMPGTD	Compare packed signed doubleword integers for greater than.

5.4.5 MMX Logical Instructions

The logical instructions perform AND, AND NOT, OR, and XOR operations on quadword operands.

PAND	Bitwise logical AND.
PANDN	Bitwise logical AND NOT.
POR	Bitwise logical OR.
PXOR	Bitwise logical exclusive OR.

5.4.6 MMX Shift and Rotate Instructions

The shift and rotate instructions shift and rotate packed bytes, words, or doublewords, or quadwords in 64-bit operands.

PSLLW	Shift packed words left logical.
PSLLD	Shift packed doublewords left logical.
PSLLQ	Shift packed quadword left logical.
PSRLW	Shift packed words right logical.
PSRLD	Shift packed doublewords right logical.
PSRLQ	Shift packed quadword right logical.

PSRAW	Shift packed words right arithmetic.
PSRAD	Shift packed doublewords right arithmetic.

5.4.7 MMX State Management Instructions

The EMMS instruction clears the MMX state from the MMX registers.

EMMS	Empty MMX state.

5.5 SSE INSTRUCTIONS

SSE instructions represent an extension of the SIMD execution model introduced with the MMX technology. For more detail on these instructions, see Chapter 10, "Programming with Intel® Streaming SIMD Extensions (Intel® SSE)."

SSE instructions can only be executed on Intel 64 and IA-32 processors that support SSE extensions. Support for these instructions can be detected with the CPUID instruction. See the description of the CPUID instruction in Chapter 3, "Instruction Set Reference, A-L," of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A.*

SSE instructions are divided into four subgroups (note that the first subgroup has subordinate subgroups of its own):

- SIMD single-precision floating-point instructions that operate on the XMM registers.
- MXCSR state management instructions.
- 64-bit SIMD integer instructions that operate on the MMX registers.
- Cacheability control, prefetch, and instruction ordering instructions.

The following sections provide an overview of these groups.

5.5.1 SSE SIMD Single-Precision Floating-Point Instructions

These instructions operate on packed and scalar single-precision floating-point values located in XMM registers and/or memory. This subgroup is further divided into the following subordinate subgroups: data transfer, packed arithmetic, comparison, logical, shuffle and unpack, and conversion instructions.

5.5.1.1 SSE Data Transfer Instructions

SSE data transfer instructions move packed and scalar single-precision floating-point operands between XMM registers and between XMM registers and memory.

MOVAPS	Move four aligned packed single-precision floating-point values between XMM registers or between and XMM register and memory.
MOVUPS	Move four unaligned packed single-precision floating-point values between XMM registers or between and XMM register and memory.
MOVHPS	Move two packed single-precision floating-point values to an from the high quadword of an XMM register and memory.
MOVHLPS	Move two packed single-precision floating-point values from the high quadword of an XMM register to the low quadword of another XMM register.
MOVLPS	Move two packed single-precision floating-point values to an from the low quadword of an XMM register and memory.
MOVLHPS	Move two packed single-precision floating-point values from the low quadword of an XMM register to the high quadword of another XMM register.
MOVMSKPS	Extract sign mask from four packed single-precision floating-point values.

MOVSS	Move scalar single-precision floating-point value between XMM registers or between an XMM register and memory.

5.5.1.2 SSE Packed Arithmetic Instructions

SSE packed arithmetic instructions perform packed and scalar arithmetic operations on packed and scalar single-precision floating-point operands.

ADDPS	Add packed single-precision floating-point values.
ADDSS	Add scalar single-precision floating-point values.
SUBPS	Subtract packed single-precision floating-point values.
SUBSS	Subtract scalar single-precision floating-point values.
MULPS	Multiply packed single-precision floating-point values.
MULSS	Multiply scalar single-precision floating-point values.
DIVPS	Divide packed single-precision floating-point values.
DIVSS	Divide scalar single-precision floating-point values.
RCPPS	Compute reciprocals of packed single-precision floating-point values.
RCPSS	Compute reciprocal of scalar single-precision floating-point values.
SQRTPS	Compute square roots of packed single-precision floating-point values.
SQRTSS	Compute square root of scalar single-precision floating-point values.
RSQRTPS	Compute reciprocals of square roots of packed single-precision floating-point values.
RSQRTSS	Compute reciprocal of square root of scalar single-precision floating-point values.
MAXPS	Return maximum packed single-precision floating-point values.
MAXSS	Return maximum scalar single-precision floating-point values.
MINPS	Return minimum packed single-precision floating-point values.
MINSS	Return minimum scalar single-precision floating-point values.

5.5.1.3 SSE Comparison Instructions

SSE compare instructions compare packed and scalar single-precision floating-point operands.

CMPPS	Compare packed single-precision floating-point values.
CMPSS	Compare scalar single-precision floating-point values.
COMISS	Perform ordered comparison of scalar single-precision floating-point values and set flags in EFLAGS register.
UCOMISS	Perform unordered comparison of scalar single-precision floating-point values and set flags in EFLAGS register.

5.5.1.4 SSE Logical Instructions

SSE logical instructions perform bitwise AND, AND NOT, OR, and XOR operations on packed single-precision floating-point operands.

ANDPS	Perform bitwise logical AND of packed single-precision floating-point values.
ANDNPS	Perform bitwise logical AND NOT of packed single-precision floating-point values.
ORPS	Perform bitwise logical OR of packed single-precision floating-point values.
XORPS	Perform bitwise logical XOR of packed single-precision floating-point values.

5.5.1.5 SSE Shuffle and Unpack Instructions

SSE shuffle and unpack instructions shuffle or interleave single-precision floating-point values in packed single-precision floating-point operands.

SHUFPS	Shuffles values in packed single-precision floating-point operands.

UNPCKHPS	Unpacks and interleaves the two high-order values from two single-precision floating-point operands.
UNPCKLPS	Unpacks and interleaves the two low-order values from two single-precision floating-point operands.

5.5.1.6 SSE Conversion Instructions

SSE conversion instructions convert packed and individual doubleword integers into packed and scalar single-precision floating-point values and vice versa.

CVTPI2PS	Convert packed doubleword integers to packed single-precision floating-point values.
CVTSI2SS	Convert doubleword integer to scalar single-precision floating-point value.
CVTPS2PI	Convert packed single-precision floating-point values to packed doubleword integers.
CVTTPS2PI	Convert with truncation packed single-precision floating-point values to packed doubleword integers.
CVTSS2SI	Convert a scalar single-precision floating-point value to a doubleword integer.
CVTTSS2SI	Convert with truncation a scalar single-precision floating-point value to a scalar doubleword integer.

5.5.2 SSE MXCSR State Management Instructions

MXCSR state management instructions allow saving and restoring the state of the MXCSR control and status register.

LDMXCSR	Load MXCSR register.
STMXCSR	Save MXCSR register state.

5.5.3 SSE 64-Bit SIMD Integer Instructions

These SSE 64-bit SIMD integer instructions perform additional operations on packed bytes, words, or doublewords contained in MMX registers. They represent enhancements to the MMX instruction set described in Section 5.4, "MMX™ Instructions."

PAVGB	Compute average of packed unsigned byte integers.
PAVGW	Compute average of packed unsigned word integers.
PEXTRW	Extract word.
PINSRW	Insert word.
PMAXUB	Maximum of packed unsigned byte integers.
PMAXSW	Maximum of packed signed word integers.
PMINUB	Minimum of packed unsigned byte integers.
PMINSW	Minimum of packed signed word integers.
PMOVMSKB	Move byte mask.
PMULHUW	Multiply packed unsigned integers and store high result.
PSADBW	Compute sum of absolute differences.
PSHUFW	Shuffle packed integer word in MMX register.

5.5.4 SSE Cacheability Control, Prefetch, and Instruction Ordering Instructions

The cacheability control instructions provide control over the caching of non-temporal data when storing data from the MMX and XMM registers to memory. The PREFETCH*h* allows data to be prefetched to a selected cache level. The SFENCE instruction controls instruction ordering on store operations.

MASKMOVQ	Non-temporal store of selected bytes from an MMX register into memory.

MOVNTQ	Non-temporal store of quadword from an MMX register into memory.
MOVNTPS	Non-temporal store of four packed single-precision floating-point values from an XMM register into memory.
PREFETCH*h*	Load 32 or more of bytes from memory to a selected level of the processor's cache hierarchy
SFENCE	Serializes store operations.

5.6 SSE2 INSTRUCTIONS

SSE2 extensions represent an extension of the SIMD execution model introduced with MMX technology and the SSE extensions. SSE2 instructions operate on packed double-precision floating-point operands and on packed byte, word, doubleword, and quadword operands located in the XMM registers. For more detail on these instructions, see Chapter 11, "Programming with Intel® Streaming SIMD Extensions 2 (Intel® SSE2)."

SSE2 instructions can only be executed on Intel 64 and IA-32 processors that support the SSE2 extensions. Support for these instructions can be detected with the CPUID instruction. See the description of the CPUID instruction in Chapter 3, "Instruction Set Reference, A-L," of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A.*

These instructions are divided into four subgroups (note that the first subgroup is further divided into subordinate subgroups):

- Packed and scalar double-precision floating-point instructions.
- Packed single-precision floating-point conversion instructions.
- 128-bit SIMD integer instructions.
- Cacheability-control and instruction ordering instructions.

The following sections give an overview of each subgroup.

5.6.1 SSE2 Packed and Scalar Double-Precision Floating-Point Instructions

SSE2 packed and scalar double-precision floating-point instructions are divided into the following subordinate subgroups: data movement, arithmetic, comparison, conversion, logical, and shuffle operations on double-precision floating-point operands. These are introduced in the sections that follow.

5.6.1.1 SSE2 Data Movement Instructions

SSE2 data movement instructions move double-precision floating-point data between XMM registers and between XMM registers and memory.

MOVAPD	Move two aligned packed double-precision floating-point values between XMM registers or between and XMM register and memory.
MOVUPD	Move two unaligned packed double-precision floating-point values between XMM registers or between and XMM register and memory.
MOVHPD	Move high packed double-precision floating-point value to an from the high quadword of an XMM register and memory.
MOVLPD	Move low packed single-precision floating-point value to an from the low quadword of an XMM register and memory.
MOVMSKPD	Extract sign mask from two packed double-precision floating-point values.
MOVSD	Move scalar double-precision floating-point value between XMM registers or between an XMM register and memory.

5.6.1.2 SSE2 Packed Arithmetic Instructions

The arithmetic instructions perform addition, subtraction, multiply, divide, square root, and maximum/minimum operations on packed and scalar double-precision floating-point operands.

ADDPD	Add packed double-precision floating-point values.
ADDSD	Add scalar double precision floating-point values.
SUBPD	Subtract packed double-precision floating-point values.
SUBSD	Subtract scalar double-precision floating-point values.
MULPD	Multiply packed double-precision floating-point values.
MULSD	Multiply scalar double-precision floating-point values.
DIVPD	Divide packed double-precision floating-point values.
DIVSD	Divide scalar double-precision floating-point values.
SQRTPD	Compute packed square roots of packed double-precision floating-point values.
SQRTSD	Compute scalar square root of scalar double-precision floating-point values.
MAXPD	Return maximum packed double-precision floating-point values.
MAXSD	Return maximum scalar double-precision floating-point values.
MINPD	Return minimum packed double-precision floating-point values.
MINSD	Return minimum scalar double-precision floating-point values.

5.6.1.3 SSE2 Logical Instructions

SSE2 logical instructions preform AND, AND NOT, OR, and XOR operations on packed double-precision floating-point values.

ANDPD	Perform bitwise logical AND of packed double-precision floating-point values.
ANDNPD	Perform bitwise logical AND NOT of packed double-precision floating-point values.
ORPD	Perform bitwise logical OR of packed double-precision floating-point values.
XORPD	Perform bitwise logical XOR of packed double-precision floating-point values.

5.6.1.4 SSE2 Compare Instructions

SSE2 compare instructions compare packed and scalar double-precision floating-point values and return the results of the comparison either to the destination operand or to the EFLAGS register.

CMPPD	Compare packed double-precision floating-point values.
CMPSD	Compare scalar double-precision floating-point values.
COMISD	Perform ordered comparison of scalar double-precision floating-point values and set flags in EFLAGS register.
UCOMISD	Perform unordered comparison of scalar double-precision floating-point values and set flags in EFLAGS register.

5.6.1.5 SSE2 Shuffle and Unpack Instructions

SSE2 shuffle and unpack instructions shuffle or interleave double-precision floating-point values in packed double-precision floating-point operands.

SHUFPD	Shuffles values in packed double-precision floating-point operands.
UNPCKHPD	Unpacks and interleaves the high values from two packed double-precision floating-point operands.
UNPCKLPD	Unpacks and interleaves the low values from two packed double-precision floating-point operands.

5.6.1.6 SSE2 Conversion Instructions

SSE2 conversion instructions convert packed and individual doubleword integers into packed and scalar double-precision floating-point values and vice versa. They also convert between packed and scalar single-precision and double-precision floating-point values.

CVTPD2PI	Convert packed double-precision floating-point values to packed doubleword integers.
CVTTPD2PI	Convert with truncation packed double-precision floating-point values to packed doubleword integers.
CVTPI2PD	Convert packed doubleword integers to packed double-precision floating-point values.
CVTPD2DQ	Convert packed double-precision floating-point values to packed doubleword integers.
CVTTPD2DQ	Convert with truncation packed double-precision floating-point values to packed doubleword integers.
CVTDQ2PD	Convert packed doubleword integers to packed double-precision floating-point values.
CVTPS2PD	Convert packed single-precision floating-point values to packed double-precision floating-point values.
CVTPD2PS	Convert packed double-precision floating-point values to packed single-precision floating-point values.
CVTSS2SD	Convert scalar single-precision floating-point values to scalar double-precision floating-point values.
CVTSD2SS	Convert scalar double-precision floating-point values to scalar single-precision floating-point values.
CVTSD2SI	Convert scalar double-precision floating-point values to a doubleword integer.
CVTTSD2SI	Convert with truncation scalar double-precision floating-point values to scalar doubleword integers.
CVTSI2SD	Convert doubleword integer to scalar double-precision floating-point value.

5.6.2 SSE2 Packed Single-Precision Floating-Point Instructions

SSE2 packed single-precision floating-point instructions perform conversion operations on single-precision floating-point and integer operands. These instructions represent enhancements to the SSE single-precision floating-point instructions.

CVTDQ2PS	Convert packed doubleword integers to packed single-precision floating-point values.
CVTPS2DQ	Convert packed single-precision floating-point values to packed doubleword integers.
CVTTPS2DQ	Convert with truncation packed single-precision floating-point values to packed doubleword integers.

5.6.3 SSE2 128-Bit SIMD Integer Instructions

SSE2 SIMD integer instructions perform additional operations on packed words, doublewords, and quadwords contained in XMM and MMX registers.

MOVDQA	Move aligned double quadword.
MOVDQU	Move unaligned double quadword.
MOVQ2DQ	Move quadword integer from MMX to XMM registers.
MOVDQ2Q	Move quadword integer from XMM to MMX registers.
PMULUDQ	Multiply packed unsigned doubleword integers.
PADDQ	Add packed quadword integers.
PSUBQ	Subtract packed quadword integers.
PSHUFLW	Shuffle packed low words.
PSHUFHW	Shuffle packed high words.
PSHUFD	Shuffle packed doublewords.

PSLLDQ	Shift double quadword left logical.
PSRLDQ	Shift double quadword right logical.
PUNPCKHQDQ	Unpack high quadwords.
PUNPCKLQDQ	Unpack low quadwords.

5.6.4 SSE2 Cacheability Control and Ordering Instructions

SSE2 cacheability control instructions provide additional operations for caching of non-temporal data when storing data from XMM registers to memory. LFENCE and MFENCE provide additional control of instruction ordering on store operations.

CLFLUSH	See Section 5.1.13.
LFENCE	Serializes load operations.
MFENCE	Serializes load and store operations.
PAUSE	Improves the performance of "spin-wait loops".
MASKMOVDQU	Non-temporal store of selected bytes from an XMM register into memory.
MOVNTPD	Non-temporal store of two packed double-precision floating-point values from an XMM register into memory.
MOVNTDQ	Non-temporal store of double quadword from an XMM register into memory.
MOVNTI	Non-temporal store of a doubleword from a general-purpose register into memory.

5.7 SSE3 INSTRUCTIONS

The SSE3 extensions offers 13 instructions that accelerate performance of Streaming SIMD Extensions technology, Streaming SIMD Extensions 2 technology, and x87-FP math capabilities. These instructions can be grouped into the following categories:

- One x87FPU instruction used in integer conversion.
- One SIMD integer instruction that addresses unaligned data loads.
- Two SIMD floating-point packed ADD/SUB instructions.
- Four SIMD floating-point horizontal ADD/SUB instructions.
- Three SIMD floating-point LOAD/MOVE/DUPLICATE instructions.
- Two thread synchronization instructions.

SSE3 instructions can only be executed on Intel 64 and IA-32 processors that support SSE3 extensions. Support for these instructions can be detected with the CPUID instruction. See the description of the CPUID instruction in Chapter 3, "Instruction Set Reference, A-L," of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A.*

The sections that follow describe each subgroup.

5.7.1 SSE3 x87-FP Integer Conversion Instruction

FISTTP	Behaves like the FISTP instruction but uses truncation, irrespective of the rounding mode specified in the floating-point control word (FCW).

5.7.2 SSE3 Specialized 128-bit Unaligned Data Load Instruction

LDDQU	Special 128-bit unaligned load designed to avoid cache line splits.

5.7.3 SSE3 SIMD Floating-Point Packed ADD/SUB Instructions

ADDSUBPS	Performs single-precision addition on the second and fourth pairs of 32-bit data elements within the operands; single-precision subtraction on the first and third pairs.
ADDSUBPD	Performs double-precision addition on the second pair of quadwords, and double-precision subtraction on the first pair.

5.7.4 SSE3 SIMD Floating-Point Horizontal ADD/SUB Instructions

HADDPS	Performs a single-precision addition on contiguous data elements. The first data element of the result is obtained by adding the first and second elements of the first operand; the second element by adding the third and fourth elements of the first operand; the third by adding the first and second elements of the second operand; and the fourth by adding the third and fourth elements of the second operand.
HSUBPS	Performs a single-precision subtraction on contiguous data elements. The first data element of the result is obtained by subtracting the second element of the first operand from the first element of the first operand; the second element by subtracting the fourth element of the first operand from the third element of the first operand; the third by subtracting the second element of the second operand from the first element of the second operand; and the fourth by subtracting the fourth element of the second operand from the third element of the second operand.
HADDPD	Performs a double-precision addition on contiguous data elements. The first data element of the result is obtained by adding the first and second elements of the first operand; the second element by adding the first and second elements of the second operand.
HSUBPD	Performs a double-precision subtraction on contiguous data elements. The first data element of the result is obtained by subtracting the second element of the first operand from the first element of the first operand; the second element by subtracting the second element of the second operand from the first element of the second operand.

5.7.5 SSE3 SIMD Floating-Point LOAD/MOVE/DUPLICATE Instructions

MOVSHDUP	Loads/moves 128 bits; duplicating the second and fourth 32-bit data elements.
MOVSLDUP	Loads/moves 128 bits; duplicating the first and third 32-bit data elements.
MOVDDUP	Loads/moves 64 bits (bits[63:0] if the source is a register) and returns the same 64 bits in both the lower and upper halves of the 128-bit result register; duplicates the 64 bits from the source.

5.7.6 SSE3 Agent Synchronization Instructions

MONITOR	Sets up an address range used to monitor write-back stores.
MWAIT	Enables a logical processor to enter into an optimized state while waiting for a write-back store to the address range set up by the MONITOR instruction.

5.8 SUPPLEMENTAL STREAMING SIMD EXTENSIONS 3 (SSSE3) INSTRUCTIONS

SSSE3 provide 32 instructions (represented by 14 mnemonics) to accelerate computations on packed integers. These include:

- Twelve instructions that perform horizontal addition or subtraction operations.
- Six instructions that evaluate absolute values.
- Two instructions that perform multiply and add operations and speed up the evaluation of dot products.
- Two instructions that accelerate packed-integer multiply operations and produce integer values with scaling.
- Two instructions that perform a byte-wise, in-place shuffle according to the second shuffle control operand.

- Six instructions that negate packed integers in the destination operand if the signs of the corresponding element in the source operand is less than zero.
- Two instructions that align data from the composite of two operands.

SSSE3 instructions can only be executed on Intel 64 and IA-32 processors that support SSSE3 extensions. Support for these instructions can be detected with the CPUID instruction. See the description of the CPUID instruction in Chapter 3, "Instruction Set Reference, A-L," of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A.*

The sections that follow describe each subgroup.

5.8.1 Horizontal Addition/Subtraction

PHADDW	Adds two adjacent, signed 16-bit integers horizontally from the source and destination operands and packs the signed 16-bit results to the destination operand.
PHADDSW	Adds two adjacent, signed 16-bit integers horizontally from the source and destination operands and packs the signed, saturated 16-bit results to the destination operand.
PHADDD	Adds two adjacent, signed 32-bit integers horizontally from the source and destination operands and packs the signed 32-bit results to the destination operand.
PHSUBW	Performs horizontal subtraction on each adjacent pair of 16-bit signed integers by subtracting the most significant word from the least significant word of each pair in the source and destination operands. The signed 16-bit results are packed and written to the destination operand.
PHSUBSW	Performs horizontal subtraction on each adjacent pair of 16-bit signed integers by subtracting the most significant word from the least significant word of each pair in the source and destination operands. The signed, saturated 16-bit results are packed and written to the destination operand.
PHSUBD	Performs horizontal subtraction on each adjacent pair of 32-bit signed integers by subtracting the most significant doubleword from the least significant double word of each pair in the source and destination operands. The signed 32-bit results are packed and written to the destination operand.

5.8.2 Packed Absolute Values

PABSB	Computes the absolute value of each signed byte data element.
PABSW	Computes the absolute value of each signed 16-bit data element.
PABSD	Computes the absolute value of each signed 32-bit data element.

5.8.3 Multiply and Add Packed Signed and Unsigned Bytes

PMADDUBSW	Multiplies each unsigned byte value with the corresponding signed byte value to produce an intermediate, 16-bit signed integer. Each adjacent pair of 16-bit signed values are added horizontally. The signed, saturated 16-bit results are packed to the destination operand.

5.8.4 Packed Multiply High with Round and Scale

PMULHRSW	Multiplies vertically each signed 16-bit integer from the destination operand with the corresponding signed 16-bit integer of the source operand, producing intermediate, signed 32-bit integers. Each intermediate 32-bit integer is truncated to the 18 most significant bits. Rounding is always performed by adding 1 to the least significant bit of the 18-bit intermediate result. The final result is obtained by selecting the 16 bits immediately to the right of the most significant bit of each 18-bit intermediate result and packed to the destination operand.

5.8.5 Packed Shuffle Bytes

PSHUFB — Permutes each byte in place, according to a shuffle control mask. The least significant three or four bits of each shuffle control byte of the control mask form the shuffle index. The shuffle mask is unaffected. If the most significant bit (bit 7) of a shuffle control byte is set, the constant zero is written in the result byte.

5.8.6 Packed Sign

PSIGNB/W/D — Negates each signed integer element of the destination operand if the sign of the corresponding data element in the source operand is less than zero.

5.8.7 Packed Align Right

PALIGNR — Source operand is appended after the destination operand forming an intermediate value of twice the width of an operand. The result is extracted from the intermediate value into the destination operand by selecting the 128 bit or 64 bit value that are right-aligned to the byte offset specified by the immediate value.

5.9 SSE4 INSTRUCTIONS

Intel® Streaming SIMD Extensions 4 (SSE4) introduces 54 new instructions. 47 of the SSE4 instructions are referred to as SSE4.1 in this document, 7 new SSE4 instructions are referred to as SSE4.2.

SSE4.1 is targeted to improve the performance of media, imaging, and 3D workloads. SSE4.1 adds instructions that improve compiler vectorization and significantly increase support for packed dword computation. The technology also provides a hint that can improve memory throughput when reading from uncacheable WC memory type.

The 47 SSE4.1 instructions include:

- Two instructions perform packed dword multiplies.
- Two instructions perform floating-point dot products with input/output selects.
- One instruction performs a load with a streaming hint.
- Six instructions simplify packed blending.
- Eight instructions expand support for packed integer MIN/MAX.
- Four instructions support floating-point round with selectable rounding mode and precision exception override.
- Seven instructions improve data insertion and extractions from XMM registers
- Twelve instructions improve packed integer format conversions (sign and zero extensions).
- One instruction improves SAD (sum absolute difference) generation for small block sizes.
- One instruction aids horizontal searching operations.
- One instruction improves masked comparisons.
- One instruction adds qword packed equality comparisons.
- One instruction adds dword packing with unsigned saturation.

The SSE4.2 instructions operating on XMM registers include:

- String and text processing that can take advantage of single-instruction multiple-data programming techniques.
- A SIMD integer instruction that enhances the capability of the 128-bit integer SIMD capability in SSE4.1.

5.10 SSE4.1 INSTRUCTIONS

SSE4.1 instructions can use an XMM register as a source or destination. Programming SSE4.1 is similar to programming 128-bit Integer SIMD and floating-point SIMD instructions in SSE/SSE2/SSE3/SSSE3. SSE4.1 does not provide any 64-bit integer SIMD instructions operating on MMX registers. The sections that follow describe each subgroup.

5.10.1 Dword Multiply Instructions

PMULLD	Returns four lower 32-bits of the 64-bit results of signed 32-bit integer multiplies.
PMULDQ	Returns two 64-bit signed result of signed 32-bit integer multiplies.

5.10.2 Floating-Point Dot Product Instructions

DPPD	Perform double-precision dot product for up to 2 elements and broadcast.
DPPS	Perform single-precision dot products for up to 4 elements and broadcast.

5.10.3 Streaming Load Hint Instruction

MOVNTDQA	Provides a non-temporal hint that can cause adjacent 16-byte items within an aligned 64-byte region (a streaming line) to be fetched and held in a small set of temporary buffers ("streaming load buffers"). Subsequent streaming loads to other aligned 16-byte items in the same streaming line may be supplied from the streaming load buffer and can improve throughput.

5.10.4 Packed Blending Instructions

BLENDPD	Conditionally copies specified double-precision floating-point data elements in the source operand to the corresponding data elements in the destination, using an immediate byte control.
BLENDPS	Conditionally copies specified single-precision floating-point data elements in the source operand to the corresponding data elements in the destination, using an immediate byte control.
BLENDVPD	Conditionally copies specified double-precision floating-point data elements in the source operand to the corresponding data elements in the destination, using an implied mask.
BLENDVPS	Conditionally copies specified single-precision floating-point data elements in the source operand to the corresponding data elements in the destination, using an implied mask.
PBLENDVB	Conditionally copies specified byte elements in the source operand to the corresponding elements in the destination, using an implied mask.
PBLENDW	Conditionally copies specified word elements in the source operand to the corresponding elements in the destination, using an immediate byte control.

5.10.5 Packed Integer MIN/MAX Instructions

PMINUW	Compare packed unsigned word integers.
PMINUD	Compare packed unsigned dword integers.
PMINSB	Compare packed signed byte integers.
PMINSD	Compare packed signed dword integers.
PMAXUW	Compare packed unsigned word integers.
PMAXUD	Compare packed unsigned dword integers.
PMAXSB	Compare packed signed byte integers.

PMAXSD	Compare packed signed dword integers.

5.10.6 Floating-Point Round Instructions with Selectable Rounding Mode

ROUNDPS	Round packed single precision floating-point values into integer values and return rounded floating-point values.
ROUNDPD	Round packed double precision floating-point values into integer values and return rounded floating-point values.
ROUNDSS	Round the low packed single precision floating-point value into an integer value and return a rounded floating-point value.
ROUNDSD	Round the low packed double precision floating-point value into an integer value and return a rounded floating-point value.

5.10.7 Insertion and Extractions from XMM Registers

EXTRACTPS	Extracts a single-precision floating-point value from a specified offset in an XMM register and stores the result to memory or a general-purpose register.
INSERTPS	Inserts a single-precision floating-point value from either a 32-bit memory location or selected from a specified offset in an XMM register to a specified offset in the destination XMM register. In addition, INSERTPS allows zeroing out selected data elements in the destination, using a mask.
PINSRB	Insert a byte value from a register or memory into an XMM register.
PINSRD	Insert a dword value from 32-bit register or memory into an XMM register.
PINSRQ	Insert a qword value from 64-bit register or memory into an XMM register.
PEXTRB	Extract a byte from an XMM register and insert the value into a general-purpose register or memory.
PEXTRW	Extract a word from an XMM register and insert the value into a general-purpose register or memory.
PEXTRD	Extract a dword from an XMM register and insert the value into a general-purpose register or memory.
PEXTRQ	Extract a qword from an XMM register and insert the value into a general-purpose register or memory.

5.10.8 Packed Integer Format Conversions

PMOVSXBW	Sign extend the lower 8-bit integer of each packed word element into packed signed word integers.
PMOVZXBW	Zero extend the lower 8-bit integer of each packed word element into packed signed word integers.
PMOVSXBD	Sign extend the lower 8-bit integer of each packed dword element into packed signed dword integers.
PMOVZXBD	Zero extend the lower 8-bit integer of each packed dword element into packed signed dword integers.
PMOVSXWD	Sign extend the lower 16-bit integer of each packed dword element into packed signed dword integers.
PMOVZXWD	Zero extend the lower 16-bit integer of each packed dword element into packed signed dword integers..
PMOVSXBQ	Sign extend the lower 8-bit integer of each packed qword element into packed signed qword integers.
PMOVZXBQ	Zero extend the lower 8-bit integer of each packed qword element into packed signed qword integers.

PMOVSXWQ	Sign extend the lower 16-bit integer of each packed qword element into packed signed qword integers.
PMOVZXWQ	Zero extend the lower 16-bit integer of each packed qword element into packed signed qword integers.
PMOVSXDQ	Sign extend the lower 32-bit integer of each packed qword element into packed signed qword integers.
PMOVZXDQ	Zero extend the lower 32-bit integer of each packed qword element into packed signed qword integers.

5.10.9 Improved Sums of Absolute Differences (SAD) for 4-Byte Blocks

MPSADBW	Performs eight 4-byte wide Sum of Absolute Differences operations to produce eight word integers.

5.10.10 Horizontal Search

PHMINPOSUW	Finds the value and location of the minimum unsigned word from one of 8 horizontally packed unsigned words. The resulting value and location (offset within the source) are packed into the low dword of the destination XMM register.

5.10.11 Packed Test

PTEST	Performs a logical AND between the destination with this mask and sets the ZF flag if the result is zero. The CF flag (zero for TEST) is set if the inverted mask AND'd with the destination is all zeroes.

5.10.12 Packed Qword Equality Comparisons

PCMPEQQ	128-bit packed qword equality test.

5.10.13 Dword Packing With Unsigned Saturation

PACKUSDW	PACKUSDW packs dword to word with unsigned saturation.

5.11 SSE4.2 INSTRUCTION SET

Five of the SSE4.2 instructions operate on XMM register as a source or destination. These include four text/string processing instructions and one packed quadword compare SIMD instruction. Programming these five SSE4.2 instructions is similar to programming 128-bit Integer SIMD in SSE2/SSSE3. SSE4.2 does not provide any 64-bit integer SIMD instructions.

CRC32 operates on general-purpose registers and is summarized in Section 5.1.6. The sections that follow summarize each subgroup.

5.11.1 String and Text Processing Instructions

PCMPESTRI	Packed compare explicit-length strings, return index in ECX/RCX.
PCMPESTRM	Packed compare explicit-length strings, return mask in XMM0.
PCMPISTRI	Packed compare implicit-length strings, return index in ECX/RCX.
PCMPISTRM	Packed compare implicit-length strings, return mask in XMM0.

5.11.2 Packed Comparison SIMD integer Instruction

PCMPGTQ	Performs logical compare of greater-than on packed integer quadwords.

5.12 AESNI AND PCLMULQDQ

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

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

5.13 INTEL® ADVANCED VECTOR EXTENSIONS (INTEL® AVX)

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

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

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

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

- Table 14-2 lists 256-bit and 128-bit floating-point arithmetic instructions promoted from legacy 128-bit SIMD instruction sets.
- Table 14-3 lists 256-bit and 128-bit data movement and processing instructions promoted from legacy 128-bit SIMD instruction sets.
- Table 14-4 lists functional enhancements of 256-bit AVX instructions not available from legacy 128-bit SIMD instruction sets.
- Table 14-5 lists 128-bit integer and floating-point instructions promoted from legacy 128-bit SIMD instruction sets.
- Table 14-6 lists functional enhancements of 128-bit AVX instructions not available from legacy 128-bit SIMD instruction sets.
- Table 14-7 lists 128-bit data movement and processing instructions promoted from legacy instruction sets.

5.14 16-BIT FLOATING-POINT CONVERSION

Conversion between single-precision floating-point (32-bit) and half-precision FP (16-bit) data are provided by VCVTPS2PH, VCVTPH2PS:

VCVTPH2PS	Convert eight/four data element containing 16-bit floating-point data into eight/four single-precision floating-point data.
VCVTPS2PH	Convert eight/four data element containing single-precision floating-point data into eight/four 16-bit floating-point data.

5.15 FUSED-MULTIPLY-ADD (FMA)

FMA extensions enhances Intel AVX with high-throughput, arithmetic capabilities covering fused multiply-add, fused multiply-subtract, fused multiply add/subtract interleave, signed-reversed multiply on fused multiply-add and multiply-subtract. FMA extensions provide 36 256-bit floating-point instructions to perform computation on 256-bit vectors and additional 128-bit and scalar FMA instructions.

- Table 14-15 lists FMA instruction sets.

5.16 INTEL® ADVANCED VECTOR EXTENSIONS 2 (INTEL® AVX2)

Intel® AVX2 extends Intel AVX by promoting most of the 128-bit SIMD integer instructions with 256-bit numeric processing capabilities. Intel AVX2 instructions follow the same programming model as AVX instructions.

In addition, AVX2 provide enhanced functionalities for broadcast/permute operations on data elements, vector shift instructions with variable-shift count per data element, and instructions to fetch non-contiguous data elements from memory.

- Table 14-18 lists promoted vector integer instructions in AVX2.
- Table 14-19 lists new instructions in AVX2 that complements AVX.

5.17 INTEL® TRANSACTIONAL SYNCHRONIZATION EXTENSIONS (INTEL® TSX)

XABORT	Abort an RTM transaction execution.
XACQUIRE	Prefix hint to the beginning of an HLE transaction region.
XRELEASE	Prefix hint to the end of an HLE transaction region.
XBEGIN	Transaction begin of an RTM transaction region.
XEND	Transaction end of an RTM transaction region.
XTEST	Test if executing in a transactional region.

5.18 INTEL® SHA EXTENSIONS

Intel® SHA extensions provide a set of instructions that target the acceleration of the Secure Hash Algorithm (SHA), specifically the SHA-1 and SHA-256 variants.

SHA1MSG1	Perform an intermediate calculation for the next four SHA1 message dwords from the previous message dwords.
SHA1MSG2	Perform the final calculation for the next four SHA1 message dwords from the intermediate message dwords.
SHA1NEXTE	Calculate SHA1 state E after four rounds.
SHA1RNDS4	Perform four rounds of SHA1 operations.
SHA256MSG1	Perform an intermediate calculation for the next four SHA256 message dwords.
SHA256MSG2	Perform the final calculation for the next four SHA256 message dwords.
SHA256RNDS2	Perform two rounds of SHA256 operations.

5.19 INTEL® ADVANCED VECTOR EXTENSIONS 512 (INTEL® AVX-512)

The Intel® AVX-512 family comprises a collection of 512-bit SIMD instruction sets to accelerate a diverse range of applications. Intel AVX-512 instructions provide a wide range of functionality that support programming in 512-bit, 256 and 128-bit vector register, plus support for opmask registers and instructions operating on opmask registers.

The collection of 512-bit SIMD instruction sets in Intel AVX-512 include new functionality not available in Intel AVX and Intel AVX2, and promoted instructions similar to equivalent ones in Intel AVX / Intel AVX2 but with enhance-

ment provided by opmask registers not available to VEX-encoded Intel AVX / Intel AVX2. Some instruction mnemonics in AVX / AVX2 that are promoted into AVX-512 can be replaced by new instruction mnemonics that are available only with EVEX encoding, e.g., VBROADCASTF128 into VBROADCASTF32X4. Details of EVEX instruction encoding are discussed in Section 2.6, "Intel® AVX-512 Encoding" of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A*.

512-bit instruction mnemonics in AVX-512F that are not AVX/AVX2 promotions include:

VALIGND/Q	Perform dword/qword alignment of two concatenated source vectors.
VBLENDMPD/PS	Replace the VBLENDVPD/PS instructions (using opmask as select control).
VCOMPRESSPD/PS	Compress packed DP or SP elements of a vector.
VCVT(T)PD2UDQ	Convert packed DP FP elements of a vector to packed unsigned 32-bit integers.
VCVT(T)PS2UDQ	Convert packed SP FP elements of a vector to packed unsigned 32-bit integers.
VCVTQQ2PD/PS	Convert packed signed 64-bit integers to packed DP/SP FP elements.
VCVT(T)SD2USI	Convert the low DP FP element of a vector to an unsigned integer.
VCVT(T)SS2USI	Convert the low SP FP element of a vector to an unsigned integer.
VCVTUDQ2PD/PS	Convert packed unsigned 32-bit integers to packed DP/SP FP elements.
VCVTUSI2USD/S	Convert an unsigned integer to the low DP/SP FP element and merge to a vector.
VEXPANDPD/PS	Expand packed DP or SP elements of a vector.
VEXTRACTF32X4/64X4	Extract a vector from a full-length vector with 32/64-bit granular update.
VEXTRACTI32X4/64X4	Extract a vector from a full-length vector with 32/64-bit granular update.
VFIXUPIMMPD/PS	Perform fix-up to special values in DP/SP FP vectors.
VFIXUPIMMSD/SS	Perform fix-up to special values of the low DP/SP FP element.
VGETEXPPD/PS	Convert the exponent of DP/SP FP elements of a vector into FP values.
VGETEXPSD/SS	Convert the exponent of the low DP/SP FP element in a vector into FP value.
VGETMANTPD/PS	Convert the mantissa of DP/SP FP elements of a vector into FP values.
VGETMANTSD/SS	Convert the mantissa of the low DP/SP FP element of a vector into FP value.
VINSERTF32X4/64X4	Insert a 128/256-bit vector into a full-length vector with 32/64-bit granular update.
VMOVDQA32/64	VMOVDQA with 32/64-bit granular conditional update.
VMOVDQU32/64	VMOVDQU with 32/64-bit granular conditional update.
VPBLENDMD/Q	Blend dword/qword elements using opmask as select control.
VPBROADCASTD/Q	Broadcast from general-purpose register to vector register.
VPCMPD/UD	Compare packed signed/unsigned dwords using specified primitive.
VPCMPQ/UQ	Compare packed signed/unsigned quadwords using specified primitive.
VPCOMPRESSQ/D	Compress packed 64/32-bit elements of a vector.
VPERMI2D/Q	Full permute of two tables of dword/qword elements overwriting the index vector.
VPERMI2PD/PS	Full permute of two tables of DP/SP elements overwriting the index vector.
VPERMT2D/Q	Full permute of two tables of dword/qword elements overwriting one source table.
VPERMT2PD/PS	Full permute of two tables of DP/SP elements overwriting one source table.
VPEXPANDD/Q	Expand packed dword/qword elements of a vector.
VPMAXSQ	Compute maximum of packed signed 64-bit integer elements.
VPMAXUD/UQ	Compute maximum of packed unsigned 32/64-bit integer elements.
VPMINSQ	Compute minimum of packed signed 64-bit integer elements.
VPMINUD/UQ	Compute minimum of packed unsigned 32/64-bit integer elements.
VPMOV(S|US)QB	Down convert qword elements in a vector to byte elements using truncation (saturation | unsigned saturation).
VPMOV(S|US)QW	Down convert qword elements in a vector to word elements using truncation (saturation | unsigned saturation).
VPMOV(S|US)QD	Down convert qword elements in a vector to dword elements using truncation (saturation | unsigned saturation).

VPMOV(S|US)DB	Down convert dword elements in a vector to byte elements using truncation (saturation | unsigned saturation).
VPMOV(S|US)DW	Down convert dword elements in a vector to word elements using truncation (saturation | unsigned saturation).
VPROLD/Q	Rotate dword/qword element left by a constant shift count with conditional update.
VPROLVD/Q	Rotate dword/qword element left by shift counts specified in a vector with conditional update.
VPRORD/Q	Rotate dword/qword element right by a constant shift count with conditional update.
VPRORRD/Q	Rotate dword/qword element right by shift counts specified in a vector with conditional update.
VPSCATTERDD/DQ	Scatter dword/qword elements in a vector to memory using dword indices.
VPSCATTERQD/QQ	Scatter dword/qword elements in a vector to memory using qword indices.
VPSRAQ	Shift qwords right by a constant shift count and shifting in sign bits.
VPSRAVQ	Shift qwords right by shift counts in a vector and shifting in sign bits.
VPTESTNMD/Q	Perform bitwise NAND of dword/qword elements of two vectors and write results to opmask.
VPTERLOGD/Q	Perform bitwise ternary logic operation of three vectors with 32/64 bit granular conditional update.
VPTESTMD/Q	Perform bitwise AND of dword/qword elements of two vectors and write results to opmask.
VRCP14PD/PS	Compute approximate reciprocals of packed DP/SP FP elements of a vector.
VRCP14SD/SS	Compute the approximate reciprocal of the low DP/SP FP element of a vector.
VRNDSCALEPD/PS	Round packed DP/SP FP elements of a vector to specified number of fraction bits.
VRNDSCALESD/SS	Round the low DP/SP FP element of a vector to specified number of fraction bits.
VRSQRT14PD/PS	Compute approximate reciprocals of square roots of packed DP/SP FP elements of a vector.
VRSQRT14SD/SS	Compute the approximate reciprocal of square root of the low DP/SP FP element of a vector.
VSCALEPD/PS	Multiply packed DP/SP FP elements of a vector by powers of two with exponents specified in a second vector.
VSCALESD/SS	Multiply the low DP/SP FP element of a vector by powers of two with exponent specified in the corresponding element of a second vector.
VSCATTERDD/DQ	Scatter SP/DP FP elements in a vector to memory using dword indices.
VSCATTERQD/QQ	Scatter SP/DP FP elements in a vector to memory using qword indices.
VSHUFF32X4/64X2	Shuffle 128-bit lanes of a vector with 32/64 bit granular conditional update.
VSHUFI32X4/64X2	Shuffle 128-bit lanes of a vector with 32/64 bit granular conditional update.

512-bit instruction mnemonics in AVX-512DQ that are not AVX/AVX2 promotions include:

VCVT(T)PD2QQ	Convert packed DP FP elements of a vector to packed signed 64-bit integers.
VCVT(T)PD2UQQ	Convert packed DP FP elements of a vector to packed unsigned 64-bit integers.
VCVT(T)PS2QQ	Convert packed SP FP elements of a vector to packed signed 64-bit integers.
VCVT(T)PS2UQQ	Convert packed SP FP elements of a vector to packed unsigned 64-bit integers.
VCVTUQQ2PD/PS	Convert packed unsigned 64-bit integers to packed DP/SP FP elements.
VEXTRACTF64X2	Extract a vector from a full-length vector with 64-bit granular update.
VEXTRACTI64X2	Extract a vector from a full-length vector with 64-bit granular update.
VFPCLASSPD/PS	Test packed DP/SP FP elements in a vector by numeric/special-value category.
VFPCLASSSD/SS	Test the low DP/SP FP element by numeric/special-value category.
VINSERTF64X2	Insert a 128-bit vector into a full-length vector with 64-bit granular update.
VINSERTI64X2	Insert a 128-bit vector into a full-length vector with 64-bit granular update.
VPMOVM2D/Q	Convert opmask register to vector register in 32/64-bit granularity.

VPMOVB2D/Q2M	Convert a vector register in 32/64-bit granularity to an opmask register.
VPMULLQ	Multiply packed signed 64-bit integer elements of two vectors and store low 64-bit signed result.
VRANGEPD/PS	Perform RANGE operation on each pair of DP/SP FP elements of two vectors using specified range primitive in imm8.
VRANGESD/SS	Perform RANGE operation on the pair of low DP/SP FP element of two vectors using specified range primitive in imm8.
VREDUCEPD/PS	Perform Reduction operation on packed DP/SP FP elements of a vector using specified reduction primitive in imm8.
VREDUCESD/SS	Perform Reduction operation on the low DP/SP FP element of a vector using specified reduction primitive in imm8.

512-bit instruction mnemonics in AVX-512BW that are not AVX/AVX2 promotions include:

VDBPSADBW	Double block packed Sum-Absolute-Differences on unsigned bytes.
VMOVDQU8/16	VMOVDQU with 8/16-bit granular conditional update.
VPBLENDMB	Replaces the VPBLENDVB instruction (using opmask as select control).
VPBLENDMW	Blend word elements using opmask as select control.
VPBROADCASTB/W	Broadcast from general-purpose register to vector register.
VPCMPB/UB	Compare packed signed/unsigned bytes using specified primitive.
VPCMPW/UW	Compare packed signed/unsigned words using specified primitive.
VPERMW	Permute packed word elements.
VPERMI2B/W	Full permute from two tables of byte/word elements overwriting the index vector.
VPMOVM2B/W	Convert opmask register to vector register in 8/16-bit granularity.
VPMOVB2M/W2M	Convert a vector register in 8/16-bit granularity to an opmask register.
VPMOV(S|US)WB	Down convert word elements in a vector to byte elements using truncation (saturation | unsigned saturation).
VPSLLVW	Shift word elements in a vector left by shift counts in a vector.
VPSRAVW	Shift words right by shift counts in a vector and shifting in sign bits.
VPSRLVW	Shift word elements in a vector right by shift counts in a vector.
VPTESTNMB/W	Perform bitwise NAND of byte/word elements of two vectors and write results to opmask.
VPTESTMB/W	Perform bitwise AND of byte/word elements of two vectors and write results to opmask.

512-bit instruction mnemonics in AVX-512CD that are not AVX/AVX2 promotions include:

VPBROADCASTM	Broadcast from opmask register to vector register.
VPCONFLICTD/Q	Detect conflicts within a vector of packed 32/64-bit integers.
VPLZCNTD/Q	Count the number of leading zero bits of packed dword/qword elements.

Opmask instructions include:

KADDB/W/D/Q	Add two 8/16/32/64-bit opmasks.
KANDB/W/D/Q	Logical AND two 8/16/32/64-bit opmasks.
KANDNB/W/D/Q	Logical AND NOT two 8/16/32/64-bit opmasks.
KMOVB/W/D/Q	Move from or move to opmask register of 8/16/32/64-bit data.
KNOTB/W/D/Q	Bitwise NOT of two 8/16/32/64-bit opmasks.
KORB/W/D/Q	Logical OR two 8/16/32/64-bit opmasks.
KORTESTB/W/D/Q	Update EFLAGS according to the result of bitwise OR of two 8/16/32/64-bit opmasks.
KSHIFTLB/W/D/Q	Shift left 8/16/32/64-bit opmask by specified count.
KSHIFTRB/W/D/Q	Shift right 8/16/32/64-bit opmask by specified count.

KTESTB/W/D/Q	Update EFLAGS according to the result of bitwise TEST of two 8/16/32/64-bit opmasks.
KUNPCKBW/WD/DQ	Unpack and interleave two 8/16/32-bit opmasks into 16/32/64-bit mask.
KXNORB/W/D/Q	Bitwise logical XNOR of two 8/16/32/64-bit opmasks.
KXORB/W/D/Q	Logical XOR of two 8/16/32/64-bit opmasks.

512-bit instruction mnemonics in AVX-512ER include:

VEXP2PD/PS	Compute approximate base-2 exponential of packed DP/SP FP elements of a vector.
VEXP2SD/SS	Compute approximate base-2 exponential of the low DP/SP FP element of a vector.
VRCP28PD/PS	Compute approximate reciprocals to 28 bits of packed DP/SP FP elements of a vector.
VRCP28SD/SS	Compute the approximate reciprocal to 28 bits of the low DP/SP FP element of a vector.
VRSQRT28PD/PS	Compute approximate reciprocals of square roots to 28 bits of packed DP/SP FP elements of a vector.
VRSQRT28SD/SS	Compute the approximate reciprocal of square root to 28 bits of the low DP/SP FP element of a vector.

512-bit instruction mnemonics in AVX-512PF include:

VGATHERPF0DPD/PS	Sparse prefetch of packed DP/SP FP vector with T0 hint using dword indices.
VGATHERPF0QPD/PS	Sparse prefetch of packed DP/SP FP vector with T0 hint using qword indices.
VGATHERPF1DPD/PS	Sparse prefetch of packed DP/SP FP vector with T1 hint using dword indices.
VGATHERPF1QPD/PS	Sparse prefetch of packed DP/SP FP vector with T1 hint using qword indices.
VSCATTERPF0DPD/PS	Sparse prefetch of packed DP/SP FP vector with T0 hint to write using dword indices.
VSCATTERPF0QPD/PS	Sparse prefetch of packed DP/SP FP vector with T0 hint to write using qword indices.
VSCATTERPF1DPD/PS	Sparse prefetch of packed DP/SP FP vector with T1 hint to write using dword indices.
VSCATTERPF1QPD/PS	Sparse prefetch of packed DP/SP FP vector with T1 hint to write using qword indices.

5.20 SYSTEM INSTRUCTIONS

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

CLAC	Clear AC Flag in EFLAGS register.
STAC	Set AC Flag in EFLAGS register.
LGDT	Load global descriptor table (GDT) register.
SGDT	Store global descriptor table (GDT) register.
LLDT	Load local descriptor table (LDT) register.
SLDT	Store local descriptor table (LDT) register.
LTR	Load task register.
STR	Store task register.
LIDT	Load interrupt descriptor table (IDT) register.
SIDT	Store interrupt descriptor table (IDT) register.
MOV	Load and store control registers.
LMSW	Load machine status word.
SMSW	Store machine status word.
CLTS	Clear the task-switched flag.
ARPL	Adjust requested privilege level.
LAR	Load access rights.
LSL	Load segment limit.

VERR	Verify segment for reading
VERW	Verify segment for writing.
MOV	Load and store debug registers.
INVD	Invalidate cache, no writeback.
WBINVD	Invalidate cache, with writeback.
INVLPG	Invalidate TLB Entry.
INVPCID	Invalidate Process-Context Identifier.
LOCK (prefix)	Lock Bus.
HLT	Halt processor.
RSM	Return from system management mode (SMM).
RDMSR	Read model-specific register.
WRMSR	Write model-specific register.
RDPMC	Read performance monitoring counters.
RDTSC	Read time stamp counter.
RDTSCP	Read time stamp counter and processor ID.
SYSENTER	Fast System Call, transfers to a flat protected mode kernel at CPL = 0.
SYSEXIT	Fast System Call, transfers to a flat protected mode kernel at CPL = 3.
XSAVE	Save processor extended states to memory.
XSAVEC	Save processor extended states with compaction to memory.
XSAVEOPT	Save processor extended states to memory, optimized.
XSAVES	Save processor supervisor-mode extended states to memory.
XRSTOR	Restore processor extended states from memory.
XRSTORS	Restore processor supervisor-mode extended states from memory.
XGETBV	Reads the state of an extended control register.
XSETBV	Writes the state of an extended control register.
RDFSBASE	Reads from FS base address at any privilege level.
RDGSBASE	Reads from GS base address at any privilege level.
WRFSBASE	Writes to FS base address at any privilege level.
WRGSBASE	Writes to GS base address at any privilege level.

5.21 64-BIT MODE INSTRUCTIONS

The following instructions are introduced in 64-bit mode. This mode is a sub-mode of IA-32e mode.

CDQE	Convert doubleword to quadword.
CMPSQ	Compare string operands.
CMPXCHG16B	Compare RDX:RAX with m128.
LODSQ	Load qword at address (R)SI into RAX.
MOVSQ	Move qword from address (R)SI to (R)DI.
MOVZX (64-bits)	Move bytes/words to doublewords/quadwords, zero-extension.
STOSQ	Store RAX at address RDI.
SWAPGS	Exchanges current GS base register value with value in MSR address C0000102H.
SYSCALL	Fast call to privilege level 0 system procedures.
SYSRET	Return from fast systemcall.

5.22 VIRTUAL-MACHINE EXTENSIONS

The behavior of the VMCS-maintenance instructions is summarized below:

VMPTRLD	Takes a single 64-bit source operand in memory. It makes the referenced VMCS active and current.
VMPTRST	Takes a single 64-bit destination operand that is in memory. Current-VMCS pointer is stored into the destination operand.
VMCLEAR	Takes a single 64-bit operand in memory. The instruction sets the launch state of the VMCS referenced by the operand to "clear", renders that VMCS inactive, and ensures that data for the VMCS have been written to the VMCS-data area in the referenced VMCS region.
VMREAD	Reads a component from the VMCS (the encoding of that field is given in a register operand) and stores it into a destination operand.
VMWRITE	Writes a component to the VMCS (the encoding of that field is given in a register operand) from a source operand.

The behavior of the VMX management instructions is summarized below:

VMLAUNCH	Launches a virtual machine managed by the VMCS. A VM entry occurs, transferring control to the VM.
VMRESUME	Resumes a virtual machine managed by the VMCS. A VM entry occurs, transferring control to the VM.
VMXOFF	Causes the processor to leave VMX operation.
VMXON	Takes a single 64-bit source operand in memory. It causes a logical processor to enter VMX root operation and to use the memory referenced by the operand to support VMX operation.

The behavior of the VMX-specific TLB-management instructions is summarized below:

INVEPT	Invalidate cached **Extended Page Table** (EPT) mappings in the processor to synchronize address translation in virtual machines with memory-resident EPT pages.
INVVPID	Invalidate cached mappings of address translation based on the **Virtual Processor ID** (VPID).

None of the instructions above can be executed in compatibility mode; they generate invalid-opcode exceptions if executed in compatibility mode.

The behavior of the guest-available instructions is summarized below:

VMCALL	Allows a guest in VMX non-root operation to call the VMM for service. A VM exit occurs, transferring control to the VMM.
VMFUNC	This instruction allows software in VMX non-root operation to invoke a VM function, which is processor functionality enabled and configured by software in VMX root operation. No VM exit occurs.

5.23 SAFER MODE EXTENSIONS

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

GETSEC[CAPABILITIES]	Returns the available leaf functions of the GETSEC instruction.
GETSEC[ENTERACCS]	Loads an authenticated code chipset module and enters authenticated code execution mode.
GETSEC[EXITAC]	Exits authenticated code execution mode.
GETSEC[SENTER]	Establishes a Measured Launched Environment (MLE) which has its dynamic root of trust anchored to a chipset supporting Intel Trusted Execution Technology.
GETSEC[SEXIT]	Exits the MLE.
GETSEC[PARAMETERS]	Returns SMX related parameter information.
GETSEC[SMCRTL]	SMX mode control.
GETSEC[WAKEUP]	Wakes up sleeping logical processors inside an MLE.

5.24 INTEL® MEMORY PROTECTION EXTENSIONS

Intel Memory Protection Extensions (MPX) provides a set of instructions to enable software to add robust bounds checking capability to memory references. Details of Intel MPX are described in Chapter 17, "Intel® MPX".

BNDMK	Create a LowerBound and a UpperBound in a register.
BNDCL	Check the address of a memory reference against a LowerBound.
BNDCU	Check the address of a memory reference against an UpperBound in 1's compliment form.
BNDCN	Check the address of a memory reference against an UpperBound not in 1's compliment form.
BNDMOV	Copy or load from memory of the LowerBound and UpperBound to a register.
BNDMOV	Store to memory of the LowerBound and UpperBound from a register.
BNDLDX	Load bounds using address translation.
BNDSTX	Store bounds using address translation.

5.25 INTEL® SOFTWARE GUARD EXTENSIONS

Intel Software Guard Extensions (Intel SGX) provide two sets of instruction leaf functions to enable application software to instantiate a protected container, referred to as an enclave. The enclave instructions are organized as leaf functions under two instruction mnemonics: ENCLS (ring 0) and ENCLU (ring 3). Details of Intel SGX are described in CHAPTER 36 through CHAPTER 42 of *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3D*.

The first implementation of Intel SGX is also referred to as SGX1, it is introduced with the 6th Generation Intel Core Processors. The leaf functions supported in SGX1 is shown in Table 5-3.

Table 5-3. Supervisor and User Mode Enclave Instruction Leaf Functions in Long-Form of SGX1

Supervisor Instruction	Description	User Instruction	Description
ENCLS[EADD]	Add a page	ENCLU[EENTER]	Enter an Enclave
ENCLS[EBLOCK]	Block an EPC page	ENCLU[EEXIT]	Exit an Enclave
ENCLS[ECREATE]	Create an enclave	ENCLU[EGETKEY]	Create a cryptographic key
ENCLS[EDBGRD]	Read data by debugger	ENCLU[EREPORT]	Create a cryptographic report
ENCLS[EDBGWR]	Write data by debugger	ENCLU[ERESUME]	Re-enter an Enclave
ENCLS[EEXTEND]	Extend EPC page measurement		
ENCLS[EINIT]	Initialize an enclave		
ENCLS[ELDB]	Load an EPC page as blocked		
ENCLS[ELDU]	Load an EPC page as unblocked		
ENCLS[EPA]	Add version array		
ENCLS[EREMOVE]	Remove a page from EPC		
ENCLS[ETRACK]	Activate EBLOCK checks		
ENCLS[EWB]	Write back/invalidate an EPC page		

3. Updates to Chapter 13, Volume 1

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

Change to this chapter: Typo corrections in Section 13.5.4 "MPX State" and Section 13.6 "Processor Tracking of XSAVE-Managed State".

CHAPTER 13
MANAGING STATE USING THE XSAVE FEATURE SET

The XSAVE feature set extends the functionality of the FXSAVE and FXRSTOR instructions (see Section 10.5, "FXSAVE and FXRSTOR Instructions") by supporting the saving and restoring of processor state in addition to the x87 execution environment (**x87 state**) and the registers used by the streaming SIMD extensions (**SSE state**).

The **XSAVE feature set** comprises eight instructions. XGETBV and XSETBV allow software to read and write the extended control register XCR0, which controls the operation of the XSAVE feature set. XSAVE, XSAVEOPT, XSAVEC, and XSAVES are four instructions that save processor state to memory; XRSTOR and XRSTORS are corresponding instructions that load processor state from memory. XGETBV, XSAVE, XSAVEOPT, XSAVEC, and XRSTOR can be executed at any privilege level; XSETBV, XSAVES, and XRSTORS can be executed only if CPL = 0. In addition to XCR0, the XSAVES and XRSTORS instructions are controlled also by the IA32_XSS MSR (index DA0H).

The XSAVE feature set organizes the state that manages into **state components**. Operation of the instructions is based on **state-component bitmaps** that have the same format as XCR0 and as the IA32_XSS MSR: each bit corresponds to a state component. Section 13.1 discusses these state components and bitmaps in more detail.

Section 13.2 describes how the processor enumerates support for the XSAVE feature set and for **XSAVE-enabled features** (those features that require use of the XSAVE feature set for their enabling). Section 13.3 explains how software can enable the XSAVE feature set and XSAVE-enabled features.

The XSAVE feature set allows saving and loading processor state from a region of memory called an **XSAVE area**. Section 13.4 presents details of the XSAVE area and its organization. Each XSAVE-managed state component is associated with a section of the XSAVE area. Section 13.5 describes in detail each of the XSAVE-managed state components.

Section 13.7 through Section 13.12 describe the operation of XSAVE, XRSTOR, XSAVEOPT, XSAVEC, XSAVES, and XRSTORS, respectively.

13.1 XSAVE-SUPPORTED FEATURES AND STATE-COMPONENT BITMAPS

The XSAVE feature set supports the saving and restoring of **state components**, each of which is a discrete set of processor registers (or parts of registers). In general, each such state component corresponds to a particular CPU feature. Such a feature is **XSAVE-supported**. Some XSAVE-supported features use registers in multiple XSAVE-managed state components.

The XSAVE feature set organizes the state components of the XSAVE-supported features using **state-component bitmaps**. A state-component bitmap comprises 64 bits; each bit in such a bitmap corresponds to a single state component. The following bits are defined in state-component bitmaps:

- Bit 0 corresponds to the state component used for the x87 FPU execution environment (**x87 state**). See Section 13.5.1.
- Bit 1 corresponds to the state component used for registers used by the streaming SIMD extensions (**SSE state**). See Section 13.5.2.
- Bit 2 corresponds to the state component used for the additional register state used by the Intel® Advanced Vector Extensions (**AVX state**). See Section 13.5.3.
- Bits 4:3 correspond to the two state components used for the additional register state used by Intel® Memory Protection Extensions (**MPX state**):
  - State component 3 is used for the 4 128-bit bounds registers BND0–BND3 (**BNDREGS state**).
  - State component 4 is used for the 64-bit user-mode MPX configuration register BNDCFGU and the 64-bit MPX status register BNDSTATUS (**BNDCSR state**).
- Bits 7:5 correspond to the three state components used for the additional register state used by Intel® Advanced Vector Extensions 512 (**AVX-512 state**):
  - State component 5 is used for the 8 64-bit opmask registers k0–k7 (**opmask state**).

  - State component 6 is used for the upper 256 bits of the registers ZMM0–ZMM15. These 16 256-bit values are denoted ZMM0_H–ZMM15_H (**ZMM_Hi256 state**).
  - State component 7 is used for the 16 512-bit registers ZMM16–ZMM31 (**Hi16_ZMM state**).
- Bit 8 corresponds to the state component used for the Intel Processor Trace MSRs (**PT state**).
- Bit 9 corresponds to the state component used for the protection-key feature's register PKRU (**PKRU state**). See Section 13.5.7.

Bits in the range 62:10 are not currently defined in state-component bitmaps and are reserved for future expansion. As individual state component is defined within bits 62:10, additional sub-sections are updated within Section 13.5 over time. Bit 63 is used for special functionality in some bitmaps and does not correspond to any state component.

The state component corresponding to bit *i* of state-component bitmaps is called **state component *i***. Thus, x87 state is state component 0; SSE state is state component 1; AVX state is state component 2; MPX state comprises state components 3–4; AVX-512 state comprises state components 5–7; PT state is state component 8; and PKRU state is state component 9.

The XSAVE feature set uses state-component bitmaps in multiple ways. Most of the instructions use an implicit operand (in EDX:EAX), called the **instruction mask**, which is the state-component bitmap that specifies the state components on which the instruction operates.

Some state components are **user state components**, and they can be managed by the entire XSAVE feature set. Other state components are **supervisor state components**, and they can be managed only by XSAVES and XRSTORS. All the state components corresponding to bits in the range 9:0 are user state components, except PT state (corresponding to bit 8), which is a supervisor state component.

Extended control register XCR0 contains a state-component bitmap that specifies the user state components that software has enabled the XSAVE feature set to manage. If the bit corresponding to a state component is clear in XCR0, instructions in the XSAVE feature set will not operate on that state component, regardless of the value of the instruction mask.

The IA32_XSS MSR (index DA0H) contains a state-component bitmap that specifies the supervisor state components that software has enabled XSAVES and XRSTORS to manage (XSAVE, XSAVEC, XSAVEOPT, and XRSTOR cannot manage supervisor state components). If the bit corresponding to a state component is clear in the IA32_XSS MSR, XSAVES and XRSTORS will not operate on that state component, regardless of the value of the instruction mask.

Some XSAVE-supported features can be used only if XCR0 has been configured so that the features' state components can be managed by the XSAVE feature set. (This applies only to features with user state components.) Such state components and features are **XSAVE-enabled**. In general, the processor will not modify (or allow modification of) the registers of a state component of an XSAVE-enabled feature if the bit corresponding to that state component is clear in XCR0. (If software clears such a bit in XCR0, the processor preserves the corresponding state component.) If an XSAVE-enabled feature has not been fully enabled in XCR0, execution of any instruction defined for that feature causes an invalid-opcode exception (#UD).

As will be explained in Section 13.3, the XSAVE feature set is enabled only if CR4.OSXSAVE[bit 18] = 1. If CR4.OSXSAVE = 0, the processor treats XSAVE-enabled state features and their state components as if all bits in XCR0 were clear; the state components cannot be modified and the features' instructions cannot be executed.

The state components for x87 state, for SSE state, for PT state, and for PKRU state are XSAVE-managed but the corresponding features are not XSAVE-enabled. Processors allow modification of this state, as well as execution of x87 FPU instructions and SSE instructions and use of Intel Processor Trace and protection keys, regardless of the value of CR4.OSXSAVE and XCR0.

13.2 ENUMERATION OF CPU SUPPORT FOR XSAVE INSTRUCTIONS AND XSAVE-SUPPORTED FEATURES

A processor enumerates support for the XSAVE feature set and for features supported by that feature set using the CPUID instruction. The following items provide specific details:

- CPUID.1:ECX.XSAVE[bit 26] enumerates general support for the XSAVE feature set:

— If this bit is 0, the processor does not support any of the following instructions: XGETBV, XRSTOR, XRSTORS, XSAVE, XSAVEC, XSAVEOPT, XSAVES, and XSETBV; the processor provides no further enumeration through CPUID function 0DH (see below).

— If this bit is 1, the processor supports the following instructions: XGETBV, XRSTOR, XSAVE, and XSETBV.[1] Further enumeration is provided through CPUID function 0DH.

CR4.OSXSAVE can be set to 1 if and only if CPUID.1:ECX.XSAVE[bit 26] is enumerated as 1.

- CPUID function 0DH enumerates details of CPU support through a set of sub-functions. Software selects a specific sub-function by the value placed in the ECX register. The following items provide specific details:

  — CPUID function 0DH, sub-function 0.

    - EDX:EAX is a bitmap of all the user state components that can be managed using the XSAVE feature set. A bit can be set in XCR0 if and only if the corresponding bit is set in this bitmap. Every processor that supports the XSAVE feature set will set EAX[0] (x87 state) and EAX[1] (SSE state).

      If EAX[*i*] = 1 (for $1 < i < 32$) or EDX[*i*–32] = 1 (for $32 \le i < 63$), sub-function *i* enumerates details for state component *i* (see below).

    - ECX enumerates the size (in bytes) required by the XSAVE instruction for an XSAVE area containing all the user state components supported by this processor.

    - EBX enumerates the size (in bytes) required by the XSAVE instruction for an XSAVE area containing all the user state components corresponding to bits currently set in XCR0.

  — CPUID function 0DH, sub-function 1.

    - EAX[0] enumerates support for the XSAVEOPT instruction. The instruction is supported if and only if this bit is 1. If EAX[0] = 0, execution of XSAVEOPT causes an invalid-opcode exception (#UD).

    - EAX[1] enumerates support for **compaction extensions** to the XSAVE feature set. The following are supported if this bit is 1:

      — The compacted format of the extended region of XSAVE areas (see Section 13.4.3).

      — The XSAVEC instruction. If EAX[1] = 0, execution of XSAVEC causes a #UD.

      — Execution of the compacted form of XRSTOR (see Section 13.8).

    - EAX[2] enumerates support for execution of XGETBV with ECX = 1. This allows software to determine the state of the init optimization. See Section 13.6.

    - EAX[3] enumerates support for XSAVES, XRSTORS, and the IA32_XSS MSR. If EAX[3] = 0, execution of XSAVES or XRSTORS causes a #UD; an attempt to access the IA32_XSS MSR using RDMSR or WRMSR causes a general-protection exception (#GP). Every processor that supports a supervisor state component sets EAX[3]. Every processor that sets EAX[3] (XSAVES, XRSTORS, IA32_XSS) will also set EAX[1] (the compaction extensions).

    - EAX[31:4] are reserved.

    - EBX enumerates the size (in bytes) required by the XSAVES instruction for an XSAVE area containing all the state components corresponding to bits currently set in XCR0 | IA32_XSS.

    - EDX:ECX is a bitmap of all the supervisor state components that can be managed by XSAVES and XRSTORS. A bit can be set in the IA32_XSS MSR if and only if the corresponding bit is set in this bitmap.

NOTE

In summary, the XSAVE feature set supports state component *i* ($0 \le i < 63$) if one of the following is true: (1) $i < 32$ and CPUID.(EAX=0DH,ECX=0):EAX[*i*] = 1; (2) $i \ge 32$ and CPUID.(EAX=0DH,ECX=0):EAX[*i*–32] = 1; (3) $i < 32$ and CPUID.(EAX=0DH,ECX=1):ECX[*i*] = 1; or (4) $i \ge 32$ and CPUID.(EAX=0DH,ECX=1):EDX[*i*–32] = 1. The XSAVE feature set supports user state component *i* if (1) or (2) holds; if (3) or (4) holds, state component *i* is a supervisor state component and support is limited to XSAVES and XRSTORS.

1. If CPUID.1:ECX.XSAVE[bit 26] = 1, XGETBV and XSETBV may be executed with ECX = 0 (to read and write XCR0). Any support for execution of these instructions with other values of ECX is enumerated separately.

— CPUID function 0DH, sub-function *i* (*i* > 1). This sub-function enumerates details for state component *i*. If the XSAVE feature set supports state component *i* (see note above), the following items provide specific details:

- EAX enumerates the size (in bytes) required for state component *i*.
- If state component *i* is a user state component, EBX enumerates the offset (in bytes, from the base of the XSAVE area) of the section used for state component *i*. (This offset applies only when the standard format for the extended region of the XSAVE area is being used; see Section 13.4.3.)
- If state component *i* is a supervisor state component, EBX returns 0.
- If state component *i* is a user state component, ECX[0] return 0; if state component *i* is a supervisor state component, ECX[0] returns 1.
- The value returned by ECX[1] indicates the alignment of state component *i* when the compacted format of the extended region of an XSAVE area is used (see Section 13.4.3). If ECX[1] returns 0, state component *i* is located immediately following the preceding state component; if ECX[1] returns 1, state component *i* is located on the next 64-byte boundary following the preceding state component.
- ECX[31:2] and EDX return 0.

If the XSAVE feature set does not support state component *i*, sub-function *i* returns 0 in EAX, EBX, ECX, and EDX.

13.3 ENABLING THE XSAVE FEATURE SET AND XSAVE-ENABLED FEATURES

Software enables the XSAVE feature set by setting CR4.OSXSAVE[bit 18] to 1 (e.g., with the MOV to CR4 instruction). If this bit is 0, execution of any of XGETBV, XRSTOR, XRSTORS, XSAVE, XSAVEC, XSAVEOPT, XSAVES, and XSETBV causes an invalid-opcode exception (#UD).

When CR4.OSXSAVE = 1 and CPL = 0, executing the XSETBV instruction with ECX = 0 writes the 64-bit value in EDX:EAX to XCR0 (EAX is written to XCR0[31:0] and EDX to XCR0[63:32]). (Execution of the XSETBV instruction causes a general-protection fault — #GP — if CPL > 0.) The following items provide details regarding individual bits in XCR0:

- XCR0[0] is associated with x87 state (see Section 13.5.1). XCR0[0] is always 1. It has that value coming out of RESET. Executing the XSETBV instruction causes a general-protection fault (#GP) if ECX = 0 and EAX[0] is 0.
- XCR0[1] is associated with SSE state (see Section 13.5.2). Software can use the XSAVE feature set to manage SSE state only if XCR0[1] = 1. The value of XCR0[1] in no way determines whether software can execute SSE instructions (these instructions can be executed even if XCR0[1] = 0).

  XCR0[1] is 0 coming out of RESET. As noted in Section 13.2, every processor that supports the XSAVE feature set allows software to set XCR0[1].
- XCR0[2] is associated with AVX state (see Section 13.5.3). Software can use the XSAVE feature set to manage AVX state only if XCR0[2] = 1. In addition, software can execute AVX instructions only if CR4.OSXSAVE = XCR0[2] = 1. Otherwise, any execution of an AVX instruction causes an invalid-opcode exception (#UD).

  XCR0[2] is 0 coming out of RESET. As noted in Section 13.2, a processor allows software to set XCR0[2] if and only if CPUID.(EAX=0DH,ECX=0):EAX[2] = 1. In addition, executing the XSETBV instruction causes a general-protection fault (#GP) if ECX = 0 and EAX[2:1] has the value 10b; that is, software cannot enable the XSAVE feature set for AVX state but not for SSE state.

  As noted in Section 13.1, the processor will preserve AVX state unmodified if software clears XCR0[2]. However, clearing XCR0[2] while AVX state is not in its initial configuration may cause SSE instructions to incur a power and performance penalty. See Section 13.5.3, "Enable the Use Of XSAVE Feature Set And XSAVE State Components" of *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*, for how system software can avoid this penalty.
- XCR0[4:3] are associated with MPX state (see Section 13.5.4). Software can use the XSAVE feature set to manage MPX state only if XCR0[4:3] = 11b. In addition, software can execute MPX instructions only if CR4.OSXSAVE = 1 and XCR0[4:3] = 11b. Otherwise, any execution of an MPX instruction causes an invalid-opcode exception (#UD).[1]

XCR0[4:3] have value 00b coming out of RESET. As noted in Section 13.2, a processor allows software to set XCR0[4:3] to 11b if and only if CPUID.(EAX=0DH,ECX=0):EAX[4:3] = 11b. In addition, executing the XSETBV instruction causes a general-protection fault (#GP) if ECX = 0, EAX[4:3] is neither 00b nor 11b; that is, software can enable the XSAVE feature set for MPX state only if it does so for both state components.

As noted in Section 13.1, the processor will preserve MPX state unmodified if software clears XCR0[4:3].

- XCR0[7:5] are associated with AVX-512 state (see Section 13.5.5). Software can use the XSAVE feature set to manage AVX-512 state only if XCR0[7:5] = 111b. In addition, software can execute AVX-512 instructions only if CR4.OSXSAVE = 1 and XCR0[7:5] = 111b. Otherwise, any execution of an AVX-512 instruction causes an invalid-opcode exception (#UD).

  XCR0[7:5] have value 000b coming out of RESET. As noted in Section 13.2, a processor allows software to set XCR0[7:5] to 111b if and only if CPUID.(EAX=0DH,ECX=0):EAX[7:5] = 111b. In addition, executing the XSETBV instruction causes a general-protection fault (#GP) if ECX = 0, EAX[7:5] is not 000b, and any bit is clear in EAX[2:1] or EAX[7:5]; that is, software can enable the XSAVE feature set for AVX-512 state only if it does so for all three state components, and only if it also does so for AVX state and SSE state. This implies that the value of XCR0[7:5] is always either 000b or 111b.

  As noted in Section 13.1, the processor will preserve AVX-512 state unmodified if software clears XCR0[7:5]. However, clearing XCR0[7:5] while AVX-512 state is not in its initial configuration may cause SSE and AVX instructions to incur a power and performance penalty. See Section 13.5.3, "Enable the Use Of XSAVE Feature Set And XSAVE State Components" of *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*, for how system software can avoid this penalty.

- XCR0[9] is associated with PKRU state (see Section 13.5.7). Software can use the XSAVE feature set to manage PKRU state only if XCR0[9] = 1. The value of XCR0[9] in no way determines whether software can use protection keys or execute other instructions that access PKRU state (these instructions can be executed even if XCR0[9] = 0).

  XCR0[9] is 0 coming out of RESET. As noted in Section 13.2, a processor allows software to set XCR0[9] if and only if CPUID.(EAX=0DH,ECX=0):EAX[9] = 1.

- XCR0[63:10] and XCR0[8] are reserved.[1] Executing the XSETBV instruction causes a general-protection fault (#GP) if ECX = 0 and any corresponding bit in EDX:EAX is not 0. These bits in XCR0 are all 0 coming out of RESET.

Software operating with CPL > 0 may need to determine whether the XSAVE feature set and certain XSAVE-enabled features have been enabled. If CPL > 0, execution of the MOV from CR4 instruction causes a general-protection fault (#GP). The following alternative mechanisms allow software to discover the enabling of the XSAVE feature set regardless of CPL:

- The value of CR4.OSXSAVE is returned in CPUID.1:ECX.OSXSAVE[bit 27]. If software determines that CPUID.1:ECX.OSXSAVE = 1, the processor supports the XSAVE feature set and the feature set has been enabled in CR4.
- Executing the XGETBV instruction with ECX = 0 returns the value of XCR0 in EDX:EAX. XGETBV can be executed if CR4.OSXSAVE = 1 (if CPUID.1:ECX.OSXSAVE = 1), regardless of CPL.

Thus, software can use the following algorithm to determine the support and enabling for the XSAVE feature set:

1. Use CPUID to discover the value of CPUID.1:ECX.OSXSAVE.
   - — If the bit is 0, either the XSAVE feature set is not supported by the processor or has not been enabled by software. Either way, the XSAVE feature set is not available, nor are XSAVE-enabled features such as AVX.
   - — If the bit is 1, the processor supports the XSAVE feature set — including the XGETBV instruction — and it has been enabled by software. The XSAVE feature set can be used to manage x87 state (because XCR0[0] is always 1). Software requiring more detailed information can go on to the next step.
2. Execute XGETBV with ECX = 0 to discover the value of XCR0. If XCR0[1] = 1, the XSAVE feature set can be used to manage SSE state. If XCR0[2] = 1, the XSAVE feature set can be used to manage AVX state and software can execute AVX instructions. If XCR0[4:3] is 11b, the XSAVE feature set can be used to manage MPX

1. If XCR0[3] = 0, executions of CALL, RET, JMP, and Jcc do not initialize the bounds registers.

1. Bit 8 corresponds to a supervisor state component. Since bits can be set in XCR0 only for user state components, that bit of XCR0 must be 0.

state and software can execute MPX instructions. If XCR0[7:5] is 111b, the XSAVE feature set can be used to manage AVX-512 state and software can execute AVX-512 instructions. If XCR0[9] = 1, the XSAVE feature set can be used to manage PKRU state.

The IA32_XSS MSR (with MSR index DA0H) is zero coming out of RESET. If CR4.OSXSAVE = 1, CPUID.(EAX=0DH,ECX=1):EAX[3] = 1, and CPL = 0, executing the WRMSR instruction with ECX = DA0H writes the 64-bit value in EDX:EAX to the IA32_XSS MSR (EAX is written to IA32_XSS[31:0] and EDX to IA32_XSS[63:32]). The following items provide details regarding individual bits in the IA32_XSS MSR:

- IA32_XSS[8] is associated with PT state (see Section 13.5.6). Software can use XSAVES and XRSTORS to manage PT state only if IA32_XSS[8] = 1. The value of IA32_XSS[8] does not determine whether software can use Intel Processor Trace (the feature can be used even if IA32_XSS[8] = 0).
- IA32_XSS[63:9] and IA32_XSS[7:0] are reserved.[1] Executing the WRMSR instruction causes a general-protection fault (#GP) if ECX = DA0H and any corresponding bit in EDX:EAX is not 0. These bits in XCR0 are all 0 coming out of RESET.

The IA32_XSS MSR is 0 coming out of RESET.

There is no mechanism by which software operating with CPL > 0 can discover the value of the IA32_XSS MSR.

13.4 XSAVE AREA

The XSAVE feature set includes instructions that save and restore the XSAVE-managed state components to and from memory: XSAVE, XSAVEOPT, XSAVEC, and XSAVES (for saving); and XRSTOR and XRSTORS (for restoring). The processor organizes the state components in a region of memory called an **XSAVE area**. Each of the save and restore instructions takes a memory operand that specifies the 64-byte aligned base address of the XSAVE area on which it operates.

Every XSAVE area has the following format:

- The **legacy region**. The legacy region of an XSAVE area comprises the 512 bytes starting at the area's base address. It is used to manage the state components for x87 state and SSE state. The legacy region is described in more detail in Section 13.4.1.
- The **XSAVE header**. The XSAVE header of an XSAVE area comprises the 64 bytes starting at an offset of 512 bytes from the area's base address. The XSAVE header is described in more detail in Section 13.4.2.
- The **extended region**. The extended region of an XSAVE area starts at an offset of 576 bytes from the area's base address. It is used to manage the state components other than those for x87 state and SSE state. The extended region is described in more detail in Section 13.4.3. The size of the extended region is determined by which state components the processor supports and which bits have been set in XCR0 and IA32_XSS (see Section 13.3).

13.4.1 Legacy Region of an XSAVE Area

The legacy region of an XSAVE area comprises the 512 bytes starting at the area's base address. It has the same format as the FXSAVE area (see Section 10.5.1). The XSAVE feature set uses the legacy area for x87 state (state component 0) and SSE state (state component 1). Table 13-1 illustrates the format of the first 416 bytes of the legacy region of an XSAVE area.

Table 13-1. Format of the Legacy Region of an XSAVE Area

15 14	13 12	11 10	9 8	7 6	5	4	3 2	1 0	
FIP[63:48] or reserved	FCS or FIP[47:32]	FIP[31:0]		FOP	Rsvd.	FTW	FSW	FCW	**0**
MXCSR_MASK		MXCSR		FDP[63:48] or reserved	FDS or FDP[47:32]		FDP[31:0]		**16**

1. Bit 9 and bits 7:0 correspond to user state components. Since bits can be set in the IA32_XSS MSR only for supervisor state components, those bits of the MSR must be 0.

Table 13-1. Format of the Legacy Region of an XSAVE Area (Contd.) (Contd.)

15 14	13 12	11 10	9 8	7 6	5 4	3 2	1 0	
Reserved			ST0/MM0					**32**
Reserved			ST1/MM1					**48**
Reserved			ST2/MM2					**64**
Reserved			ST3/MM3					**80**
Reserved			ST4/MM4					**96**
Reserved			ST5/MM5					**112**
Reserved			ST6/MM6					**128**
Reserved			ST7/MM7					**144**
XMM0								**160**
XMM1								**176**
XMM2								**192**
XMM3								**208**
XMM4								**224**
XMM5								**240**
XMM6								**256**
XMM7								**272**
XMM8								**288**
XMM9								**304**
XMM10								**320**
XMM11								**336**
XMM12								**352**
XMM13								**368**
XMM14								**384**
XMM15								**400**

The x87 state component comprises bytes 23:0 and bytes 159:32. The SSE state component comprises bytes 31:24 and bytes 415:160. The XSAVE feature set does not use bytes 511:416; bytes 463:416 are reserved.

Section 13.7 through Section 13.9 provide details of how instructions in the XSAVE feature set use the legacy region of an XSAVE area.

13.4.2 XSAVE Header

The XSAVE header of an XSAVE area comprises the 64 bytes starting at offset 512 from the area's base address:

- Bytes 7:0 of the XSAVE header is a state-component bitmap (see Section 13.1) called **XSTATE_BV**. It identifies the state components in the XSAVE area.
- Bytes 15:8 of the XSAVE header is a state-component bitmap called **XCOMP_BV**. It is used as follows:
  - XCOMP_BV[63] indicates the format of the extended region of the XSAVE area (see Section 13.4.3). If it is clear, the standard format is used. If it is set, the compacted format is used; XCOMP_BV[62:0] provide format specifics as specified in Section 13.4.3.
  - XCOMP_BV[63] determines which form of the XRSTOR instruction is used. If the bit is set, the compacted form is used; otherwise, the standard form is used. See Section 13.8.

  - All bits in XCOMP_BV should be 0 if the processor does not support the compaction extensions to the XSAVE feature set.
- Bytes 63:16 of the XSAVE header are reserved.

Section 13.7 through Section 13.9 provide details of how instructions in the XSAVE feature set use the XSAVE header of an XSAVE area.

13.4.3 Extended Region of an XSAVE Area

The extended region of an XSAVE area starts at byte offset 576 from the area's base address. The size of the extended region is determined by which state components the processor supports and which bits have been set in XCR0 | IA32_XSS (see Section 13.3).

The XSAVE feature set uses the extended area for each state component *i*, where $i \geq 2$. The following state components are currently supported in the extended area: state component 2 contains AVX state; state components 5–7 contain AVX-512 state; and state component 9 contains PKRU state.

The extended region of the an XSAVE area may have one of two formats. The **standard format** is supported by all processors that support the XSAVE feature set; the **compacted format** is supported by those processors that support the compaction extensions to the XSAVE feature set (see Section 13.2). Bit 63 of the XCOMP_BV field in the XSAVE header (see Section 13.4.2) indicates which format is used.

The following items describe the two possible formats of the extended region:

- **Standard format.** Each state component *i* ($i \geq 2$) is located at the byte offset from the base address of the XSAVE area enumerated in CPUID.(EAX=0DH,ECX=*i*):EBX. (CPUID.(EAX=0DH,ECX=*i*):EAX enumerates the number of bytes required for state component *i*.
- **Compacted format.** Each state component *i* ($i \geq 2$) is located at a byte offset from the base address of the XSAVE area based on the XCOMP_BV field in the XSAVE header:
  - If XCOMP_BV[*i*] = 0, state component *i* is not in the XSAVE area.
  - If XCOMP_BV[*i*] = 1, state component *i* is located at a byte offset $location_i$ from the base address of the XSAVE area, where $location_i$ is determined by the following items:
    - If XCOMP_BV[*j*] = 0 for every *j*, $2 \leq j < i$, $location_i$ is 576. (This item applies if *i* is the first bit set in bits 62:2 of the XCOMP_BV; it implies that state component *i* is located at the beginning of the extended region.)
    - Otherwise, let *j*, $2 \leq j < i$, be the greatest value such that XCOMP_BV[*j*] = 1. Then $location_i$ is determined by the following values: $location_j$; $size_j$, as enumerated in CPUID.(EAX=0DH,ECX=*j*):EAX; and the value of $align_i$, as enumerated in CPUID.(EAX=0DH,ECX=*i*):ECX[1]:
      - If $align_i = 0$, $location_i = location_j + size_j$. (This item implies that state component *i* is located immediately following the preceding state component whose bit is set in XCOMP_BV.)
      - If $align_i = 1$, $location_i$ = ceiling($location_j + size_j$, 64). (This item implies that state component *i* is located on the next 64-byte boundary following the preceding state component whose bit is set in XCOMP_BV.)

13.5 XSAVE-MANAGED STATE

The section provides details regarding how the XSAVE feature set interacts with the various XSAVE-managed state components.

Unless otherwise state, the state pertaining to a particular state component is saved beginning at byte 0 of the section of the XSAVE are corresponding to that state component.

13.5.1 x87 State

Instructions in the XSAVE feature set can manage the same state of the x87 FPU execution environment (**x87 state**) that can be managed using the FXSAVE and FXRSTOR instructions. They organize all x87 state as a user state component in the legacy region of the XSAVE area (see Section 13.4.1). This region is illustrated in Table 13-1; the x87 state is listed below, along with details of its interactions with the XSAVE feature set:

- Bytes 1:0, 3:2, 7:6. These are used for the x87 FPU Control Word (FCW), the x87 FPU Status Word (FSW), and the x87 FPU Opcode (FOP), respectively.
- Byte 4 is used for an abridged version of the x87 FPU Tag Word (FTW). The following items describe its usage:
  - For each j, $0 \le j \le 7$, XSAVE, XSAVEOPT, XSAVEC, and XSAVES save a 0 into bit j of byte 4 if x87 FPU data register STj has a empty tag; otherwise, XSAVE, XSAVEOPT, XSAVEC, and XSAVES save a 1 into bit j of byte 4.
  - For each j, $0 \le j \le 7$, XRSTOR and XRSTORS establish the tag value for x87 FPU data register STj as follows. If bit j of byte 4 is 0, the tag for STj in the tag register for that data register is marked empty (11B); otherwise, the x87 FPU sets the tag for STj based on the value being loaded into that register (see below).
- Bytes 15:8 are used as follows:
  - If the instruction has no REX prefix, or if REX.W = 0:
    - Bytes 11:8 are used for bits 31:0 of the x87 FPU Instruction Pointer Offset (FIP).
    - If CPUID.(EAX=07H,ECX=0H):EBX[bit 13] = 0, bytes 13:12 are used for x87 FPU Instruction Pointer Selector (FCS). Otherwise, XSAVE, XSAVEOPT, XSAVEC, and XSAVES save these bytes as 0000H, and XRSTOR and XRSTORS ignore them.
    - Bytes 15:14 are not used.
  - If the instruction has a REX prefix with REX.W = 1, bytes 15:8 are used for the full 64 bits of FIP.
- Bytes 23:16 are used as follows:
  - If the instruction has no REX prefix, or if REX.W = 0:
    - Bytes 19:16 are used for bits 31:0 of the x87 FPU Data Pointer Offset (FDP).
    - If CPUID.(EAX=07H,ECX=0H):EBX[bit 13] = 0, bytes 21:20 are used for x87 FPU Data Pointer Selector (FDS). Otherwise, XSAVE, XSAVEOPT, XSAVEC, and XSAVES save these bytes as 0000H; and XRSTOR and XRSTORS ignore them.
    - Bytes 23:22 are not used.
  - If the instruction has a REX prefix with REX.W = 1, bytes 23:16 are used for the full 64 bits of FDP.
- Bytes 31:24 are used for SSE state (see Section 13.5.2).
- Bytes 159:32 are used for the registers ST0–ST7 (MM0–MM7). Each of the 8 register is allocated a 128-bit region, with the low 80 bits used for the register and the upper 48 bits unused.

x87 state is XSAVE-managed but the x87 FPU feature is not XSAVE-enabled. The XSAVE feature set can operate on x87 state only if the feature set is enabled (CR4.OSXSAVE = 1).[1] Software can otherwise use x87 state even if the XSAVE feature set is not enabled.

13.5.2 SSE State

Instructions in the XSAVE feature set can manage the registers used by the streaming SIMD extensions (**SSE state**) just as the FXSAVE and FXRSTOR instructions do. They organize all SSE state as a user state component in the legacy region of the XSAVE area (see Section 13.4.1). This region is illustrated in Table 13-1; the SSE state is listed below, along with details of its interactions with the XSAVE feature set:

- Bytes 23:0 are used for x87 state (see Section 13.5.1).
- Bytes 27:24 are used for the MXCSR register. XRSTOR and XRSTORS generate general-protection faults (#GP) in response to attempts to set any of the reserved bits of the MXCSR register.[2]

1. The processor ensures that XCR0[0] is always 1.

- Bytes 31:28 are used for the MXCSR_MASK value. XRSTOR and XRSTORS ignore this field.
- Bytes 159:32 are used for x87 state.
- Bytes 287:160 are used for the registers XMM0–XMM7.
- Bytes 415:288 are used for the registers XMM8–XMM15. These fields are used only in 64-bit mode. Executions of XSAVE, XSAVEOPT, XSAVEC, and XSAVES outside 64-bit mode do not modify these bytes; executions of XRSTOR and XRSTORS outside 64-bit mode do not update XMM8–XMM15. See Section 13.13.

SSE state is XSAVE-managed but the SSE feature is not XSAVE-enabled. The XSAVE feature set can operate on SSE state only if the feature set is enabled (CR4.OSXSAVE = 1) and has been configured to manage SSE state (XCR0[1] = 1). Software can otherwise use SSE state even if the XSAVE feature set is not enabled or has not been configured to manage SSE state.

13.5.3 AVX State

The register state used by the Intel® Advanced Vector Extensions (AVX) comprises the MXCSR register and 16 256-bit vector registers called YMM0–YMM15. The low 128 bits of each register YMM*i* is identical to the SSE register XMM*i*. Thus, the new state register state added by AVX comprises the upper 128 bits of the registers YMM0–YMM15. These 16 128-bit values are denoted YMM0_H–YMM15_H and are collectively called **AVX state**.

As noted in Section 13.1, the XSAVE feature set manages AVX state as user state component 2. Thus, AVX state is located in the extended region of the XSAVE area (see Section 13.4.3).

As noted in Section 13.2, CPUID.(EAX=0DH,ECX=2):EBX enumerates the offset (in bytes, from the base of the XSAVE area) of the section of the extended region of the XSAVE area used for AVX state (when the standard format of the extended region is used). CPUID.(EAX=0DH,ECX=2):EAX enumerates the size (in bytes) required for AVX state.

The XSAVE feature set partitions YMM0_H–YMM15_H in a manner similar to that used for the XMM registers (see Section 13.5.2). Bytes 127:0 of the AVX-state section are used for YMM0_H–YMM7_H. Bytes 255:128 are used for YMM8_H–YMM15_H, but they are used only in 64-bit mode. Executions of XSAVE, XSAVEOPT, XSAVEC, and XSAVES outside 64-bit mode do not modify bytes 255:128; executions of XRSTOR and XRSTORS outside 64-bit mode do not update YMM8_H–YMM15_H. See Section 13.13. In general, bytes 16*i*+15:16*i* are used for YMM*i*_H (for $0 \le i \le 15$).

AVX state is XSAVE-managed and the AVX feature is XSAVE-enabled. The XSAVE feature set can operate on AVX state only if the feature set is enabled (CR4.OSXSAVE = 1) and has been configured to manage AVX state (XCR0[2] = 1). AVX instructions cannot be used unless the XSAVE feature set is enabled and has been configured to manage AVX state.

13.5.4 MPX State

The register state used by the Intel® Memory Protection Extensions (MPX) comprises the 4 128-bit bounds registers BND0–BND3 (**BNDREGS state**); and the 64-bit user-mode configuration register BNDCFGU and the 64-bit MPX status register BNDSTATUS (collectively, **BNDCSR state**). Together, these two user state components compose **MPX state**.

As noted in Section 13.1, the XSAVE feature set manages MPX state as state components 3–4. Thus, MPX state is located in the extended region of the XSAVE area (see Section 13.4.3). The following items detail how these state components are organized in this region:

- **BNDREGS state.**
  As noted in Section 13.2, CPUID.(EAX=0DH,ECX=3):EBX enumerates the offset (in bytes, from the base of the XSAVE area) of the section of the extended region of the XSAVE area used for BNDREGS state (when the standard format of the extended region is used). CPUID.(EAX=0DH,ECX=3):EAX enumerates the size (in bytes) required for BNDREGS state. The BNDREGS section is used for the 4 128-bit bound registers BND0–BND3, with bytes 16*i*+15:16*i* being used for BND*i*.

2. While MXCSR and MXCSR_MASK are part of SSE state, their treatment by the XSAVE feature set is not the same as that of the XMM registers. See Section 13.7 through Section 13.11 for details.

- **BNDCSR state.**
  As noted in Section 13.2, CPUID.(EAX=0DH,ECX=4):EBX enumerates the offset of the section of the extended region of the XSAVE area used for BNDCSR state (when the standard format of the extended region is used). CPUID.(EAX=0DH,ECX=4):EAX enumerates the size (in bytes) required for BNDCSR state. In the BNDSCR section, bytes 7:0 are used for BNDCFGU and bytes 15:8 are used for BNDSTATUS.

Both components of MPX state are XSAVE-managed and the MPX feature is XSAVE-enabled. The XSAVE feature set can operate on MPX state only if the feature set is enabled (CR4.OSXSAVE = 1) and has been configured to manage MPX state (XCR0[4:3] = 11b). MPX instructions cannot be used unless the XSAVE feature set is enabled and has been configured to manage MPX state.

13.5.5 AVX-512 State

The register state used by the Intel® Advanced Vector Extensions 512 (AVX-512) comprises the MXCSR register, the 8 64-bit opmask registers k0–k7, and 32 512-bit vector registers called ZMM0–ZMM31. For each *i*, 0 <= *i* <= 15, the low 256 bits of register ZMM*i* is identical to the AVX register YMM*i*. Thus, the new state register state added by AVX comprises the following user state components:

- The opmask registers, collectively called **opmask state**.
- The upper 256 bits of the registers ZMM0–ZMM15. These 16 256-bit values are denoted ZMM0_H–ZMM15_H and are collectively called **ZMM_Hi256 state**.
- The 16 512-bit registers ZMM16–ZMM31, collectively called **Hi16_ZMM state**.

Together, these three state components compose **AVX-512 state**.

As noted in Section 13.1, the XSAVE feature set manages AVX-512 state as state components 5–7. Thus, AVX-512 state is located in the extended region of the XSAVE area (see Section 13.4.3). The following items detail how these state components are organized in this region:

- **Opmask state.**
  As noted in Section 13.2, CPUID.(EAX=0DH,ECX=5):EBX enumerates the offset (in bytes, from the base of the XSAVE area) of the section of the extended region of the XSAVE area used for opmask state (when the standard format of the extended region is used). CPUID.(EAX=0DH,ECX=5):EAX enumerates the size (in bytes) required for opmask state. The opmask section is used for the 8 64-bit bound registers k0–k7, with bytes 8*i*+7:8*i* being used for k*i*.
- **ZMM_Hi256 state.**
  As noted in Section 13.2, CPUID.(EAX=0DH,ECX=6):EBX enumerates the offset of the section of the extended region of the XSAVE area used for ZMM_Hi256 state (when the standard format of the extended region is used). CPUID.(EAX=0DH,ECX=6):EAX enumerates the size (in bytes) required for ZMM_Hi256 state.

  The XSAVE feature set partitions ZMM0_H–ZMM15_H in a manner similar to that used for the XMM registers (see Section 13.5.2). Bytes 255:0 of the ZMM_Hi256-state section are used for ZMM0_H–ZMM7_H. Bytes 511:256 are used for ZMM8_H–ZMM15_H, but they are used only in 64-bit mode. Executions of XSAVE, XSAVEOPT, XSAVEC, and XSAVES outside 64-bit mode do not modify bytes 511:256; executions of XRSTOR and XRSTORS outside 64-bit mode do not update ZMM8_H–ZMM15_H. See Section 13.13. In general, bytes 32*i*+31:32*i* are used for ZMM*i*_H (for 0 ≤ *i* ≤ 15).
- **Hi16_ZMM state.**
  As noted in Section 13.2, CPUID.(EAX=0DH,ECX=7):EBX enumerates the offset of the section of the extended region of the XSAVE area used for Hi16_ZMM state (when the standard format of the extended region is used). CPUID.(EAX=0DH,ECX=7):EAX enumerates the size (in bytes) required for Hi16_ZMM state.

  The XSAVE feature set accesses Hi16_ZMM state only in 64-bit mode. Executions of XSAVE, XSAVEOPT, XSAVEC, and XSAVES outside 64-bit mode do not modify the Hi16_ZMM section; executions of XRSTOR and XRSTORS outside 64-bit mode do not update ZMM16–ZMM31. See Section 13.13. In general, bytes 64(*i*-16)+63:64(*i*-16) are used for ZMM*i* (for 16 ≤ *i* ≤ 31).

All three components of AVX-512 state are XSAVE-managed and the AVX-512 feature is XSAVE-enabled. The XSAVE feature set can operate on AVX-512 state only if the feature set is enabled (CR4.OSXSAVE = 1) and has been configured to manage AVX-512 state (XCR0[7:5] = 111b). AVX-512 instructions cannot be used unless the XSAVE feature set is enabled and has been configured to manage AVX-512 state.

13.5.6 PT State

The register state used by Intel Processor Trace (**PT state**) comprises the following 9 MSRs: IA32_RTIT_CTL, IA32_RTIT_OUTPUT_BASE, IA32_RTIT_OUTPUT_MASK_PTRS, IA32_RTIT_STATUS, IA32_RTIT_CR3_MATCH, IA32_RTIT_ADDR0_A, IA32_RTIT_ADDR0_B, IA32_RTIT_ADDR1_A, and IA32_RTIT_ADDR1_B.[1]

As noted in Section 13.1, the XSAVE feature set manages PT state as supervisor state component 8. Thus, PT state is located in the extended region of the XSAVE area (see Section 13.4.3). As noted in Section 13.2, CPUID.(EAX=0DH,ECX=8):EAX enumerates the size (in bytes) required for PT state. The MSRs are each allocated 8 bytes in the state component in the order given above. Thus, IA32_RTIT_CTL is at byte offset 0, IA32_RTIT_OUTPUT_BASE at byte offset 8, etc. Any locations in the state component at or beyond byte offset 72 are reserved.

PT state is XSAVE-managed but Intel Processor Trace is not XSAVE-enabled. The XSAVE feature set can operate on PT state only if the feature set is enabled (CR4.OSXSAVE = 1) and has been configured to manage PT state (IA32_XSS[8] = 1). Software can otherwise use Intel Processor Trace and access its MSRs (using RDMSR and WRMSR) even if the XSAVE feature set is not enabled or has not been configured to manage PT state.

The following items describe special treatment of PT state by the XSAVES and XRSTORS instructions:

- If XSAVES saves PT state, the instruction clears IA32_RTIT_CTL.TraceEn (bit 0) after saving the value of the IA32_RTIT_CTL MSR and before saving any other PT state. If XSAVES causes a fault or a VM exit, it restores IA32_RTIT_CTL.TraceEn to its original value.
- If XSAVES saves PT state, the instruction saves zeroes in the reserved portions of the state component.
- If XRSTORS would restore (or initialize) PT state and IA32_RTIT_CTL.TraceEn = 1, the instruction causes a general-protection exception (#GP) before modifying PT state.
- If XRSTORS causes an exception or a VM exit, it does so before any modification to IA32_RTIT_CTL.TraceEn (even if it has loaded other PT state).

13.5.7 PKRU State

The register state used by the protection-key feature (**PKRU state**) is the 32-bit PKRU register. As noted in Section 13.1, the XSAVE feature set manages PKRU state as user state component 9. Thus, PKRU state is located in the extended region of the XSAVE area (see Section 13.4.3).

As noted in Section 13.2, CPUID.(EAX=0DH,ECX=9):EBX enumerates the offset (in bytes, from the base of the XSAVE area) of the section of the extended region of the XSAVE area used for PKRU state (when the standard format of the extended region is used). CPUID.(EAX=0DH,ECX=9):EAX enumerates the size (in bytes) required for PKRU state. The XSAVE feature set uses bytes 3:0 of the PK-state section for the PKRU register.

PKRU state is XSAVE-managed but the protection-key feature is not XSAVE-enabled. The XSAVE feature set can operate on PKRU state only if the feature set is enabled (CR4.OSXSAVE = 1) and has been configured to manage PKRU state (XCR0[9] = 1). Software can otherwise use protection keys and access PKRU state even if the XSAVE feature set is not enabled or has not been configured to manage PKRU state.

The value of the PKRU register determines the access rights for user-mode linear addresses. (See Section 4.6, "Access Rights," of *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*.) The access rights that pertain to an execution of the XRSTOR and XRSTORS instructions are determined by the value of the register before the execution and not by any value that the execution might load into the PKRU register.

13.6 PROCESSOR TRACKING OF XSAVE-MANAGED STATE

The XSAVEOPT, XSAVEC, and XSAVES instructions use two optimization to reduce the amount of data that they write to memory. They avoid writing data for any state component known to be in its initial configuration (the **init optimization**). In addition, if either XSAVEOPT or XSAVES is using the same XSAVE area as that used by the most

1. These MSRs might not be supported by every processor that supports Intel Processor Trace. Software can use the CPUID instruction to discover which are supported; see Section 35.3.1, "Detection of Intel Processor Trace and Capability Enumeration," of *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3C*.

recent execution of XRSTOR or XRSTORS, it may avoid writing data for any state component whose configuration is known not to have been modified since then (the **modified optimization**). (XSAVE does not use these optimizations, and XSAVEC does not use the modified optimization.) The operation of XSAVEOPT, XSAVEC, and XSAVES are described in more detail in Section 13.9 through Section 13.11.

A processor can support the init and modified optimizations with special hardware that tracks the state components that might benefit from those optimizations. Other implementations might not include such hardware; such a processor would always consider each such state component as not in its initial configuration and as modified since the last execution of XRSTOR or XRSTORS.

The following notation describes the state of the init and modified optimizations:

- XINUSE denotes the state-component bitmap corresponding to the init optimization. If XINUSE[*i*] = 0, state component *i* is known to be in its initial configuration; otherwise XINUSE[*i*] = 1. It is possible for XINUSE[*i*] to be 1 even when state component *i* is in its initial configuration. On a processor that does not support the init optimization, XINUSE[*i*] is always 1 for every value of *i*.

  Executing XGETBV with ECX = 1 returns in EDX:EAX the logical-AND of XCR0 and the current value of the XINUSE state-component bitmap. Such an execution of XGETBV always sets EAX[1] to 1 if XCR0[1] = 1 and MXCSR does not have its RESET value of 1F80H. Section 13.2 explains how software can determine whether a processor supports this use of XGETBV.
- XMODIFIED denotes the state-component bitmap corresponding to the modified optimization. If XMODIFIED[*i*] = 0, state component *i* is known not to have been modified since the most recent execution of XRSTOR or XRSTORS; otherwise XMODIFIED[*i*] = 1. It is possible for XMODIFIED[*i*] to be 1 even when state component *i* has not been modified since the most recent execution of XRSTOR or XRSTORS. On a processor that does not support the modified optimization, XMODIFIED[*i*] is always 1 for every value of *i*.

A processor that implements the modified optimization saves information about the most recent execution of XRSTOR or XRSTORS in a quantity called **XRSTOR_INFO**, a 4-tuple containing the following: (1) the CPL; (2) whether the logical processor was in VMX non-root operation; (3) the linear address of the XSAVE area; and (4) the XCOMP_BV field in the XSAVE area. An execution of XSAVEOPT or XSAVES uses the modified optimization only if that execution corresponds to XRSTOR_INFO on these four parameters.

This mechanism implies that, depending on details of the operating system, the processor might determine that an execution of XSAVEOPT by one user application corresponds to an earlier execution of XRSTOR by a different application. For this reason, Intel recommends the application software not use the XSAVEOPT instruction.

The following items specify the initial configuration each state component (for the purposes of defining the XINUSE bitmap):

- **x87 state.** x87 state is in its initial configuration if the following all hold: FCW is 037FH; FSW is 0000H; FTW is FFFFH; FCS and FDS are each 0000H; FIP and FDP are each 00000000_00000000H; each of ST0–ST7 is 0000_00000000_00000000H.
- **SSE state.** In 64-bit mode, SSE state is in its initial configuration if each of XMM0–XMM15 is 0. Outside 64-bit mode, SSE state is in its initial configuration if each of XMM0–XMM7 is 0. XINUSE[1] pertains only to the state of the XMM registers and not to MXCSR. An execution of XRSTOR or XRSTORS outside 64-bit mode does not update XMM8–XMM15. (See Section 13.13.)
- **AVX state.** In 64-bit mode, AVX state is in its initial configuration if each of YMM0_H–YMM15_H is 0. Outside 64-bit mode, AVX state is in its initial configuration if each of YMM0_H–YMM7_H is 0. An execution of XRSTOR or XRSTORS outside 64-bit mode does not update YMM8_H–YMM15_H. (See Section 13.13.)
- **BNDREGS state.** BNDREGS state is in its initial configuration if the value of each of BND0–BND3 is 0.
- **BNDCSR state.** BNDCSR state is in its initial configuration if BNDCFGU and BNDCSR each has value 0.
- **Opmask state.** Opmask state is in its initial configuration if each of the opmask registers k0–k7 is 0.
- **ZMM_Hi256 state.** In 64-bit mode, ZMM_Hi256 state is in its initial configuration if each of ZMM0_H–ZMM15_H is 0. Outside 64-bit mode, ZMM_Hi256 state is in its initial configuration if each of ZMM0_H–ZMM7_H is 0. An execution of XRSTOR or XRSTORS outside 64-bit mode does not update ZMM8_H–ZMM15_H. (See Section 13.13.)
- **Hi16_ZMM state.** In 64-bit mode, Hi16_ZMM state is in its initial configuration if each of ZMM16–ZMM31 is 0. Outside 64-bit mode, Hi16_ZMM state is always in its initial configuration. An execution of XRSTOR or XRSTORS outside 64-bit mode does not update ZMM31–ZMM31. (See Section 13.13.)

- **PT state.** PT state is in its initial configuration if each of the 9 MSRs is 0.
- **PKRU state.** PKRU state is in its initial configuration if the value of the PKRU is 0.

13.7 OPERATION OF XSAVE

The XSAVE instruction takes a single memory operand, which is an XSAVE area. In addition, the register pair EDX:EAX is an implicit operand used as a state-component bitmap (see Section 13.1) called the **instruction mask**. The logical-AND of XCR0 and the instruction mask is the **requested-feature bitmap** (**RFBM**) of the user state components to be saved.

The following conditions cause execution of the XSAVE instruction to generate a fault:

- If the XSAVE feature set is not enabled (CR4.OSXSAVE = 0), an invalid-opcode exception (#UD) occurs.
- If CR0.TS[bit 3] is 1, a device-not-available exception (#NM) occurs.
- If the address of the XSAVE area is not 64-byte aligned, a general-protection exception (#GP) occurs.[1]

If none of these conditions cause a fault, execution of XSAVE reads the XSTATE_BV field of the XSAVE header (see Section 13.4.2) and writes it back to memory, setting XSTATE_BV[i] ($0 \le i \le 63$) as follows:

- If RFBM[i] = 0, XSTATE_BV[i] is not changed.
- If RFBM[i] = 1, XSTATE_BV[i] is set to the value of XINUSE[i]. Section 13.6 defines XINUSE to describe the processor init optimization and specifies the initial configuration of each state component. The nature of that optimization implies the following:
  - If state component i is in its initial configuration, XINUSE[i] may be either 0 or 1, and XSTATE_BV[i] may be written with either 0 or 1.

    XINUSE[1] pertains only to the state of the XMM registers and not to MXCSR. Thus, XSTATE_BV[1] may be written with 0 even if MXCSR does not have its RESET value of 1F80H.
  - If state component i is not in its initial configuration, XINUSE[i] = 1 and XSTATE_BV[i] is written with 1.

  (As explained in Section 13.6, the initial configurations of some state components may depend on whether the processor is in 64-bit mode.)

The XSAVE instruction does not write any part of the XSAVE header other than the XSTATE_BV field; in particular, it does **not** write to the XCOMP_BV field.

Execution of XSAVE saves into the XSAVE area those state components corresponding to bits that are set in RFBM. State components 0 and 1 are located in the legacy region of the XSAVE area (see Section 13.4.1). Each state component i, $2 \le i \le 62$, is located in the extended region; the XSAVE instruction always uses the standard format for the extended region (see Section 13.4.3).

The MXCSR register and MXCSR_MASK are part of SSE state (see Section 13.5.2) and are thus associated with RFBM[1]. However, the XSAVE instruction also saves these values when RFBM[2] = 1 (even if RFBM[1] = 0).

See Section 13.5 for specifics for each state component and for details regarding mode-specific operation and operation determined by instruction prefixes. See Section 13.13 for details regarding faults caused by memory accesses.

13.8 OPERATION OF XRSTOR

The XRSTOR instruction takes a single memory operand, which is an XSAVE area. In addition, the register pair EDX:EAX is an implicit operand used as a state-component bitmap (see Section 13.1) called the **instruction mask**. The logical-AND of XCR0 and the instruction mask is the **requested-feature bitmap** (**RFBM**) of the user state components to be restored.

The following conditions cause execution of the XRSTOR instruction to generate a fault:

- If the XSAVE feature set is not enabled (CR4.OSXSAVE = 0), an invalid-opcode exception (#UD) occurs.

1. If CR0.AM = 1, CPL = 3, and EFLAGS.AC =1, an alignment-check exception (#AC) may occur instead of #GP.

- If CR0.TS[bit 3] is 1, a device-not-available exception (#NM) occurs.
- If the address of the XSAVE area is not 64-byte aligned, a general-protection exception (#GP) occurs.[1]

After checking for these faults, the XRSTOR instruction reads the XCOMP_BV field in the XSAVE area's XSAVE header (see Section 13.4.2). If XCOMP_BV[63] = 0, the **standard form of XRSTOR** is executed (see Section 13.8.1); otherwise, the **compacted form of XRSTOR** is executed (see Section 13.8.2).[2]

See Section 13.2 for details of how to determine whether the compacted form of XRSTOR is supported.

13.8.1 Standard Form of XRSTOR

The standard from of XRSTOR performs additional fault checking. Either of the following conditions causes a general-protection exception (#GP):

- The XSTATE_BV field of the XSAVE header sets a bit that is not set in XCR0.
- Bytes 23:8 of the XSAVE header are not all 0 (this implies that all bits in XCOMP_BV are 0).[3]

If none of these conditions cause a fault, the processor updates each state component *i* for which RFBM[*i*] = 1. XRSTOR updates state component *i* based on the value of bit *i* in the XSTATE_BV field of the XSAVE header:

- If XSTATE_BV[*i*] = 0, the state component is set to its initial configuration. Section 13.6 specifies the initial configuration of each state component.

  The initial configuration of state component 1 pertains only to the XMM registers and not to MXCSR. See below for the treatment of MXCSR
- If XSTATE_BV[*i*] = 1, the state component is loaded with data from the XSAVE area. See Section 13.5 for specifics for each state component and for details regarding mode-specific operation and operation determined by instruction prefixes. See Section 13.13 for details regarding faults caused by memory accesses.

  State components 0 and 1 are located in the legacy region of the XSAVE area (see Section 13.4.1). Each state component *i*, $2 \le i \le 62$, is located in the extended region; the standard form of XRSTOR uses the standard format for the extended region (see Section 13.4.3).

The MXCSR register is part of state component 1, SSE state (see Section 13.5.2). However, the standard form of XRSTOR loads the MXCSR register from memory whenever the RFBM[1] (SSE) or RFBM[2] (AVX) is set, regardless of the values of XSTATE_BV[1] and XSTATE_BV[2]. The standard form of XRSTOR causes a general-protection exception (#GP) if it would load MXCSR with an illegal value.

13.8.2 Compacted Form of XRSTOR

The compacted from of XRSTOR performs additional fault checking. Any of the following conditions causes a #GP:

- The XCOMP_BV field of the XSAVE header sets a bit in the range 62:0 that is not set in XCR0.
- The XSTATE_BV field of the XSAVE header sets a bit (including bit 63) that is not set in XCOMP_BV.
- Bytes 63:16 of the XSAVE header are not all 0.

If none of these conditions cause a fault, the processor updates each state component *i* for which RFBM[*i*] = 1. XRSTOR updates state component *i* based on the value of bit *i* in the XSTATE_BV field of the XSAVE header:

- If XSTATE_BV[*i*] = 0, the state component is set to its initial configuration. Section 13.6 specifies the initial configuration of each state component.

---

1. If CR0.AM = 1, CPL = 3, and EFLAGS.AC =1, an alignment-check exception (#AC) may occur instead of #GP.
2. If the processor does not support the compacted form of XRSTOR, it may execute the standard form of XRSTOR without first reading the XCOMP_BV field. A processor supports the compacted form of XRSTOR only if it enumerates CPUID.(EAX=0DH,ECX=1):EAX[1] as 1.
3. Bytes 63:24 of the XSAVE header are also reserved. Software should ensure that bytes 63:16 of the XSAVE header are all 0 in any XSAVE area. (Bytes 15:8 should also be 0 if the XSAVE area is to be used on a processor that does not support the compaction extensions to the XSAVE feature set.)

If XSTATE_BV[1] = 0, the compacted form XRSTOR initializes MXCSR to 1F80H. (This differs from the standard from of XRSTOR, which loads MXCSR from the XSAVE area whenever either RFBM[1] or RFBM[2] is set.)

State component *i* is set to its initial configuration as indicated above if RFBM[*i*] = 1 and XSTATE_BV[*i*] = 0 — **even if XCOMP_BV[*i*] = 0**. This is true for all values of *i*, including 0 (x87 state) and 1 (SSE state).

- If XSTATE_BV[*i*] = 1, the state component is loaded with data from the XSAVE area.[1] See Section 13.5 for specifics for each state component and for details regarding mode-specific operation and operation determined by instruction prefixes. See Section 13.13 for details regarding faults caused by memory accesses.

  State components 0 and 1 are located in the legacy region of the XSAVE area (see Section 13.4.1). Each state component *i*, $2 \le i \le 62$, is located in the extended region; the compacted form of the XRSTOR instruction uses the compacted format for the extended region (see Section 13.4.3).

The MXCSR register is part of SSE state (see Section 13.5.2) and is thus loaded from memory if RFBM[1] = XSTATE_BV[*i*] = 1. The compacted form of XRSTOR does not consider RFBM[2] (AVX) when determining whether to update MXCSR. (This is a difference from the standard form of XRSTOR.) The compacted form of XRSTOR causes a general-protection exception (#GP) if it would load MXCSR with an illegal value.

13.8.3 XRSTOR and the Init and Modified Optimizations

Execution of the XRSTOR instruction causes the processor to update is tracking for the init and modified optimizations (see Section 13.6). The following items provide details:

- The processor updates its tracking for the init optimization as follows:
  - If RFBM[*i*] = 0, XINUSE[*i*] is not changed.
  - If RFBM[*i*] = 1 and XSTATE_BV[*i*] = 0, state component *i* may be tracked as init; XINUSE[*i*] may be set to 0 or 1. (As noted in Section 13.6, a processor need not implement the init optimization for state component *i*; a processor that does not do so implicitly maintains XINUSE[*i*] = 1 at all times.)
  - If RFBM[*i*] = 1 and XSTATE_BV[*i*] = 1, state component *i* is tracked as not init; XINUSE[*i*] is set to 1.
- The processor updates its tracking for the modified optimization and records information about the XRSTOR execution for future interaction with the XSAVEOPT and XSAVES instructions (see Section 13.9 and Section 13.11) as follows:
  - If RFBM[*i*] = 0, state component *i* is tracked as modified; XMODIFIED[*i*] is set to 1.
  - If RFBM[*i*] = 1, state component *i* may be tracked as unmodified; XMODIFIED[*i*] may be set to 0 or 1. (As noted in Section 13.6, a processor need not implement the modified optimization for state component *i*; a processor that does not do so implicitly maintains XMODIFIED[*i*] = 1 at all times.)
  - XRSTOR_INFO is set to the 4-tuple $\langle w,x,y,z \rangle$, where w is the CPL (0); x is 1 if the logical processor is in VMX non-root operation and 0 otherwise; *y* is the linear address of the XSAVE area; and *z* is XCOMP_BV. In particular, the standard form of XRSTOR always sets *z* to all zeroes, while the compacted form of XRSTORS never does so (because it sets at least bit 63 to 1).

13.9 OPERATION OF XSAVEOPT

The operation of XSAVEOPT is similar to that of XSAVE. Unlike XSAVE, XSAVEOPT uses the init optimization (by which it may omit saving state components that are in their initial configuration) and the modified optimization (by which it may omit saving state components that have not been modified since the last execution of XRSTOR); see Section 13.6. See Section 13.2 for details of how to determine whether XSAVEOPT is supported.

The XSAVEOPT instruction takes a single memory operand, which is an XSAVE area. In addition, the register pair EDX:EAX is an implicit operand used as a state-component bitmap (see Section 13.1) called the **instruction mask**. The logical (bitwise) AND of XCR0 and the instruction mask is the **requested-feature bitmap** (**RFBM**) of the user state components to be saved.

1. Earlier fault checking ensured that, if the instruction has reached this point in execution and XSTATE_BV[*i*] is 1, then XCOMP_BV[*i*] is also 1.

The following conditions cause execution of the XSAVEOPT instruction to generate a fault:

- If the XSAVE feature set is not enabled (CR4.OSXSAVE = 0), an invalid-opcode exception (#UD) occurs.
- If CR0.TS[bit 3] is 1, a device-not-available exception (#NM) occurs.
- If the address of the XSAVE area is not 64-byte aligned, a general-protection exception (#GP) occurs.[1]

If none of these conditions cause a fault, execution of XSAVEOPT reads the XSTATE_BV field of the XSAVE header (see Section 13.4.2) and writes it back to memory, setting XSTATE_BV[*i*] ($0 \le i \le 63$) as follows:

- If RFBM[*i*] = 0, XSTATE_BV[*i*] is not changed.
- If RFBM[*i*] = 1, XSTATE_BV[*i*] is set to the value of XINUSE[*i*]. Section 13.6 defines XINUSE to describe the processor init optimization and specifies the initial configuration of each state component. The nature of that optimization implies the following:
  - If the state component is in its initial configuration, XINUSE[*i*] may be either 0 or 1, and XSTATE_BV[*i*] may be written with either 0 or 1.

    XINUSE[1] pertains only to the state of the XMM registers and not to MXCSR. Thus, XSTATE_BV[1] may be written with 0 even if MXCSR does not have its RESET value of 1F80H.
  - If the state component is not in its initial configuration, XSTATE_BV[*i*] is written with 1.

  (As explained in Section 13.6, the initial configurations of some state components may depend on whether the processor is in 64-bit mode.)

The XSAVEOPT instruction does not write any part of the XSAVE header other than the XSTATE_BV field; in particular, it does not write to the XCOMP_BV field.

Execution of XSAVEOPT saves into the XSAVE area those state components corresponding to bits that are set in RFBM (subject to the optimizations described below). State components 0 and 1 are located in the legacy region of the XSAVE area (see Section 13.4.1). Each state component *i*, $2 \le i \le 62$, is located in the extended region; the XSAVEOPT instruction always uses the standard format for the extended region (see Section 13.4.3).

See Section 13.5 for specifics for each state component and for details regarding mode-specific operation and operation determined by instruction prefixes. See Section 13.13 for details regarding faults caused by memory accesses.

Execution of XSAVEOPT performs two optimizations that reduce the amount of data written to memory:

- **Init optimization.**
  If XINUSE[*i*] = 0, state component *i* is not saved to the XSAVE area (even if RFBM[*i*] = 1). (See below for exceptions made for MXCSR.)
- **Modified optimization.**
  Each execution of XRSTOR and XRSTORS establishes XRSTOR_INFO as a 4-tuple $\langle w,x,y,z \rangle$ (see Section 13.8.3 and Section 13.12). Execution of XSAVEOPT uses the modified optimization only if the following all hold for the current value of XRSTOR_INFO:
  - *w* = CPL;
  - *x* = 1 if and only if the logical processor is in VMX non-root operation;
  - *y* is the linear address of the XSAVE area being used by XSAVEOPT; and
  - *z* is 00000000_00000000H. (This last item implies that XSAVEOPT does not use the modified optimization if the last execution of XRSTOR used the compacted form, or if an execution of XRSTORS followed the last execution of XRSTOR.)

  If XSAVEOPT uses the modified optimization and XMODIFIED[*i*] = 0 (see Section 13.6), state component *i* is not saved to the XSAVE area.

  (In practice, the benefit of the modified optimization for state component *i* depends on how the processor is tracking state component *i*; see Section 13.6. Limitations on the tracking ability may result in state component *i* being saved even though is in the same configuration that was loaded by the previous execution of XRSTOR.)

1. If CR0.AM = 1, CPL = 3, and EFLAGS.AC =1, an alignment-check exception (#AC) may occur instead of #GP.

Depending on details of the operating system, an execution of XSAVEOPT by a user application might use the modified optimization when the most recent execution of XRSTOR was by a different application. Because of this, Intel recommends the application software not use the XSAVEOPT instruction.

The MXCSR register and MXCSR_MASK are part of SSE state (see Section 13.5.2) and are thus associated with bit 1 of RFBM. However, the XSAVEOPT instruction also saves these values when RFBM[2] = 1 (even if RFBM[1] = 0). The init and modified optimizations do not apply to the MXCSR register and MXCSR_MASK.

13.10 OPERATION OF XSAVEC

The operation of XSAVEC is similar to that of XSAVE. Two main differences are (1) XSAVEC uses the compacted format for the extended region of the XSAVE area; and (2) XSAVEC uses the init optimization (see Section 13.6). Unlike XSAVEOPT, XSAVEC does not use the modified optimization. See Section 13.2 for details of how to determine whether XSAVEC is supported.

The XSAVEC instruction takes a single memory operand, which is an XSAVE area. In addition, the register pair EDX:EAX is an implicit operand used as a state-component bitmap (see Section 13.1) called the **instruction mask**. The logical (bitwise) AND of XCR0 and the instruction mask is the **requested-feature bitmap** (**RFBM**) of the user state components to be saved.

The following conditions cause execution of the XSAVEC instruction to generate a fault:

- If the XSAVE feature set is not enabled (CR4.OSXSAVE = 0), an invalid-opcode exception (#UD) occurs.
- If CR0.TS[bit 3] is 1, a device-not-available exception (#NM) occurs.
- If the address of the XSAVE area is not 64-byte aligned, a general-protection exception (#GP) occurs.[1]

If none of these conditions cause a fault, execution of XSAVEC writes the XSTATE_BV field of the XSAVE header (see Section 13.4.2), setting XSTATE_BV[i] ($0 \le i \le 63$) as follows:[2]

- If RFBM[i] = 0, XSTATE_BV[i] is written as 0.
- If RFBM[i] = 1, XSTATE_BV[i] is set to the value of XINUSE[i] (see below for an exception made for XSTATE_BV[1]). Section 13.6 defines XINUSE to describe the processor init optimization and specifies the initial configuration of each state component. The nature of that optimization implies the following:
  - If state component i is in its initial configuration, XSTATE_BV[i] may be written with either 0 or 1.
  - If state component i is not in its initial configuration, XSTATE_BV[i] is written with 1.

  XINUSE[1] pertains only to the state of the XMM registers and not to MXCSR. However, if RFBM[1] = 1 and MXCSR does not have the value 1F80H, XSAVEC writes XSTATE_BV[1] as 1 even if XINUSE[1] = 0.

  (As explained in Section 13.6, the initial configurations of some state components may depend on whether the processor is in 64-bit mode.)

The XSAVEC instructions sets bit 63 of the XCOMP_BV field of the XSAVE header while writing RFBM[62:0] to XCOMP_BV[62:0]. The XSAVEC instruction does not write any part of the XSAVE header other than the XSTATE_BV and XCOMP_BV fields.

Execution of XSAVEC saves into the XSAVE area those state components corresponding to bits that are set in RFBM (subject to the init optimization described below). State components 0 and 1 are located in the legacy region of the XSAVE area (see Section 13.4.1). Each state component i, $2 \le i \le 62$, is located in the extended region; the XSAVEC instruction always uses the compacted format for the extended region (see Section 13.4.3).

See Section 13.5 for specifics for each state component and for details regarding mode-specific operation and operation determined by instruction prefixes. See Section 13.13 for details regarding faults caused by memory accesses.

Execution of XSAVEC performs the init optimization to reduce the amount of data written to memory. If XINUSE[i] = 0, state component i is not saved to the XSAVE area (even if RFBM[i] = 1). However, if RFBM[1] = 1 and MXCSR does not have the value 1F80H, XSAVEC writes saves all of state component 1 (SSE — including the

1. If CR0.AM = 1, CPL = 3, and EFLAGS.AC =1, an alignment-check exception (#AC) may occur instead of #GP.
2. Unlike the XSAVE and XSAVEOPT instructions, the XSAVEC instruction does **not** read the XSTATE_BV field of the XSAVE header.

XMM registers) even if XINUSE[1] = 0. Unlike the XSAVE instruction, RFBM[2] does not determine whether XSAVEC saves MXCSR and MXCSR_MASK.

13.11 OPERATION OF XSAVES

The operation of XSAVES is similar to that of XSAVEC. The main differences are (1) XSAVES can be executed only if CPL = 0; (2) XSAVES can operate on the state components whose bits are set in XCR0 | IA32_XSS and can thus operate on supervisor state components; and (3) XSAVES uses the modified optimization (see Section 13.6). See Section 13.2 for details of how to determine whether XSAVES is supported.

The XSAVES instruction takes a single memory operand, which is an XSAVE area. In addition, the register pair EDX:EAX is an implicit operand used as a state-component bitmap (see Section 13.1) called the **instruction mask**. EDX:EAX & (XCR0 | IA32_XSS) (the logical AND the instruction mask with the logical OR of XCR0 and IA32_XSS) is the **requested-feature bitmap** (**RFBM**) of the state components to be saved.

The following conditions cause execution of the XSAVES instruction to generate a fault:

- If the XSAVE feature set is not enabled (CR4.OSXSAVE = 0), an invalid-opcode exception (#UD) occurs.
- If CR0.TS[bit 3] is 1, a device-not-available exception (#NM) occurs.
- If CPL > 0 or if the address of the XSAVE area is not 64-byte aligned, a general-protection exception (#GP) occurs.[1]

If none of these conditions cause a fault, execution of XSAVES writes the XSTATE_BV field of the XSAVE header (see Section 13.4.2), setting XSTATE_BV[i] ($0 \le i \le 63$) as follows:

- If RFBM[i] = 0, XSTATE_BV[i] is written as 0.
- If RFBM[i] = 1, XSTATE_BV[i] is set to the value of XINUSE[i] (see below for an exception made for XSTATE_BV[1]). Section 13.6 defines XINUSE to describe the processor init optimization and specifies the initial configuration of each state component. The nature of that optimization implies the following:
  - If state component i is in its initial configuration, XSTATE_BV[i] may be written with either 0 or 1.
  - If state component i is not in its initial configuration, XSTATE_BV[i] is written with 1.

  XINUSE[1] pertains only to the state of the XMM registers and not to MXCSR. However, if RFBM[1] = 1 and MXCSR does not have the value 1F80H, XSAVES writes XSTATE_BV[1] as 1 even if XINUSE[1] = 0.

  (As explained in Section 13.6, the initial configurations of some state components may depend on whether the processor is in 64-bit mode.)

The XSAVES instructions sets bit 63 of the XCOMP_BV field of the XSAVE header while writing RFBM[62:0] to XCOMP_BV[62:0]. The XSAVES instruction does not write any part of the XSAVE header other than the XSTATE_BV and XCOMP_BV fields.

Execution of XSAVES saves into the XSAVE area those state components corresponding to bits that are set in RFBM (subject to the optimizations described below). State components 0 and 1 are located in the legacy region of the XSAVE area (see Section 13.4.1). Each state component i, $2 \le i \le 62$, is located in the extended region; the XSAVES instruction always uses the compacted format for the extended region (see Section 13.4.3).

See Section 13.5 for specifics for each state component and for details regarding mode-specific operation and operation determined by instruction prefixes; in particular, see Section 13.5.6 for some special treatment of PT state by XSAVES. See Section 13.13 for details regarding faults caused by memory accesses.

Execution of XSAVES performs the init optimization to reduce the amount of data written to memory. If XINUSE[i] = 0, state component i is not saved to the XSAVE area (even if RFBM[i] = 1). However, if RFBM[1] = 1 and MXCSR does not have the value 1F80H, XSAVES writes saves all of state component 1 (SSE — including the XMM registers) even if XINUSE[1] = 0.

Like XSAVEOPT, XSAVES may perform the modified optimization. Each execution of XRSTOR and XRSTORS establishes XRSTOR_INFO as a 4-tuple $\langle w,x,y,z \rangle$ (see Section 13.8.3 and Section 13.12). Execution of XSAVES uses the modified optimization only if the following all hold:

1. If CR0.AM = 1, CPL = 3, and EFLAGS.AC =1, an alignment-check exception (#AC) may occur instead of #GP.

- *w* = CPL;
- *x* = 1 if and only if the logical processor is in VMX non-root operation;
- *y* is the linear address of the XSAVE area being used by XSAVEOPT; and
- *z*[63] is 1 and *z*[62:0] = RFBM[62:0]. (This last item implies that XSAVES does not use the modified optimization if the last execution of XRSTOR used the standard form and followed the last execution of XRSTORS.)

If XSAVES uses the modified optimization and XMODIFIED[*i*] = 0 (see Section 13.6), state component *i* is not saved to the XSAVE area.

13.12 OPERATION OF XRSTORS

The operation of XRSTORS is similar to that of XRSTOR. Three main differences are (1) XRSTORS can be executed only if CPL = 0; (2) XRSTORS can operate on the state components whose bits are set in XCR0 | IA32_XSS and can thus operate on supervisor state components; and (3) XRSTORS has only a compacted form (no standard form; see Section 13.8). See Section 13.2 for details of how to determine whether XRSTORS is supported.

The XRSTORS instruction takes a single memory operand, which is an XSAVE area. In addition, the register pair EDX:EAX is an implicit operand used as a state-component bitmap (see Section 13.1) called the **instruction mask**. EDX:EAX & (XCR0 | IA32_XSS) (the logical AND the instruction mask with the logical OR of XCR0 and IA32_XSS) is the **requested-feature bitmap** (**RFBM**) of the state components to be restored.

The following conditions cause execution of the XRSTOR instruction to generate a fault:

- If the XSAVE feature set is not enabled (CR4.OSXSAVE = 0), an invalid-opcode exception (#UD) occurs.
- If CR0.TS[bit 3] is 1, a device-not-available exception (#NM) occurs.
- If CPL > 0 or if the address of the XSAVE area is not 64-byte aligned, a general-protection exception (#GP) occurs.[1]

After checking for these faults, the XRSTORS instruction reads the first 64 bytes of the XSAVE header, including the XSTATE_BV and XCOMP_BV fields (see Section 13.4.2). A #GP occurs if any of the following conditions hold for the values read:

- XCOMP_BV[63] = 0.
- XCOMP_BV sets a bit in the range 62:0 that is not set in XCR0 | IA32_XSS.
- XSTATE_BV sets a bit (including bit 63) that is not set in XCOMP_BV.
- Bytes 63:16 of the XSAVE header are not all 0.

If none of these conditions cause a fault, the processor updates each state component *i* for which RFBM[*i*] = 1. XRSTORS updates state component *i* based on the value of bit *i* in the XSTATE_BV field of the XSAVE header:

- If XSTATE_BV[*i*] = 0, the state component is set to its initial configuration. Section 13.6 specifies the initial configuration of each state component. If XSTATE_BV[1] = 0, XRSTORS initializes MXCSR to 1F80H.

  State component *i* is set to its initial configuration as indicated above if RFBM[*i*] = 1 and XSTATE_BV[*i*] = 0 — **even if XCOMP_BV[*i*] = 0**. This is true for all values of *i*, including 0 (x87 state) and 1 (SSE state).
- If XSTATE_BV[*i*] = 1, the state component is loaded with data from the XSAVE area.[2] See Section 13.5 for specifics for each state component and for details regarding mode-specific operation and operation determined by instruction prefixes; in particular, see Section 13.5.6 for some special treatment of PT state by XRSTORS. See Section 13.13 for details regarding faults caused by memory accesses.

  If XRSTORS is restoring a supervisor state component, the instruction causes a general-protection exception (#GP) if it would load any element of that component with an unsupported value (e.g., by setting a reserved bit in an MSR) or if a bit is set in any reserved portion of the state component in the XSAVE area.

1. If CR0.AM = 1, CPL = 3, and EFLAGS.AC =1, an alignment-check exception (#AC) may occur instead of #GP.
2. Earlier fault checking ensured that, if the instruction has reached this point in execution and XSTATE_BV[*i*] is 1, then XCOMP_BV[*i*] is also 1.

State components 0 and 1 are located in the legacy region of the XSAVE area (see Section 13.4.1). Each state component i, $2 \le i \le 62$, is located in the extended region; XRSTORS uses the compacted format for the extended region (see Section 13.4.3).

The MXCSR register is part of SSE state (see Section 13.5.2) and is thus loaded from memory if RFBM[1] = XSTATE_BV[i] = 1. XRSTORS causes a general-protection exception (#GP) if it would load MXCSR with an illegal value.

If an execution of XRSTORS causes an exception or a VM exit during or after restoring a supervisor state component, each element of that state component may have the value it held before the XRSTORS execution, the value loaded from the XSAVE area, or the element's initial value (as defined in Section 13.6). See Section 13.5.6 for some special treatment of PT state for the case in which XRSTORS causes an exception or a VM exit.

Like XRSTOR, execution of XRSTORS causes the processor to update is tracking for the init and modified optimizations (see Section 13.6 and Section 13.8.3). The following items provide details:

- The processor updates its tracking for the init optimization as follows:
  - If RFBM[i] = 0, XINUSE[i] is not changed.
  - If RFBM[i] = 1 and XSTATE_BV[i] = 0, state component i may be tracked as init; XINUSE[i] may be set to 0 or 1.
  - If RFBM[i] = 1 and XSTATE_BV[i] = 1, state component i is tracked as not init; XINUSE[i] is set to 1.
- The processor updates its tracking for the modified optimization and records information about the XRSTORS execution for future interaction with the XSAVEOPT and XSAVES instructions as follows:
  - If RFBM[i] = 0, state component i is tracked as modified; XMODIFIED[i] is set to 1.
  - If RFBM[i] = 1, state component i may be tracked as unmodified; XMODIFIED[i] may be set to 0 or 1.
  - XRSTOR_INFO is set to the 4-tuple $\langle w,x,y,z \rangle$, where w is the CPL; x is 1 if the logical processor is in VMX non-root operation and 0 otherwise; y is the linear address of the XSAVE area; and z is XCOMP_BV (this implies that $z[63] = 1$).

13.13 MEMORY ACCESSES BY THE XSAVE FEATURE SET

Each instruction in the XSAVE feature set operates on a set of XSAVE-managed state components. The specific set of components on which an instruction operates is determined by the values of XCR0, the IA32_XSS MSR, EDX:EAX, and (for XRSTOR and XRSTORS) the XSAVE header.

Section 13.4 provides the details necessary to determine the location of each state component for any execution of an instruction in the XSAVE feature set. An execution of an instruction in the XSAVE feature set may access any byte of any state component on which that execution operates.

Section 13.5 provides details of the different XSAVE-managed state components. Some portions of some of these components are accessible only in 64-bit mode. Executions of XRSTOR and XRSTORS outside 64-bit mode will not update those portions; executions of XSAVE, XSAVEC, XSAVEOPT, and XSAVES will not modify the corresponding locations in memory.

Despite this fact, any execution of these instructions outside 64-bit mode may access any byte in any state component on which that execution operates — even those at addresses corresponding to registers that are accessible only in 64-bit mode. As result, such an execution may incur a fault due to an attempt to access such an address.

For example, an execution of XSAVE outside 64-bit mode may incur a page fault if paging does not map as read/write the section of the XSAVE area containing state component 7 (Hi16_ZMM state) — despite the fact that state component 7 can be accessed only in 64-bit mode.

4. Updates to Chapter 16, Volume 1

Change bars show changes to Chapter 16 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1:* Basic Architecture.

Change to this chapter: Additions/changes to lists in Section 16.3.8.1 "Instruction Based Considerations".

CHAPTER 16
PROGRAMMING WITH INTEL® TRANSACTIONAL SYNCHRONIZATION EXTENSIONS

16.1 OVERVIEW

This chapter describes the software programming interface to the Intel® Transactional Synchronization Extensions of the Intel 64 architecture.

Multithreaded applications take advantage of increasing number of cores to achieve high performance. However, writing multi-threaded applications requires programmers to reason about data sharing among multiple threads. Access to shared data typically requires synchronization mechanisms. These mechanisms ensure multiple threads update shared data by serializing operations on the shared data, often through the use of a critical section protected by a lock. Since serialization limits concurrency, programmers try to limit synchronization overheads. They do this either through minimizing the use of synchronization or through the use of fine-grain locks; where multiple locks each protect different shared data. Unfortunately, this process is difficult and error prone; a missed or incorrect synchronization can cause an application to fail. Conservatively adding synchronization and using coarser granularity locks, where a few locks each protect many items of shared data, helps avoid correctness problems but limits performance due to excessive serialization. While programmers must use static information to determine when to serialize, the determination as to whether actually to serialize is best done dynamically.

Intel® Transactional Synchronization Extensions aim to improve the performance of lock-protected critical sections while maintaining the lock-based programming model.

16.2 INTEL® TRANSACTIONAL SYNCHRONIZATION EXTENSIONS

Intel® Transactional Synchronization Extensions (Intel® TSX) allow the processor to determine dynamically whether threads need to serialize through lock-protected critical sections, and to perform serialization only when required. This lets the hardware expose and exploit concurrency hidden in an application due to dynamically unnecessary synchronization through a technique known as lock elision.

With lock elision, the hardware executes the programmer-specified critical sections (also referred to as transactional regions) transactionally. In such an execution, the lock variable is only read within the transactional region; it is not written to (and therefore not acquired) with the expectation that the lock variable remains unchanged after the transactional region, thus exposing concurrency.

If the transactional execution completes successfully, then the hardware ensures that all memory operations performed within the transactional region will appear to have occurred instantaneously when viewed from other logical processors, a process referred to as an **atomic commit**. Any updates performed within the transactional region are made visible to other processors only on an atomic commit.

Since a successful transactional execution ensures an atomic commit, the processor can execute the programmer-specified code section optimistically without synchronization. If synchronization was unnecessary for that specific execution, execution can commit without any cross-thread serialization.

If the transactional execution is unsuccessful, the processor cannot commit the updates atomically. When this happens, the processor will roll back the execution, a process referred to as a **transactional abort**. On a transactional abort, the processor will discard all updates performed in the region, restore architectural state to appear as if the optimistic execution never occurred, and resume execution non-transactionally. Depending on the policy in place, lock elision may be retried or the lock may be explicitly acquired to ensure forward progress.

Intel TSX provides two software interfaces for programmers.

- Hardware Lock Elision (HLE) is a legacy compatible instruction set extension (comprising the XACQUIRE and XRELEASE prefixes).
- Restricted Transactional Memory (RTM) is a new instruction set interface (comprising the XBEGIN and XEND instructions).

Programmers who would like to run Intel TSX-enabled software on legacy hardware would use the HLE interface to implement lock elision. On the other hand, programmers who do not have legacy hardware requirements and who deal with more complex locking primitives would use the RTM software interface of Intel TSX to implement lock elision. In the latter case when using new instructions, the programmer must always provide a non-transactional path (which would have code to eventually acquire the lock being elided) to execute following a transactional abort and must not rely on the transactional execution alone.

In addition, Intel TSX also provides the XTEST instruction to test whether a logical processor is executing transactionally, and the XABORT instruction to abort a transactional region.

A processor can perform a transactional abort for numerous reasons. A primary cause is due to conflicting accesses between the transactionally executing logical processor and another logical processor. Such conflicting accesses may prevent a successful transactional execution. Memory addresses read from within a transactional region constitute the **read-set** of the transactional region and addresses written to within the transactional region constitute the **write-set** of the transactional region. Intel TSX maintains the read- and write-sets at the granularity of a cache line.

A conflicting data access occurs if another logical processor either reads a location that is part of the transactional region's write-set or writes a location that is a part of either the read- or write-set of the transactional region. We refer to this as a **data conflict**. Since Intel TSX detects data conflicts at the granularity of a cache line, unrelated data locations placed in the same cache line will be detected as conflicts. Transactional aborts may also occur due to limited transactional resources. For example, the amount of data accessed in the region may exceed an implementation-specific capacity. Additionally, some instructions and system events may cause transactional aborts.

16.2.1 HLE Software Interface

HLE provides two new instruction prefix hints: XACQUIRE and XRELEASE.

The programmer uses the XACQUIRE prefix in front of the instruction that is used to acquire the lock that is protecting the critical section. The processor treats the indication as a hint to elide the write associated with the lock acquire operation. Even though the lock acquire has an associated write operation to the lock, the processor does not add the address of the lock to the transactional region's write-set nor does it issue any write requests to the lock. Instead, the address of the lock is added to the read-set. The logical processor enters transactional execution. If the lock was available before the XACQUIRE prefixed instruction, all other processors will continue to see it as available afterwards. Since the transactionally executing logical processor neither added the address of the lock to its write-set nor performed externally visible write operations to it, other logical processors can read the lock without causing a data conflict. This allows other logical processors to also enter and concurrently execute the critical section protected by the lock. The processor automatically detects any data conflicts that occur during the transactional execution and will perform a transactional abort if necessary.

Even though the eliding processor did not perform any external write operations to the lock, the hardware ensures program order of operations on the lock. If the eliding processor itself reads the value of the lock in the critical section, it will appear as if the processor had acquired the lock, i.e. the read will return the non-elided value. This behavior makes an HLE execution functionally equivalent to an execution without the HLE prefixes.

The programmer uses the XRELEASE prefix in front of the instruction that is used to release the lock protecting the critical section. This involves a write to the lock. If the instruction is restoring the value of the lock to the value it had prior to the XACQUIRE prefixed lock acquire operation on the same lock, then the processor elides the external write request associated with the release of the lock and does not add the address of the lock to the write-set. The processor then attempts to commit the transactional execution.

With HLE, if multiple threads execute critical sections protected by the same lock but they do not perform any conflicting operations on each other's data, then the threads can execute concurrently and without serialization. Even though the software uses lock acquisition operations on a common lock, the hardware recognizes this, elides the lock, and executes the critical sections on the two threads without requiring any communication through the lock — if such communication was dynamically unnecessary.

If the processor is unable to execute the region transactionally, it will execute the region non-transactionally and without elision. HLE enabled software has the same forward progress guarantees as the underlying non-HLE lock-based execution. For successful HLE execution, the lock and the critical section code must follow certain guidelines (discussed in Section 16.3.3 and Section 16.3.8). These guidelines only affect performance; not following these guidelines will not cause a functional failure.

Hardware without HLE support will ignore the XACQUIRE and XRELEASE prefix hints and will not perform any elision since these prefixes correspond to the REPNE/REPE IA-32 prefixes which are ignored on the instructions where XACQUIRE and XRELEASE are valid. Importantly, HLE is compatible with the existing lock-based programming model. Improper use of hints will not cause functional bugs though it may expose latent bugs already in the code.

16.2.2 RTM Software Interface

RTM provides three new instructions: XBEGIN, XEND, and XABORT.

Software uses the XBEGIN instruction to specify the start of the transactional region and the XEND instruction to specify the end of the transactional region. The XBEGIN instruction takes an operand that provides a relative offset to the **fallback instruction address** if the transactional region could not be successfully executed transactionally. Software using these instructions to implement lock elision must test the lock within the transactional region, and only if free should try to commit. Further, the software may also define a policy to retry if the lock is not free.

A processor may abort transactional execution for many reasons. The hardware automatically detects transactional abort conditions and restarts execution from the fallback instruction address with the architectural state corresponding to that at the start of the XBEGIN instruction and the EAX register updated to describe the abort status.

The XABORT instruction allows programmers to abort the execution of a transactional region explicitly. The XABORT instruction takes an 8 bit immediate argument that is loaded into the EAX register and will thus be available to software following a transactional abort.

Hardware provides no guarantees as to whether a transactional execution will ever successfully commit. Programmers must always provide an alternative code sequence in the fallback path to guarantee forward progress. When using the instructions for lock elision, this may be as simple as acquiring a lock and executing the specified code region non-transactionally. Further, a transactional region that always aborts on a given implementation may complete transactionally on a future implementation. Therefore, programmers must ensure the code paths for the transactional region and the alternative code sequence are functionally tested.

If the RTM software interface is used for anything other than lock elision, the programmer must similarly ensure that the fallback path is inter-operable with the transactionally executing path.

16.3 INTEL® TSX APPLICATION PROGRAMMING MODEL

16.3.1 Detection of Transactional Synchronization Support

16.3.1.1 Detection of HLE Support

A processor supports HLE execution if CPUID.07H.EBX.HLE [bit 4] = 1. However, an application can use the HLE prefixes (XACQUIRE and XRELEASE) without checking whether the processor supports HLE. Processors without HLE support ignore these prefixes and will execute the code without entering transactional execution.

16.3.1.2 Detection of RTM Support

A processor supports RTM execution if CPUID.07H.EBX.RTM [bit 11] = 1. An application must check if the processor supports RTM before it uses the RTM instructions (XBEGIN, XEND, XABORT). These instructions will generate a #UD exception when used on a processor that does not support RTM.

16.3.1.3 Detection of XTEST Instruction

A processor supports the XTEST instruction if it supports either HLE or RTM. An application must check either of these feature flags before using the XTEST instruction. This instruction will generate a #UD exception when used on a processor that does not support either HLE or RTM.

16.3.2 Querying Transactional Execution Status

The XTEST instruction can be used to determine the transactional status of a transactional region specified by HLE or RTM. Note, while the HLE prefixes are ignored on processors that do not support HLE, the XTEST instruction will generate a #UD exception when used on processors that do not support either HLE or RTM.

16.3.3 Requirements for HLE Locks

For HLE execution to successfully commit transactionally, the lock must satisfy certain properties and access to the lock must follow certain guidelines.

- An XRELEASE prefixed instruction must restore the value of the elided lock to the value it had before the lock acquisition. This allows hardware to safely elide locks by not adding them to the write-set. The data size and data address of the lock release (XRELEASE prefixed) instruction must match that of the lock acquire (XACQUIRE prefixed) and the lock must not cross a cache line boundary.
- Software should not write to the elided lock inside a transactional HLE region with any instruction other than an XRELEASE prefixed instruction, otherwise it may cause a transactional abort. In addition, recursive locks (where a thread acquires the same lock multiple times without first releasing the lock) may also cause a transactional abort. Note that software can observe the result of the elided lock acquire inside the critical section. Such a read operation will return the value of the write to the lock.

The processor automatically detects violations to these guidelines, and safely transitions to a non-transactional execution without elision. Since Intel TSX detects conflicts at the granularity of a cache line, writes to data collocated on the same cache line as the elided lock may be detected as data conflicts by other logical processors eliding the same lock.

16.3.4 Transactional Nesting

Both HLE- and RTM-based transactional executions support nested transactional regions. However, a transactional abort restores state to the operation that started transactional execution: either the outermost XACQUIRE prefixed HLE eligible instruction or the outermost XBEGIN instruction. The processor treats all nested transactional regions as one monolithic transactional region.

16.3.4.1 HLE Nesting and Elision

Programmers can nest HLE regions up to an implementation specific depth of MAX_HLE_NEST_COUNT. Each logical processor tracks the nesting count internally but this count is not available to software. An XACQUIRE prefixed HLE-eligible instruction increments the nesting count, and an XRELEASE prefixed HLE-eligible instruction decrements it. The logical processor enters transactional execution when the nesting count goes from zero to one. The logical processor attempts to commit only when the nesting count becomes zero. A transactional abort may occur if the nesting count exceeds MAX_HLE_NEST_COUNT.

In addition to supporting nested HLE regions, the processor can also elide multiple nested locks. The processor tracks a lock for elision beginning with the XACQUIRE prefixed HLE eligible instruction for that lock and ending with the XRELEASE prefixed HLE eligible instruction for that same lock. The processor can, at any one time, track up to a MAX_HLE_ELIDED_LOCKS number of locks. For example, if the implementation supports a MAX_HLE_ELIDED_LOCKS value of two and if the programmer nests three HLE identified critical sections (by performing XACQUIRE prefixed HLE eligible instructions on three distinct locks without performing an intervening XRELEASE prefixed HLE eligible instruction on any one of the locks), then the first two locks will be elided, but the third won't be elided (but will be added to the transaction’s write-set). However, the execution will still continue transactionally. Once an XRELEASE for one of the two elided locks is encountered, a subsequent lock acquired through the XACQUIRE prefixed HLE eligible instruction will be elided.

The processor attempts to commit the HLE execution when all elided XACQUIRE and XRELEASE pairs have been matched, the nesting count goes to zero, and the locks have satisfied the requirements described earlier. If execution cannot commit atomically, then execution transitions to a non-transactional execution without elision as if the first instruction did not have an XACQUIRE prefix.

16.3.4.2 RTM Nesting

Programmers can nest RTM-based transactional regions up to an implementation specific MAX_RTM_NEST_COUNT. The logical processor tracks the nesting count internally but this count is not available to software. An XBEGIN instruction increments the nesting count, and an XEND instruction decrements it. The logical processor attempts to commit only if the nesting count becomes zero. A transactional abort occurs if the nesting count exceeds MAX_RTM_NEST_COUNT.

16.3.4.3 Nesting HLE and RTM

HLE and RTM provide two alternative software interfaces to a common transactional execution capability. The behavior when HLE and RTM are nested together—HLE inside RTM or RTM inside HLE—is implementation specific. However, in all cases, the implementation will maintain HLE and RTM semantics. An implementation may choose to ignore HLE hints when used inside RTM regions, and may cause a transactional abort when RTM instructions are used inside HLE regions. In the latter case, the transition from transactional to non-transactional execution occurs seamlessly since the processor will re-execute the HLE region without actually doing elision, and then execute the RTM instructions.

16.3.5 RTM Abort Status Definition

RTM uses the EAX register to communicate abort status to software. Following an RTM abort the EAX register has the following definition.

Table 16-1. RTM Abort Status Definition

EAX Register Bit Position	Meaning
0	Set if abort caused by XABORT instruction.
1	If set, the transactional execution may succeed on a retry. This bit is always clear if bit 0 is set.
2	Set if another logical processor conflicted with a memory address that was part of the transactional execution that aborted.
3	Set if an internal buffer to track transactional state overflowed.
4	Set if a debug exception (#DB) or breakpoint exception (#BP) was hit.
5	Set if an abort occurred during execution of a nested transactional execution.
23:6	Reserved.
31:24	XABORT argument (only valid if bit 0 set, otherwise reserved).

The EAX abort status for RTM only provides causes for aborts. It does not by itself encode whether an abort or commit occurred for the RTM region. The value of EAX can be 0 following an RTM abort. For example, a CPUID instruction when used inside an RTM region causes a transactional abort and may not satisfy the requirements for setting any of the EAX bits. This may result in an EAX value of 0.

16.3.6 RTM Memory Ordering

A successful RTM commit causes all memory operations in the RTM region to appear to execute atomically. A successfully committed RTM region consisting of an XBEGIN followed by an XEND, even with no memory operations in the RTM region, has the same ordering semantics as a LOCK prefixed instruction.

The XBEGIN instruction does not have fencing semantics. However, if an RTM execution aborts, all memory updates from within the RTM region are discarded and never made visible to any other logical processor.

16.3.7 RTM-Enabled Debugger Support

Any debug exception (#DB) or breakpoint exception (#BP) inside an RTM region causes a transactional abort and, by default, redirects control flow to the fallback instruction address with architectural state recovered and bit 4 in EAX set. However, to allow software debuggers to intercept execution on debug or breakpoint exceptions, the RTM architecture provides additional capability called **advanced debugging of RTM transactional regions**.

Advanced debugging of RTM transactional regions is enabled if bit 11 of DR7 and bit 15 of the IA32_DEBUGCTL MSR are both 1. In this case, any RTM transactional abort due to a #DB or #BP causes execution to roll back to just before the XBEGIN instruction (EAX is restored to the value it had before XBEGIN) and then delivers a #DB. (A #DB is delivered even if the transactional abort was caused by a #BP.) DR6[16] is cleared to indicate that the exception resulted from a debug or breakpoint exception inside an RTM region. See also Section 17.3.3, "Debug Exceptions, Breakpoint Exceptions, and Restricted Transactional Memory (RTM)," of *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3B*.

16.3.8 Programming Considerations

Typical programmer-identified regions are expected to execute transactionally and to commit successfully. However, Intel TSX does not provide any such guarantee. A transactional execution may abort for many reasons. To take full advantage of the transactional capabilities, programmers should follow certain guidelines to increase the probability of their transactional execution committing successfully.

This section discusses various events that may cause transactional aborts. The architecture ensures that updates performed within a transactional region that subsequently aborts execution will never become visible. Only a committed transactional execution updates architectural state. Transactional aborts never cause functional failures and only affect performance.

16.3.8.1 Instruction Based Considerations

Programmers can use any instruction safely inside a transactional region. Further, programmers can use the Intel TSX instructions and prefixes at any privilege level. However, some instructions will always abort the transactional execution and cause execution to seamlessly and safely transition to a non-transactional path.

Intel TSX allows for most common instructions to be used inside transactional regions without causing aborts. The following operations inside a transactional region do not typically cause an abort.

- Operations on the instruction pointer register, general purpose registers (GPRs) and the status flags (CF, OF, SF, PF, AF, and ZF).
- Operations on XMM and YMM registers and the MXCSR register

However, programmers must be careful when intermixing SSE and AVX operations inside a transactional region. Intermixing SSE instructions accessing XMM registers and AVX instructions accessing YMM registers may cause transactional regions to abort.

CLD and STD instructions when used inside transactional regions may cause aborts if they change the value of the DF flag. However, if DF is 1, the STD instruction will not cause an abort. Similarly, if DF is 0, the CLD instruction will not cause an abort.

Instructions not enumerated here as causing abort when used inside a transactional region will typically not cause the execution to abort (examples include but are not limited to MFENCE, LFENCE, SFENCE, RDTSC, RDTSCP, etc.).

The following instructions will abort transactional execution on any implementation:

- XABORT
- CPUID
- PAUSE
- ENCLS
- ENCLU

In addition, in some implementations, the following instructions may always cause transactional aborts. These instructions are not expected to be commonly used inside typical transactional regions. However, programmers must not rely on these instructions to force a transactional abort, since whether they cause transactional aborts is implementation dependent.

- Operations on X87 and MMX architecture state. This includes all MMX and X87 instructions, including the FXRSTOR and FXSAVE instructions.
- Update to non-status portion of EFLAGS: CLI, STI, POPFD, POPFQ, CLAC and STAC.
- Instructions that update segment registers, debug registers and/or control registers: MOV to DS/ES/FS/GS/SS, POP DS/ES/FS/GS/SS, LDS, LES, LFS, LGS, LSS, SWAPGS, WRFSBASE, WRGSBASE, LGDT, SGDT, LIDT, SIDT, LLDT, SLDT, LTR, STR, Far CALL, Far JMP, Far RET, IRET, MOV to DRx, MOV to CR0/CR2/CR3/CR4/CR8, CLTS, and LMSW.
- Ring transitions: SYSENTER, SYSCALL, SYSEXIT, and SYSRET.
- TLB and Cacheability control: CLFLUSH, CLFLUSHOPT, CLWB, INVD, WBINVD, INVLPG, INVPCID, and memory instructions with a non-temporal hint (V/MOVNTDQA, V/MOVNTDQ, V/MOVNTI, V/MOVNTPD, V/MOVNTPS, V/MOVNTQ, V/MASKMOVQ, and V/MASKMOVDQU).
- Processor state save: XRSTOR, XRSTORS, XSAVE, XSAVEC, XSAVEOPT, XSAVES, and XSETBV.
- Interrupts: INTn, INTO.
- IO: IN, INS, REP INS, OUT, OUTS, REP OUTS and their variants.
- VMX: VMPTRLD, VMPTRST, VMCLEAR, VMREAD, VMWRITE, VMCALL, VMLAUNCH, VMRESUME, VMXOFF, VMXON, INVEPT, INVVPID, and VMFUNC.
- SMX: GETSEC.
- UD, RSM, RDMSR, WRMSR, WRPKRU, HLT, MONITOR, MWAIT, and VZEROUPPER.

16.3.8.2 Runtime Considerations

In addition to the instruction-based considerations, runtime events may cause transactional execution to abort. These may be due to data access patterns or micro-architectural implementation causes. Keep in mind that the following list is not a comprehensive discussion of all abort causes.

Any fault or trap in a transactional region that must be exposed to software will be suppressed. Transactional execution will abort and execution will transition to a non-transactional execution, as if the fault or trap had never occurred. If any exception is not masked, that will result in a transactional abort and it will be as if the exception had never occurred.

When executed in VMX non-root operation, certain instructions may result in a VM exit. When such instructions are executed inside a transactional region, then instead of causing a VM exit, they will cause a transactional abort and the execution will appear as if instruction that would have caused a VM exit never executed.

Synchronous exception events (#DE, #OF, #NP, #SS, #GP, #BR, #UD, #AC, #XM, #PF, #NM, #TS, #MF, #DB, #BP/INT3) that occur during transactional execution may cause an execution not to commit transactionally, and require a non-transactional execution. These events are suppressed as if they had never occurred. With HLE, since the non-transactional code path is identical to the transactional code path, these events will typically re-appear when the instruction that caused the exception is re-executed non-transactionally, causing the associated synchronous events to be delivered appropriately in the non-transactional execution. The same behavior also applies to synchronous events (EPT violations, EPT misconfigurations, and accesses to the APIC-access page) that occur in VMX non-root operation.

Asynchronous events (NMI, SMI, INTR, IPI, PMI, etc.) occurring during transactional execution may cause the transactional execution to abort and transition to a non-transactional execution. The asynchronous events will be pended and handled after the transactional abort is processed. The same behavior also applies to asynchronous events (VMX-preemption timer expiry, virtual-interrupt delivery, and interrupt-window exiting) that occur in VMX non-root operation.

Transactional execution only supports write-back cacheable memory type operations. A transactional region may always abort if it includes operations on any other memory type. This includes instruction fetches to UC memory type.

Memory accesses within a transactional region may require the processor to set the Accessed and Dirty flags of the referenced page table entry. The behavior of how the processor handles this is implementation specific. Some implementations may allow the updates to these flags to become externally visible even if the transactional region subsequently aborts. Some Intel TSX implementations may choose to abort the transactional execution if these flags need to be updated. Further, a processor's page-table walk may generate accesses to its own transactionally written but uncommitted state. Some Intel TSX implementations may choose to abort the execution of a transac-

tional region in such situations. Regardless, the architecture ensures that, if the transactional region aborts, then the transactionally written state will not be made architecturally visible through the behavior of structures such as TLBs.

Executing self-modifying code transactionally may also cause transactional aborts. Programmers must continue to follow the Intel recommended guidelines for writing self-modifying and cross-modifying code even when employing Intel TSX.

While an Intel TSX implementation will typically provide sufficient resources for executing common transactional regions, implementation constraints and excessive sizes for transactional regions may cause a transactional execution to abort and transition to a non-transactional execution. The architecture provides no guarantee of the amount of resources available to do transactional execution and does not guarantee that a transactional execution will ever succeed.

Conflicting requests to a cache line accessed within a transactional region may prevent the transactional region from executing successfully. For example, if logical processor P0 reads line A in a transactional region and another logical processor P1 writes A (either inside or outside a transactional region) then logical processor P0 may abort if logical processor P1’s write interferes with processor P0's ability to execute transactionally. Similarly, if P0 writes line A in a transactional region and P1reads or writes A (either inside or outside a transactional region), then P0 may abort if P1's access to A interferes with P0's ability to execute transactionally. In addition, other coherence traffic may at times appear as conflicting requests and may cause aborts. While these false conflicts may happen, they are expected to be uncommon. The conflict resolution policy to determine whether P0 or P1 aborts in the above scenarios is implementation specific.

5. Updates to Chapter 1, Volume 2A

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

Change to this chapter: Updates to list of processors supported and minor typo correction in Figure 1-2 "Syntax for CPUID, CR, and MSR Data Presentation".

CHAPTER 1
ABOUT THIS MANUAL

The *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volumes 2A, 2B, 2C & 2D*: *Instruction Set Reference* (order numbers 253666, 253667, 326018 and 334569) are part of a set that describes the architecture and programming environment of all Intel 64 and IA-32 architecture processors. Other volumes in this set are:

- The *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1: Basic Architecture* (Order Number 253665).
- The *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volumes 3A, 3B, 3C & 3D: System Programming Guide* (order numbers 253668, 253669, 326019 and 332831).
- The *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 4: Model-Specific Registers* (order number 335592).

The *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*, describes the basic architecture and programming environment of Intel 64 and IA-32 processors. The *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volumes 2A, 2B, 2C & 2D*, describe the instruction set of the processor and the opcode structure. These volumes apply to application programmers and to programmers who write operating systems or executives. The *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volumes 3A, 3B, 3C & 3D*, describe the operating-system support environment of Intel 64 and IA-32 processors. These volumes target operating-system and BIOS designers. In addition, the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3B*, addresses the programming environment for classes of software that host operating systems. The *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 4*, describes the model-specific registers of Intel 64 and IA-32 processors.

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

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

- Pentium® processors
- P6 family processors
- Pentium® 4 processors
- Pentium® M processors
- Intel® Xeon® processors
- Pentium® D processors
- Pentium® processor Extreme Editions
- 64-bit Intel® Xeon® processors
- Intel® Core™ Duo processor
- Intel® Core™ Solo processor
- Dual-Core Intel® Xeon® processor LV
- Intel® Core™2 Duo processor
- Intel® Core™2 Quad processor Q6000 series
- Intel® Xeon® processor 3000, 3200 series
- Intel® Xeon® processor 5000 series
- Intel® Xeon® processor 5100, 5300 series
- Intel® Core™2 Extreme processor X7000 and X6800 series
- Intel® Core™2 Extreme processor QX6000 series
- Intel® Xeon® processor 7100 series

- Intel® Pentium® Dual-Core processor
- Intel® Xeon® processor 7200, 7300 series
- Intel® Xeon® processor 5200, 5400, 7400 series
- Intel® Core™2 Extreme processor QX9000 and X9000 series
- Intel® Core™2 Quad processor Q9000 series
- Intel® Core™2 Duo processor E8000, T9000 series
- Intel® Atom™ processor family
- Intel® Atom™ processors 200, 300, D400, D500, D2000, N200, N400, N2000, E2000, Z500, Z600, Z2000, C1000 series are built from 45 nm and 32 nm processes
- Intel® Core™ i7 processor
- Intel® Core™ i5 processor
- Intel® Xeon® processor E7-8800/4800/2800 product families
- Intel® Core™ i7-3930K processor
- 2nd generation Intel® Core™ i7-2xxx, Intel® Core™ i5-2xxx, Intel® Core™ i3-2xxx processor series
- Intel® Xeon® processor E3-1200 product family
- Intel® Xeon® processor E5-2400/1400 product family
- Intel® Xeon® processor E5-4600/2600/1600 product family
- 3rd generation Intel® Core™ processors
- Intel® Xeon® processor E3-1200 v2 product family
- Intel® Xeon® processor E5-2400/1400 v2 product families
- Intel® Xeon® processor E5-4600/2600/1600 v2 product families
- Intel® Xeon® processor E7-8800/4800/2800 v2 product families
- 4th generation Intel® Core™ processors
- The Intel® Core™ M processor family
- Intel® Core™ i7-59xx Processor Extreme Edition
- Intel® Core™ i7-49xx Processor Extreme Edition
- Intel® Xeon® processor E3-1200 v3 product family
- Intel® Xeon® processor E5-2600/1600 v3 product families
- 5th generation Intel® Core™ processors
- Intel® Xeon® processor D-1500 product family
- Intel® Xeon® processor E5 v4 family
- Intel® Atom™ processor X7-Z8000 and X5-Z8000 series
- Intel® Atom™ processor Z3400 series
- Intel® Atom™ processor Z3500 series
- 6th generation Intel® Core™ processors
- Intel® Xeon® processor E3-1500m v5 product family
- 7th generation Intel® Core™ processors
- Intel® Xeon Phi™ Processor 3200, 5200, 7200 Series
- Intel® Xeon® Processor Scalable Family

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

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

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

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

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

The Intel® Atom™ processors 200, 300, D400, D500, D2000, N200, N400, N2000, E2000, Z500, Z600, Z2000, C1000 series are based on the Intel® Atom™ microarchitecture and supports Intel 64 architecture.

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

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

The Intel® Core™ i7 processor and Intel® Xeon® processor 3400, 5500, 7500 series are based on 45 nm Intel® microarchitecture code name Nehalem. Intel® microarchitecture code name Westmere is a 32 nm version of Intel® microarchitecture code name Nehalem. Intel® Xeon® processor 5600 series, Intel Xeon processor E7 and various Intel Core i7, i5, i3 processors are based on Intel® microarchitecture code name Westmere. These processors support Intel 64 architecture.

The Intel® Xeon® processor E5 family, Intel® Xeon® processor E3-1200 family, Intel® Xeon® processor E7-8800/4800/2800 product families, Intel® Core™ i7-3930K processor, and 2nd generation Intel® Core™ i7-2xxx, Intel® Core™ i5-2xxx, Intel® Core™ i3-2xxx processor series are based on the Intel® microarchitecture code name Sandy Bridge and support Intel 64 architecture.

The Intel® Xeon® processor E7-8800/4800/2800 v2 product families, Intel® Xeon® processor E3-1200 v2 product family and 3rd generation Intel® Core™ processors are based on the Intel® microarchitecture code name Ivy Bridge and support Intel 64 architecture.

The Intel® Xeon® processor E5-4600/2600/1600 v2 product families, Intel® Xeon® processor E5-2400/1400 v2 product families and Intel® Core™ i7-49xx Processor Extreme Edition are based on the Intel® microarchitecture code name Ivy Bridge-E and support Intel 64 architecture.

The Intel® Xeon® processor E3-1200 v3 product family and 4th Generation Intel® Core™ processors are based on the Intel® microarchitecture code name Haswell and support Intel 64 architecture.

The Intel® Xeon® processor E5-2600/1600 v3 product families and the Intel® Core™ i7-59xx Processor Extreme Edition are based on the Intel® microarchitecture code name Haswell-E and support Intel 64 architecture.

The Intel® Atom™ processor Z8000 series is based on the Intel microarchitecture code name Airmont.

The Intel® Atom™ processor Z3400 series and the Intel® Atom™ processor Z3500 series are based on the Intel microarchitecture code name Silvermont.

The Intel® Core™ M processor family, 5th generation Intel® Core™ processors, Intel® Xeon® processor D-1500 product family and the Intel® Xeon® processor E5 v4 family are based on the Intel® microarchitecture code name Broadwell and support Intel 64 architecture.

The Intel® Xeon® Processor Scalable Family, Intel® Xeon® processor E3-1500m v5 product family and 6th generation Intel® Core™ processors are based on the Intel® microarchitecture code name Skylake and support Intel 64 architecture.

The 7th generation Intel® Core™ processors are based on the Intel® microarchitecture code name Kaby Lake and support Intel 64 architecture.

The Intel® Xeon Phi™ Processor 3200, 5200, 7200 Series is based on the Intel® microarchitecture code name Knights Landing and supports Intel 64 architecture.

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

1.2 OVERVIEW OF VOLUME 2A, 2B, 2C AND 2D: INSTRUCTION SET REFERENCE

A description of *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volumes 2A, 2B, 2C & 2D* content follows:

Chapter 1 — About This Manual. Gives an overview of all seven volumes of the *Intel® 64 and IA-32 Architectures Software Developer's Manual*. It also describes the notational conventions in these manuals and lists related Intel® manuals and documentation of interest to programmers and hardware designers.

Chapter 2 — Instruction Format. Describes the machine-level instruction format used for all IA-32 instructions and gives the allowable encodings of prefixes, the operand-identifier byte (ModR/M byte), the addressing-mode specifier byte (SIB byte), and the displacement and immediate bytes.

Chapter 3 — Instruction Set Reference, A-L. Describes Intel 64 and IA-32 instructions in detail, including an algorithmic description of operations, the effect on flags, the effect of operand- and address-size attributes, and the exceptions that may be generated. The instructions are arranged in alphabetical order. General-purpose, x87 FPU, Intel MMX™ technology, SSE/SSE2/SSE3/SSSE3/SSE4 extensions, and system instructions are included.

Chapter 4 — Instruction Set Reference, M-U. Continues the description of Intel 64 and IA-32 instructions started in Chapter 3. It starts *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2B*.

Chapter 5 — Instruction Set Reference, V-Z. Continues the description of Intel 64 and IA-32 instructions started in chapters 3 and 4. It provides the balance of the alphabetized list of instructions and starts *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2C*.

Chapter 6— Safer Mode Extensions Reference. Describes the safer mode extensions (SMX). SMX is intended for a system executive to support launching a measured environment in a platform where the identity of the software controlling the platform hardware can be measured for the purpose of making trust decisions. This chapter starts *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2D*.

Appendix A — Opcode Map. Gives an opcode map for the IA-32 instruction set.

Appendix B — Instruction Formats and Encodings. Gives the binary encoding of each form of each IA-32 instruction.

Appendix C — Intel® C/C++ Compiler Intrinsics and Functional Equivalents. Lists the Intel® C/C++ compiler intrinsics and their assembly code equivalents for each of the IA-32 MMX and SSE/SSE2/SSE3 instructions.

1.3 NOTATIONAL CONVENTIONS

This manual uses specific notation for data-structure formats, for symbolic representation of instructions, and for hexadecimal and binary numbers. A review of this notation makes the manual easier to read.

1.3.1 Bit and Byte Order

In illustrations of data structures in memory, smaller addresses appear toward the bottom of the figure; addresses increase toward the top. Bit positions are numbered from right to left. The numerical value of a set bit is equal to two raised to the power of the bit position. IA-32 processors are "little endian" machines; this means the bytes of a word are numbered starting from the least significant byte. Figure 1-1 illustrates these conventions.

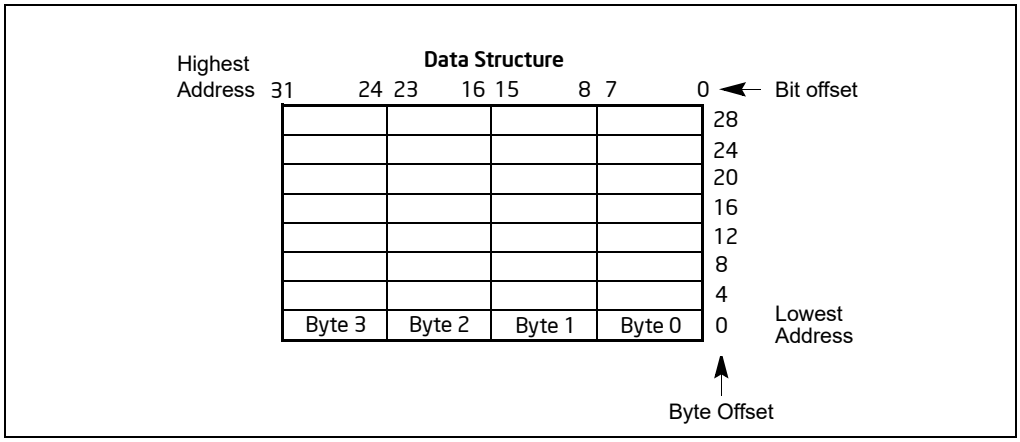


Figure 1-1. Bit and Byte Order

1.3.2 Reserved Bits and Software Compatibility

In many register and memory layout descriptions, certain bits are marked as **reserved**. When bits are marked as reserved, it is essential for compatibility with future processors that software treat these bits as having a future, though unknown, effect. The behavior of reserved bits should be regarded as not only undefined, but unpredictable. Software should follow these guidelines in dealing with reserved bits:

- Do not depend on the states of any reserved bits when testing the values of registers which contain such bits. Mask out the reserved bits before testing.
- Do not depend on the states of any reserved bits when storing to memory or to a register.
- Do not depend on the ability to retain information written into any reserved bits.
- When loading a register, always load the reserved bits with the values indicated in the documentation, if any, or reload them with values previously read from the same register.

NOTE

Avoid any software dependence upon the state of reserved bits in IA-32 registers. Depending upon the values of reserved register bits will make software dependent upon the unspecified manner in which the processor handles these bits. Programs that depend upon reserved values risk incompatibility with future processors.

1.3.3 Instruction Operands

When instructions are represented symbolically, a subset of the IA-32 assembly language is used. In this subset, an instruction has the following format:

```
label: mnemonic argument1, argument2, argument3
```

where:

- A **label** is an identifier which is followed by a colon.
- A **mnemonic** is a reserved name for a class of instruction opcodes which have the same function.
- The operands *argument1*, *argument2*, and *argument3* are optional. There may be from zero to three operands, depending on the opcode. When present, they take the form of either literals or identifiers for data items. Operand identifiers are either reserved names of registers or are assumed to be assigned to data items declared in another part of the program (which may not be shown in the example).

When two operands are present in an arithmetic or logical instruction, the right operand is the source and the left operand is the destination.

For example:

```
LOADREG: MOV EAX, SUBTOTAL
```

In this example, LOADREG is a label, MOV is the mnemonic identifier of an opcode, EAX is the destination operand, and SUBTOTAL is the source operand. Some assembly languages put the source and destination in reverse order.

1.3.4 Hexadecimal and Binary Numbers

Base 16 (hexadecimal) numbers are represented by a string of hexadecimal digits followed by the character H (for example, F82EH). A hexadecimal digit is a character from the following set: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, and F.

Base 2 (binary) numbers are represented by a string of 1s and 0s, sometimes followed by the character B (for example, 1010B). The "B" designation is only used in situations where confusion as to the type of number might arise.

1.3.5 Segmented Addressing

The processor uses byte addressing. This means memory is organized and accessed as a sequence of bytes. Whether one or more bytes are being accessed, a byte address is used to locate the byte or bytes in memory. The range of memory that can be addressed is called an **address space**.

The processor also supports segmented addressing. This is a form of addressing where a program may have many independent address spaces, called **segments**. For example, a program can keep its code (instructions) and stack in separate segments. Code addresses would always refer to the code space, and stack addresses would always refer to the stack space. The following notation is used to specify a byte address within a segment:

```
Segment-register:Byte-address
```

For example, the following segment address identifies the byte at address FF79H in the segment pointed by the DS register:

```
DS:FF79H
```

The following segment address identifies an instruction address in the code segment. The CS register points to the code segment and the EIP register contains the address of the instruction.

```
CS:EIP
```

1.3.6 Exceptions

An exception is an event that typically occurs when an instruction causes an error. For example, an attempt to divide by zero generates an exception. However, some exceptions, such as breakpoints, occur under other conditions. Some types of exceptions may provide error codes. An error code reports additional information about the error. An example of the notation used to show an exception and error code is shown below:

```
#PF(fault code)
```

This example refers to a page-fault exception under conditions where an error code naming a type of fault is reported. Under some conditions, exceptions which produce error codes may not be able to report an accurate code. In this case, the error code is zero, as shown below for a general-protection exception:

```
#GP(0)
```

1.3.7 A New Syntax for CPUID, CR, and MSR Values

Obtain feature flags, status, and system information by using the CPUID instruction, by checking control register bits, and by reading model-specific registers. We are moving toward a new syntax to represent this information. See Figure 1-2.

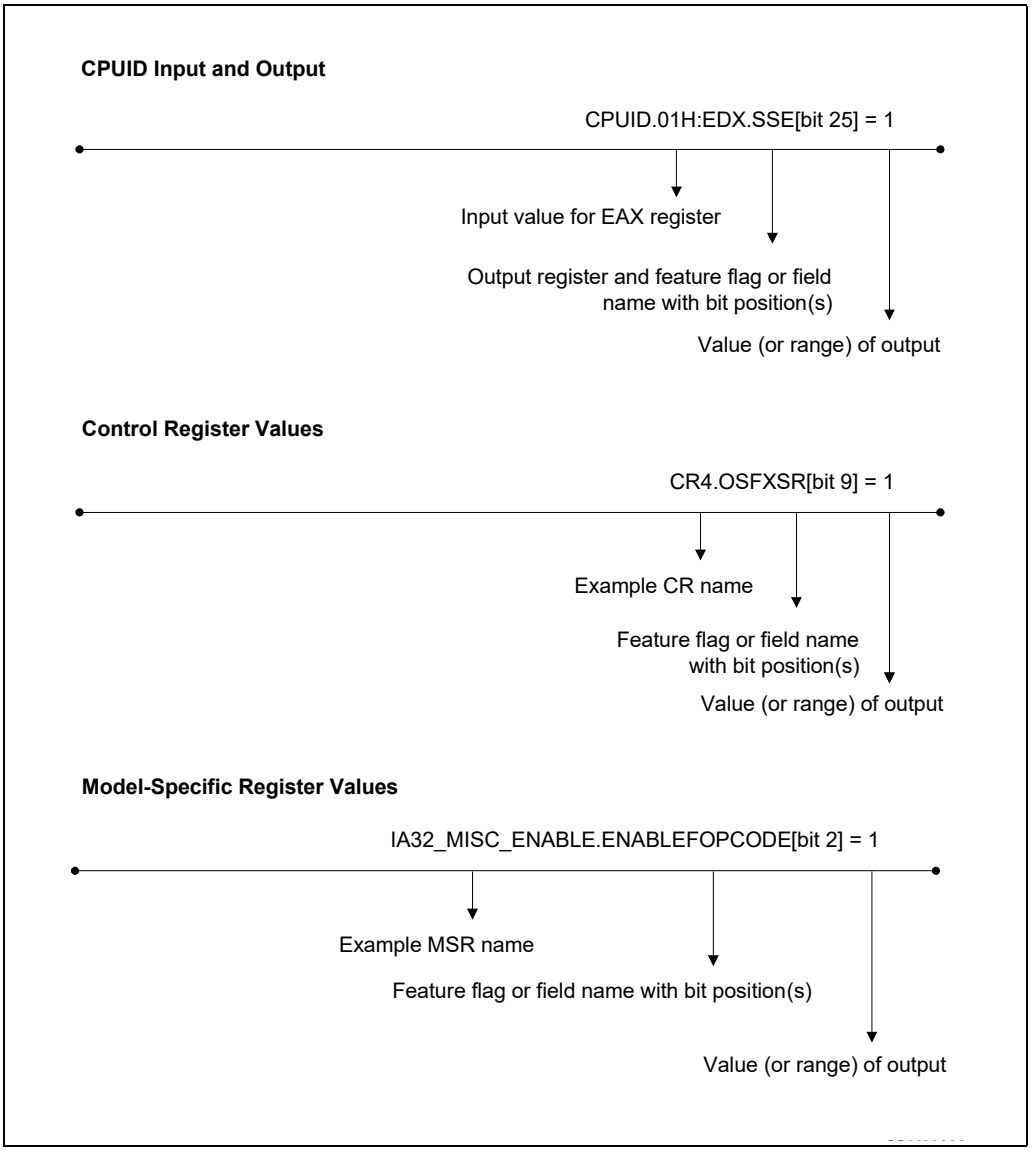


Figure 1-2. Syntax for CPUID, CR, and MSR Data Presentation

1.4 RELATED LITERATURE

Literature related to Intel 64 and IA-32 processors is listed and viewable on-line at:

http://www.intel.com/content/www/us/en/processors/architectures-software-developer-manuals.html

See also:

- The data sheet for a particular Intel 64 or IA-32 processor
- The specification update for a particular Intel 64 or IA-32 processor
- Intel® C++ Compiler documentation and online help:
  http://software.intel.com/en-us/articles/intel-compilers/
- Intel® Fortran Compiler documentation and online help:
  http://software.intel.com/en-us/articles/intel-compilers/
- Intel® Software Development Tools:
  https://software.intel.com/en-us/intel-sdp-home

- Intel® 64 and IA-32 Architectures Software Developer's Manual (in one, four or ten volumes): https://software.intel.com/en-us/articles/intel-sdm
- Intel® 64 and IA-32 Architectures Optimization Reference Manual: https://software.intel.com/en-us/articles/intel-sdm#optimization
- Intel 64 Architecture x2APIC Specification:

  http://www.intel.com/content/www/us/en/architecture-and-technology/64-architecture-x2apic-specification.html
- Intel® Trusted Execution Technology Measured Launched Environment Programming Guide:

  http://www.intel.com/content/www/us/en/software-developers/intel-txt-software-development-guide.html
- Developing Multi-threaded Applications: A Platform Consistent Approach: https://software.intel.com/sites/default/files/article/147714/51534-developing-multithreaded-applications.pdf
- Using Spin-Loops on Intel® Pentium® 4 Processor and Intel® Xeon® Processor: https://software.intel.com/sites/default/files/22/30/25602
- Performance Monitoring Unit Sharing Guide http://software.intel.com/file/30388

Literature related to selected features in future Intel processors are available at:

- Intel® Architecture Instruction Set Extensions Programming Reference https://software.intel.com/en-us/isa-extensions
- Intel® Software Guard Extensions (Intel® SGX) Programming Reference https://software.intel.com/en-us/isa-extensions/intel-sgx

More relevant links are:

- Intel® Developer Zone:

  https://software.intel.com/en-us
- Developer centers:

  http://www.intel.com/content/www/us/en/hardware-developers/developer-centers.html
- Processor support general link:

  http://www.intel.com/support/processors/
- Intel® Hyper-Threading Technology (Intel® HT Technology):

  http://www.intel.com/technology/platform-technology/hyper-threading/index.htm

6. Updates to Chapter 2, Volume 2A

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

Change to this chapter: Clarification in Sections 2.1.1 "Instruction Prefixes" and 2.3.5.5 "3-byte VEX byte 2, bit[7] - 'W'". Typo correction in Section 2.6.8 "Static Rounding Support in EVEX".

CHAPTER 2
INSTRUCTION FORMAT

This chapter describes the instruction format for all Intel 64 and IA-32 processors. The instruction format for protected mode, real-address mode and virtual-8086 mode is described in Section 2.1. Increments provided for IA-32e mode and its sub-modes are described in Section 2.2.

2.1 INSTRUCTION FORMAT FOR PROTECTED MODE, REAL-ADDRESS MODE, AND VIRTUAL-8086 MODE

The Intel 64 and IA-32 architectures instruction encodings are subsets of the format shown in Figure 2-1. Instructions consist of optional instruction prefixes (in any order), primary opcode bytes (up to three bytes), an addressing-form specifier (if required) consisting of the ModR/M byte and sometimes the SIB (Scale-Index-Base) byte, a displacement (if required), and an immediate data field (if required).

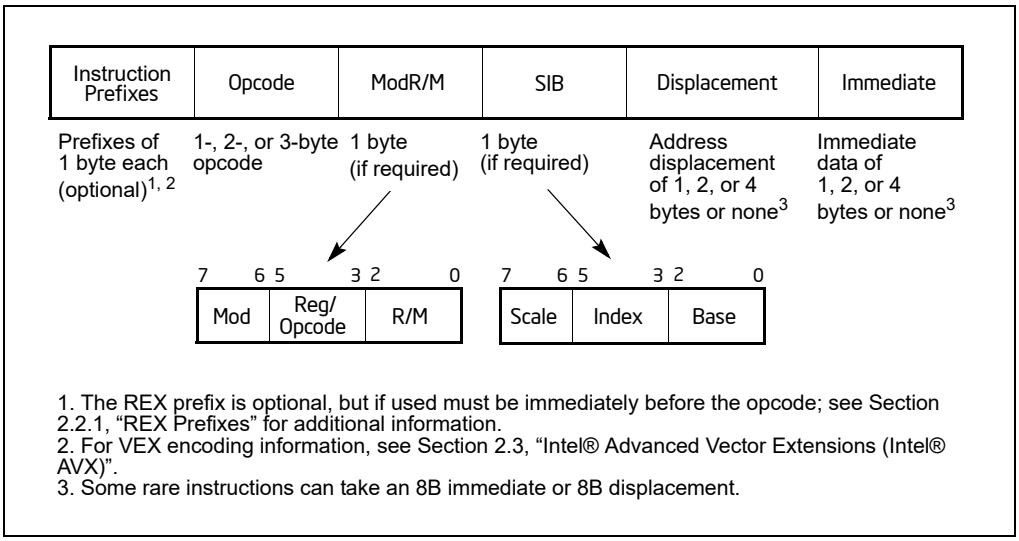


Figure 2-1. Intel 64 and IA-32 Architectures Instruction Format

2.1.1 Instruction Prefixes

Instruction prefixes are divided into four groups, each with a set of allowable prefix codes. For each instruction, it is only useful to include up to one prefix code from each of the four groups (Groups 1, 2, 3, 4). Groups 1 through 4 may be placed in any order relative to each other.

- Group 1
  - — Lock and repeat prefixes:
    - LOCK prefix is encoded using F0H.
    - REPNE/REPNZ prefix is encoded using F2H. Repeat-Not-Zero prefix applies only to string and input/output instructions. (F2H is also used as a mandatory prefix for some instructions.)
    - REP or REPE/REPZ is encoded using F3H. The repeat prefix applies only to string and input/output instructions. F3H is also used as a mandatory prefix for POPCNT, LZCNT and ADOX instructions.

  - BND prefix is encoded using F2H if the following conditions are true:
    - CPUID.(EAX=07H, ECX=0):EBX.MPX[bit 14] is set.
    - BNDCFGU.EN and/or IA32_BNDCFGS.EN is set.
    - When the F2 prefix precedes a near CALL, a near RET, a near JMP, a short Jcc, or a near Jcc instruction (see Chapter 17, "Intel® MPX," of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*).
- Group 2
  - Segment override prefixes:
    - 2EH—CS segment override (use with any branch instruction is reserved).
    - 36H—SS segment override prefix (use with any branch instruction is reserved).
    - 3EH—DS segment override prefix (use with any branch instruction is reserved).
    - 26H—ES segment override prefix (use with any branch instruction is reserved).
    - 64H—FS segment override prefix (use with any branch instruction is reserved).
    - 65H—GS segment override prefix (use with any branch instruction is reserved).
  - Branch hints[1]:
    - 2EH—Branch not taken (used only with J*cc* instructions).
    - 3EH—Branch taken (used only with J*cc* instructions).
- Group 3
    - Operand-size override prefix is encoded using 66H (66H is also used as a mandatory prefix for some instructions).
- Group 4
    - 67H—Address-size override prefix.

The LOCK prefix (F0H) forces an operation that ensures exclusive use of shared memory in a multiprocessor environment. See "LOCK—Assert LOCK# Signal Prefix" in Chapter 3, "Instruction Set Reference, A-L," for a description of this prefix.

Repeat prefixes (F2H, F3H) cause an instruction to be repeated for each element of a string. Use these prefixes only with string and I/O instructions (MOVS, CMPS, SCAS, LODS, STOS, INS, and OUTS). Use of repeat prefixes and/or undefined opcodes with other Intel 64 or IA-32 instructions is reserved; such use may cause unpredictable behavior.

Some instructions may use F2H,F3H as a mandatory prefix to express distinct functionality.

Branch hint prefixes (2EH, 3EH) allow a program to give a hint to the processor about the most likely code path for a branch. Use these prefixes only with conditional branch instructions (J*cc*). Other use of branch hint prefixes and/or other undefined opcodes with Intel 64 or IA-32 instructions is reserved; such use may cause unpredictable behavior.

The operand-size override prefix allows a program to switch between 16- and 32-bit operand sizes. Either size can be the default; use of the prefix selects the non-default size.

Some SSE2/SSE3/SSSE3/SSE4 instructions and instructions using a three-byte sequence of primary opcode bytes may use 66H as a mandatory prefix to express distinct functionality.

Other use of the 66H prefix is reserved; such use may cause unpredictable behavior.

The address-size override prefix (67H) allows programs to switch between 16- and 32-bit addressing. Either size can be the default; the prefix selects the non-default size. Using this prefix and/or other undefined opcodes when operands for the instruction do not reside in memory is reserved; such use may cause unpredictable behavior.

1. Some earlier microarchitectures used these as branch hints, but recent generations have not and they are reserved for future hint usage.

2.1.2 Opcodes

A primary opcode can be 1, 2, or 3 bytes in length. An additional 3-bit opcode field is sometimes encoded in the ModR/M byte. Smaller fields can be defined within the primary opcode. Such fields define the direction of operation, size of displacements, register encoding, condition codes, or sign extension. Encoding fields used by an opcode vary depending on the class of operation.

Two-byte opcode formats for general-purpose and SIMD instructions consist of one of the following:

- An escape opcode byte 0FH as the primary opcode and a second opcode byte.
- A mandatory prefix (66H, F2H, or F3H), an escape opcode byte, and a second opcode byte (same as previous bullet).

For example, CVTDQ2PD consists of the following sequence: F3 0F E6. The first byte is a mandatory prefix (it is not considered as a repeat prefix).

Three-byte opcode formats for general-purpose and SIMD instructions consist of one of the following:

- An escape opcode byte 0FH as the primary opcode, plus two additional opcode bytes.
- A mandatory prefix (66H, F2H, or F3H), an escape opcode byte, plus two additional opcode bytes (same as previous bullet).

For example, PHADDW for XMM registers consists of the following sequence: 66 0F 38 01. The first byte is the mandatory prefix.

Valid opcode expressions are defined in Appendix A and Appendix B.

2.1.3 ModR/M and SIB Bytes

Many instructions that refer to an operand in memory have an addressing-form specifier byte (called the ModR/M byte) following the primary opcode. The ModR/M byte contains three fields of information:

- The *mod* field combines with the r/m field to form 32 possible values: eight registers and 24 addressing modes.
- The *reg/opcode* field specifies either a register number or three more bits of opcode information. The purpose of the reg/opcode field is specified in the primary opcode.
- The *r/m* field can specify a register as an operand or it can be combined with the mod field to encode an addressing mode. Sometimes, certain combinations of the *mod* field and the *r/m* field are used to express opcode information for some instructions.

Certain encodings of the ModR/M byte require a second addressing byte (the SIB byte). The base-plus-index and scale-plus-index forms of 32-bit addressing require the SIB byte. The SIB byte includes the following fields:

- The *scale* field specifies the scale factor.
- The *index* field specifies the register number of the index register.
- The *base* field specifies the register number of the base register.

See Section 2.1.5 for the encodings of the ModR/M and SIB bytes.

2.1.4 Displacement and Immediate Bytes

Some addressing forms include a displacement immediately following the ModR/M byte (or the SIB byte if one is present). If a displacement is required, it can be 1, 2, or 4 bytes.

If an instruction specifies an immediate operand, the operand always follows any displacement bytes. An immediate operand can be 1, 2 or 4 bytes.

2.1.5 Addressing-Mode Encoding of ModR/M and SIB Bytes

The values and corresponding addressing forms of the ModR/M and SIB bytes are shown in Table 2-1 through Table 2-3: 16-bit addressing forms specified by the ModR/M byte are in Table 2-1 and 32-bit addressing forms are in Table 2-2. Table 2-3 shows 32-bit addressing forms specified by the SIB byte. In cases where the reg/opcode field in the ModR/M byte represents an extended opcode, valid encodings are shown in Appendix B.

In Table 2-1 and Table 2-2, the Effective Address column lists 32 effective addresses that can be assigned to the first operand of an instruction by using the Mod and R/M fields of the ModR/M byte. The first 24 options provide ways of specifying a memory location; the last eight (Mod = 11B) provide ways of specifying general-purpose, MMX technology and XMM registers.

The Mod and R/M columns in Table 2-1 and Table 2-2 give the binary encodings of the Mod and R/M fields required to obtain the effective address listed in the first column. For example: see the row indicated by Mod = 11B, R/M = 000B. The row identifies the general-purpose registers EAX, AX or AL; MMX technology register MM0; or XMM register XMM0. The register used is determined by the opcode byte and the operand-size attribute.

Now look at the seventh row in either table (labeled "REG ="). This row specifies the use of the 3-bit Reg/Opcode field when the field is used to give the location of a second operand. The second operand must be a general-purpose, MMX technology, or XMM register. Rows one through five list the registers that may correspond to the value in the table. Again, the register used is determined by the opcode byte along with the operand-size attribute.

If the instruction does not require a second operand, then the Reg/Opcode field may be used as an opcode extension. This use is represented by the sixth row in the tables (labeled "/digit (Opcode)"). Note that values in row six are represented in decimal form.

The body of Table 2-1 and Table 2-2 (under the label "Value of ModR/M Byte (in Hexadecimal)") contains a 32 by 8 array that presents all of 256 values of the ModR/M byte (in hexadecimal). Bits 3, 4 and 5 are specified by the column of the table in which a byte resides. The row specifies bits 0, 1 and 2; and bits 6 and 7. The figure below demonstrates interpretation of one table value.

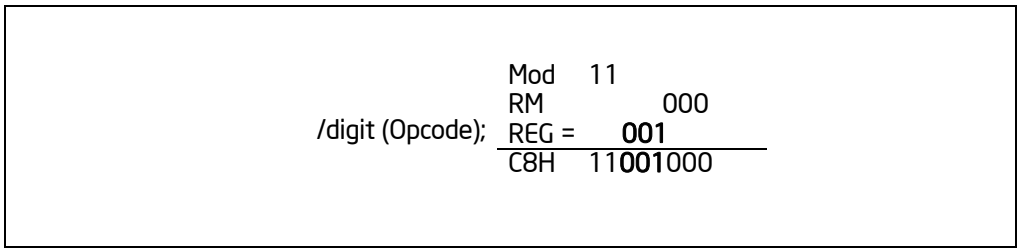


Figure 2-2. Table Interpretation of ModR/M Byte (C8H)

Table 2-1. 16-Bit Addressing Forms with the ModR/M Byte

r8(/r) r16(/r) r32(/r) mm(/r) xmm(/r) (In decimal) /digit (Opcode) (In binary) REG =			AL AX EAX MM0 XMM0 0 000	CL CX ECX MM1 XMM1 1 001	DL DX EDX MM2 XMM2 2 010	BL BX EBX MM3 XMM3 3 011	AH SP ESP MM4 XMM4 4 100	CH BP[1] EBP MM5 XMM5 5 101	DH SI ESI MM6 XMM6 6 110	BH DI EDI MM7 XMM7 7 111
Effective Address	**Mod**	**R/M**	**Value of ModR/M Byte (in Hexadecimal)**							
[BX+SI] [BX+DI] [BP+SI] [BP+DI] [SI] [DI] disp16[2] [BX]	00	000 001 010 011 100 101 110 111	00 01 02 03 04 05 06 07	08 09 0A 0B 0C 0D 0E 0F	10 11 12 13 14 15 16 17	18 19 1A 1B 1C 1D 1E 1F	20 21 22 23 24 25 26 27	28 29 2A 2B 2C 2D 2E 2F	30 31 32 33 34 35 36 37	38 39 3A 3B 3C 3D 3E 3F
[BX+SI]+disp8[3] [BX+DI]+disp8 [BP+SI]+disp8 [BP+DI]+disp8 [SI]+disp8 [DI]+disp8 [BP]+disp8 [BX]+disp8	01	000 001 010 011 100 101 110 111	40 41 42 43 44 45 46 47	48 49 4A 4B 4C 4D 4E 4F	50 51 52 53 54 55 56 57	58 59 5A 5B 5C 5D 5E 5F	60 61 62 63 64 65 66 67	68 69 6A 6B 6C 6D 6E 6F	70 71 72 73 74 75 76 77	78 79 7A 7B 7C 7D 7E 7F
[BX+SI]+disp16 [BX+DI]+disp16 [BP+SI]+disp16 [BP+DI]+disp16 [SI]+disp16 [DI]+disp16 [BP]+disp16 [BX]+disp16	10	000 001 010 011 100 101 110 111	80 81 82 83 84 85 86 87	88 89 8A 8B 8C 8D 8E 8F	90 91 92 93 94 95 96 97	98 99 9A 9B 9C 9D 9E 9F	A0 A1 A2 A3 A4 A5 A6 A7	A8 A9 AA AB AC AD AE AF	B0 B1 B2 B3 B4 B5 B6 B7	B8 B9 BA BB BC BD BE BF
EAX/AX/AL/MM0/XMM0 ECX/CX/CL/MM1/XMM1 EDX/DX/DL/MM2/XMM2 EBX/BX/BL/MM3/XMM3 ESP/SP/AHMM4/XMM4 EBP/BP/CH/MM5/XMM5 ESI/SI/DH/MM6/XMM6 EDI/DI/BH/MM7/XMM7	11	000 001 010 011 100 101 110 111	C0 C1 C2 C3 C4 C5 C6 C7	C8 C9 CA CB CC CD CE CF	D0 D1 D2 D3 D4 D5 D6 D7	D8 D9 DA DB DC DD DE DF	E0 EQ E2 E3 E4 E5 E6 E7	E8 E9 EA EB EC ED EE EF	F0 F1 F2 F3 F4 F5 F6 F7	F8 F9 FA FB FC FD FE FF

NOTES:

1. The default segment register is SS for the effective addresses containing a BP index, DS for other effective addresses.
2. The disp16 nomenclature denotes a 16-bit displacement that follows the ModR/M byte and that is added to the index.
3. The disp8 nomenclature denotes an 8-bit displacement that follows the ModR/M byte and that is sign-extended and added to the index.

Table 2-2. 32-Bit Addressing Forms with the ModR/M Byte

r8(/r) r16(/r) r32(/r) mm(/r) xmm(/r) (In decimal) /digit (Opcode) (In binary) REG =			AL AX EAX MM0 XMM0 0 000	CL CX ECX MM1 XMM1 1 001	DL DX EDX MM2 XMM2 2 010	BL BX EBX MM3 XMM3 3 011	AH SP ESP MM4 XMM4 4 100	CH BP EBP MM5 XMM5 5 101	DH SI ESI MM6 XMM6 6 110	BH DI EDI MM7 XMM7 7 111
Effective Address	**Mod**	**R/M**	**Value of ModR/M Byte (in Hexadecimal)**							
[EAX] [ECX] [EDX] [EBX] [--][--][1] disp32[2] [ESI] [EDI]	00	000 001 010 011 100 101 110 111	00 01 02 03 04 05 06 07	08 09 0A 0B 0C 0D 0E 0F	10 11 12 13 14 15 16 17	18 19 1A 1B 1C 1D 1E 1F	20 21 22 23 24 25 26 27	28 29 2A 2B 2C 2D 2E 2F	30 31 32 33 34 35 36 37	38 39 3A 3B 3C 3D 3E 3F
[EAX]+disp8[3] [ECX]+disp8 [EDX]+disp8 [EBX]+disp8 [--][--]+disp8 [EBP]+disp8 [ESI]+disp8 [EDI]+disp8	01	000 001 010 011 100 101 110 111	40 41 42 43 44 45 46 47	48 49 4A 4B 4C 4D 4E 4F	50 51 52 53 54 55 56 57	58 59 5A 5B 5C 5D 5E 5F	60 61 62 63 64 65 66 67	68 69 6A 6B 6C 6D 6E 6F	70 71 72 73 74 75 76 77	78 79 7A 7B 7C 7D 7E 7F
[EAX]+disp32 [ECX]+disp32 [EDX]+disp32 [EBX]+disp32 [--][--]+disp32 [EBP]+disp32 [ESI]+disp32 [EDI]+disp32	10	000 001 010 011 100 101 110 111	80 81 82 83 84 85 86 87	88 89 8A 8B 8C 8D 8E 8F	90 91 92 93 94 95 96 97	98 99 9A 9B 9C 9D 9E 9F	A0 A1 A2 A3 A4 A5 A6 A7	A8 A9 AA AB AC AD AE AF	B0 B1 B2 B3 B4 B5 B6 B7	B8 B9 BA BB BC BD BE BF
EAX/AX/AL/MM0/XMM0 ECX/CX/CL/MM/XMM1 EDX/DX/DL/MM2/XMM2 EBX/BX/BL/MM3/XMM3 ESP/SP/AH/MM4/XMM4 EBP/BP/CH/MM5/XMM5 ESI/SI/DH/MM6/XMM6 EDI/DI/BH/MM7/XMM7	11	000 001 010 011 100 101 110 111	C0 C1 C2 C3 C4 C5 C6 C7	C8 C9 CA CB CC CD CE CF	D0 D1 D2 D3 D4 D5 D6 D7	D8 D9 DA DB DC DD DE DF	E0 E1 E2 E3 E4 E5 E6 E7	E8 E9 EA EB EC ED EE EF	F0 F1 F2 F3 F4 F5 F6 F7	F8 F9 FA FB FC FD FE FF

NOTES:

1. The [--][--] nomenclature means a SIB follows the ModR/M byte.
2. The disp32 nomenclature denotes a 32-bit displacement that follows the ModR/M byte (or the SIB byte if one is present) and that is added to the index.
3. The disp8 nomenclature denotes an 8-bit displacement that follows the ModR/M byte (or the SIB byte if one is present) and that is sign-extended and added to the index.

Table 2-3 is organized to give 256 possible values of the SIB byte (in hexadecimal). General purpose registers used as a base are indicated across the top of the table, along with corresponding values for the SIB byte's base field. Table rows in the body of the table indicate the register used as the index (SIB byte bits 3, 4 and 5) and the scaling factor (determined by SIB byte bits 6 and 7).

Table 2-3. 32-Bit Addressing Forms with the SIB Byte

r32 (In decimal) Base = (In binary) Base =			EAX 0 000	ECX 1 001	EDX 2 010	EBX 3 011	ESP 4 100	[*] 5 101	ESI 6 110	EDI 7 111
Scaled Index	**SS**	**Index**	**Value of SIB Byte (in Hexadecimal)**							
[EAX] [ECX] [EDX] [EBX] none [EBP] [ESI] [EDI]	00	000 001 010 011 100 101 110 111	00 08 10 18 20 28 30 38	01 09 11 19 21 29 31 39	02 0A 12 1A 22 2A 32 3A	03 0B 13 1B 23 2B 33 3B	04 0C 14 1C 24 2C 34 3C	05 0D 15 1D 25 2D 35 3D	06 0E 16 1E 26 2E 36 3E	07 0F 17 1F 27 2F 37 3F
[EAX*2] [ECX*2] [EDX*2] [EBX*2] none [EBP*2] [ESI*2] [EDI*2]	01	000 001 010 011 100 101 110 111	40 48 50 58 60 68 70 78	41 49 51 59 61 69 71 79	42 4A 52 5A 62 6A 72 7A	43 4B 53 5B 63 6B 73 7B	44 4C 54 5C 64 6C 74 7C	45 4D 55 5D 65 6D 75 7D	46 4E 56 5E 66 6E 76 7E	47 4F 57 5F 67 6F 77 7F
[EAX*4] [ECX*4] [EDX*4] [EBX*4] none [EBP*4] [ESI*4] [EDI*4]	10	000 001 010 011 100 101 110 111	80 88 90 98 A0 A8 B0 B8	81 89 91 99 A1 A9 B1 B9	82 8A 92 9A A2 AA B2 BA	83 8B 93 9B A3 AB B3 BB	84 8C 94 9C A4 AC B4 BC	85 8D 95 9D A5 AD B5 BD	86 8E 96 9E A6 AE B6 BE	87 8F 97 9F A7 AF B7 BF
[EAX*8] [ECX*8] [EDX*8] [EBX*8] none [EBP*8] [ESI*8] [EDI*8]	11	000 001 010 011 100 101 110 111	C0 C8 D0 D8 E0 E8 F0 F8	C1 C9 D1 D9 E1 E9 F1 F9	C2 CA D2 DA E2 EA F2 FA	C3 CB D3 DB E3 EB F3 FB	C4 CC D4 DC E4 EC F4 FC	C5 CD D5 DD E5 ED F5 FD	C6 CE D6 DE E6 EE F6 FE	C7 CF D7 DF E7 EF F7 FF

NOTES:

1. The [*] nomenclature means a disp32 with no base if the MOD is 00B. Otherwise, [*] means disp8 or disp32 + [EBP]. This provides the following address modes:

MOD bits	Effective Address
00	[scaled index] + disp32
01	[scaled index] + disp8 + [EBP]
10	[scaled index] + disp32 + [EBP]

2.2 IA-32E MODE

IA-32e mode has two sub-modes. These are:

- **Compatibility Mode.** Enables a 64-bit operating system to run most legacy protected mode software unmodified.
- **64-Bit Mode.** Enables a 64-bit operating system to run applications written to access 64-bit address space.

2.2.1 REX Prefixes

REX prefixes are instruction-prefix bytes used in 64-bit mode. They do the following:

- Specify GPRs and SSE registers.
- Specify 64-bit operand size.
- Specify extended control registers.

Not all instructions require a REX prefix in 64-bit mode. A prefix is necessary only if an instruction references one of the extended registers or uses a 64-bit operand. If a REX prefix is used when it has no meaning, it is ignored.

Only one REX prefix is allowed per instruction. If used, the REX prefix byte must immediately precede the opcode byte or the escape opcode byte (0FH). When a REX prefix is used in conjunction with an instruction containing a mandatory prefix, the mandatory prefix must come before the REX so the REX prefix can be immediately preceding the opcode or the escape byte. For example, CVTDQ2PD with a REX prefix should have REX placed between F3 and 0F E6. Other placements are ignored. The instruction-size limit of 15 bytes still applies to instructions with a REX prefix. See Figure 2-3.

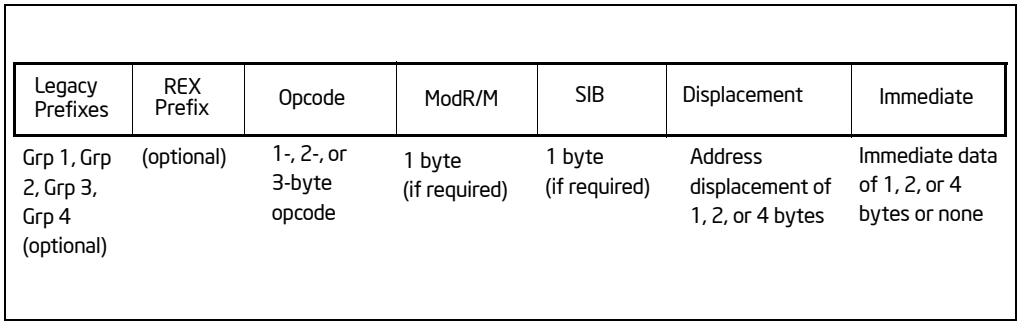


Figure 2-3. Prefix Ordering in 64-bit Mode

2.2.1.1 Encoding

Intel 64 and IA-32 instruction formats specify up to three registers by using 3-bit fields in the encoding, depending on the format:

- ModR/M: the reg and r/m fields of the ModR/M byte.
- ModR/M with SIB: the reg field of the ModR/M byte, the base and index fields of the SIB (scale, index, base) byte.
- Instructions without ModR/M: the reg field of the opcode.

In 64-bit mode, these formats do not change. Bits needed to define fields in the 64-bit context are provided by the addition of REX prefixes.

2.2.1.2 More on REX Prefix Fields

REX prefixes are a set of 16 opcodes that span one row of the opcode map and occupy entries 40H to 4FH. These opcodes represent valid instructions (INC or DEC) in IA-32 operating modes and in compatibility mode. In 64-bit mode, the same opcodes represent the instruction prefix REX and are not treated as individual instructions.

The single-byte-opcode forms of the INC/DEC instructions are not available in 64-bit mode. INC/DEC functionality is still available using ModR/M forms of the same instructions (opcodes FF/0 and FF/1).

See Table 2-4 for a summary of the REX prefix format. Figure 2-4 though Figure 2-7 show examples of REX prefix fields in use. Some combinations of REX prefix fields are invalid. In such cases, the prefix is ignored. Some additional information follows:

- Setting REX.W can be used to determine the operand size but does not solely determine operand width. Like the 66H size prefix, 64-bit operand size override has no effect on byte-specific operations.
- For non-byte operations: if a 66H prefix is used with prefix (REX.W = 1), 66H is ignored.
- If a 66H override is used with REX and REX.W = 0, the operand size is 16 bits.

- REX.R modifies the ModR/M reg field when that field encodes a GPR, SSE, control or debug register. REX.R is ignored when ModR/M specifies other registers or defines an extended opcode.
- REX.X bit modifies the SIB index field.
- REX.B either modifies the base in the ModR/M r/m field or SIB base field; or it modifies the opcode reg field used for accessing GPRs.

Table 2-4. REX Prefix Fields [BITS: 0100WRXB]

Field Name	Bit Position	Definition
-	7:4	0100
W	3	0 = Operand size determined by CS.D
		1 = 64 Bit Operand Size
R	2	Extension of the ModR/M reg field
X	1	Extension of the SIB index field
B	0	Extension of the ModR/M r/m field, SIB base field, or Opcode reg field

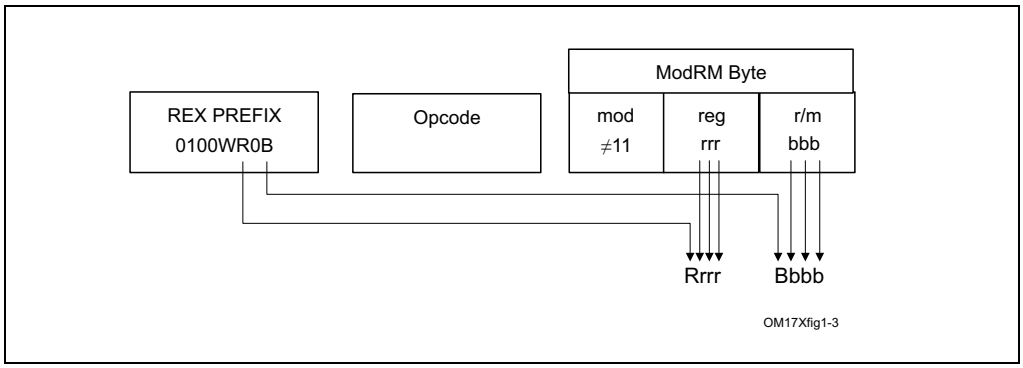


Figure 2-4. Memory Addressing Without an SIB Byte; REX.X Not Used

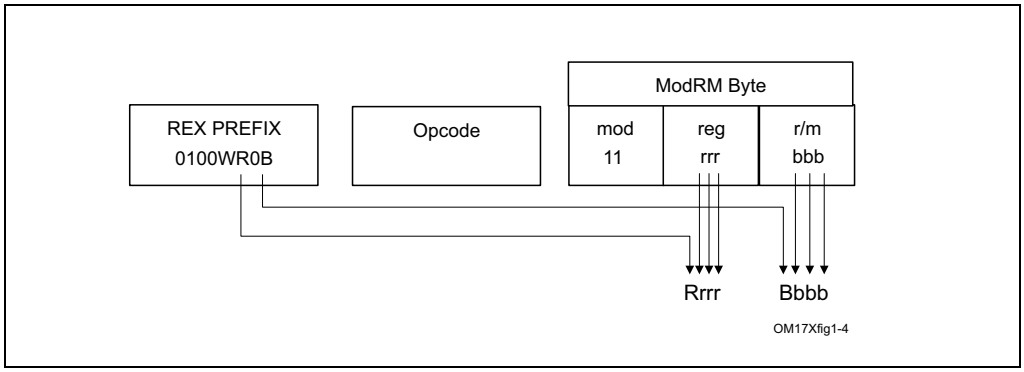


Figure 2-5. Register-Register Addressing (No Memory Operand); REX.X Not Used

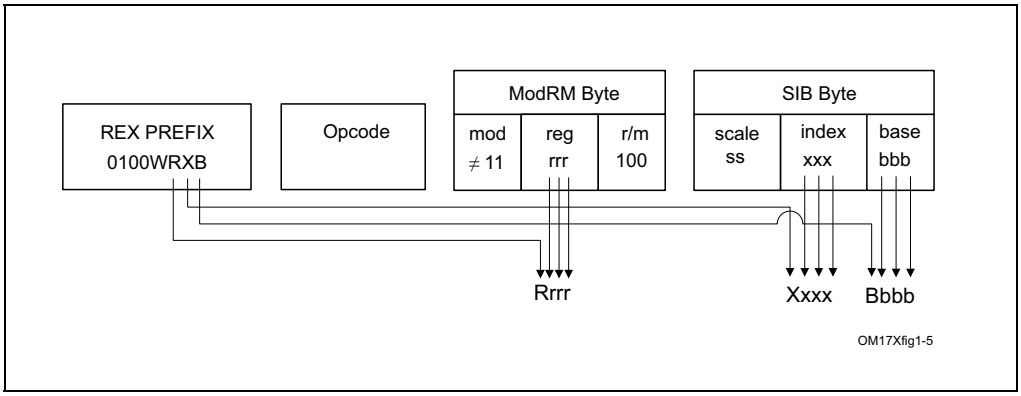


Figure 2-6. Memory Addressing With a SIB Byte

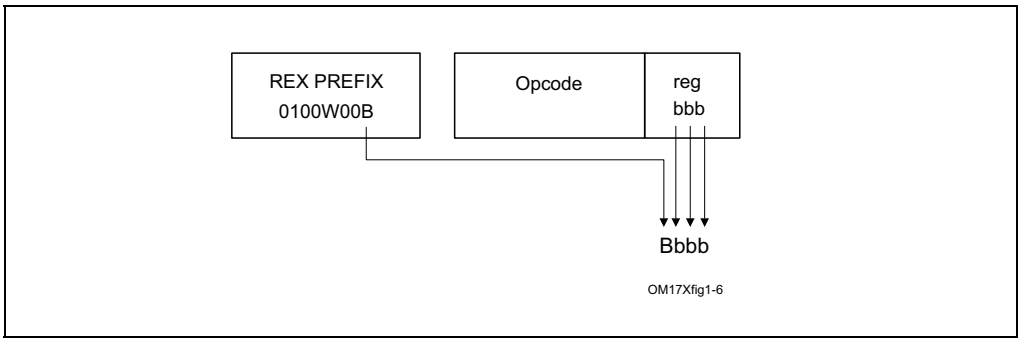


Figure 2-7. Register Operand Coded in Opcode Byte; REX.X & REX.R Not Used

In the IA-32 architecture, byte registers (AH, AL, BH, BL, CH, CL, DH, and DL) are encoded in the ModR/M byte's reg field, the r/m field or the opcode reg field as registers 0 through 7. REX prefixes provide an additional addressing capability for byte-registers that makes the least-significant byte of GPRs available for byte operations.

Certain combinations of the fields of the ModR/M byte and the SIB byte have special meaning for register encodings. For some combinations, fields expanded by the REX prefix are not decoded. Table 2-5 describes how each case behaves.

Table 2-5. Special Cases of REX Encodings

ModR/M or SIB	Sub-field Encodings	Compatibility Mode Operation	Compatibility Mode Implications	Additional Implications
ModR/M Byte	mod ≠ 11 r/m = b*100(ESP)	SIB byte present.	SIB byte required for ESP-based addressing.	REX prefix adds a fourth bit (b) which is not decoded (don't care). SIB byte also required for R12-based addressing.
ModR/M Byte	mod = 0 r/m = b*101(EBP)	Base register not used.	EBP without a displacement must be done using mod = 01 with displacement of 0.	REX prefix adds a fourth bit (b) which is not decoded (don't care). Using RBP or R13 without displacement must be done using mod = 01 with a displacement of 0.
SIB Byte	index = 0100(ESP)	Index register not used.	ESP cannot be used as an index register.	REX prefix adds a fourth bit (b) which is decoded. There are no additional implications. The expanded index field allows distinguishing RSP from R12, therefore R12 can be used as an index.
SIB Byte	base = 0101(EBP)	Base register is unused if mod = 0.	Base register depends on mod encoding.	REX prefix adds a fourth bit (b) which is not decoded. This requires explicit displacement to be used with EBP/RBP or R13.

NOTES:

* Don't care about value of REX.B

2.2.1.3 Displacement

Addressing in 64-bit mode uses existing 32-bit ModR/M and SIB encodings. The ModR/M and SIB displacement sizes do not change. They remain 8 bits or 32 bits and are sign-extended to 64 bits.

2.2.1.4 Direct Memory-Offset MOVs

In 64-bit mode, direct memory-offset forms of the MOV instruction are extended to specify a 64-bit immediate absolute address. This address is called a moffset. No prefix is needed to specify this 64-bit memory offset. For these MOV instructions, the size of the memory offset follows the address-size default (64 bits in 64-bit mode). See Table 2-6.

Table 2-6. Direct Memory Offset Form of MOV

Opcode	Instruction
A0	MOV AL, moffset
A1	MOV EAX, moffset
A2	MOV moffset, AL
A3	MOV moffset, EAX

2.2.1.5 Immediates

In 64-bit mode, the typical size of immediate operands remains 32 bits. When the operand size is 64 bits, the processor sign-extends all immediates to 64 bits prior to their use.

Support for 64-bit immediate operands is accomplished by expanding the semantics of the existing move (MOV reg, imm16/32) instructions. These instructions (opcodes B8H – BFH) move 16-bits or 32-bits of immediate data (depending on the effective operand size) into a GPR. When the effective operand size is 64 bits, these instructions can be used to load an immediate into a GPR. A REX prefix is needed to override the 32-bit default operand size to a 64-bit operand size.

For example:

```
48 B8   8877665544332211   MOV RAX,1122334455667788H
```

2.2.1.6 RIP-Relative Addressing

A new addressing form, RIP-relative (relative instruction-pointer) addressing, is implemented in 64-bit mode. An effective address is formed by adding displacement to the 64-bit RIP of the next instruction.

In IA-32 architecture and compatibility mode, addressing relative to the instruction pointer is available only with control-transfer instructions. In 64-bit mode, instructions that use ModR/M addressing can use RIP-relative addressing. Without RIP-relative addressing, all ModR/M modes address memory relative to zero.

RIP-relative addressing allows specific ModR/M modes to address memory relative to the 64-bit RIP using a signed 32-bit displacement. This provides an offset range of ±2GB from the RIP. Table 2-7 shows the ModR/M and SIB encodings for RIP-relative addressing. Redundant forms of 32-bit displacement-addressing exist in the current ModR/M and SIB encodings. There is one ModR/M encoding and there are several SIB encodings. RIP-relative addressing is encoded using a redundant form.

In 64-bit mode, the ModR/M Disp32 (32-bit displacement) encoding is re-defined to be RIP+Disp32 rather than displacement-only. See Table 2-7.

Table 2-7. RIP-Relative Addressing

ModR/M and SIB Sub-field Encodings		Compatibility Mode Operation	64-bit Mode Operation	Additional Implications in 64-bit mode
ModR/M Byte	mod = 00	Disp32	RIP + Disp32	Must use SIB form with normal (zero-based) displacement addressing
	r/m = 101 (none)			
SIB Byte	base = 101 (none)	if mod = 00, Disp32	Same as legacy	None
	index = 100 (none)			
	scale = 0, 1, 2, 4			

The ModR/M encoding for RIP-relative addressing does not depend on using a prefix. Specifically, the r/m bit field encoding of 101B (used to select RIP-relative addressing) is not affected by the REX prefix. For example, selecting R13 (REX.B = 1, r/m = 101B) with mod = 00B still results in RIP-relative addressing. The 4-bit r/m field of REX.B combined with ModR/M is not fully decoded. In order to address R13 with no displacement, software must encode R13 + 0 using a 1-byte displacement of zero.

RIP-relative addressing is enabled by 64-bit mode, not by a 64-bit address-size. The use of the address-size prefix does not disable RIP-relative addressing. The effect of the address-size prefix is to truncate and zero-extend the computed effective address to 32 bits.

2.2.1.7 Default 64-Bit Operand Size

In 64-bit mode, two groups of instructions have a default operand size of 64 bits (do not need a REX prefix for this operand size). These are:

- Near branches.
- All instructions, except far branches, that implicitly reference the RSP.

2.2.2 Additional Encodings for Control and Debug Registers

In 64-bit mode, more encodings for control and debug registers are available. The REX.R bit is used to modify the ModR/M reg field when that field encodes a control or debug register (see Table 2-4). These encodings enable the processor to address CR8-CR15 and DR8- DR15. An additional control register (CR8) is defined in 64-bit mode. CR8 becomes the Task Priority Register (TPR).

In the first implementation of IA-32e mode, CR9-CR15 and DR8-DR15 are not implemented. Any attempt to access unimplemented registers results in an invalid-opcode exception (#UD).

2.3 INTEL® ADVANCED VECTOR EXTENSIONS (INTEL® AVX)

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

2.3.1 Instruction Format

Instruction encoding using VEX prefix provides several advantages:

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

Figure 2-8 shows the Intel 64 instruction encoding format with VEX prefix support. Legacy instruction without a VEX prefix is fully supported and unchanged. The use of VEX prefix in an Intel 64 instruction is optional, but a VEX prefix is required for Intel 64 instructions that operate on YMM registers or support three and four operand syntax. VEX prefix is not a constant-valued, "single-purpose" byte like 0FH, 66H, F2H, F3H in legacy SSE instructions. VEX prefix provides substantially richer capability than the REX prefix.

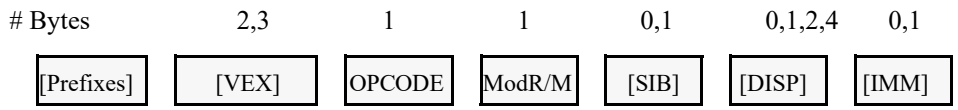


Figure 2-8. Instruction Encoding Format with VEX Prefix

2.3.2 VEX and the LOCK prefix

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

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

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

2.3.4 VEX and the REX prefix

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

2.3.5 The VEX Prefix

The VEX prefix is encoded in either the two-byte form (the first byte must be C5H) or in the three-byte form (the first byte must be C4H). The two-byte VEX is used mainly for 128-bit, scalar, and the most common 256-bit AVX instructions; while the three-byte VEX provides a compact replacement of REX and 3-byte opcode instructions (including AVX and FMA instructions). Beyond the first byte of the VEX prefix, it consists of a number of bit fields providing specific capability, they are shown in Figure 2-9.

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

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

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

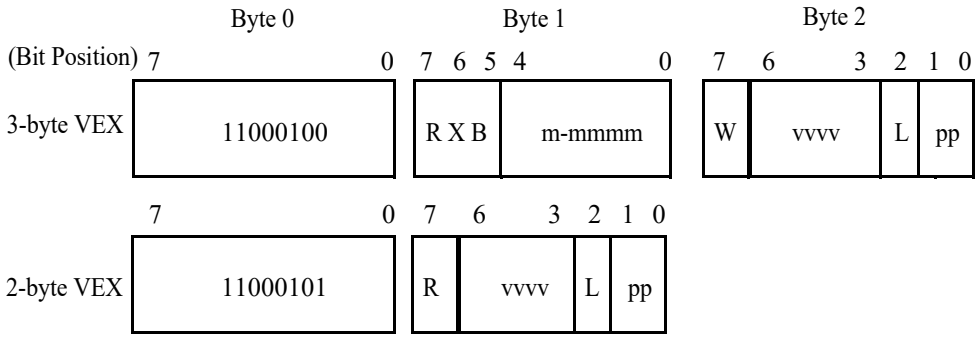
The VEX prefix is required to be the last prefix and immediately precedes the opcode bytes. It must follow any other prefixes. If VEX prefix is present a REX prefix is not supported.

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

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

The VEX prefix supports 256-bit versions of floating-point SSE, SSE2, SSE3, and SSE4 instructions. Note, certain new instruction functionality can only be encoded with the VEX prefix.

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



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

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

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

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

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

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

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

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

Figure 2-9. VEX bit fields

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

2.3.5.1 VEX Byte 0, bits[7:0]

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

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

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

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

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

This bit is stored in bit inverted format.

2.3.5.3 3-byte VEX byte 1, bit[6] - 'X'

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

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

This bit is stored in bit inverted format.

2.3.5.4 3-byte VEX byte 1, bit[5] - 'B'

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

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

This bit is stored in bit inverted format.

2.3.5.5 3-byte VEX byte 2, bit[7] - 'W'

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

- For AVX instructions that have equivalent legacy SSE instructions (typically these SSE instructions have a general-purpose register operand with its operand size attribute promotable by REX.W), if REX.W promotes the operand size attribute of the general-purpose register operand in legacy SSE instruction, VEX.W has same meaning in the corresponding AVX equivalent form. In 32-bit modes for these instructions, VEX.W is silently ignored.
- For AVX instructions that have equivalent legacy SSE instructions (typically these SSE instructions have operands with their operand size attribute fixed and not promotable by REX.W), if REX.W is don't care in legacy SSE instruction, VEX.W is ignored in the corresponding AVX equivalent form irrespective of mode.
- For new AVX instructions where VEX.W has no defined function (typically these meant the combination of the opcode byte and VEX.mmmmm did not have any equivalent SSE functions), VEX.W is reserved as zero and setting to other than zero will cause instruction to #UD.

2.3.5.6 2-byte VEX Byte 1, bits[6:3] and 3-byte VEX Byte 2, bits [6:3]- 'vvvv' the Source or Dest Register Specifier

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

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

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

In 64-bit mode all 4 bits may be used. See Table 2-8 for the encoding of the XMM or YMM registers. In 32-bit and 16-bit modes bit 6 must be 1 (if bit 6 is not 1, the 2-byte VEX version will generate LDS instruction and the 3-byte VEX version will ignore this bit).

Table 2-8. VEX.vvvv to register name mapping

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

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

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

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

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

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

- VEX.vvvv encodes the first source register operand, specified in inverted (1's complement) form and is valid for instructions with 2 or more source operands.
- VEX.vvvv encodes the destination register operand, specified in 1's complement form for certain vector shifts. The instructions where VEX.vvvv is used as a destination are listed in Table 2-9. The notation in the "Opcode" column in Table 2-9 is described in detail in section 3.1.1.
- VEX.vvvv does not encode any operand, the field is reserved and should contain 1111b.

Table 2-9. Instructions with a VEX.vvvv destination

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

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

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

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

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

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

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

2.3.6.1 3-byte VEX byte 1, bits[4:0] - "m-mmmm"

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

Table 2-10. VEX.m-mmmm interpretation

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

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

2.3.6.2 2-byte VEX byte 1, bit[2], and 3-byte VEX byte 2, bit [2]- "L"

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

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

See the following table.

Table 2-11. VEX.L interpretation

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

2.3.6.3 2-byte VEX byte 1, bits[1:0], and 3-byte VEX byte 2, bits [1:0]- "pp"

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

See the following table.

Table 2-12. VEX.pp interpretation

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

2.3.7 The Opcode Byte

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

2.3.8 The MODRM, SIB, and Displacement Bytes

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

2.3.9 The Third Source Operand (Immediate Byte)

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

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

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

2.3.10.1 Vector Length Transition and Programming Considerations

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

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

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

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

Processors that support further vector-register extensions (defining bits beyond bit 255) will also extend the XSAVE and XRSTOR instructions to save and restore these extensions. To ensure forward compatibility, software that handles involuntary calls and that uses instructions encoded with VEX.128 and VEX.256 prefixes should first save and then restore the vector registers (with any extensions) using the XSAVE and XRSTOR instructions with save/restore masks that set bits that correspond to all vector-register extensions. Ideally, software should rely on a mechanism that is cognizant of which bits to set. (E.g., an OS mechanism that sets the save/restore mask bits for all vector-register extensions that are enabled in XCR0.) Saving and restoring state with instructions other than XSAVE and XRSTOR will, on future processors with wider vector registers, corrupt the extended state of the vector registers - even if doing so functions correctly on processors supporting 256-bit vector registers. (The same is true

if XSAVE and XRSTOR are used with a save/restore mask that does not set bits corresponding to all supported extensions to the vector registers.)

2.3.11 AVX Instruction Length

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

2.3.12 Vector SIB (VSIB) Memory Addressing

In Intel® Advanced Vector Extensions 2 (Intel® AVX2), an SIB byte that follows the ModR/M byte can support VSIB memory addressing to an array of linear addresses. VSIB addressing is only supported in a subset of Intel AVX2 instructions. VSIB memory addressing requires 32-bit or 64-bit effective address. In 32-bit mode, VSIB addressing is not supported when address size attribute is overridden to 16 bits. In 16-bit protected mode, VSIB memory addressing is permitted if address size attribute is overridden to 32 bits. Additionally, VSIB memory addressing is supported only with VEX prefix.

In VSIB memory addressing, the SIB byte consists of:

- The scale field (bit 7:6) specifies the scale factor.
- The index field (bits 5:3) specifies the register number of the vector index register, each element in the vector register specifies an index.
- The base field (bits 2:0) specifies the register number of the base register.

Table 2-3 shows the 32-bit VSIB addressing form. It is organized to give 256 possible values of the SIB byte (in hexadecimal). General purpose registers used as a base are indicated across the top of the table, along with corresponding values for the SIB byte's base field. The register names also include R8L-R15L applicable only in 64-bit mode (when address size override prefix is used, but the value of VEX.B is not shown in Table 2-3). In 32-bit mode, R8L-R15L does not apply.

Table rows in the body of the table indicate the vector index register used as the index field and each supported scaling factor shown separately. Vector registers used in the index field can be XMM or YMM registers. The left-most column includes vector registers VR8-VR15 (i.e. XMM8/YMM8-XMM15/YMM15), which are only available in 64-bit mode and does not apply if encoding in 32-bit mode.

Table 2-13. 32-Bit VSIB Addressing Forms of the SIB Byte

r32 (In decimal) Base = (In binary) Base =				EAX/ R8L 0 000	ECX/ R9L 1 001	EDX/ R10L 2 010	EBX/ R11L 3 011	ESP/ R12L 4 100	EBP/ R13L[1] 5 101	ESI/ R14L 6 110	EDI/ R15L 7 111
Scaled Index		**SS**	**Index**	**Value of SIB Byte (in Hexadecimal)**							
VR0/VR8 VR1/VR9 VR2/VR10 VR3/VR11 VR4/VR12 VR5/VR13 VR6/VR14 VR7/VR15	*1	00	000 001 010 011 100 101 110 111	00 08 10 18 20 28 30 38	01 09 11 19 21 29 31 39	02 0A 12 1A 22 2A 32 3A	03 0B 13 1B 23 2B 33 3B	04 0C 14 1C 24 2C 34 3C	05 0D 15 1D 25 2D 35 3D	06 0E 16 1E 26 2E 36 3E	07 0F 17 1F 27 2F 37 3F
VR0/VR8 VR1/VR9 VR2/VR10 VR3/VR11 VR4/VR12 VR5/VR13 VR6/VR14 VR7/VR15	*2	01	000 001 010 011 100 101 110 111	40 48 50 58 60 68 70 78	41 49 51 59 61 69 71 79	42 4A 52 5A 62 6A 72 7A	43 4B 53 5B 63 6B 73 7B	44 4C 54 5C 64 6C 74 7C	45 4D 55 5D 65 6D 75 7D	46 4E 56 5E 66 6E 76 7E	47 4F 57 5F 67 6F 77 7F

Table 2-13. 32-Bit VSIB Addressing Forms of the SIB Byte (Contd.)

VR0/VR8 VR1/VR9 VR2/VR10 VR3/VR11 VR4/VR12 VR5/VR13 VR6/VR14 VR7/VR15	*4	10	000 001 010 011 100 101 110 111	80 88 90 98 A0 A8 B0 B8	81 89 91 89 A1 A9 B1 B9	82 8A 92 9A A2 AA B2 BA	83 8B 93 9B A3 AB B3 BB	84 8C 94 9C A4 AC B4 BC	85 8D 95 9D A5 AD B5 BD	86 8E 96 9E A6 AE B6 BE	87 8F 97 9F A7 AF B7 BF
VR0/VR8 VR1/VR9 VR2/VR10 VR3/VR11 VR4/VR12 VR5/VR13 VR6/VR14 VR7/VR15	*8	11	000 001 010 011 100 101 110 111	C0 C8 D0 D8 E0 E8 F0 F8	C1 C9 D1 D9 E1 E9 F1 F9	C2 CA D2 DA E2 EA F2 FA	C3 CB D3 DB E3 EB F3 FB	C4 CC D4 DC E4 EC F4 FC	C5 CD D5 DD E5 ED F5 FD	C6 CE D6 DE E6 EE F6 FE	C7 CF D7 DF E7 EF F7 FF

NOTES:

1. If ModR/M.mod = 00b, the base address is zero, then effective address is computed as [scaled vector index] + disp32. Otherwise the base address is computed as [EBP/R13]+ disp, the displacement is either 8 bit or 32 bit depending on the value of ModR/M.mod:

MOD	Effective Address
00b	[Scaled Vector Register] + Disp32
01b	[Scaled Vector Register] + Disp8 + [EBP/R13]
10b	[Scaled Vector Register] + Disp32 + [EBP/R13]

2.3.12.1 64-bit Mode VSIB Memory Addressing

In 64-bit mode VSIB memory addressing uses the VEX.B field and the base field of the SIB byte to encode one of the 16 general-purpose register as the base register. The VEX.X field and the index field of the SIB byte encode one of the 16 vector registers as the vector index register.

In 64-bit mode the top row of Table 2-13 base register should be interpreted as the full 64-bit of each register.

2.4 AVX AND SSE INSTRUCTION EXCEPTION SPECIFICATION

To look up the exceptions of legacy 128-bit SIMD instruction, 128-bit VEX-encoded instructions, and 256-bit VEX-encoded instruction, Table 2-14 summarizes the exception behavior into separate classes, with detailed exception conditions defined in sub-sections 2.4.1 through 2.5.1. For example, ADDPS contains the entry:

"See Exceptions Type 2"

In this entry, *"Type2"* can be looked up in Table 2-14.

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

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

NOTE

Instructions that operate only with MMX, X87, or general-purpose registers are not covered by the exception classes defined in this section. For instructions that operate on MMX registers, see Section 22.25.3, "Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3B*.

Table 2-14. Exception class description

Exception Class	Instruction set	Mem arg	Floating-Point Exceptions (#XM)
Type 1	AVX, Legacy SSE	16/32 byte explicitly aligned	None
Type 2	AVX, Legacy SSE	16/32 byte not explicitly aligned	Yes
Type 3	AVX, Legacy SSE	< 16 byte	Yes
Type 4	AVX, Legacy SSE	16/32 byte not explicitly aligned	No
Type 5	AVX, Legacy SSE	< 16 byte	No
Type 6	AVX (no Legacy SSE)	Varies	(At present, none do)
Type 7	AVX, Legacy SSE	None	None
Type 8	AVX	None	None
Type 11	F16C	8 or 16 byte, Not explicitly aligned, no AC#	Yes
Type 12	AVX2	Not explicitly aligned, no AC#	No

See Table 2-15 for lists of instructions in each exception class.

Table 2-15. Instructions in each Exception Class

Exception Class	Instruction
Type 1	(V)MOVAPD, (V)MOVAPS, (V)MOVDQA, (V)MOVNTDQ, (V)MOVNTDQA, (V)MOVNTPD, (V)MOVNTPS
Type 2	(V)ADDPD, (V)ADDPS, (V)ADDSUBPD, (V)ADDSUBPS, (V)CMPPD, (V)CMPPS, (V)CVTDQ2PS, (V)CVTPD2DQ, (V)CVTPD2PS, (V)CVTPS2DQ, (V)CVTTPD2DQ, (V)CVTTPS2DQ, (V)DIVPD, (V)DIVPS, (V)DPPD*, (V)DPPS*, VFMADD132PD, VFMADD213PD, VFMADD231PD, VFMADD132PS, VFMADD213PS, VFMADD231PS, VFMADDSUB132PD, VFMADDSUB213PD, VFMADDSUB231PD, VFMADDSUB132PS, VFMADDSUB213PS, VFMADDSUB231PS, VFMSUBADD132PD, VFMSUBADD213PD, VFMSUBADD231PD, VFMSUBADD132PS, VFMSUBADD213PS, VFMSUBADD231PS, VFMSUB132PD, VFMSUB213PD, VFMSUB231PD, VFMSUB132PS, VFMSUB213PS, VFMSUB231PS, VFNMADD132PD, VFNMADD213PD, VFNMADD231PD, VFNMADD132PS, VFNMADD213PS, VFNMADD231PS, VFNMSUB132PD, VFNMSUB213PD, VFNMSUB231PD, VFNMSUB132PS, VFNMSUB213PS, VFNMSUB231PS, (V)HADDPD, (V)HADDPS, (V)HSUBPD, (V)HSUBPS, (V)MAXPD, (V)MAXPS, (V)MINPD, (V)MINPS, (V)MULPD, (V)MULPS, (V)ROUNDPS, (V)SQRTPD, (V)SQRTPS, (V)SUBPD, (V)SUBPS
Type 3	(V)ADDSD, (V)ADDSS, (V)CMPSD, (V)CMPSS, (V)COMISD, (V)COMISS, (V)CVTPS2PD, (V)CVTSD2SI, (V)CVTSD2SS, (V)CVTSI2SD, (V)CVTSI2SS, (V)CVTSS2SD, (V)CVTSS2SI, (V)CVTTSD2SI, (V)CVTTSS2SI, (V)DIVSD, (V)DIVSS, VFMADD132SD, VFMADD213SD, VFMADD231SD, VFMADD132SS, VFMADD213SS, VFMADD231SS, VFMSUB132SD, VFMSUB213SD, VFMSUB231SD, VFMSUB132SS, VFMSUB213SS, VFMSUB231SS, VFNMADD132SD, VFNMADD213SD, VFNMADD231SD, VFNMADD132SS, VFNMADD213SS, VFNMADD231SS, VFNMSUB132SD, VFNMSUB213SD, VFNMSUB231SD, VFNMSUB132SS, VFNMSUB213SS, VFNMSUB231SS, (V)MAXSD, (V)MAXSS, (V)MINSD, (V)MINSS, (V)MULSD, (V)MULSS, (V)ROUNDSD, (V)ROUNDSS, (V)SQRTSD, (V)SQRTSS, (V)SUBSD, (V)SUBSS, (V)UCOMISD, (V)UCOMISS
Type 4	(V)AESDEC, (V)AESDECLAST, (V)AESENC, (V)AESENCLAST, (V)AESIMC, (V)AESKEYGENASSIST, (V)ANDPD, (V)ANDPS, (V)ANDNPD, (V)ANDNPS, (V)BLENDPD, (V)BLENDPS, VBLENDVPD, VBLENDVPS, (V)LDDQU***, (V)MASKMOVDQU, (V)PTEST, VTESTPS, VTESTPD, (V)MOVDQU*, (V)MOVSHDUP, (V)MOVSLDUP, (V)MOVUPD*, (V)MOVUPS*, (V)MPSADBW, (V)ORPD, (V)ORPS, (V)PABSB, (V)PABSW, (V)PABSD, (V)PACKSSWB, (V)PACKSSDW, (V)PACKUSWB, (V)PACKUSDW, (V)PADDB, (V)PADDW, (V)PADDD, (V)PADDQ, (V)PADDSB, (V)PADDSW, (V)PADDUSB, (V)PADDUSW, (V)PALIGNR, (V)PAND, (V)PANDN, (V)PAVGB, (V)PAVGW, (V)PBLENDVB, (V)PBLENDW, (V)PCMP(E/I)STRI/M***, (V)PCMPEQB, (V)PCMPEQW, (V)PCMPEQD, (V)PCMPEQQ, (V)PCMPGTB, (V)PCMPGTW, (V)PCMPGTD, (V)PCMPGTQ, (V)PCLMULQDQ, (V)PHADDW, (V)PHADDD, (V)PHADDSW, (V)PHMINPOSUW, (V)PHSUBD, (V)PHSUBW, (V)PHSUBSW, (V)PMADDWD, (V)PMADDUBSW, (V)PMAXSB, (V)PMAXSW, (V)PMAXSD, (V)PMAXUB, (V)PMAXUW, (V)PMAXUD, (V)PMINSB, (V)PMINSW, (V)PMINSD, (V)PMINUB, (V)PMINUW, (V)PMINUD, (V)PMULHUW, (V)PMULHRSW, (V)PMULHW, (V)PMULLW, (V)PMULLD, (V)PMULUDQ, (V)PMULDQ, (V)POR, (V)PSADBW, (V)PSHUFB, (V)PSHUFD, (V)PSHUFHW, (V)PSHUFLW, (V)PSIGNB, (V)PSIGNW, (V)PSIGND, (V)PSLLW, (V)PSLLD, (V)PSLLQ, (V)PSRAW, (V)PSRAD, (V)PSRLW, (V)PSRLD, (V)PSRLQ, (V)PSUBB, (V)PSUBW, (V)PSUBD, (V)PSUBQ, (V)PSUBSB, (V)PSUBSW, (V)PUNPCKHBW, (V)PUNPCKHWD, (V)PUNPCKHDQ, (V)PUNPCKHQDQ, (V)PUNPCKLBW, (V)PUNPCKLWD, (V)PUNPCKLDQ, (V)PUNPCKLQDQ, (V)PXOR, (V)RCPPS, (V)RSQRTPS, (V)SHUFPD, (V)SHUFPS, (V)UNPCKHPD, (V)UNPCKHPS, (V)UNPCKLPD, (V)UNPCKLPS, (V)XORPD, (V)XORPS, VPBLENDD, VPERMD, VPERMPS, VPERMPD, VPERMQ, VPSLLVD, VPSLLVQ, VPSRAVD, VPSRLVD, VPSRLVQ, VPERMILPD, VPERMILPS, VPERM2F128
Type 5	(V)CVTDQ2PD, (V)EXTRACTPS, (V)INSERTPS, (V)MOVD, (V)MOVQ, (V)MOVDDUP, (V)MOVLPD, (V)MOVLPS, (V)MOVHPD, (V)MOVHPS, (V)MOVSD, (V)MOVSS, (V)PEXTRB, (V)PEXTRD, (V)PEXTRW, (V)PEXTRQ, (V)PINSRB, (V)PINSRD, (V)PINSRW, (V)PINSRQ, (V)RCPSS, (V)RSQRTSS, (V)PMOVSX/ZX, VLDMXCSR*, VSTMXCSR
Type 6	VEXTRACTF128, VBROADCASTSS, VBROADCASTSD, VBROADCASTF128, VINSERTF128, VMASKMOVPS**, VMASKMOVPD**, VPMASKMOVD, VPMASKMOVQ, VBROADCASTI128, VPBROADCASTB, VPBROADCASTD, VPBROADCASTW, VPBROADCASTQ, VEXTRACTI128, VINSERTI128, VPERM2I128
Type 7	(V)MOVLHPS, (V)MOVHLPS, (V)MOVMSKPD, (V)MOVMSKPS, (V)PMOVMSKB, (V)PSLLDQ, (V)PSRLDQ, (V)PSLLW, (V)PSLLD, (V)PSLLQ, (V)PSRAW, (V)PSRAD, (V)PSRLW, (V)PSRLD, (V)PSRLQ
Type 8	VZEROALL, VZEROUPPER
Type 11	VCVTPH2PS, VCVTPS2PH
Type 12	VGATHERDPS, VGATHERDPD, VGATHERQPS, VGATHERQPD, VPGATHERDD, VPGATHERDQ, VPGATHERQD, VPGATHERQQ

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

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

(***) - PCMPESTRI, PCMPESTRM, PCMPISTRI, PCMPISTRM and LDDQU instructions do not cause #GP if the memory operand is not aligned to 16-Byte boundary.

Table 2-15 classifies exception behaviors for AVX instructions. Within each class of exception conditions that are listed in Table 2-18 through Table 2-27, certain subsets of AVX instructions may be subject to #UD exception depending on the encoded value of the VEX.L field. Table 2-17 provides supplemental information of AVX instructions that may be subject to #UD exception if encoded with incorrect values in the VEX.W or VEX.L field.

Table 2-16. #UD Exception and VEX.W=1 Encoding

Exception Class	#UD If VEX.W = 1 in all modes	#UD If VEX.W = 1 in non-64-bit modes
Type 1		
Type 2		
Type 3		
Type 4	VBLENDVPD, VBLENDVPS, VPBLENDVB, VTESTPD, VTESTPS, VPBLENDD, VPERMD, VPERMPS, VPERM2I128, VPSRAVD, VPERMILPD, VPERMILPS, VPERM2F128	
Type 5		
Type 6	VEXTRACTF128, VBROADCASTSS, VBROADCASTSD, VBROADCASTF128, VINSERTF128, VMASKMOVPS, VMASKMOVPD, VBROADCASTI128, VPBROADCASTB/W/D, VEXTRACTI128, VINSERTI128	
Type 7		
Type 8		
Type 11	VCVTPH2PS, VCVTPS2PH	
Type 12		

Table 2-17. #UD Exception and VEX.L Field Encoding

Exception Class	#UD If VEX.L = 0	#UD If (VEX.L = 1 && AVX2 not present && AVX present)	#UD If (VEX.L = 1 && AVX2 present)
Type 1		VMOVNTDQA	
Type 2		VDPPD	VDPPD
Type 3			
Type 4		VMASKMOVDQU, VMPSADBW, VPABSB/W/D, VPACKSSWB/DW, VPACKUSWB/DW, VPADDB/W/D, VPADDQ, VPADDSB/W, VPADDUSB/W, VPALIGNR, VPAND, VPANDN, VPAVGB/W, VPBLENDVB, VPBLENDW, VPCMP(E/I)STRI/M, VPCMPEQB/W/D/Q, VPCMPGTB/W/D/Q, VPHADDW/D, VPHADDSW, VPHMINPOSUW, VPHSUBD/W, VPHSUBSW, VPMADDWD, VPMADDUBSW, VPMAXSB/W/D, VPMAXUB/W/D, VPMINSB/W/D, VPMINUB/W/D, VPMULHUW, VPMULHRSW, VPMULHW/LW, VPMULLD, VPMULUDQ, VPMULDQ, VPOR, VPSADBW, VPSHUFB/D, VPSHUFHW/LW, VPSIGNB/W/D, VPSLLW/D/Q, VPSRAW/D, VPSRLW/D/Q, VPSUBB/W/D/Q, VPSUBSB/W, VPUNPCKHBW/WD/DQ, VPUNPCKHQDQ, VPUNPCKLBW/WD/DQ, VPUNPCKLQDQ, VPXOR	VPCMP(E/I)STRI/M, PHMINPOSUW
Type 5		VEXTRACTPS, VINSERTPS, VMOVD, VMOVQ, VMOVLPD, VMOVLPS, VMOVHPD, VMOVHPS, VPEXTRB, VPEXTRD, VPEXTRW, VPEXTRQ, VPINSRB, VPINSRD, VPINSRW, VPINSRQ, VPMOVSX/ZX, VLDMXCSR, VSTMXCSR	Same as column 3
Type 6	VEXTRACTF128, VPERM2F128, VBROADCASTSD, VBROADCASTF128, VINSERTF128,		
Type 7		VMOVLHPS, VMOVHLPS, VPMOVMSKB, VPSLLDQ, VPSRLDQ, VPSLLW, VPSLLD, VPSLLQ, VPSRAW, VPSRAD, VPSRLW, VPSRLD, VPSRLQ	VMOVLHPS, VMOVHLPS
Type 8			
Type 11			
Type 12			

2.4.1 Exceptions Type 1 (Aligned memory reference)

Table 2-18. Type 1 Class Exception Conditions

Exception	Real	Virtual-8086	Protected and Compatibility	64-bit	Cause of Exception
Invalid Opcode, #UD	X	X			VEX prefix.
			X	X	VEX prefix: If XCR0[2:1] ≠ '11b'. If CR4.OSXSAVE[bit 18]=0.
	X	X	X	X	Legacy SSE instruction: If CR0.EM[bit 2] = 1. If CR4.OSFXSR[bit 9] = 0.
	X	X	X	X	If preceded by a LOCK prefix (F0H).
			X	X	If any REX, F2, F3, or 66 prefixes precede a VEX prefix.
	X	X	X	X	If any corresponding CPUID feature flag is '0'.
Device Not Available, #NM	X	X	X	X	If CR0.TS[bit 3]=1.
Stack, SS(0)			X		For an illegal address in the SS segment.
				X	If a memory address referencing the SS segment is in a non-canonical form.
General Protection, #GP(0)			X	X	VEX.256: Memory operand is not 32-byte aligned. VEX.128: Memory operand is not 16-byte aligned.
	X	X	X	X	Legacy SSE: Memory operand is not 16-byte aligned.
			X		For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.
				X	If the memory address is in a non-canonical form.
	X	X			If any part of the operand lies outside the effective address space from 0 to FFFFH.
Page Fault #PF(fault-code)		X	X	X	For a page fault.

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

Table 2-19. Type 2 Class Exception Conditions

Exception	Real	Virtual 8086	Protected and Compatibility	64-bit	Cause of Exception
Invalid Opcode, #UD	X	X			VEX prefix.
	X	X	X	X	If an unmasked SIMD floating-point exception and CR4.OSXMMEXCPT[bit 10] = 0.
			X	X	VEX prefix: If XCR0[2:1] ≠ '11b'. If CR4.OSXSAVE[bit 18]=0.
	X	X	X	X	Legacy SSE instruction: If CR0.EM[bit 2] = 1. If CR4.OSFXSR[bit 9] = 0.
	X	X	X	X	If preceded by a LOCK prefix (F0H).
			X	X	If any REX, F2, F3, or 66 prefixes precede a VEX prefix.
	X	X	X	X	If any corresponding CPUID feature flag is '0'.
Device Not Available, #NM	X	X	X	X	If CR0.TS[bit 3]=1.
Stack, SS(0)			X		For an illegal address in the SS segment.
				X	If a memory address referencing the SS segment is in a non-canonical form.
General Protection, #GP(0)	X	X	X	X	Legacy SSE: Memory operand is not 16-byte aligned.
			X		For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.
				X	If the memory address is in a non-canonical form.
	X	X			If any part of the operand lies outside the effective address space from 0 to FFFFH.
Page Fault #PF(fault-code)		X	X	X	For a page fault.
SIMD Floating-point Exception, #XM	X	X	X	X	If an unmasked SIMD floating-point exception and CR4.OSXMMEXCPT[bit 10] = 1.

2.4.3 Exceptions Type 3 (<16 Byte memory argument)

Table 2-20. Type 3 Class Exception Conditions

Exception	Real	Virtual-8086	Protected and Compatibility	64-bit	Cause of Exception
Invalid Opcode, #UD	X	X			VEX prefix.
	X	X	X	X	If an unmasked SIMD floating-point exception and CR4.OSXMMEXCPT[bit 10] = 0.
			X	X	VEX prefix: If XCR0[2:1] ≠ '11b'. If CR4.OSXSAVE[bit 18]=0.
	X	X	X	X	Legacy SSE instruction: If CR0.EM[bit 2] = 1. If CR4.OSFXSR[bit 9] = 0.
	X	X	X	X	If preceded by a LOCK prefix (F0H).
			X	X	If any REX, F2, F3, or 66 prefixes precede a VEX prefix.
	X	X	X	X	If any corresponding CPUID feature flag is '0'.
Device Not Available, #NM	X	X	X	X	If CR0.TS[bit 3]=1.
Stack, SS(0)			X		For an illegal address in the SS segment.
				X	If a memory address referencing the SS segment is in a non-canonical form.
General Protection, #GP(0)			X		For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.
				X	If the memory address is in a non-canonical form.
	X	X			If any part of the operand lies outside the effective address space from 0 to FFFFH.
Page Fault #PF(fault-code)		X	X	X	For a page fault.
Alignment Check #AC(0)		X	X	X	If alignment checking is enabled and an unaligned memory reference of 8 Bytes or less is made while the current privilege level is 3.
SIMD Floating-point Exception, #XM	X	X	X	X	If an unmasked SIMD floating-point exception and CR4.OSXMMEXCPT[bit 10] = 1.

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

Table 2-21. Type 4 Class Exception Conditions

Exception	Real	Virtual-8086	Protected and Compatibility	64-bit	Cause of Exception
Invalid Opcode, #UD	X	X			VEX prefix.
			X	X	VEX prefix: If XCR0[2:1] ≠ '11b'. If CR4.OSXSAVE[bit 18]=0.
	X	X	X	X	Legacy SSE instruction: If CR0.EM[bit 2] = 1. If CR4.OSFXSR[bit 9] = 0.
	X	X	X	X	If preceded by a LOCK prefix (F0H).
			X	X	If any REX, F2, F3, or 66 prefixes precede a VEX prefix.
	X	X	X	X	If any corresponding CPUID feature flag is '0'.
Device Not Available, #NM	X	X	X	X	If CR0.TS[bit 3]=1.
Stack, SS(0)			X		For an illegal address in the SS segment.
				X	If a memory address referencing the SS segment is in a non-canonical form.
General Protection, #GP(0)	X	X	X	X	Legacy SSE: Memory operand is not 16-byte aligned.[1]
			X		For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.
				X	If the memory address is in a non-canonical form.
	X	X			If any part of the operand lies outside the effective address space from 0 to FFFFH.
Page Fault #PF(fault-code)		X	X	X	For a page fault.

NOTES:

1. PCMPESTRI, PCMPESTRM, PCMPISTRI, PCMPISTRM and LDDQU instructions do not cause #GP if the memory operand is not aligned to 16-Byte boundary.

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

Table 2-22. Type 5 Class Exception Conditions

Exception	Real	Virtual-8086	Protected and Compatibility	64-bit	Cause of Exception
Invalid Opcode, #UD	X	X			VEX prefix.
			X	X	VEX prefix: If XCR0[2:1] ≠ '11b'. If CR4.OSXSAVE[bit 18]=0.
	X	X	X	X	Legacy SSE instruction: If CR0.EM[bit 2] = 1. If CR4.OSFXSR[bit 9] = 0.
	X	X	X	X	If preceded by a LOCK prefix (F0H).
			X	X	If any REX, F2, F3, or 66 prefixes precede a VEX prefix.
	X	X	X	X	If any corresponding CPUID feature flag is '0'.
Device Not Available, #NM	X	X	X	X	If CR0.TS[bit 3]=1.
Stack, SS(0)			X		For an illegal address in the SS segment.
				X	If a memory address referencing the SS segment is in a non-canonical form.
General Protection, #GP(0)			X		For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.
				X	If the memory address is in a non-canonical form.
	X	X			If any part of the operand lies outside the effective address space from 0 to FFFFH.
Page Fault #PF(fault-code)		X	X	X	For a page fault.
Alignment Check #AC(0)		X	X	X	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

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

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

Table 2-23. Type 6 Class Exception Conditions

Exception	Real	Virtual-8086	Protected and Compatibility	64-bit	Cause of Exception
Invalid Opcode, #UD	X	X			VEX prefix.
			X	X	If XCR0[2:1] ≠ '11b'. If CR4.OSXSAVE[bit 18]=0.
			X	X	If preceded by a LOCK prefix (F0H).
			X	X	If any REX, F2, F3, or 66 prefixes precede a VEX prefix.
			X	X	If any corresponding CPUID feature flag is '0'.
Device Not Available, #NM			X	X	If CR0.TS[bit 3]=1.
Stack, SS(0)			X		For an illegal address in the SS segment.
				X	If a memory address referencing the SS segment is in a non-canonical form.
General Protection, #GP(0)			X		For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.
				X	If the memory address is in a non-canonical form.
Page Fault #PF(fault-code)			X	X	For a page fault.
Alignment Check #AC(0)			X	X	For 4 or 8 byte memory references if alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

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

Table 2-24. Type 7 Class Exception Conditions

Exception	Real	Virtual-8086	Protected and Compatibility	64-bit	Cause of Exception
Invalid Opcode, #UD	X	X			VEX prefix.
			X	X	VEX prefix: If XCR0[2:1] ≠ '11b'. If CR4.OSXSAVE[bit 18]=0.
	X	X	X	X	Legacy SSE instruction: If CR0.EM[bit 2] = 1. If CR4.OSFXSR[bit 9] = 0.
	X	X	X	X	If preceded by a LOCK prefix (F0H).
			X	X	If any REX, F2, F3, or 66 prefixes precede a VEX prefix.
	X	X	X	X	If any corresponding CPUID feature flag is '0'.
Device Not Available, #NM			X	X	If CR0.TS[bit 3]=1.

2.4.8 Exceptions Type 8 (AVX and no memory argument)

Table 2-25. Type 8 Class Exception Conditions

Exception	Real	Virtual-8086	Protected and Compatibility	64-bit	Cause of Exception
Invalid Opcode, #UD	X	X			Always in Real or Virtual-8086 mode.
			X	X	If XCR0[2:1] ≠ '11b'. If CR4.OSXSAVE[bit 18]=0. If CPUID.01H.ECX.AVX[bit 28]=0. If VEX.vvvv ≠ 1111B.
	X	X	X	X	If proceeded by a LOCK prefix (F0H).
Device Not Available, #NM			X	X	If CR0.TS[bit 3]=1.

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

Table 2-26. Type 11 Class Exception Conditions

Exception	Real	Virtual-8086	Protected and Compatibility	64-bit	Cause of Exception
Invalid Opcode, #UD	X	X			VEX prefix.
			X	X	VEX prefix: If XCR0[2:1] ≠ '11b'. If CR4.OSXSAVE[bit 18]=0.
	X	X	X	X	If preceded by a LOCK prefix (F0H).
			X	X	If any REX, F2, F3, or 66 prefixes precede a VEX prefix.
	X	X	X	X	If any corresponding CPUID feature flag is '0'.
Device Not Available, #NM	X	X	X	X	If CR0.TS[bit 3]=1.
Stack, SS(0)			X		For an illegal address in the SS segment.
				X	If a memory address referencing the SS segment is in a non-canonical form.
General Protection, #GP(0)			X		For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.
				X	If the memory address is in a non-canonical form.
	X	X			If any part of the operand lies outside the effective address space from 0 to FFFFH.
Page Fault #PF (fault-code)		X	X	X	For a page fault.
SIMD Floating-Point Exception, #XM	X	X	X	X	If an unmasked SIMD floating-point exception and CR4.OSXMMEXCPT[bit 10] = 1.

2.4.10 Exception Type 12 (VEX-only, VSIB mem arg, no AC, no floating-point exceptions)

Table 2-27. Type 12 Class Exception Conditions

Exception	Real	Virtual-8086	Protected and Compatibility	64-bit	Cause of Exception
Invalid Opcode, #UD	X	X			VEX prefix.
			X	X	VEX prefix: If XCR0[2:1] ≠ '11b'. If CR4.OSXSAVE[bit 18]=0.
	X	X	X	X	If preceded by a LOCK prefix (F0H).
			X	X	If any REX, F2, F3, or 66 prefixes precede a VEX prefix.
	X	X	X	NA	If address size attribute is 16 bit.
	X	X	X	X	If ModR/M.mod = '11b'.
	X	X	X	X	If ModR/M.rm ≠ '100b'.
	X	X	X	X	If any corresponding CPUID feature flag is '0'.
	X	X	X	X	If any vector register is used more than once between the destination register, mask register and the index register in VSIB addressing.
Device Not Available, #NM	X	X	X	X	If CR0.TS[bit 3]=1.
Stack, SS(0)			X		For an illegal address in the SS segment.
				X	If a memory address referencing the SS segment is in a non-canonical form.
General Protection, #GP(0)			X		For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.
				X	If the memory address is in a non-canonical form.
	X	X			If any part of the operand lies outside the effective address space from 0 to FFFFH.
Page Fault #PF (fault-code)		X	X	X	For a page fault.

2.5 VEX ENCODING SUPPORT FOR GPR INSTRUCTIONS

VEX prefix may be used to encode instructions that operate on neither YMM nor XMM registers. VEX-encoded general-purpose-register instructions have the following properties:

- Instruction syntax support for three encodable operands.
- Encoding support for instruction syntax of non-destructive source operand, destination operand encoded via VEX.vvvv, and destructive three-operand syntax.
- Elimination of escape opcode byte (0FH), two-byte escape via a compact bit field representation within the VEX prefix.
- Elimination of the need to use REX prefix to encode the extended half of general-purpose register sets (R8-R15) for direct register access or memory addressing.
- Flexible and more compact bit fields are provided in the VEX prefix to retain the full functionality provided by REX prefix. REX.W, REX.X, REX.B functionalities are provided in the three-byte VEX prefix only.
- VEX-encoded GPR instructions are encoded with VEX.L=0.

Any VEX-encoded GPR instruction with a 66H, F2H, or F3H prefix preceding VEX will #UD.
Any VEX-encoded GPR instruction with a REX prefix proceeding VEX will #UD.
VEX-encoded GPR instructions are not supported in real and virtual 8086 modes.

2.5.1 Exception Conditions for VEX-Encoded GPR Instructions

The exception conditions applicable to VEX-encoded GPR instruction differs from those of legacy GPR instructions. Table 2-28 lists VEX-encoded GPR instructions. The exception conditions for VEX-encoded GRP instructions are found in Table 2-29 for those instructions which have a default operand size of 32 bits and 16-bit operand size is not encodable.

Table 2-28. VEX-Encoded GPR Instructions

Exception Class	Instruction
See Table 2-29	ANDN, BLSI, BLSMSK, BLSR, BZHI, MULX, PDEP, PEXT, RORX, SARX, SHLX, SHRX

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

Table 2-29. Exception Definition (VEX-Encoded GPR Instructions)

Exception	Real	Virtual-8086	Protected and Compatibility	64-bit	Cause of Exception
Invalid Opcode, #UD	X	X	X	X	If BMI1/BMI2 CPUID feature flag is '0'.
	X	X			If a VEX prefix is present.
			X	X	If any REX, F2, F3, or 66 prefixes precede a VEX prefix.
Stack, SS(0)	X	X	X		For an illegal address in the SS segment.
				X	If a memory address referencing the SS segment is in a non-canonical form.
General Protection, #GP(0)			X		For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments. If the DS, ES, FS, or GS register is used to access memory and it contains a null segment selector.
				X	If the memory address is in a non-canonical form.
	X	X			If any part of the operand lies outside the effective address space from 0 to FFFFH.
Page Fault #PF(fault-code)		X	X	X	For a page fault.
Alignment Check #AC(0)		X	X	X	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

2.6 INTEL® AVX-512 ENCODING

The majority of the Intel AVX-512 family of instructions (operating on 512/256/128-bit vector register operands) are encoded using a new prefix (called EVEX). Opmask instructions (operating on opmask register operands) are encoded using the VEX prefix. The EVEX prefix has some parts resembling the instruction encoding scheme using the VEX prefix, and many other capabilities not available with the VEX prefix.

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

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

2.6.1 Instruction Format and EVEX

The placement of the EVEX prefix in an IA instruction is represented in Figure 2-10.

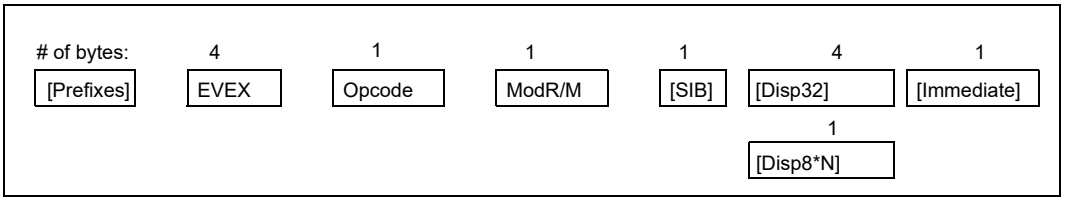


Figure 2-10. AVX-512 Instruction Format and the EVEX Prefix

The EVEX prefix is a 4-byte prefix, with the first two bytes derived from unused encoding form of the 32-bit-mode-only BOUND instruction. The layout of the EVEX prefix is shown in Figure 2-11. The first byte must be 62H, followed by three payload bytes, denoted as P0, P1, and P2 individually or collectively as P[23:0] (see Figure 2-11).

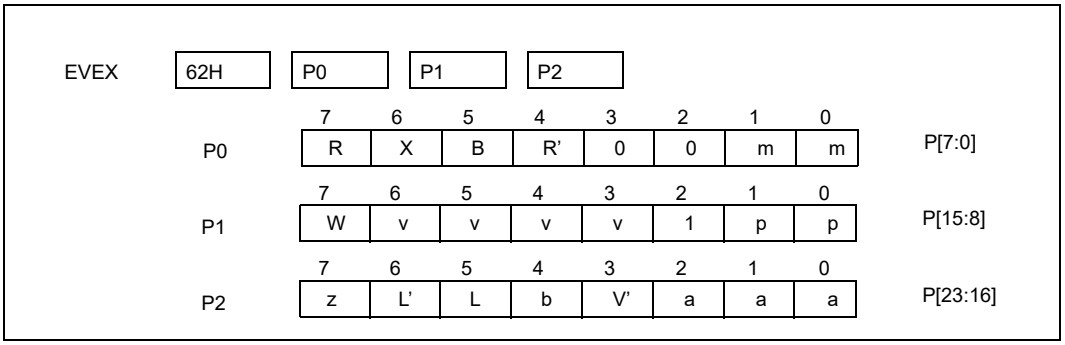


Figure 2-11. Bit Field Layout of the EVEX Prefix

Table 2-30. EVEX Prefix Bit Field Functional Grouping

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

The bit fields in P[23:0] are divided into the following functional groups (Table 2-30 provides a tabular summary):

- Reserved bits: P[3:2] must be 0, otherwise #UD.
- Fixed-value bit: P[10] must be 1, otherwise #UD.
- Compressed legacy prefix/escape bytes: P[1:0] is identical to the lowest 2 bits of VEX.mmmmm; P[9:8] is identical to VEX.pp.
- Operand specifier modifier bits for vector register, general purpose register, memory addressing: P[7:5] allows access to the next set of 8 registers beyond the low 8 registers when combined with ModR/M register specifiers.
- Operand specifier modifier bit for vector register: P[4] (or EVEX.R') allows access to the high 16 vector register set when combined with P[7] and ModR/M.reg specifier; P[6] can also provide access to a high 16 vector register when SIB or VSIB addressing are not needed.
- Non-destructive source /vector index operand specifier: P[19] and P[14:11] encode the second source vector register operand in a non-destructive source syntax, vector index register operand can access an upper 16 vector register using P[19].
- Op-mask register specifiers: P[18:16] encodes op-mask register set k0-k7 in instructions operating on vector registers.
- EVEX.W: P[15] is similar to VEX.W which serves either as opcode extension bit or operand size promotion to 64-bit in 64-bit mode.
- Vector destination merging/zeroing: P[23] encodes the destination result behavior which either zeroes the masked elements or leave masked element unchanged.
- Broadcast/Static-rounding/SAE context bit: P[20] encodes multiple functionality, which differs across different classes of instructions and can affect the meaning of the remaining field (EVEX.L'L). The functionality for the following instruction classes are:
  - — Broadcasting a single element across the destination vector register: this applies to the instruction class with Load+Op semantic where one of the source operand is from memory.
  - — Redirect L'L field (P[22:21]) as static rounding control for floating-point instructions with rounding semantic. Static rounding control overrides MXCSR.RC field and implies "Suppress all exceptions" (SAE).
  - — Enable SAE for floating -point instructions with arithmetic semantic that is not rounding.
  - — For instruction classes outside of the afore-mentioned three classes, setting EVEX.b will cause #UD.

- Vector length/rounding control specifier: P[22:21] can serve one of three options.
  - — Vector length information for packed vector instructions.
  - — Ignored for instructions operating on vector register content as a single data element.
  - — Rounding control for floating-point instructions that have a rounding semantic and whose source and destination operands are all vector registers.

2.6.2 Register Specifier Encoding and EVEX

EVEX-encoded instruction can access 8 opmask registers, 16 general-purpose registers and 32 vector registers in 64-bit mode (8 general-purpose registers and 8 vector registers in non-64-bit modes). EVEX-encoding can support instruction syntax that access up to 4 instruction operands. Normal memory addressing modes and VSIB memory addressing are supported with EVEX prefix encoding. The mapping of register operands used by various instruction syntax and memory addressing in 64-bit mode are shown in Table 2-31. Opmask register encoding is described in Section 2.6.3.

Table 2-31. 32-Register Support in 64-bit Mode Using EVEX with Embedded REX Bits

	4[1]	3	[2:0]	Reg. Type	Common Usages
REG	EVEX.R'	REX.R	modrm.reg	GPR, Vector	Destination or Source
NDS/NDD	EVEX.V'	EVEX.vvvv		GPR, Vector	2ndSource or Destination
RM	EVEX.X	EVEX.B	modrm.r/m	GPR, Vector	1st Source or Destination
BASE	0	EVEX.B	modrm.r/m	GPR	memory addressing
INDEX	0	EVEX.X	sib.index	GPR	memory addressing
VIDX	EVEX.V'	EVEX.X	sib.index	Vector	VSIB memory addressing

NOTES:

1. Not applicable for accessing general purpose registers.

The mapping of register operands used by various instruction syntax and memory addressing in 32-bit modes are shown in Table 2-32.

Table 2-32. EVEX Encoding Register Specifiers in 32-bit Mode

	[2:0]	Reg. Type	Common Usages
REG	modrm.reg	GPR, Vector	Destination or Source
NDS/NDD	EVEX.vvv	GPR, Vector	2nd Source or Destination
RM	modrm.r/m	GPR, Vector	1st Source or Destination
BASE	modrm.r/m	GPR	Memory Addressing
INDEX	sib.index	GPR	Memory Addressing
VIDX	sib.index	Vector	VSIB Memory Addressing

2.6.3 Opmask Register Encoding

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

- Opmask registers that are the source or destination operands of an instruction treating the content of opmask register as a scalar value, are encoded using the VEX prefix scheme. It can support up to three operands using standard modR/M byte's reg field and rm field and VEX.vvvv. Such a scalar opmask instruction does not support conditional update of the destination operand.
- An opmask register providing conditional processing and/or conditional update of the destination register of a vector instruction is encoded using EVEX.aaa field (see Section 2.6.4).

- An opmask register serving as the destination or source operand of a vector instruction is encoded using standard modR/M byte's reg field and rm fields.

Table 2-33. Opmask Register Specifier Encoding

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

NOTES:

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

2.6.4 Masking Support in EVEX

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

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

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

AVX-512 Foundation instructions can be divided into the following groups:

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

2.6.5 Compressed Displacement (disp8*N) Support in EVEX

For memory addressing using disp8 form, EVEX-encoded instructions always use a compressed displacement scheme by multiplying disp8 in conjunction with a scaling factor N that is determined based on the vector length, the value of EVEX.b bit (embedded broadcast) and the input element size of the instruction. In general, the factor N corresponds to the number of bytes characterizing the internal memory operation of the input operand (e.g., 64 when the accessing a full 512-bit memory vector). The scale factor N is listed in Table 2-34 and Table 2-35 below, where EVEX encoded instructions are classified using the **tupletype** attribute. The scale factor N of each tupletype is listed based on the vector length (VL) and other factors affecting it.

Table 2-34 covers EVEX-encoded instructions which has a load semantic in conjunction with additional computational or data element movement operation, operating either on the full vector or half vector (due to conversion of

numerical precision from a wider format to narrower format). EVEX.b is supported for such instructions for data element sizes which are either dword or qword (see Section 2.6.11).

EVEX-encoded instruction that are pure load/store, and "Load+op" instruction semantic that operate on data element size less then dword do not support broadcasting using EVEX.b. These are listed in Table 2-35. Table 2-35 also includes many broadcast instructions which perform broadcast using a subset of data elements without using EVEX.b. These instructions and a few data element size conversion instruction are covered in Table 2-35. Instruction classified in Table 2-35 do not use EVEX.b and EVEX.b must be 0, otherwise #UD will occur.

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

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

Table 2-34. Compressed Displacement (DISP8*N) Affected by Embedded Broadcast

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

Table 2-35. EVEX DISP8*N for Instructions Not Affected by Embedded Broadcast

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

2.6.6 EVEX Encoding of Broadcast/Rounding/SAE Support

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

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

2.6.7 Embedded Broadcast Support in EVEX

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

The following instruction classes do not support embedded broadcasting:

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

2.6.8 Static Rounding Support in EVEX

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

2.6.9 SAE Support in EVEX

The EVEX encoding system allows arithmetic floating-point instructions without rounding semantic to be encoded with the SAE attribute. This capability applies to scalar and 512-bit vector lengths, register-to-register only, by setting EVEX.b. When EVEX.b is set, "suppress all exceptions" is implied. The processor behaves as if all MXCSR masking controls are set.

2.6.10 Vector Length Orthogonality

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

When EVEX.b bit is set for a register-register instructions with floating-point rounding semantic, the same two bits P2[6:5] specifies rounding mode for the instruction, with implied SAE behavior. The mapping of different instruction classes relative to the embedded broadcast/rounding/SAE control and the EVEX.L'L fields are summarized in Table 2-36.

Table 2-36. EVEX Embedded Broadcast/Rounding/SAE and Vector Length on Vector Instructions

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

2.6.11 #UD Equations for EVEX

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

2.6.11.1 State Dependent #UD

In general, attempts of execute an instruction, which required OS support for incremental extended state component, will #UD if required state components were not enabled by OS. Table 2-37 lists instruction categories with respect to required processor state components. Attempts to execute a given category of instructions while enabled states were less than the required bit vector in XCR0 shown in Table 2-37 will cause #UD.

Table 2-37. OS XSAVE Enabling Requirements of Instruction Categories

Instruction Categories	Vector Register State Access	Required XCR0 Bit Vector [7:0]
Legacy SIMD prefix encoded Instructions (e.g SSE)	XMM	xxxxxx11b
VEX-encoded instructions operating on YMM	YMM	xxxxx111b
EVEX-encoded 128-bit instructions	ZMM	111xx111b
EVEX-encoded 256-bit instructions	ZMM	111xx111b
EVEX-encoded 512-bit instructions	ZMM	111xx111b
VEX-encoded instructions operating on opmask	k-reg	xx1xxx11b

2.6.11.2 Opcode Independent #UD

A number of bit fields in EVEX encoded instruction must obey mode-specific but opcode-independent patterns listed in Table 2-38.

Table 2-38. Opcode Independent, State Dependent EVEX Bit Fields

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

2.6.11.3 Opcode Dependent #UD

This section describes legal values for the rest of the EVEX bit fields. Table 2-39 lists the #UD conditions of EVEX prefix bit fields which encodes or modifies register operands.

Table 2-39. #UD Conditions of Operand-Encoding EVEX Prefix Bit Fields

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

Table 2-40 lists the #UD conditions of instruction encoding of opmask register using EVEX.aaa and EVEX.z

Table 2-40. #UD Conditions of Opmask Related Encoding Field

Notation	Position	Operand Encoding	64-bit #UD	Non-64-bit #UD
EVEX.aaa	P[18 : 16]	Instructions do not use opmask for conditional processing[1].	if aaa != 000b	if aaa != 000b
		Opmask used as conditional processing mask and updated at completion[2].	if aaa = 000b	if aaa = 000b;
		Opmask used as conditional processing.	None (valid[3])	None (valid[1])
EVEX.z	P[23]	Vector instruction using opmask as source or destination[4].	if EVEX.z != 0	if EVEX.z != 0
		Store instructions or gather/scatter instructions.	if EVEX.z != 0	if EVEX.z != 0
		Instruction supporting conditional processing mask with EVEX.aaa = 000b.	if EVEX.z != 0	if EVEX.z != 0

NOTES:

1. E.g., VBROADCASTMxxx, VPMOVM2x, VPMOVx2M.
2. E.g., Gather/Scatter family.
3. aaa can take any value. A value of 000 indicates that there is no masking on the instruction; in this case, all elements will be processed as if there was a mask of 'all ones' regardless of the actual value in K0.
4. E.g., VFPCLASSPD/PS, VCMPB/D/Q/W family, VPMOVM2x, VPMOVx2M.

Table 2-41 lists the #UD conditions of EVEX bit fields that depends on the context of EVEX.b.

Table 2-41. #UD Conditions Dependent on EVEX.b Context

Notation	Position	Operand Encoding	64-bit #UD	Non-64-bit #UD
EVEX.L'Lb	P[22 : 20]	Reg-reg, FP instructions with rounding semantic.	None (valid[1])	None (valid[1])
		Other reg-reg, FP instructions that can cause #XF.	None (valid[2])	None (valid[2])
		Other reg-mem instructions in Table 2-34.	None (valid[3])	None (valid[3])
		Other instruction classes[4] in Table 2-35.	If EVEX.b > 0	If EVEX.b > 0

NOTES:

1. L'L specifies rounding control, see Table 2-36, supports {er} syntax.
2. L'L specifies vector length, see Table 2-36, supports {sae} syntax.
3. L'L specifies vector length, see Table 2-36, supports embedded broadcast syntax
4. L'L specifies either vector length or ignored.

2.6.12 Device Not Available

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

2.6.13 Scalar Instructions

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

2.7 EXCEPTION CLASSIFICATIONS OF EVEX-ENCODED INSTRUCTIONS

The exception behavior of EVEX-encoded instructions can be classified into the classes shown in the rest of this section. The classification of EVEX-encoded instructions follow a similar framework as those of AVX and AVX2 instructions using the VEX prefix. Exception types for EVEX-encoded instructions are named in the style of "E##" or with a suffix "E##XX". The "##" designation generally follows that of AVX/AVX2 instructions. The majority of EVEX encoded instruction with "Load+op" semantic supports memory fault suppression, which is represented by E##. The instructions with "Load+op" semantic but do not support fault suppression are named "E##NF". A summary table of exception classes by class names are shown below.

Table 2-42. EVEX-Encoded Instruction Exception Class Summary

Exception Class	Instruction set	Mem arg	(#XM)
Type E1	Vector Moves/Load/Stores	Explicitly aligned, w/ fault suppression	None
Type E1NF	Vector Non-temporal Stores	Explicitly aligned, no fault suppression	None
Type E2	FP Vector Load+op	Support fault suppression	Yes
Type E2NF	FP Vector Load+op	No fault suppression	Yes
Type E3	FP Scalar/Partial Vector, Load+Op	Support fault suppression	Yes
Type E3NF	FP Scalar/Partial Vector, Load+Op	No fault suppression	Yes
Type E4	Integer Vector Load+op	Support fault suppression	No
Type E4NF	Integer Vector Load+op	No fault suppression	No
Type E5	Legacy-like Promotion	Varies, Support fault suppression	No
Type E5NF	Legacy-like Promotion	Varies, No fault suppression	No

Table 2-42. EVEX-Encoded Instruction Exception Class Summary

Exception Class	Instruction set	Mem arg	(#XM)
Type E6	Post AVX Promotion	Varies, w/ fault suppression	No
Type E6NF	Post AVX Promotion	Varies, no fault suppression	No
Type E7NM	Register-to-register op	None	None
Type E9NF	Miscellaneous 128-bit	Vector-length Specific, no fault suppression	None
Type E10	Non-XF Scalar	Vector Length ignored, w/ fault suppression	None
Type E10NF	Non-XF Scalar	Vector Length ignored, no fault suppression	None
Type E11	VCVTPH2PS	Half Vector Length, w/ fault suppression	Yes
Type E11NF	VCVTPS2PH	Half Vector Length, no fault suppression	Yes
Type E12	Gather and Scatter Family	VSIB addressing, w/ fault suppression	None
Type E12NP	Gather and Scatter Prefetch Family	VSIB addressing, w/o page fault	None

Table 2-43 lists EVEX-encoded instruction mnemonic by exception classes.

Table 2-43. EVEX Instructions in each Exception Class

Exception Class	Instruction
Type E1	VMOVAPD, VMOVAPS, VMOVDQA32, VMOVDQA64
Type E1NF	VMOVNTDQ, VMOVNTDQA, VMOVNTPD, VMOVNTPS
Type E2	VADDPD, VADDPS, VCMPPD, VCMPPS, VCVTDQ2PS, VCVTPD2DQ, VCVTPD2PS, VCVTPS2DQ, VCVTTPD2DQ, VCVTTPS2DQ, VDIVPD, VDIVPS, VFMADDxxxPD, VFMADDxxxPS, VFMSUBADDxxxPD, VFMSUBADDxxxPS, VFMSUBxxxPD, VFMSUBxxxPS, VFNMADDxxxPD, VFNMADDxxxPS, VFNMSUBxxxPD, VFNMSUBxxxPS, VMAXPD, VMAXPS, VMINPD, VMINPS, VMULPD, VMULPS, VSQRTPD, VSQRTPS, VSUBPD, VSUBPS VCVTPD2QQ, VCVTPD2UQQ, VCVTPD2UDQ, VCVTPS2UDQS, VCVTQQ2PD, VCVTQQ2PS, VCVTTPD2DQ, VCVTTPD2QQ, VCVTTPD2UDQ, VCVTTPD2UQQ, VCVTTPS2DQ, VCVTTPS2UDQ, VCVTUDQ2PS, VCVTUQQ2PD, VCVTUQQ2PS, VFIXUPIMMPD, VFIXUPIMMPS, VGETEXPPD, VGETEXPPS, VGETMANTPD, VGETMANTPS, VRANGEPD, VRANGEPS, VREDUCEPD, VREDUCEPS, VRNDSCALEPD, VRNDSCALEPS, VSCALEFPD, VSCALEFPS, VRCP28PD, VRCP28PS, VRSQRT28PD, VRSQRT28PS
Type E3	VADDSD, VADDSS, VCMPSD, VCMPSS, VCVTPS2PD, VCVTSD2SS, VCVTSS2SD, VDIVSD, VDIVSS, VMAXSD, VMAXSS, VMINSD, VMINSS, VMULSD, VMULSS, VSQRTSD, VSQRTSS, VSUBSD, VSUBSS VCVTPS2QQ, VCVTPS2UQQ, VCVTTPS2QQ, VCVTTPS2UQQ, VFMADDxxxSD, VFMADDxxxSS, VFMSUBxxxSD, VFMSUBxxxSS, VFNMADDxxxSD, VFNMADDxxxSS, VFNMSUBxxxSD, VFNMSUBxxxSS, VFIXUPIMMSD, VFIXUPIMMSS, VGETEXPSD, VGETEXPSS, VGETMANTSD, VGETMANTSS, VRANGESD, VRANGESS, VREDUCESD, VREDUCESS, VRNDSCALESD, VRNDSCALESS, VSCALEFSD, VSCALEFSS, VRCP28SD, VRCP28SS, VRSQRT28SD, VRSQRT28SS
Type E3NF	VCOMISD, VCOMISS, VCVTSD2SI, VCVTSI2SD, VCVTSI2SS, VCVTSS2SI, VCVTTSD2SI, VCVTTSS2SI, VUCOMISD, VUCOMISS VCVTSD2USI, VCVTTSD2USI, VCVTSS2USI, VCVTTSS2USI, VCVTUSI2SD, VCVTUSI2SS

Table 2-43. EVEX Instructions in each Exception Class (Contd.)

Exception Class	Instruction
Type E4	VANDPD, VANDPS, VANDNPD, VANDNPS, VORPD, VORPS, VPABSD, VPABSQ, VPADDD, VPADDQ, VPANDD, VPANDQ, VPANDND, VPANDNQ, VPCMPEQD, VPCMPEQQ, VPCMPGTD, VPCMPGTQ, VPMAXSD, VPMAXSQ, VPMAXUD, VPMAXUQ, VPMINSD, VPMINSQ, VPMINUD, VPMINUQ, VPMULLD, VPMULLQ, VPMULUDQ, VPMULDQ, VPORD, VPORQ, VPSUBD, VPSUBQ, VPXORD, VPXORQ, VXORPD, VXORPS, VPSLLVD, VPSLLVQ, VBLENDMPD, VBLENDMPS, VPBLENDMD, VPBLENDMQ, VFPCLASSPD, VFPCLASSPS, VPCMPD, VPCMPQ, VPCMPUD, VPCMPUQ, VPLZCNTD, VPLZCNTQ, VPROLD, VPROLQ, (VPSLLD, VPSLLQ, VPSRAD, VPSRAQ, VPSRLD, VPSRLQ)[1], VPTERNLOGD, VPTERNLOGQ, VPTESTMD, VPTESTMQ, VPTESTNMD, VPTESTNMQ, VRCP14PD, VRCP14PS, VRSQRT14PD, VRSQRT14PS, VPCONFLICTD, VPCONFLICTQ, VPSRAVW, VPSRAVD, VPSRAVW, VPSRAVQ, VPMADD52LUQ, VPMADD52HUQ
E4.nb[2]	VMOVUPD, VMOVUPS, VMOVDQU8, VMOVDQU16, VMOVDQU32, VMOVDQU64, VPCMPB, VPCMPW, VPCMPUB, VPCMPUW, VEXPANDPD, VEXPANDPS, VPCOMPRESSD, VPCOMPRESSQ, VPEXPANDD, VPEXPANDQ, VCOMPRESSPD, VCOMPRESSPS, VPABSB, VPABSW, VPADDB, VPADDW, VPADDSB, VPADDSW, VPADDUSB, VPADDUSW, VPAVGB, VPAVGW, VPCMPEQB, VPCMPEQW, VPCMPGTB, VPCMPGTW, VPMAXSB, VPMAXSW, VPMAXUB, VPMAXUW, VPMINSB, VPMINSW, VPMINUB, VPMINUW, VPMULHRSW, VPMULHUW, VPMULHW, VPMULLW, VPSUBB, VPSUBW, VPSUBSB, VPSUBSW, VPTESTMB, VPTESTMW, VPTESTNMB, VPTESTNMW, VPSLLW, VPSRAW, VPSRLW, VPSLLVW, VPSRLVW
Type E4NF	VPACKSSDW, VPACKUSDW VPSHUFD, VPUNPCKHDQ, VPUNPCKHQDQ, VPUNPCKLDQ, VPUNPCKLQDQ, VSHUFPD, VSHUFPS, VUNPCKHPD, VUNPCKHPS, VUNPCKLPD, VUNPCKLPS, VPERMD, VPERMPS, VPERMPD, VPERMQ, VALIGND, VALIGNQ, VPERMI2D, VPERMI2PS, VPERMI2PD, VPERMI2Q, VPERMT2D, VPERMT2PS, VPERMT2Q, VPERMT2PD, VPERMILPD, VPERMILPS, VSHUFI32X4, VSHUFI64X2, VSHUFF32X4, VSHUFF64X2, VPMULTISHIFTQB
E4NF.nb[2]	VDBPSADBW, VPACKSSWB, VPACKUSWB, VPALIGNR, VPMADDWD, VPMADDUBSW, VMOVSHDUP, VMOVSLDUP, VPSADBW, VPSHUFB, VPSHUFHW, VPSHUFLW, VPSLLDQ, VPSRLDQ, VPSLLW, VPSRAW, VPSRLW, (VPSLLD, VPSLLQ, VPSRAD, VPSRAQ, VPSRLD, VPSRLQ)[3], VPUNPCKHBW, VPUNPCKHWD, VPUNPCKLBW, VPUNPCKLWD, VPERMW, VPERMI2W, VPERMT2W
Type E5	VCVTDQ2PD, PMOVSXBW, PMOVSXBW, PMOVSXBD, PMOVSXBQ, PMOVSXWD, PMOVSXWQ, PMOVSXDQ, PMOVZXBW, PMOVZXBD, PMOVZXBQ, PMOVZXWD, PMOVZXWQ, PMOVZXDQ VCVTUDQ2PD
Type E5NF	VMOVDDUP
Type E6	VBROADCASTSS, VBROADCASTSD, VBROADCASTF32X4, VBROADCASTI32X4, VPBROADCASTB, VPBROADCASTD, VPBROADCASTW, VPBROADCASTQ, VBROADCASTF32X2, VBROADCASTF32X4, VBROADCASTF64X2, VBROADCASTF32X8, VBROADCASTF64X4, VBROADCASTI32X2, VBROADCASTI32X4, VBROADCASTI64X2, VBROADCASTI32X8, VBROADCASTI64X4, VFPCLASSSD, VFPCLASSSS, VPMOVQB, VPMOVSQB, VPMOVUSQB, VPMOVQW, VPMOVSQW, VPMOVUSQW, VPMOVQD, VPMOVSQD, VPMOVUSQD, VPMOVDB, VPMOVSDB, VPMOVUSDB, VPMOVDW, VPMOVSDW, VPMOVUSDW
Type E6NF	VEXTRACTF32X4, VEXTRACTF64X2, VEXTRACTF32X8, VINSERTF32X4, VINSERTF64X2, VINSERTF64X4, VINSERTF32X8, VINSERTI32X4, VINSERTI64X2, VINSERTI64X4, VINSERTI32X8, VEXTRACTI32X4, VEXTRACTI64X2, VEXTRACTI32X8, VEXTRACTI64X4, VPBROADCASTMB2Q, VPBROADCASTMW2D, VPMOVWB, VPMOVSWB, VPMOVUSWB
Type E7NM.128[4]	VMOVLHPS, VMOVHLPS
Type E7NM.	(VPBROADCASTD, VPBROADCASTQ, VPBROADCASTB, VPBROADCASTW)[5], VPMOVM2B, VPMOVM2D, VPMOVM2Q, VPMOVM2W, VPMOVB2M, VPMOVD2M, VPMOVQ2M, VPMOVW2M

Table 2-43. EVEX Instructions in each Exception Class (Contd.)

Exception Class	Instruction
Type E9NF	VEXTRACTPS, VINSERTPS, VMOVHPD, VMOVHPS, VMOVLPD, VMOVLPS, VMOVD, VMOVQ, VPEXTRB, VPEXTRD, VPEXTRW, VPEXTRQ, VPINSRB, VPINSRD, VPINSRW, VPINSRQ
Type E10	VMOVSD, VMOVSS, VRCP14SD, VRCP14SS, VRSQRT14SD, VRSQRT14SS,
Type E10NF	(VCVTSI2SD, VCVTUSI2SD)[6]
Type E11	VCVTPH2PS, VCVTPS2PH
Type E12	VGATHERDPS, VGATHERDPD, VGATHERQPS, VGATHERQPD, VPGATHERDD, VPGATHERDQ, VPGATHERQD, VPGATHERQQ, VPSCATTERDD, VPSCATTERDQ, VPSCATTERQD, VPSCATTERQQ, VSCATTERDPD, VSCATTERDPS, VSCATTERQPD, VSCATTERQPS
Type E12NP	VGATHERPF0DPD, VGATHERPF0DPS, VGATHERPF0QPD, VGATHERPF0QPS, VGATHERPF1DPD, VGATHERPF1DPS, VGATHERPF1QPD, VGATHERPF1QPS, VSCATTERPF0DPD, VSCATTERPF0DPS, VSCATTERPF0QPD, VSCATTERPF0QPS, VSCATTERPF1DPD, VSCATTERPF1DPS, VSCATTERPF1QPD, VSCATTERPF1QPS

NOTES:

1. Operand encoding FVI tupletype with immediate.
2. Embedded broadcast is not supported with the ".nb" suffix.
3. Operand encoding M128 tupletype.
4. #UD raised if EVEX.L'L !=00b (VL=128).
5. The source operand is a general purpose register.
6. W0 encoding only.

2.7.1 Exceptions Type E1 and E1NF of EVEX-Encoded Instructions

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

Table 2-44. Type E1 Class Exception Conditions

Exception	Real	Virtual 80x86	Protected and Compatibility	64-bit	Cause of Exception
Invalid Opcode, #UD	X	X			If EVEX prefix present.
			X	X	If CR4.OSXSAVE[bit 18]=0. If any one of following conditions applies: ▪ State requirement, Table 2-37 not met. ▪ Opcode independent #UD condition in Table 2-38. ▪ Operand encoding #UD conditions in Table 2-39. ▪ Opmask encoding #UD condition of Table 2-40. ▪ If EVEX.b != 0. ▪ If EVEX.L'L != 10b (VL=512).
	X	X	X	X	If preceded by a LOCK prefix (F0H).
			X	X	If any REX, F2, F3, or 66 prefixes precede a EVEX prefix.
	X	X	X	X	If any corresponding CPUID feature flag is '0'.
Device Not Available, #NM	X	X	X	X	If CR0.TS[bit 3]=1.
Stack, SS(0)			X		If fault suppression not set, and an illegal address in the SS segment.
				X	If fault suppression not set, and a memory address referencing the SS segment is in a non-canonical form.
General Protection, #GP(0)			X	X	EVEX.512: Memory operand is not 64-byte aligned. EVEX.256: Memory operand is not 32-byte aligned. EVEX.128: Memory operand is not 16-byte aligned.
			X		If fault suppression not set, and an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.
				X	If fault suppression not set, and the memory address is in a non-canonical form.
	X	X			If fault suppression not set, and any part of the operand lies outside the effective address space from 0 to FFFFH.
Page Fault #PF(fault-code)		X	X	X	If fault suppression not set, and a page fault.

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

Table 2-45. Type E1NF Class Exception Conditions

Exception	Real	Virtual 80x86	Protected and Compatibility	64-bit	Cause of Exception
Invalid Opcode, #UD	X	X			If EVEX prefix present.
			X	X	If CR4.OSXSAVE[bit 18]=0. If any one of following conditions applies: ▪ State requirement, Table 2-37 not met. ▪ Opcode independent #UD condition in Table 2-38. ▪ Operand encoding #UD conditions in Table 2-39. ▪ Opmask encoding #UD condition of Table 2-40. ▪ If EVEX.b != 0. ▪ If EVEX.L'L != 10b (VL=512).
	X	X	X	X	If preceded by a LOCK prefix (F0H).
			X	X	If any REX, F2, F3, or 66 prefixes precede a EVEX prefix.
	X	X	X	X	If any corresponding CPUID feature flag is '0'.
Device Not Available, #NM	X	X	X	X	If CR0.TS[bit 3]=1.
Stack, SS(0)			X		For an illegal address in the SS segment.
				X	If a memory address referencing the SS segment is in a non-canonical form.
General Protection, #GP(0)			X	X	EVEX.512: Memory operand is not 64-byte aligned. EVEX.256: Memory operand is not 32-byte aligned. EVEX.128: Memory operand is not 16-byte aligned.
			X		For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.
				X	If the memory address is in a non-canonical form.
	X	X			If any part of the operand lies outside the effective address space from 0 to FFFFH.
Page Fault #PF(fault-code)		X	X	X	For a page fault.

2.7.2 Exceptions Type E2 of EVEX-Encoded Instructions

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

Table 2-46. Type E2 Class Exception Conditions

Exception	Real	Virtual 8086	Protected and Compatibility	64-bit	Cause of Exception
Invalid Opcode, #UD	X	X			If EVEX prefix present.
	X	X	X	X	If an unmasked SIMD floating-point exception and CR4.OSXMMEXCPT[bit 10] = 0.
			X	X	If CR4.OSXSAVE[bit 18]=0. If any one of following conditions applies: ▪ State requirement, Table 2-37 not met. ▪ Opcode independent #UD condition in Table 2-38. ▪ Operand encoding #UD conditions in Table 2-39. ▪ Opmask encoding #UD condition of Table 2-40. ▪ If EVEX.L'L != 10b (VL=512).
	X	X	X	X	If preceded by a LOCK prefix (F0H).
			X	X	If any REX, F2, F3, or 66 prefixes precede a EVEX prefix.
	X	X	X	X	If any corresponding CPUID feature flag is '0'.
Device Not Available, #NM	X	X	X	X	If CR0.TS[bit 3]=1.
Stack, SS(0)			X		If fault suppression not set, and an illegal address in the SS segment.
				X	If fault suppression not set, and a memory address referencing the SS segment is in a non-canonical form.
General Protection, #GP(0)			X		If fault suppression not set, and an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.
				X	If fault suppression not set, and the memory address is in a non-canonical form.
	X	X			If fault suppression not set, and any part of the operand lies outside the effective address space from 0 to FFFFH.
Page Fault #PF(fault-code)		X	X	X	If fault suppression not set, and a page fault.
SIMD Floating-point Exception, #XM	X	X	X	X	If an unmasked SIMD floating-point exception, {sae} or {er} not set, and CR4.OSXMMEXCPT[bit 10] = 1.

2.7.3 Exceptions Type E3 and E3NF of EVEX-Encoded Instructions

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

Table 2-47. Type E3 Class Exception Conditions

Exception	Real	Virtual 80x86	Protected and Compatibility	64-bit	Cause of Exception
Invalid Opcode, #UD	X	X			If EVEX prefix present.
	X	X	X	X	If an unmasked SIMD floating-point exception and CR4.OSXMMEXCPT[bit 10] = 0.
			X	X	If CR4.OSXSAVE[bit 18]=0. If any one of following conditions applies: ▪ State requirement, Table 2-37 not met. ▪ Opcode independent #UD condition in Table 2-38. ▪ Operand encoding #UD conditions in Table 2-39. ▪ Opmask encoding #UD condition of Table 2-40. ▪ If EVEX.b != 0.
	X	X	X	X	If preceded by a LOCK prefix (F0H).
			X	X	If any REX, F2, F3, or 66 prefixes precede a EVEX prefix.
	X	X	X	X	If any corresponding CPUID feature flag is '0'.
Device Not Available, #NM	X	X	X	X	If CR0.TS[bit 3]=1.
Stack, SS(0)			X		If fault suppression not set, and an illegal address in the SS segment.
				X	If fault suppression not set, and a memory address referencing the SS segment is in a non-canonical form.
General Protection, #GP(0)			X		If fault suppression not set, and an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.
				X	If fault suppression not set, and the memory address is in a non-canonical form.
	X	X			If fault suppression not set, and any part of the operand lies outside the effective address space from 0 to FFFFH.
Page Fault #PF(fault-code)		X	X	X	If fault suppression not set, and a page fault.
Alignment Check #AC(0)		X	X	X	If alignment checking is enabled and an unaligned memory reference of 8 bytes or less is made while the current privilege level is 3.
SIMD Floating-point Exception, #XM	X	X	X	X	If an unmasked SIMD floating-point exception, {sae} or {er} not set, and CR4.OSX-MMEXCPT[bit 10] = 1.

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

Table 2-48. Type E3NF Class Exception Conditions

Exception	Real	Virtual 80x86	Protected and Compatibility	64-bit	Cause of Exception
Invalid Opcode, #UD	X	X			EVEX prefix.
	X	X	X	X	If an unmasked SIMD floating-point exception and CR4.OSXMMEXCPT[bit 10] = 0.
			X	X	If CR4.OSXSAVE[bit 18]=0. If any one of following conditions applies: ▪ State requirement, Table 2-37 not met. ▪ Opcode independent #UD condition in Table 2-38. ▪ Operand encoding #UD conditions in Table 2-39. ▪ Opmask encoding #UD condition of Table 2-40. ▪ If EVEX.b != 0.
	X	X	X	X	If preceded by a LOCK prefix (F0H).
			X	X	If any REX, F2, F3, or 66 prefixes precede a EVEX prefix.
	X	X	X	X	If any corresponding CPUID feature flag is '0'.
Device Not Available, #NM	X	X	X	X	If CR0.TS[bit 3]=1.
Stack, SS(0)			X		For an illegal address in the SS segment.
				X	If a memory address referencing the SS segment is in a non-canonical form.
General Protection, #GP(0)			X		For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.
				X	If the memory address is in a non-canonical form.
	X	X			If any part of the operand lies outside the effective address space from 0 to FFFFH.
Page Fault #PF(fault-code)		X	X	X	For a page fault.
Alignment Check #AC(0)		X	X	X	If alignment checking is enabled and an unaligned memory reference of 8 bytes or less is made while the current privilege level is 3.
SIMD Floating-point Exception, #XM	X	X	X	X	If an unmasked SIMD floating-point exception, {sae} or {er} not set, and CR4.OSXMMEXCPT[bit 10] = 1.

2.7.4 Exceptions Type E4 and E4NF of EVEX-Encoded Instructions

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

Table 2-49. Type E4 Class Exception Conditions

Exception	Real	Virtual 80x86	Protected and Compatibility	64-bit	Cause of Exception
Invalid Opcode, #UD	X	X			If EVEX prefix present.
			X	X	If CR4.OSXSAVE[bit 18]=0. If any one of following conditions applies: ▪ State requirement, Table 2-37 not met. ▪ Opcode independent #UD condition in Table 2-38. ▪ Operand encoding #UD conditions in Table 2-39. ▪ Opmask encoding #UD condition of Table 2-40. ▪ If EVEX.b != 0 and in E4.nb subclass (see E4.nb entries in Table 2-43). ▪ If EVEX.L'L != 10b (VL=512).
	X	X	X	X	If preceded by a LOCK prefix (F0H).
			X	X	If any REX, F2, F3, or 66 prefixes precede a EVEX prefix.
	X	X	X	X	If any corresponding CPUID feature flag is '0'.
Device Not Available, #NM	X	X	X	X	If CR0.TS[bit 3]=1.
Stack, SS(0)			X		If fault suppression not set, and an illegal address in the SS segment.
				X	If fault suppression not set, and a memory address referencing the SS segment is in a non-canonical form.
General Protection, #GP(0)			X		If fault suppression not set, and an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.
				X	If fault suppression not set, and the memory address is in a non-canonical form.
	X	X			If fault suppression not set, and any part of the operand lies outside the effective address space from 0 to FFFFH.
Page Fault #PF(fault-code)		X	X	X	If fault suppression not set, and a page fault.

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

Table 2-50. Type E4NF Class Exception Conditions

Exception	Real	Virtual 80x86	Protected and Compatibility	64-bit	Cause of Exception
Invalid Opcode, #UD	X	X			If EVEX prefix present.
			X	X	If CR4.OSXSAVE[bit 18]=0. If any one of following conditions applies: ▪ State requirement, Table 2-37 not met. ▪ Opcode independent #UD condition in Table 2-38. ▪ Operand encoding #UD conditions in Table 2-39. ▪ Opmask encoding #UD condition of Table 2-40. ▪ If EVEX.b != 0 and in E4NF.nb subclass (see E4NF.nb entries in Table 2-43). ▪ If EVEX.L'L != 10b (VL=512).
	X	X	X	X	If preceded by a LOCK prefix (F0H).
			X	X	If any REX, F2, F3, or 66 prefixes precede a EVEX prefix.
	X	X	X	X	If any corresponding CPUID feature flag is '0'.
Device Not Available, #NM	X	X	X	X	If CR0.TS[bit 3]=1.
Stack, SS(0)			X		For an illegal address in the SS segment.
				X	If a memory address referencing the SS segment is in a non-canonical form.
General Protection, #GP(0)			X		For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.
				X	If the memory address is in a non-canonical form.
	X	X			If any part of the operand lies outside the effective address space from 0 to FFFFH.
Page Fault #PF(fault-code)		X	X	X	For a page fault.

2.7.5 Exceptions Type E5 and E5NF

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

Table 2-51. Type E5 Class Exception Conditions

Exception	Real	Virtual 80x86	Protected and Compatibility	64-bit	Cause of Exception
Invalid Opcode, #UD	X	X			If EVEX prefix present.
			X	X	If CR4.OSXSAVE[bit 18]=0. If any one of following conditions applies: ▪ State requirement, Table 2-37 not met. ▪ Opcode independent #UD condition in Table 2-38. ▪ Operand encoding #UD conditions in Table 2-39. ▪ Opmask encoding #UD condition of Table 2-40. ▪ If EVEX.b != 0. ▪ If EVEX.L'L != 10b (VL=512).
	X	X	X	X	If preceded by a LOCK prefix (F0H).
			X	X	If any REX, F2, F3, or 66 prefixes precede a EVEX prefix.
	X	X	X	X	If any corresponding CPUID feature flag is '0'.
Device Not Available, #NM	X	X	X	X	If CR0.TS[bit 3]=1.
Stack, SS(0)			X		If fault suppression not set, and an illegal address in the SS segment.
				X	If fault suppression not set, and a memory address referencing the SS segment is in a non-canonical form.
General Protection, #GP(0)			X		If fault suppression not set, and an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.
				X	If fault suppression not set, and the memory address is in a non-canonical form.
	X	X			If fault suppression not set, and any part of the operand lies outside the effective address space from 0 to FFFFH.
Page Fault #PF(fault-code)		X	X	X	If fault suppression not set, and a page fault.
Alignment Check #AC(0)		X	X	X	If alignment checking is enabled and an unaligned memory reference of 8 bytes or less is made while the current privilege level is 3.

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

Table 2-52. Type E5NF Class Exception Conditions

Exception	Real	Virtual 80x86	Protected and Compatibility	64-bit	Cause of Exception
Invalid Opcode, #UD	X	X			If EVEX prefix present.
			X	X	If CR4.OSXSAVE[bit 18]=0. If any one of following conditions applies: ▪ State requirement, Table 2-37 not met. ▪ Opcode independent #UD condition in Table 2-38. ▪ Operand encoding #UD conditions in Table 2-39. ▪ Opmask encoding #UD condition of Table 2-40. ▪ If EVEX.b != 0. ▪ If EVEX.L'L != 10b (VL=512).
	X	X	X	X	If preceded by a LOCK prefix (F0H).
			X	X	If any REX, F2, F3, or 66 prefixes precede a EVEX prefix.
	X	X	X	X	If any corresponding CPUID feature flag is '0'.
Device Not Available, #NM	X	X	X	X	If CR0.TS[bit 3]=1.
Stack, SS(0)			X		If an illegal address in the SS segment.
				X	If a memory address referencing the SS segment is in a non-canonical form.
General Protection, #GP(0)			X		If an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.
				X	If the memory address is in a non-canonical form.
	X	X			If any part of the operand lies outside the effective address space from 0 to FFFFH.
Page Fault #PF(fault-code)		X	X	X	For a page fault.
Alignment Check #AC(0)		X	X	X	If alignment checking is enabled and an unaligned memory reference of 8 bytes or less is made while the current privilege level is 3.

2.7.6 Exceptions Type E6 and E6NF

Table 2-53. Type E6 Class Exception Conditions

Exception	Real	Virtual 80x86	Protected and Compatibility	64-bit	Cause of Exception
Invalid Opcode, #UD	X	X			If EVEX prefix present.
			X	X	If CR4.OSXSAVE[bit 18]=0. If any one of following conditions applies: ▪ State requirement, Table 2-37 not met. ▪ Opcode independent #UD condition in Table 2-38. ▪ Operand encoding #UD conditions in Table 2-39. ▪ Opmask encoding #UD condition of Table 2-40. ▪ If EVEX.b != 0. ▪ If EVEX.L'L != 10b (VL=512).
			X	X	If preceded by a LOCK prefix (F0H).
			X	X	If any REX, F2, F3, or 66 prefixes precede a EVEX prefix.
			X	X	If any corresponding CPUID feature flag is '0'.
Device Not Available, #NM			X	X	If CR0.TS[bit 3]=1.
Stack, SS(0)			X		If fault suppression not set, and an illegal address in the SS segment.
				X	If fault suppression not set, and a memory address referencing the SS segment is in a non-canonical form.
General Protection, #GP(0)			X		If fault suppression not set, and an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.
				X	If fault suppression not set, and the memory address is in a non-canonical form.
Page Fault #PF(fault-code)			X	X	If fault suppression not set, and a page fault.
Alignment Check #AC(0)			X	X	For 4 or 8 byte memory references if alignment checking is enabled and an unaligned memory reference of 8 bytes or less is made while the current privilege level is 3.

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

Table 2-54. Type E6NF Class Exception Conditions

Exception	Real	Virtual 80x86	Protected and Compatibility	64-bit	Cause of Exception
Invalid Opcode, #UD	X	X			If EVEX prefix present.
			X	X	If CR4.OSXSAVE[bit 18]=0. If any one of following conditions applies: ▪ State requirement, Table 2-37 not met. ▪ Opcode independent #UD condition in Table 2-38. ▪ Operand encoding #UD conditions in Table 2-39. ▪ Opmask encoding #UD condition of Table 2-40. ▪ If EVEX.b != 0. ▪ If EVEX.L'L != 10b (VL=512).
			X	X	If preceded by a LOCK prefix (F0H).
			X	X	If any REX, F2, F3, or 66 prefixes precede a EVEX prefix.
			X	X	If any corresponding CPUID feature flag is '0'.
Device Not Available, #NM			X	X	If CR0.TS[bit 3]=1.
Stack, SS(0)			X		For an illegal address in the SS segment.
				X	If a memory address referencing the SS segment is in a non-canonical form.
General Protection, #GP(0)			X		For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.
				X	If the memory address is in a non-canonical form.
Page Fault #PF(fault-code)			X	X	For a page fault.
Alignment Check #AC(0)			X	X	For 4 or 8 byte memory references if alignment checking is enabled and an unaligned memory reference of 8 bytes or less is made while the current privilege level is 3.

2.7.7 Exceptions Type E7NM

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

Table 2-55. Type E7NM Class Exception Conditions

Exception	Real	Virtual 80x86	Protected and Compatibility	64-bit	Cause of Exception
Invalid Opcode, #UD	X	X			If EVEX prefix present.
			X	X	If CR4.OSXSAVE[bit 18]=0. If any one of following conditions applies: ▪ State requirement, Table 2-37 not met. ▪ Opcode independent #UD condition in Table 2-38. ▪ Operand encoding #UD conditions in Table 2-39. ▪ Opmask encoding #UD condition of Table 2-40. ▪ If EVEX.b != 0. ▪ Instruction specific EVEX.L'L restriction not met.
	X	X	X	X	If preceded by a LOCK prefix (F0H).
			X	X	If any REX, F2, F3, or 66 prefixes precede a EVEX prefix.
	X	X	X	X	If any corresponding CPUID feature flag is '0'.
Device Not Available, #NM			X	X	If CR0.TS[bit 3]=1.

2.7.8 Exceptions Type E9 and E9NF

EVEX-encoded vector or partial-vector instructions that do not cause no SIMD FP exception and support memory fault suppression follow exception class E9.

Table 2-56. Type E9 Class Exception Conditions

Exception	Real	Virtual 80x86	Protected and Compatibility	64-bit	Cause of Exception
Invalid Opcode, #UD	X	X			If EVEX prefix present.
			X	X	If CR4.OSXSAVE[bit 18]=0. If any one of following conditions applies: ▪ State requirement, Table 2-37 not met. ▪ Opcode independent #UD condition in Table 2-38. ▪ Operand encoding #UD conditions in Table 2-39. ▪ Opmask encoding #UD condition of Table 2-40. ▪ If EVEX.b != 0. ▪ If EVEX.L'L != 00b (VL=128).
	X	X	X	X	If preceded by a LOCK prefix (F0H).
			X	X	If any REX, F2, F3, or 66 prefixes precede a EVEX prefix.
	X	X	X	X	If any corresponding CPUID feature flag is '0'.
Device Not Available, #NM	X	X	X	X	If CR0.TS[bit 3]=1.
Stack, SS(0)			X		If fault suppression not set, and an illegal address in the SS segment.
				X	If fault suppression not set, and a memory address referencing the SS segment is in a non-canonical form.
General Protection, #GP(0)			X		If fault suppression not set, and an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.
				X	If fault suppression not set, and the memory address is in a non-canonical form.
	X	X			If fault suppression not set, and any part of the operand lies outside the effective address space from 0 to FFFFH.
Page Fault #PF(fault-code)		X	X	X	If fault suppression not set, and a page fault.
Alignment Check #AC(0)		X	X	X	If alignment checking is enabled and an unaligned memory reference of 8 bytes or less is made while the current privilege level is 3.

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

Table 2-57. Type E9NF Class Exception Conditions

Exception	Real	Virtual 80x86	Protected and Compatibility	64-bit	Cause of Exception
Invalid Opcode, #UD	X	X			If EVEX prefix present.
			X	X	If CR4.OSXSAVE[bit 18]=0. If any one of following conditions applies: ▪ State requirement, Table 2-37 not met. ▪ Opcode independent #UD condition in Table 2-38. ▪ Operand encoding #UD conditions in Table 2-39. ▪ Opmask encoding #UD condition of Table 2-40. ▪ If EVEX.b != 0. ▪ If EVEX.L'L != 00b (VL=128).
	X	X	X	X	If preceded by a LOCK prefix (F0H).
			X	X	If any REX, F2, F3, or 66 prefixes precede a EVEX prefix.
	X	X	X	X	If any corresponding CPUID feature flag is '0'.
Device Not Available, #NM	X	X	X	X	If CR0.TS[bit 3]=1.
Stack, SS(0)			X		If an illegal address in the SS segment.
				X	If a memory address referencing the SS segment is in a non-canonical form.
General Protection, #GP(0)			X		If an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.
				X	If the memory address is in a non-canonical form.
	X	X			If any part of the operand lies outside the effective address space from 0 to FFFFH.
Page Fault #PF(fault-code)		X	X	X	For a page fault.
Alignment Check #AC(0)		X	X	X	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

2.7.9 Exceptions Type E10

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

Table 2-58. Type E10 Class Exception Conditions

Exception	Real	Virtual 80x86	Protected and Compatibility	64-bit	Cause of Exception
Invalid Opcode, #UD	X	X			If EVEX prefix present.
			X	X	If CR4.OSXSAVE[bit 18]=0. If any one of following conditions applies: ▪ State requirement, Table 2-37 not met. ▪ Opcode independent #UD condition in Table 2-38. ▪ Operand encoding #UD conditions in Table 2-39. ▪ Opmask encoding #UD condition of Table 2-40. ▪ If EVEX.b != 0.
	X	X	X	X	If preceded by a LOCK prefix (F0H).
			X	X	If any REX, F2, F3, or 66 prefixes precede a EVEX prefix.
	X	X	X	X	If any corresponding CPUID feature flag is '0'.
Device Not Available, #NM	X	X	X	X	If CR0.TS[bit 3]=1.
Stack, SS(0)			X		If fault suppression not set, and an illegal address in the SS segment.
				X	If fault suppression not set, and a memory address referencing the SS segment is in a non-canonical form.
General Protection, #GP(0)			X		If fault suppression not set, and an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.
				X	If fault suppression not set, and the memory address is in a non-canonical form.
	X	X			If fault suppression not set, and any part of the operand lies outside the effective address space from 0 to FFFFH.
Page Fault #PF(fault-code)		X	X	X	If fault suppression not set, and a page fault.
Alignment Check #AC(0)		X	X	X	If alignment checking is enabled and an unaligned memory reference of 8 bytes or less is made while the current privilege level is 3.

EVEX-encoded scalar instructions that must be encoded with VEX.L'L = 0, do not cause SIMD FP exception nor support memory fault suppression follow exception class E10NF.

Table 2-59. Type E10NF Class Exception Conditions

Exception	Real	Virtual 80x86	Protected and Compatibility	64-bit	Cause of Exception
Invalid Opcode, #UD	X	X			If EVEX prefix present.
			X	X	If CR4.OSXSAVE[bit 18]=0. If any one of following conditions applies: ▪ State requirement, Table 2-37 not met. ▪ Opcode independent #UD condition in Table 2-38. ▪ Operand encoding #UD conditions in Table 2-39. ▪ Opmask encoding #UD condition of Table 2-40. ▪ If EVEX.b != 0.
	X	X	X	X	If preceded by a LOCK prefix (F0H).
			X	X	If any REX, F2, F3, or 66 prefixes precede a EVEX prefix.
	X	X	X	X	If any corresponding CPUID feature flag is '0'.
Device Not Available, #NM	X	X	X	X	If CR0.TS[bit 3]=1.
Stack, SS(0)			X		If fault suppression not set, and an illegal address in the SS segment.
				X	If fault suppression not set, and a memory address referencing the SS segment is in a non-canonical form.
General Protection, #GP(0)			X		If fault suppression not set, and an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.
				X	If fault suppression not set, and the memory address is in a non-canonical form.
	X	X			If fault suppression not set, and any part of the operand lies outside the effective address space from 0 to FFFFH.
Page Fault #PF(fault-code)		X	X	X	If fault suppression not set, and a page fault.
Alignment Check #AC(0)		X	X	X	If alignment checking is enabled and an unaligned memory reference of 8 bytes or less is made while the current privilege level is 3.

2.7.10 Exception Type E11 (EVEX-only, mem arg no AC, floating-point exceptions)

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

Table 2-60. Type E11 Class Exception Conditions

Exception	Real	Virtual 80x86	Protected and Compatibility	64-bit	Cause of Exception
Invalid Opcode, #UD	X	X			If EVEX prefix present.
			X	X	If CR4.OSXSAVE[bit 18]=0. If any one of following conditions applies: ▪ State requirement, Table 2-37 not met. ▪ Opcode independent #UD condition in Table 2-38. ▪ Operand encoding #UD conditions in Table 2-39. ▪ Opmask encoding #UD condition of Table 2-40. ▪ If EVEX.b != 0. ▪ If EVEX.L'L != 10b (VL=512).
	X	X	X	X	If preceded by a LOCK prefix (F0H).
			X	X	If any REX, F2, F3, or 66 prefixes precede a EVEX prefix.
	X	X	X	X	If any corresponding CPUID feature flag is '0'.
Device Not Available, #NM	X	X	X	X	If CR0.TS[bit 3]=1.
Stack, SS(0)			X		If fault suppression not set, and an illegal address in the SS segment.
				X	If fault suppression not set, and a memory address referencing the SS segment is in a non-canonical form.
General Protection, #GP(0)			X		If fault suppression not set, and an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.
				X	If fault suppression not set, and the memory address is in a non-canonical form.
	X	X			If fault suppression not set, and any part of the operand lies outside the effective address space from 0 to FFFFH.
Page Fault #PF (fault-code)		X	X	X	If fault suppression not set, and a page fault.
SIMD Floating-Point Exception, #XM	X	X	X	X	If an unmasked SIMD floating-point exception, {sae} not set, and CR4.OSXMMEX-CPT[bit 10] = 1.

2.7.11 Exception Type E12 and E12NP (VSIB mem arg, no AC, no floating-point exceptions)

Table 2-61. Type E12 Class Exception Conditions

Exception	Real	Virtual 80x86	Protected and Compatibility	64-bit	Cause of Exception
Invalid Opcode, #UD	X	X			If EVEX prefix present.
			X	X	If CR4.OSXSAVE[bit 18]=0. If any one of following conditions applies: ▪ State requirement, Table 2-37 not met. ▪ Opcode independent #UD condition in Table 2-38. ▪ Operand encoding #UD conditions in Table 2-39. ▪ Opmask encoding #UD condition of Table 2-40. ▪ If EVEX.b != 0. ▪ If EVEX.L'L != 10b (VL=512). ▪ If vvvv != 1111b.
	X	X	X	X	If preceded by a LOCK prefix (F0H).
			X	X	If any REX, F2, F3, or 66 prefixes precede a VEX prefix.
	X	X	X	NA	If address size attribute is 16 bit.
	X	X	X	X	If ModR/M.mod = '11b'.
	X	X	X	X	If ModR/M.rm != '100b'.
	X	X	X	X	If any corresponding CPUID feature flag is '0'.
	X	X	X	X	If k0 is used (gather or scatter operation).
	X	X	X	X	If index = destination register (gather operation).
Device Not Available, #NM	X	X	X	X	If CR0.TS[bit 3]=1.
Stack, SS(0)			X		For an illegal address in the SS segment.
				X	If a memory address referencing the SS segment is in a non-canonical form.
General Protection, #GP(0)			X		For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.
				X	If the memory address is in a non-canonical form.
	X	X			If any part of the operand lies outside the effective address space from 0 to FFFFH.
Page Fault #PF (fault-code)		X	X	X	For a page fault.

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

Table 2-62. Type E12NP Class Exception Conditions

Exception	Real	Virtual 80x86	Protected and Compatibility	64-bit	Cause of Exception
Invalid Opcode, #UD	X	X			If EVEX prefix present.
			X	X	If CR4.OSXSAVE[bit 18]=0. If any one of following conditions applies: ▪ State requirement, Table 2-37 not met. ▪ Opcode independent #UD condition in Table 2-38. ▪ Operand encoding #UD conditions in Table 2-39. ▪ Opmask encoding #UD condition of Table 2-40. ▪ If EVEX.b != 0. ▪ If EVEX.L'L != 10b (VL=512).
	X	X	X	X	If preceded by a LOCK prefix (F0H).
			X	X	If any REX, F2, F3, or 66 prefixes precede a VEX prefix.
	X	X	X	NA	If address size attribute is 16 bit.
	X	X	X	X	If ModR/M.mod = '11b'.
	X	X	X	X	If ModR/M.rm != '100b'.
	X	X	X	X	If any corresponding CPUID feature flag is '0'.
	X	X	X	X	If k0 is used (gather or scatter operation).
Device Not Available, #NM	X	X	X	X	If CR0.TS[bit 3]=1.
Stack, SS(0)			X		For an illegal address in the SS segment.
				X	If a memory address referencing the SS segment is in a non-canonical form.
General Protection, #GP(0)			X		For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.
				X	If the memory address is in a non-canonical form.
	X	X			If any part of the operand lies outside the effective address space from 0 to FFFFH.

2.8 EXCEPTION CLASSIFICATIONS OF OPMASK INSTRUCTIONS

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

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

Table 2-63. TYPE K20 Exception Definition (VEX-Encoded OpMask Instructions w/o Memory Arg)

Exception	Real	Virtual 80x86	Protected and Compatibility	64-bit	Cause of Exception
Invalid Opcode, #UD	X	X	X	X	If relevant CPUID feature flag is '0'.
	X	X			If a VEX prefix is present.
			X	X	If CR4.OSXSAVE[bit 18]=0. If any one of following conditions applies: ▪ State requirement, Table 2-37 not met. ▪ Opcode independent #UD condition in Table 2-38. ▪ Operand encoding #UD conditions in Table 2-39.
			X	X	If any REX, F2, F3, or 66 prefixes precede a VEX prefix.
			X	X	If ModRM:[7:6] != 11b.
Device Not Available, #NM	X	X	X	X	If CR0.TS[bit 3]=1.

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

Table 2-64. TYPE K21 Exception Definition (VEX-Encoded OpMask Instructions Addressing Memory)

Exception	Real	Virtual 80x86	Protected and Compatibility	64-bit	Cause of Exception
Invalid Opcode, #UD	X	X	X	X	If relevant CPUID feature flag is '0'.
	X	X			If a VEX prefix is present.
			X	X	If CR4.OSXSAVE[bit 18]=0. If any one of following conditions applies: ▪ State requirement, Table 2-37 not met. ▪ Opcode independent #UD condition in Table 2-38. ▪ Operand encoding #UD conditions in Table 2-39.
Device Not Available, #NM	X	X	X	X	If CR0.TS[bit 3]=1.
			X	X	If any REX, F2, F3, or 66 prefixes precede a VEX prefix.
Stack, SS(0)	X	X	X		For an illegal address in the SS segment.
				X	If a memory address referencing the SS segment is in a non-canonical form.
General Protection, #GP(0)			X		For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments. If the DS, ES, FS, or GS register is used to access memory and it contains a null segment selector.
				X	If the memory address is in a non-canonical form.
	X	X			If any part of the operand lies outside the effective address space from 0 to FFFFH.
Page Fault #PF(fault-code)		X	X	X	For a page fault.
Alignment Check #AC(0)		X	X	X	If alignment checking is enabled and an unaligned memory reference of 8 bytes or less is made while the current privilege level is 3.

7. Updates to Chapter 3, Volume 2A

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

Change to this chapter: Updates to the following instructions: CALL, CLI, CPUID, CVTSI2SD, INSERTPS, INT n/INTO/ INT 3, IRET/IRETD, JMP. Updated operand encoding table for instructions with tuple types, breaking out tuple types into a separate column. Corrected naming typos in operation section of various instructions.

CHAPTER 3
INSTRUCTION SET REFERENCE, A-L

This chapter describes the instruction set for the Intel 64 and IA-32 architectures (A-L) in IA-32e, protected, virtual-8086, and real-address modes of operation. The set includes general-purpose, x87 FPU, MMX, SSE/SSE2/SSE3/SSSE3/SSE4, AESNI/PCLMULQDQ, AVX and system instructions. See also Chapter 4, "Instruction Set Reference, M-U," in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2B*, and Chapter 5, "Instruction Set Reference, V-Z," in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2C*.

For each instruction, each operand combination is described. A description of the instruction and its operand, an operational description, a description of the effect of the instructions on flags in the EFLAGS register, and a summary of exceptions that can be generated are also provided.

3.1 INTERPRETING THE INSTRUCTION REFERENCE PAGES

This section describes the format of information contained in the instruction reference pages in this chapter. It explains notational conventions and abbreviations used in these sections.

3.1.1 Instruction Format

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

CMC—Complement Carry Flag [this is an example]

Opcode	Instruction	Op/En	64/32-bit Mode	CPUID Feature Flag	Description
F5	CMC	ZO	V/V	NA	Complement carry flag.

Instruction Operand Encoding

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

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

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

- **NP** — Indicates the use of 66/F2/F3 prefixes (beyond those already part of the instructions opcode) are not allowed with the instruction. Such use will either cause an invalid-opcode exception (#UD) or result in the encoding for a different instruction.
- **REX.W** — Indicates the use of a REX prefix that affects operand size or instruction semantics. The ordering of the REX prefix and other optional/mandatory instruction prefixes are discussed Chapter 2. Note that REX prefixes that promote legacy instructions to 64-bit behavior are not listed explicitly in the opcode column.
- ***/digit*** — A digit between 0 and 7 indicates that the ModR/M byte of the instruction uses only the r/m (register or memory) operand. The reg field contains the digit that provides an extension to the instruction's opcode.
- ***/r*** — Indicates that the ModR/M byte of the instruction contains a register operand and an r/m operand.
- **cb, cw, cd, cp, co, ct** — A 1-byte (cb), 2-byte (cw), 4-byte (cd), 6-byte (cp), 8-byte (co) or 10-byte (ct) value following the opcode. This value is used to specify a code offset and possibly a new value for the code segment register.
- **ib, iw, id, io** — A 1-byte (ib), 2-byte (iw), 4-byte (id) or 8-byte (io) immediate operand to the instruction that follows the opcode, ModR/M bytes or scale-indexing bytes. The opcode determines if the operand is a signed value. All words, doublewords and quadwords are given with the low-order byte first.
- **+rb, +rw, +rd, +ro** — Indicated the lower 3 bits of the opcode byte is used to encode the register operand without a modR/M byte. The instruction lists the corresponding hexadecimal value of the opcode byte with low 3 bits as 000b. In non-64-bit mode, a register code, from 0 through 7, is added to the hexadecimal value of the opcode byte. In 64-bit mode, indicates the four bit field of REX.b and opcode[2:0] field encodes the register operand of the instruction. "+ro" is applicable only in 64-bit mode. See Table 3-1 for the codes.
- **+i** — A number used in floating-point instructions when one of the operands is ST(i) from the FPU register stack. The number i (which can range from 0 to 7) is added to the hexadecimal byte given at the left of the plus sign to form a single opcode byte.

Table 3-1. Register Codes Associated With +rb, +rw, +rd, +ro

byte register			word register			dword register			quadword register (64-Bit Mode only)		
Register	REX.B	Reg Field	Register	REX.B	Reg Field	Register	REX.B	Reg Field	Register	REX.B	Reg Field
AL	None	0	AX	None	0	EAX	None	0	RAX	None	0
CL	None	1	CX	None	1	ECX	None	1	RCX	None	1
DL	None	2	DX	None	2	EDX	None	2	RDX	None	2
BL	None	3	BX	None	3	EBX	None	3	RBX	None	3
AH	Not encodable (N.E.)	4	SP	None	4	ESP	None	4	N/A	N/A	N/A
CH	N.E.	5	BP	None	5	EBP	None	5	N/A	N/A	N/A
DH	N.E.	6	SI	None	6	ESI	None	6	N/A	N/A	N/A
BH	N.E.	7	DI	None	7	EDI	None	7	N/A	N/A	N/A
SPL	Yes	4	SP	None	4	ESP	None	4	RSP	None	4
BPL	Yes	5	BP	None	5	EBP	None	5	RBP	None	5
SIL	Yes	6	SI	None	6	ESI	None	6	RSI	None	6
DIL	Yes	7	DI	None	7	EDI	None	7	RDI	None	7
Registers R8 - R15 (see below): Available in 64-Bit Mode Only											

Table 3-1. Register Codes Associated With +rb, +rw, +rd, +ro (Contd.)

byte register			word register			dword register			quadword register (64-Bit Mode only)		
Register	REX.B	Reg Field	Register	REX.B	Reg Field	Register	REX.B	Reg Field	Register	REX.B	Reg Field
R8L	Yes	0	R8W	Yes	0	R8D	Yes	0	R8	Yes	0
R9L	Yes	1	R9W	Yes	1	R9D	Yes	1	R9	Yes	1
R10L	Yes	2	R10W	Yes	2	R10D	Yes	2	R10	Yes	2
R11L	Yes	3	R11W	Yes	3	R11D	Yes	3	R11	Yes	3
R12L	Yes	4	R12W	Yes	4	R12D	Yes	4	R12	Yes	4
R13L	Yes	5	R13W	Yes	5	R13D	Yes	5	R13	Yes	5
R14L	Yes	6	R14W	Yes	6	R14D	Yes	6	R14	Yes	6
R15L	Yes	7	R15W	Yes	7	R15D	Yes	7	R15	Yes	7

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

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

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

- **VEX** — Indicates the presence of the VEX prefix is required. The VEX prefix can be encoded using the three-byte form (the first byte is C4H), or using the two-byte form (the first byte is C5H). The two-byte form of VEX only applies to those instructions that do not require the following fields to be encoded: VEX.mmmmm, VEX.W, VEX.X, VEX.B. Refer to Section 2.3 for more detail on the VEX prefix.

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

  — **NDS, NDD, DDS:** Specifies that VEX.vvvv field is valid for the encoding of a register operand:

    - VEX.NDS: VEX.vvvv encodes the first source register in an instruction syntax where the content of source registers will be preserved.
    - VEX.NDD: VEX.vvvv encodes the destination register that cannot be encoded by ModR/M:reg field.
    - VEX.DDS: VEX.vvvv encodes the second source register in a three-operand instruction syntax where the content of first source register will be overwritten by the result.
    - If none of NDS, NDD, and DDS is present, VEX.vvvv must be 1111b (i.e. VEX.vvvv does not encode an operand). The VEX.vvvv field can be encoded using either the 2-byte or 3-byte form of the VEX prefix.

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

    - If VEX.256 is present in the opcode column: The semantics of the instruction must be encoded with VEX.L = 1. An attempt to encode this instruction with VEX.L= 0 can result in one of two situations: (a) if VEX.128 version is defined, the processor will behave according to the defined VEX.128 behavior; (b) an #UD occurs if there is no VEX.128 version defined.
    - If VEX.128 is present in the opcode column but there is no VEX.256 version defined for the same opcode byte: Two situations apply: (a) For VEX-encoded, 128-bit SIMD integer instructions, software must encode the instruction with VEX.L = 0. The processor will treat the opcode byte encoded with VEX.L= 1 by causing an #UD exception; (b) For VEX-encoded, 128-bit packed floating-point instructions, software must encode the instruction with VEX.L = 0. The processor will treat the opcode byte encoded with VEX.L= 1 by causing an #UD exception (e.g. VMOVLPS).
    - If VEX.LIG is present in the opcode column: The VEX.L value is ignored. This generally applies to VEX-encoded scalar SIMD floating-point instructions. Scalar SIMD floating-point instruction can be distin-

guished from the mnemonic of the instruction. Generally, the last two letters of the instruction mnemonic would be either "SS", "SD", or "SI" for SIMD floating-point conversion instructions.

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

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

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

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

— **W0:** VEX.W=0.

— **W1:** VEX.W=1.

— The presence of W0/W1 in the opcode column applies to two situations: (a) it is treated as an extended opcode bit, (b) the instruction semantics support an operand size promotion to 64-bit of a general-purpose register operand or a 32-bit memory operand. The presence of W1 in the opcode column implies the opcode must be encoded using the 3-byte form of the VEX prefix. The presence of W0 in the opcode column does not preclude the opcode to be encoded using the C5H form of the VEX prefix, if the semantics of the opcode does not require other VEX subfields not present in the two-byte form of the VEX prefix. Please see Section 2.3 on the subfield definitions within VEX.

— **WIG:** can use C5H form (if not requiring VEX.mmmmm) or VEX.W value is ignored in the C4H form of VEX prefix.

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

- **opcode** — Instruction opcode.
- **/is4** — An 8-bit immediate byte is present containing a source register specifier in either imm8[7:4] (for 64-bit mode) or imm8[6:4] (for 32-bit mode), and instruction-specific payload in imm8[3:0].
- In general, the encoding o f VEX.R, VEX.X, VEX.B field are not shown explicitly in the opcode column. The encoding scheme of VEX.R, VEX.X, VEX.B fields must follow the rules defined in Section 2.3.

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

- **EVEX** — The EVEX prefix is encoded using the four-byte form (the first byte is 62H). Refer to Section 2.6.1 for more detail on the EVEX prefix.

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

  — **NDS, NDD, DDS:** implies that EVEX.vvvv (and EVEX.v') field is valid for the encoding of an operand. It may specify either the source register (NDS) or the destination register (NDD). DDS expresses a syntax where vvvv encodes the second source register in a three-operand instruction syntax where the content of first source register will be overwritten by the result. If both NDS and NDD absent (i.e. EVEX.vvvv does not encode an operand), EVEX.vvvv must be 1111b (and EVEX.v' must be 1b).

  — **128, 256, 512, LIG:** This corresponds to the vector length; three values are allowed by EVEX: 512-bit, 256-bit and 128-bit. Alternatively, vector length is ignored (LIG) for certain instructions; this typically applies to scalar instructions operating on one data element of a vector register.

  - — **66,F2,F3:** The presence of these value maps to the EVEX.pp field encodings. The corresponding VEX.pp value affects the "opcode" byte in the same way as if a SIMD prefix (66H, F2H or F3H) does to the ensuing opcode byte. Thus a non-zero encoding of VEX.pp may be considered as an implied 66H/F2H/F3H prefix.
  - — **0F,0F3A,0F38:** The presence maps to a valid encoding of the EVEX.mmm field. Only three encoded values of EVEX.mmm are defined as valid, corresponding to the escape byte sequence of 0FH, 0F3AH and 0F38H. The effect of a valid EVEX.mmm encoding on the ensuing opcode byte is the same as if the corresponding escape byte sequence on the ensuing opcode byte for non-EVEX encoded instructions. Thus a valid encoding of EVEX.mmm may be considered as an implied escape byte sequence of either 0FH, 0F3AH or 0F38H.
  - — **W0:** EVEX.W=0.
  - — **W1:** EVEX.W=1.
  - — **WIG:** EVEX.W bit ignored
- **opcode** — Instruction opcode.
- In general, the encoding of EVEX.R and R', EVEX.X and X', and EVEX.B and B' fields are not shown explicitly in the opcode column.

3.1.1.3 Instruction Column in the Opcode Summary Table

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

- **rel8** — A relative address in the range from 128 bytes before the end of the instruction to 127 bytes after the end of the instruction.
- **rel16, rel32** — A relative address within the same code segment as the instruction assembled. The rel16 symbol applies to instructions with an operand-size attribute of 16 bits; the rel32 symbol applies to instructions with an operand-size attribute of 32 bits.
- **ptr16:16, ptr16:32** — A far pointer, typically to a code segment different from that of the instruction. The notation *16:16* indicates that the value of the pointer has two parts. The value to the left of the colon is a 16-bit selector or value destined for the code segment register. The value to the right corresponds to the offset within the destination segment. The ptr16:16 symbol is used when the instruction's operand-size attribute is 16 bits; the ptr16:32 symbol is used when the operand-size attribute is 32 bits.
- **r8** — One of the byte general-purpose registers: AL, CL, DL, BL, AH, CH, DH, BH, BPL, SPL, DIL and SIL; or one of the byte registers (R8L - R15L) available when using REX.R and 64-bit mode.
- **r16** — One of the word general-purpose registers: AX, CX, DX, BX, SP, BP, SI, DI; or one of the word registers (R8-R15) available when using REX.R and 64-bit mode.
- **r32** — One of the doubleword general-purpose registers: EAX, ECX, EDX, EBX, ESP, EBP, ESI, EDI; or one of the doubleword registers (R8D - R15D) available when using REX.R in 64-bit mode.
- **r64** — One of the quadword general-purpose registers: RAX, RBX, RCX, RDX, RDI, RSI, RBP, RSP, R8–R15. These are available when using REX.R and 64-bit mode.
- **imm8** — An immediate byte value. The imm8 symbol is a signed number between –128 and +127 inclusive. For instructions in which imm8 is combined with a word or doubleword operand, the immediate value is sign-extended to form a word or doubleword. The upper byte of the word is filled with the topmost bit of the immediate value.
- **imm16** — An immediate word value used for instructions whose operand-size attribute is 16 bits. This is a number between –32,768 and +32,767 inclusive.
- **imm32** — An immediate doubleword value used for instructions whose operand-size attribute is 32 bits. It allows the use of a number between +2,147,483,647 and –2,147,483,648 inclusive.
- **imm64** — An immediate quadword value used for instructions whose operand-size attribute is 64 bits. The value allows the use of a number between +9,223,372,036,854,775,807 and –9,223,372,036,854,775,808 inclusive.

- **r/m8** — A byte operand that is either the contents of a byte general-purpose register (AL, CL, DL, BL, AH, CH, DH, BH, BPL, SPL, DIL and SIL) or a byte from memory. Byte registers R8L - R15L are available using REX.R in 64-bit mode.
- **r/m16** — A word general-purpose register or memory operand used for instructions whose operand-size attribute is 16 bits. The word general-purpose registers are: AX, CX, DX, BX, SP, BP, SI, DI. The contents of memory are found at the address provided by the effective address computation. Word registers R8W - R15W are available using REX.R in 64-bit mode.
- **r/m32** — A doubleword general-purpose register or memory operand used for instructions whose operand-size attribute is 32 bits. The doubleword general-purpose registers are: EAX, ECX, EDX, EBX, ESP, EBP, ESI, EDI. The contents of memory are found at the address provided by the effective address computation. Doubleword registers R8D - R15D are available when using REX.R in 64-bit mode.
- **r/m64** — A quadword general-purpose register or memory operand used for instructions whose operand-size attribute is 64 bits when using REX.W. Quadword general-purpose registers are: RAX, RBX, RCX, RDX, RDI, RSI, RBP, RSP, R8–R15; these are available only in 64-bit mode. The contents of memory are found at the address provided by the effective address computation.
- **m** — A 16-, 32- or 64-bit operand in memory.
- **m8** — A byte operand in memory, usually expressed as a variable or array name, but pointed to by the DS:(E)SI or ES:(E)DI registers. In 64-bit mode, it is pointed to by the RSI or RDI registers.
- **m16** — A word operand in memory, usually expressed as a variable or array name, but pointed to by the DS:(E)SI or ES:(E)DI registers. This nomenclature is used only with the string instructions.
- **m32** — A doubleword operand in memory, usually expressed as a variable or array name, but pointed to by the DS:(E)SI or ES:(E)DI registers. This nomenclature is used only with the string instructions.
- **m64** — A memory quadword operand in memory.
- **m128** — A memory double quadword operand in memory.
- **m16:16, m16:32 & m16:64** — A memory operand containing a far pointer composed of two numbers. The number to the left of the colon corresponds to the pointer's segment selector. The number to the right corresponds to its offset.
- **m16&32, m16&16, m32&32, m16&64** — A memory operand consisting of data item pairs whose sizes are indicated on the left and the right side of the ampersand. All memory addressing modes are allowed. The m16&16 and m32&32 operands are used by the BOUND instruction to provide an operand containing an upper and lower bounds for array indices. The m16&32 operand is used by LIDT and LGDT to provide a word with which to load the limit field, and a doubleword with which to load the base field of the corresponding GDTR and IDTR registers. The m16&64 operand is used by LIDT and LGDT in 64-bit mode to provide a word with which to load the limit field, and a quadword with which to load the base field of the corresponding GDTR and IDTR registers.
- **moffs8, moffs16, moffs32, moffs64** — A simple memory variable (memory offset) of type byte, word, or doubleword used by some variants of the MOV instruction. The actual address is given by a simple offset relative to the segment base. No ModR/M byte is used in the instruction. The number shown with moffs indicates its size, which is determined by the address-size attribute of the instruction.
- **Sreg** — A segment register. The segment register bit assignments are ES = 0, CS = 1, SS = 2, DS = 3, FS = 4, and GS = 5.
- **m32fp, m64fp, m80fp** — A single-precision, double-precision, and double extended-precision (respectively) floating-point operand in memory. These symbols designate floating-point values that are used as operands for x87 FPU floating-point instructions.
- **m16int, m32int, m64int** — A word, doubleword, and quadword integer (respectively) operand in memory. These symbols designate integers that are used as operands for x87 FPU integer instructions.
- **ST or ST(0)** — The top element of the FPU register stack.
- **ST(i)** — The i^{th} element from the top of the FPU register stack ($i \leftarrow 0$ through 7).
- **mm** — An MMX register. The 64-bit MMX registers are: MM0 through MM7.
- **mm/m32** — The low order 32 bits of an MMX register or a 32-bit memory operand. The 64-bit MMX registers are: MM0 through MM7. The contents of memory are found at the address provided by the effective address computation.

- **mm/m64** — An MMX register or a 64-bit memory operand. The 64-bit MMX registers are: MM0 through MM7. The contents of memory are found at the address provided by the effective address computation.
- **xmm** — An XMM register. The 128-bit XMM registers are: XMM0 through XMM7; XMM8 through XMM15 are available using REX.R in 64-bit mode.
- **xmm/m32**— An XMM register or a 32-bit memory operand. The 128-bit XMM registers are XMM0 through XMM7; XMM8 through XMM15 are available using REX.R in 64-bit mode. The contents of memory are found at the address provided by the effective address computation.
- **xmm/m64** — An XMM register or a 64-bit memory operand. The 128-bit SIMD floating-point registers are XMM0 through XMM7; XMM8 through XMM15 are available using REX.R in 64-bit mode. The contents of memory are found at the address provided by the effective address computation.
- **xmm/m128** — An XMM register or a 128-bit memory operand. The 128-bit XMM registers are XMM0 through XMM7; XMM8 through XMM15 are available using REX.R in 64-bit mode. The contents of memory are found at the address provided by the effective address computation.
- **<XMM0>**— Indicates implied use of the XMM0 register.

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

  Some instructions use the XMM0 register as the third source operand, indicated by <XMM0>. The use of the third XMM register operand is implicit in the instruction encoding and does not affect the ModR/M encoding.
- **ymm** — A YMM register. The 256-bit YMM registers are: YMM0 through YMM7**;** YMM8 through YMM15 are available in 64-bit mode.
- **m256** — A 32-byte operand in memory. This nomenclature is used only with AVX instructions.
- **ymm/m256** — A YMM register or 256-bit memory operand.
- **<YMM0>**— Indicates use of the YMM0 register as an implicit argument.
- **bnd** — A 128-bit bounds register. BND0 through BND3.
- **mib** — A memory operand using SIB addressing form, where the index register is not used in address calculation, Scale is ignored. Only the base and displacement are used in effective address calculation.
- **m512** — A 64-byte operand in memory.
- **zmm/m512** — A ZMM register or 512-bit memory operand.
- **{k1}{z}** — A mask register used as instruction writemask. The 64-bit k registers are: k1 through k7. Writemask specification is available exclusively via EVEX prefix. The masking can either be done as a merging-masking, where the old values are preserved for masked out elements or as a zeroing masking. The type of masking is determined by using the EVEX.z bit.
- **{k1}** — Without {z}: a mask register used as instruction writemask for instructions that do not allow zeroing-masking but support merging-masking. This corresponds to instructions that require the value of the aaa field to be different than 0 (e.g., gather) and store-type instructions which allow only merging-masking.
- **k1** — A mask register used as a regular operand (either destination or source). The 64-bit k registers are: k0 through k7.
- **mV** — A vector memory operand; the operand size is dependent on the instruction.
- **vm32{x,y, z}** — A vector array of memory operands specified using VSIB memory addressing. The array of memory addresses are specified using a common base register, a constant scale factor, and a vector index register with individual elements of 32-bit index value in an XMM register (vm32x), a YMM register (vm32y) or a ZMM register (vm32z).
- **vm64{x,y, z}** — A vector array of memory operands specified using VSIB memory addressing. The array of memory addresses are specified using a common base register, a constant scale factor, and a vector index register with individual elements of 64-bit index value in an XMM register (vm64x), a YMM register (vm64y) or a ZMM register (vm64z).
- **zmm/m512/m32bcst** — An operand that can be a ZMM register, a 512-bit memory location or a 512-bit vector loaded from a 32-bit memory location.
- **zmm/m512/m64bcst** — An operand that can be a ZMM register, a 512-bit memory location or a 512-bit vector loaded from a 64-bit memory location.

- **<ZMM0>** — Indicates use of the ZMM0 register as an implicit argument.
- **{er}** — Indicates support for embedded rounding control, which is only applicable to the register-register form of the instruction. This also implies support for SAE (Suppress All Exceptions).
- **{sae}** — Indicates support for SAE (Suppress All Exceptions). This is used for instructions that support SAE, but do not support embedded rounding control.
- **SRC1** — Denotes the first source operand in the instruction syntax of an instruction encoded with the VEX/EVEX prefix and having two or more source operands.
- **SRC2** — Denotes the second source operand in the instruction syntax of an instruction encoded with the VEX/EVEX prefix and having two or more source operands.
- **SRC3** — Denotes the third source operand in the instruction syntax of an instruction encoded with the VEX/EVEX prefix and having three source operands.
- **SRC** — The source in a single-source instruction.
- **DST** — the destination in an instruction. This field is encoded by reg_field.

3.1.1.4 Operand Encoding Column in the Instruction Summary Table

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

EVEX encoded instructions employ compressed disp8*N encoding of the displacement bytes, where N is defined in Table 2-34 and Table 2-35, according to tupletypes. The Op/En column of an EVEX encoded instruction uses an abbreviation that corresponds to the tupletype abbreviation (and may include an additional abbreviation related to ModR/M and vvvv encoding). Most EVEX encoded instructions with VEX encoded equivalent have the ModR/M and vvvv encoding order. In such cases, the Tuple abbreviation is shown and the ModR/M, vvvv encoding abbreviation may be omitted.

NOTES

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

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

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

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

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

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

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

3.1.1.6 CPUID Support Column in the Instruction Summary Table

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

3.1.1.7 Description Column in the Instruction Summary Table

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

3.1.1.8 Description Section

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

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

- **Legacy SSE** — Refers to SSE, SSE2, SSE3, SSSE3, SSE4, AESNI, PCLMULQDQ and any future instruction sets referencing XMM registers and encoded without a VEX prefix.
- **VEX.vvvv** — The VEX bit field specifying a source or destination register (in 1's complement form).
- **rm_field** — shorthand for the ModR/M *r/m* field and any REX.B
- **reg_field** — shorthand for the ModR/M *reg* field and any REX.R

3.1.1.9 Operation Section

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

- Comments are enclosed within the symbol pairs "(*" and "*)".
- Compound statements are enclosed in keywords, such as: IF, THEN, ELSE and FI for an if statement; DO and OD for a do statement; or CASE... OF for a case statement.
- A register name implies the contents of the register. A register name enclosed in brackets implies the contents of the location whose address is contained in that register. For example, ES:[DI] indicates the contents of the location whose ES segment relative address is in register DI. [SI] indicates the contents of the address contained in register SI relative to the SI register's default segment (DS) or the overridden segment.
- Parentheses around the "E" in a general-purpose register name, such as (E)SI, indicates that the offset is read from the SI register if the address-size attribute is 16, from the ESI register if the address-size attribute is 32. Parentheses around the "R" in a general-purpose register name, (R)SI, in the presence of a 64-bit register definition such as (R)SI, indicates that the offset is read from the 64-bit RSI register if the address-size attribute is 64.
- Brackets are used for memory operands where they mean that the contents of the memory location is a segment-relative offset. For example, [SRC] indicates that the content of the source operand is a segment-relative offset.
- A ← B indicates that the value of B is assigned to A.
- The symbols =, ≠, >, <, ≥, and ≤ are relational operators used to compare two values: meaning equal, not equal, greater or equal, less or equal, respectively. A relational expression such as A = B is TRUE if the value of A is equal to B; otherwise it is FALSE.

- The expression "« COUNT" and "» COUNT" indicates that the destination operand should be shifted left or right by the number of bits indicated by the count operand.

The following identifiers are used in the algorithmic descriptions:

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

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

  See "Operand-Size and Address-Size Attributes" in Chapter 3 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*, for guidelines on how these attributes are determined.

- **StackAddrSize** — Represents the stack address-size attribute associated with the instruction, which has a value of 16, 32 or 64-bits. See "Address-Size Attribute for Stack" in Chapter 6, "Procedure Calls, Interrupts, and Exceptions," of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*.
- **SRC** — Represents the source operand.
- **DEST** — Represents the destination operand.
- **MAXVL** — The maximum vector register width pertaining to the instruction. This is not the vector-length encoding in the instruction's encoding but is instead determined by the current value of XCR0. For details, refer to the table below. Note that the value of MAXVL is the largest of the features enabled. Future processors may define new bits in XCR0 whose setting may imply other values for MAXVL.

MAXVL Definition

XCR0 Component	MAXVL
XCR0.SSE	128
XCR0.AVX	256
XCR0.{ZMM_Hi256, Hi16_ZMM, OPMASK}	512

The following functions are used in the algorithmic descriptions:

- **ZeroExtend(value)** — Returns a value zero-extended to the operand-size attribute of the instruction. For example, if the operand-size attribute is 32, zero extending a byte value of –10 converts the byte from F6H to a doubleword value of 000000F6H. If the value passed to the ZeroExtend function and the operand-size attribute are the same size, ZeroExtend returns the value unaltered.
- **SignExtend(value)** — Returns a value sign-extended to the operand-size attribute of the instruction. For example, if the operand-size attribute is 32, sign extending a byte containing the value –10 converts the byte from F6H to a doubleword value of FFFFFFF6H. If the value passed to the SignExtend function and the operand-size attribute are the same size, SignExtend returns the value unaltered.
- **SaturateSignedWordToSignedByte** — Converts a signed 16-bit value to a signed 8-bit value. If the signed 16-bit value is less than –128, it is represented by the saturated value -128 (80H); if it is greater than 127, it is represented by the saturated value 127 (7FH).

- **SaturateSignedDwordToSignedWord** — Converts a signed 32-bit value to a signed 16-bit value. If the signed 32-bit value is less than –32768, it is represented by the saturated value –32768 (8000H); if it is greater than 32767, it is represented by the saturated value 32767 (7FFFH).
- **SaturateSignedWordToUnsignedByte** — Converts a signed 16-bit value to an unsigned 8-bit value. If the signed 16-bit value is less than zero, it is represented by the saturated value zero (00H); if it is greater than 255, it is represented by the saturated value 255 (FFH).
- **SaturateToSignedByte** — Represents the result of an operation as a signed 8-bit value. If the result is less than –128, it is represented by the saturated value –128 (80H); if it is greater than 127, it is represented by the saturated value 127 (7FH).
- **SaturateToSignedWord** — Represents the result of an operation as a signed 16-bit value. If the result is less than –32768, it is represented by the saturated value –32768 (8000H); if it is greater than 32767, it is represented by the saturated value 32767 (7FFFH).
- **SaturateToUnsignedByte** — Represents the result of an operation as a signed 8-bit value. If the result is less than zero it is represented by the saturated value zero (00H); if it is greater than 255, it is represented by the saturated value 255 (FFH).
- **SaturateToUnsignedWord** — Represents the result of an operation as a signed 16-bit value. If the result is less than zero it is represented by the saturated value zero (00H); if it is greater than 65535, it is represented by the saturated value 65535 (FFFFH).
- **LowOrderWord(DEST * SRC)** — Multiplies a word operand by a word operand and stores the least significant word of the doubleword result in the destination operand.
- **HighOrderWord(DEST * SRC)** — Multiplies a word operand by a word operand and stores the most significant word of the doubleword result in the destination operand.
- **Push(value)** — Pushes a value onto the stack. The number of bytes pushed is determined by the operand-size attribute of the instruction. See the "Operation" subsection of the "PUSH—Push Word, Doubleword or Quadword Onto the Stack" section in Chapter 4 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2B*.
- **Pop()** — removes the value from the top of the stack and returns it. The statement EAX ← Pop(); assigns to EAX the 32-bit value from the top of the stack. Pop will return either a word, a doubleword or a quadword depending on the operand-size attribute. See the "Operation" subsection in the "POP—Pop a Value from the Stack" section of Chapter 4 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2B*.
- **PopRegisterStack** — Marks the FPU ST(0) register as empty and increments the FPU register stack pointer (TOP) by 1.
- **Switch-Tasks** — Performs a task switch.
- **Bit(BitBase, BitOffset)** — Returns the value of a bit within a bit string. The bit string is a sequence of bits in memory or a register. Bits are numbered from low-order to high-order within registers and within memory bytes. If the BitBase is a register, the BitOffset can be in the range 0 to [15, 31, 63] depending on the mode and register size. See Figure 3-1: the function Bit[RAX, 21] is illustrated.

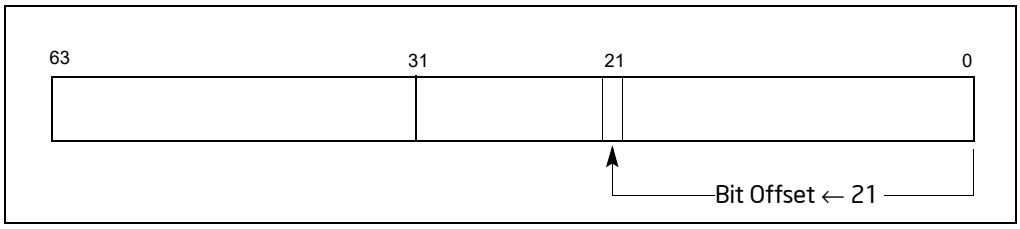


Figure 3-1. Bit Offset for BIT[RAX, 21]

If BitBase is a memory address, the BitOffset can range has different ranges depending on the operand size (see Table 3-2).

Table 3-2. Range of Bit Positions Specified by Bit Offset Operands

Operand Size	Immediate BitOffset	Register BitOffset
16	0 to 15	-2^{15} to $2^{15}-1$
32	0 to 31	-2^{31} to $2^{31}-1$
64	0 to 63	-2^{63} to $2^{63}-1$

The addressed bit is numbered (Offset MOD 8) within the byte at address (BitBase + (BitOffset DIV 8)) where DIV is signed division with rounding towards negative infinity and MOD returns a positive number (see Figure 3-2).

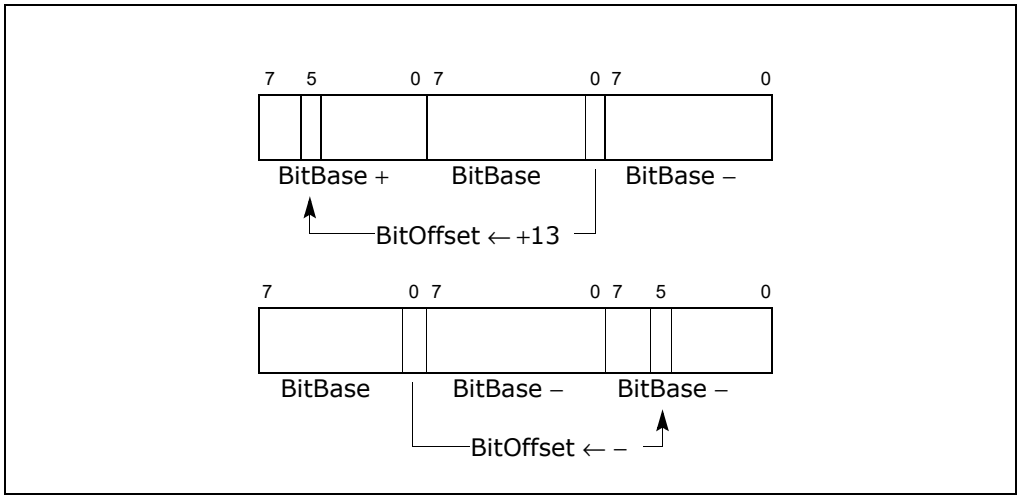


Figure 3-2. Memory Bit Indexing

3.1.1.10 Intel® C/C++ Compiler Intrinsics Equivalents Section

The Intel C/C++ compiler intrinsic functions give access to the full power of the Intel Architecture Instruction Set, while allowing the compiler to optimize register allocation and instruction scheduling for faster execution. Most of these functions are associated with a single IA instruction, although some may generate multiple instructions or different instructions depending upon how they are used. In particular, these functions are used to invoke instructions that perform operations on vector registers that can hold multiple data elements. These SIMD instructions use the following data types.

- __m128, __m256 and __m512 can represent 4, 8 or 16 packed single-precision floating-point values, and are used with the vector registers and SSE, AVX, or AVX-512 instruction set extension families. The __m128 data type is also used with various single-precision floating-point scalar instructions that perform calculations using only the lowest 32 bits of a vector register; the remaining bits of the result come from one of the sources or are set to zero depending upon the instruction.
- __m128d, __m256d and __m512d can represent 2, 4 or 8 packed double-precision floating-point values, and are used with the vector registers and SSE, AVX, or AVX-512 instruction set extension families. The __m128d data type is also used with various double-precision floating-point scalar instructions that perform calculations using only the lowest 64 bits of a vector register; the remaining bits of the result come from one of the sources or are set to zero depending upon the instruction.
- __m128i, __m256i and __m512i can represent integer data in bytes, words, doublewords, quadwords, and occasionally larger data types.

Each of these data types incorporates in its name the number of bits it can hold. For example, the __m128 type holds 128 bits, and because each single-precision floating-point value is 32 bits long the __m128 type holds (128/32) or four values. Normally the compiler will allocate memory for these data types on an even multiple of the size of the type. Such aligned memory locations may be faster to read and write than locations at other addresses.

These SIMD data types are not basic Standard C data types or C++ objects, so they may be used only with the assignment operator, passed as function arguments, and returned from a function call. If you access the internal members of these types directly, or indirectly by using them in a union, there may be side effects affecting optimization, so it is recommended to use them only with the SIMD instruction intrinsic functions described in this manual or the Intel C/C++ compiler documentation.

Many intrinsic functions names are prefixed with an indicator of the vector length and suffixed by an indicator of the vector element data type, although some functions do not follow the rules below. The prefixes are:

- _mm_ indicates that the function operates on 128-bit (or sometimes 64-bit) vectors.
- _mm256_ indicates the function operates on 256-bit vectors.
- _mm512_ indicates that the function operates on 512-bit vectors.

The suffixes include:

- _ps, which indicates a function that operates on packed single-precision floating-point data. Packed single-precision floating-point data corresponds to arrays of the C/C++ type *float* with either 4, 8 or 16 elements. Values of this type can be loaded from an array using the *_mm_loadu_ps*, *_mm256_loadu_ps*, or *_mm512_loadu_ps* functions, or created from individual values using *_mm_set_ps*, *_mm256_set_ps*, or *_mm512_set_ps* functions, and they can be stored in an array using *_mm_storeu_ps*, *_mm256_storeu_ps*, or *_mm512_storeu_ps*.
- _ss, which indicates a function that operates on scalar single-precision floating-point data. Single-precision floating-point data corresponds to the C/C++ type *float*, and values of type float can be converted to type __m128 for use with these functions using the *_mm_set_ss* function, and converted back using the *_mm_cvtss_f32* function. When used with functions that operate on packed single-precision floating-point data the scalar element corresponds with the first packed value.
- _pd, which indicates a function that operates on packed double-precision floating-point data. Packed double-precision floating-point data corresponds to arrays of the C/C++ type *double* with either 2, 4, or 8 elements. Values of this type can be loaded from an array using the *_mm_loadu_pd*, *_mm256_loadu_pd*, or *_mm512_loadu_pd* functions, or created from individual values using *_mm_set_pd*, *_mm2566_set_pd*, or *_mm512_set_pd* functions, and they can be stored in an array using *_mm_storeu_pd*, *_mm256_storeu_pd*, or *_mm512_storeu_pd*.
- _sd, which indicates a function that operates on scalar double-precision floating-point data. Double-precision floating-point data corresponds to the C/C++ type *double*, and values of type double can be converted to type __m128d for use with these functions using the *_mm_set_sd* function, and converted back using the *_mm_cvtsd_f64* function. When used with functions that operate on packed double-precision floating-point data the scalar element corresponds with the first packed value.
- _epi8, which indicates a function that operates on packed 8-bit signed integer values. Packed 8-bit signed integers correspond to an array of *signed char* with 16, 32 or 64 elements. Values of this type can be created from individual elements using *_mm_set_epi8*, *_mm256_set_epi8*, or *_mm512_set_epi8* functions.
- _epi16, which indicates a function that operates on packed 16-bit signed integer values. Packed 16-bit signed integers correspond to an array of *short* with 8, 16 or 32 elements. Values of this type can be created from individual elements using *_mm_set_epi16*, *_mm256_set_epi16*, or *_mm512_set_epi16* functions.
- _epi32, which indicates a function that operates on packed 32-bit signed integer values. Packed 32-bit signed integers correspond to an array of *int* with 4, 8 or 16 elements. Values of this type can be created from individual elements using *_mm_set_epi32*, *_mm256_set_epi32*, or *_mm512_set_epi32* functions.
- _epi64, which indicates a function that operates on packed 64-bit signed integer values. Packed 64-bit signed integers correspond to an array of *long long* (or *long* if it is a 64-bit data type) with 2, 4 or 8 elements. Values of this type can be created from individual elements using *_mm_set_epi32*, *_mm256_set_epi32*, or *_mm512_set_epi32* functions.
- _epu8, which indicates a function that operates on packed 8-bit unsigned integer values. Packed 8-bit unsigned integers correspond to an array of *unsigned char* with 16, 32 or 64 elements.

- _epu16, which indicates a function that operates on packed 16-bit unsigned integer values. Packed 16-bit unsigned integers correspond to an array of *unsigned short* with 8, 16 or 32 elements.
- _epu32, which indicates a function that operates on packed 32-bit unsigned integer values. Packed 32-bit unsigned integers correspond to an array of *unsigned* with 4, 8 or 16 elements.
- _epu64, which indicates a function that operates on packed 64-bit unsigned integer values. Packed 64-bit unsigned integers correspond to an array of *unsigned long long* (or *unsigned long* if it is a 64-bit data type) with 2, 4 or 8 elements.
- _si128, which indicates a function that operates on a single 128-bit value of type __m128i.
- _si256, which indicates a function that operates on a single a 256-bit value of type __m256i.
- _si512, which indicates a function that operates on a single a 512-bit value of type __m512i.

Values of any packed integer type can be loaded from an array using the *_mm_loadu_si128*, *_mm256_loadu_si256*, or *_mm512_loadu_si512* functions, and they can be stored in an array using *_mm_storeu_si128*, *_mm256_storeu_si256*, or *_mm512_storeu_si512*.

These functions and data types are used with the SSE, AVX, and AVX-512 instruction set extension families. In addition there are similar functions that correspond to MMX instructions. These are less frequently used because they require additional state management, and only operate on 64-bit packed integer values.

The declarations of Intel C/C++ compiler intrinsic functions may reference some non-standard data types, such as __int64. The C Standard header *stdint.h* defines similar platform-independent types, and the documentation for that header gives characteristics that apply to corresponding non-standard types according to the following table.

Table 3-3. Standard and Non-standard Data Types

Non-standard Type	Standard Type (from stdint.h)
__int64	int64_t
unsigned __int64	uint64_t
__int32	int32_t
unsigned __int32	uint32_t
__int16	int16_t
unsigned __int16	uint16_t

For a more detailed description of each intrinsic function and additional information related to its usage, refer to the online Intel Intrinsics Guide, https://software.intel.com/sites/landingpage/IntrinsicsGuide.

3.1.1.11 Flags Affected Section

The "Flags Affected" section lists the flags in the EFLAGS register that are affected by the instruction. When a flag is cleared, it is equal to 0; when it is set, it is equal to 1. The arithmetic and logical instructions usually assign values to the status flags in a uniform manner (see Appendix A, "EFLAGS Cross-Reference," in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*). Non-conventional assignments are described in the "Operation" section. The values of flags listed as **undefined** may be changed by the instruction in an indeterminate manner. Flags that are not listed are unchanged by the instruction.

3.1.1.12 FPU Flags Affected Section

The floating-point instructions have an "FPU Flags Affected" section that describes how each instruction can affect the four condition code flags of the FPU status word.

3.1.1.13 Protected Mode Exceptions Section

The "Protected Mode Exceptions" section lists the exceptions that can occur when the instruction is executed in protected mode and the reasons for the exceptions. Each exception is given a mnemonic that consists of a pound

sign (#) followed by two letters and an optional error code in parentheses. For example, #GP(0) denotes a general protection exception with an error code of 0. Table 3-4 associates each two-letter mnemonic with the corresponding exception vector and name. See Chapter 6, "Procedure Calls, Interrupts, and Exceptions," in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*, for a detailed description of the exceptions.

Application programmers should consult the documentation provided with their operating systems to determine the actions taken when exceptions occur.

Table 3-4. Intel 64 and IA-32 General Exceptions

Vector	Name	Source	Protected Mode[1]	Real Address Mode	Virtual 8086 Mode
0	#DE—Divide Error	DIV and IDIV instructions.	Yes	Yes	Yes
1	#DB—Debug	Any code or data reference.	Yes	Yes	Yes
3	#BP—Breakpoint	INT 3 instruction.	Yes	Yes	Yes
4	#OF—Overflow	INTO instruction.	Yes	Yes	Yes
5	#BR—BOUND Range Exceeded	BOUND instruction.	Yes	Yes	Yes
6	#UD—Invalid Opcode (Undefined Opcode)	UD instruction or reserved opcode.	Yes	Yes	Yes
7	#NM—Device Not Available (No Math Coprocessor)	Floating-point or WAIT/FWAIT instruction.	Yes	Yes	Yes
8	#DF—Double Fault	Any instruction that can generate an exception, an NMI, or an INTR.	Yes	Yes	Yes
10	#TS—Invalid TSS	Task switch or TSS access.	Yes	Reserved	Yes
11	#NP—Segment Not Present	Loading segment registers or accessing system segments.	Yes	Reserved	Yes
12	#SS—Stack Segment Fault	Stack operations and SS register loads.	Yes	Yes	Yes
13	#GP—General Protection[2]	Any memory reference and other protection checks.	Yes	Yes	Yes
14	#PF—Page Fault	Any memory reference.	Yes	Reserved	Yes
16	#MF—Floating-Point Error (Math Fault)	Floating-point or WAIT/FWAIT instruction.	Yes	Yes	Yes
17	#AC—Alignment Check	Any data reference in memory.	Yes	Reserved	Yes
18	#MC—Machine Check	Model dependent machine check errors.	Yes	Yes	Yes
19	#XM—SIMD Floating-Point Numeric Error	SSE/SSE2/SSE3 floating-point instructions.	Yes	Yes	Yes

NOTES:

1. Apply to protected mode, compatibility mode, and 64-bit mode.
2. In the real-address mode, vector 13 is the segment overrun exception.

3.1.1.14 Real-Address Mode Exceptions Section

The "Real-Address Mode Exceptions" section lists the exceptions that can occur when the instruction is executed in real-address mode (see Table 3-4).

3.1.1.15 Virtual-8086 Mode Exceptions Section

The "Virtual-8086 Mode Exceptions" section lists the exceptions that can occur when the instruction is executed in virtual-8086 mode (see Table 3-4).

3.1.1.16 Floating-Point Exceptions Section

The "Floating-Point Exceptions" section lists exceptions that can occur when an x87 FPU floating-point instruction is executed. All of these exception conditions result in a floating-point error exception (#MF, exception 16) being generated. Table 3-5 associates a one- or two-letter mnemonic with the corresponding exception name. See "Floating-Point Exception Conditions" in Chapter 8 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*, for a detailed description of these exceptions.

Table 3-5. x87 FPU Floating-Point Exceptions

Mnemonic	Name	Source
#IS #IA	Floating-point invalid operation: - Stack overflow or underflow - Invalid arithmetic operation	 - x87 FPU stack overflow or underflow - Invalid FPU arithmetic operation
#Z	Floating-point divide-by-zero	Divide-by-zero
#D	Floating-point denormal operand	Source operand that is a denormal number
#O	Floating-point numeric overflow	Overflow in result
#U	Floating-point numeric underflow	Underflow in result
#P	Floating-point inexact result (precision)	Inexact result (precision)

3.1.1.17 SIMD Floating-Point Exceptions Section

The "SIMD Floating-Point Exceptions" section lists exceptions that can occur when an SSE/SSE2/SSE3 floating-point instruction is executed. All of these exception conditions result in a SIMD floating-point error exception (#XM, exception 19) being generated. Table 3-6 associates a one-letter mnemonic with the corresponding exception name. For a detailed description of these exceptions, refer to "SSE and SSE2 Exceptions", in Chapter 11 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*.

Table 3-6. SIMD Floating-Point Exceptions

Mnemonic	Name	Source
#I	Floating-point invalid operation	Invalid arithmetic operation or source operand
#Z	Floating-point divide-by-zero	Divide-by-zero
#D	Floating-point denormal operand	Source operand that is a denormal number
#O	Floating-point numeric overflow	Overflow in result
#U	Floating-point numeric underflow	Underflow in result
#P	Floating-point inexact result	Inexact result (precision)

3.1.1.18 Compatibility Mode Exceptions Section

This section lists exceptions that occur within compatibility mode.

3.1.1.19 64-Bit Mode Exceptions Section

This section lists exceptions that occur within 64-bit mode.

3.2 INSTRUCTIONS (A-L)

The remainder of this chapter provides descriptions of Intel 64 and IA-32 instructions (A-L). See also: Chapter 4, "Instruction Set Reference, M-U," in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2B*, and Chapter 5, "Instruction Set Reference, V-Z," in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2C*.

AAA—ASCII Adjust After Addition

Opcode	Instruction	Op/ En	64-bit Mode	Compat/ Leg Mode	Description
37	AAA	ZO	Invalid	Valid	ASCII adjust AL after addition.

Instruction Operand Encoding

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

Description

Adjusts the sum of two unpacked BCD values to create an unpacked BCD result. The AL register is the implied source and destination operand for this instruction. The AAA instruction is only useful when it follows an ADD instruction that adds (binary addition) two unpacked BCD values and stores a byte result in the AL register. The AAA instruction then adjusts the contents of the AL register to contain the correct 1-digit unpacked BCD result.

If the addition produces a decimal carry, the AH register increments by 1, and the CF and AF flags are set. If there was no decimal carry, the CF and AF flags are cleared and the AH register is unchanged. In either case, bits 4 through 7 of the AL register are set to 0.

This instruction executes as described in compatibility mode and legacy mode. It is not valid in 64-bit mode.

Operation

```
IF 64-Bit Mode
    THEN
        #UD;
    ELSE
        IF ((AL AND 0FH) > 9) or (AF = 1)
            THEN
                AX ← AX + 106H;
                AF ← 1;
                CF ← 1;
            ELSE
                AF ← 0;
                CF ← 0;
        FI;
        AL ← AL AND 0FH;
FI;
```

Flags Affected

The AF and CF flags are set to 1 if the adjustment results in a decimal carry; otherwise they are set to 0. The OF, SF, ZF, and PF flags are undefined.

Protected Mode Exceptions

#UD	If the LOCK prefix is used.

Real-Address Mode Exceptions

Same exceptions as protected mode.

Virtual-8086 Mode Exceptions

Same exceptions as protected mode.

Compatibility Mode Exceptions

Same exceptions as protected mode.

64-Bit Mode Exceptions

#UD	If in 64-bit mode.

AAD—ASCII Adjust AX Before Division

Opcode	Instruction	Op/ En	64-bit Mode	Compat/ Leg Mode	Description
D5 0A	AAD	ZO	Invalid	Valid	ASCII adjust AX before division.
D5 *ib*	AAD imm8	ZO	Invalid	Valid	Adjust AX before division to number base *imm8.*

Instruction Operand Encoding

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

Description

Adjusts two unpacked BCD digits (the least-significant digit in the AL register and the most-significant digit in the AH register) so that a division operation performed on the result will yield a correct unpacked BCD value. The AAD instruction is only useful when it precedes a DIV instruction that divides (binary division) the adjusted value in the AX register by an unpacked BCD value.

The AAD instruction sets the value in the AL register to (AL + (10 * AH)), and then clears the AH register to 00H. The value in the AX register is then equal to the binary equivalent of the original unpacked two-digit (base 10) number in registers AH and AL.

The generalized version of this instruction allows adjustment of two unpacked digits of any number base (see the "Operation" section below), by setting the *imm8* byte to the selected number base (for example, 08H for octal, 0AH for decimal, or 0CH for base 12 numbers). The AAD mnemonic is interpreted by all assemblers to mean adjust ASCII (base 10) values. To adjust values in another number base, the instruction must be hand coded in machine code (D5 *imm8*).

This instruction executes as described in compatibility mode and legacy mode. It is not valid in 64-bit mode.

Operation

```
IF 64-Bit Mode
    THEN
        #UD;
    ELSE
        tempAL ← AL;
        tempAH ← AH;
        AL ← (tempAL + (tempAH * imm8)) AND FFH;
        (* imm8 is set to 0AH for the AAD mnemonic.*)
        AH ← 0;
FI;
```

The immediate value (*imm8*) is taken from the second byte of the instruction.

Flags Affected

The SF, ZF, and PF flags are set according to the resulting binary value in the AL register; the OF, AF, and CF flags are undefined.

Protected Mode Exceptions

#UD	If the LOCK prefix is used.

Real-Address Mode Exceptions

Same exceptions as protected mode.

Virtual-8086 Mode Exceptions

Same exceptions as protected mode.

Compatibility Mode Exceptions

Same exceptions as protected mode.

64-Bit Mode Exceptions

#UD	If in 64-bit mode.

AAM—ASCII Adjust AX After Multiply

Opcode	Instruction	Op/ En	64-bit Mode	Compat/ Leg Mode	Description
D4 0A	AAM	ZO	Invalid	Valid	ASCII adjust AX after multiply.
D4 *ib*	AAM imm8	ZO	Invalid	Valid	Adjust AX after multiply to number base *imm8.*

Instruction Operand Encoding

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

Description

Adjusts the result of the multiplication of two unpacked BCD values to create a pair of unpacked (base 10) BCD values. The AX register is the implied source and destination operand for this instruction. The AAM instruction is only useful when it follows an MUL instruction that multiplies (binary multiplication) two unpacked BCD values and stores a word result in the AX register. The AAM instruction then adjusts the contents of the AX register to contain the correct 2-digit unpacked (base 10) BCD result.

The generalized version of this instruction allows adjustment of the contents of the AX to create two unpacked digits of any number base (see the "Operation" section below). Here, the *imm8* byte is set to the selected number base (for example, 08H for octal, 0AH for decimal, or 0CH for base 12 numbers). The AAM mnemonic is interpreted by all assemblers to mean adjust to ASCII (base 10) values. To adjust to values in another number base, the instruction must be hand coded in machine code (D4 *imm8*).

This instruction executes as described in compatibility mode and legacy mode. It is not valid in 64-bit mode.

Operation

```
IF 64-Bit Mode
    THEN
        #UD;
    ELSE
        tempAL ← AL;
        AH ← tempAL / imm8; (* imm8 is set to 0AH for the AAM mnemonic *)
        AL ← tempAL MOD imm8;
FI;
```

The immediate value (*imm8*) is taken from the second byte of the instruction.

Flags Affected

The SF, ZF, and PF flags are set according to the resulting binary value in the AL register. The OF, AF, and CF flags are undefined.

Protected Mode Exceptions

#DE	If an immediate value of 0 is used.
#UD	If the LOCK prefix is used.

Real-Address Mode Exceptions

Same exceptions as protected mode.

Virtual-8086 Mode Exceptions

Same exceptions as protected mode.

Compatibility Mode Exceptions

Same exceptions as protected mode.

64-Bit Mode Exceptions

#UD	If in 64-bit mode.

AAS—ASCII Adjust AL After Subtraction

Opcode	Instruction	Op/ En	64-bit Mode	Compat/ Leg Mode	Description
3F	AAS	ZO	Invalid	Valid	ASCII adjust AL after subtraction.

Instruction Operand Encoding

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

Description

Adjusts the result of the subtraction of two unpacked BCD values to create a unpacked BCD result. The AL register is the implied source and destination operand for this instruction. The AAS instruction is only useful when it follows a SUB instruction that subtracts (binary subtraction) one unpacked BCD value from another and stores a byte result in the AL register. The AAA instruction then adjusts the contents of the AL register to contain the correct 1-digit unpacked BCD result.

If the subtraction produced a decimal carry, the AH register decrements by 1, and the CF and AF flags are set. If no decimal carry occurred, the CF and AF flags are cleared, and the AH register is unchanged. In either case, the AL register is left with its top four bits set to 0.

This instruction executes as described in compatibility mode and legacy mode. It is not valid in 64-bit mode.

Operation

```
IF 64-bit mode
    THEN
        #UD;
    ELSE
        IF ((AL AND 0FH) > 9) or (AF = 1)
            THEN
                AX ← AX - 6;
                AH ← AH - 1;
                AF ← 1;
                CF ← 1;
                AL ← AL AND 0FH;
            ELSE
                CF ← 0;
                AF ← 0;
                AL ← AL AND 0FH;
        FI;
FI;
```

Flags Affected

The AF and CF flags are set to 1 if there is a decimal borrow; otherwise, they are cleared to 0. The OF, SF, ZF, and PF flags are undefined.

Protected Mode Exceptions

#UD	If the LOCK prefix is used.

Real-Address Mode Exceptions

Same exceptions as protected mode.

Virtual-8086 Mode Exceptions

Same exceptions as protected mode.

Compatibility Mode Exceptions

Same exceptions as protected mode.

64-Bit Mode Exceptions

#UD	If in 64-bit mode.

ADC—Add with Carry

Opcode	Instruction	Op/ En	64-bit Mode	Compat/ Leg Mode	Description
14 *ib*	ADC AL, *imm8*	I	Valid	Valid	Add with carry *imm8* to AL.
15 *iw*	ADC AX, *imm16*	I	Valid	Valid	Add with carry *imm16* to AX.
15 *id*	ADC EAX, *imm32*	I	Valid	Valid	Add with carry *imm32* to EAX.
REX.W + 15 *id*	ADC RAX, *imm32*	I	Valid	N.E.	Add with carry *imm32 sign extended to 64-bits* to RAX.
80 /2 *ib*	ADC *r/m8, imm8*	MI	Valid	Valid	Add with carry *imm8* to *r/m8.*
REX + 80 /2 *ib*	ADC *r/m8**, *imm8*	MI	Valid	N.E.	Add with carry *imm8* to *r/m8.*
81 /2 *iw*	ADC *r/m16, imm16*	MI	Valid	Valid	Add with carry *imm16* to *r/m16.*
81 /2 *id*	ADC *r/m32, imm32*	MI	Valid	Valid	Add with CF *imm32* to *r/m32.*
REX.W + 81 /2 *id*	ADC *r/m64, imm32*	MI	Valid	N.E.	Add with CF *imm32* sign extended to 64-bits to *r/m64.*
83 /2 *ib*	ADC *r/m16, imm8*	MI	Valid	Valid	Add with CF sign-extended *imm8* to *r/m16.*
83 /2 *ib*	ADC *r/m32, imm8*	MI	Valid	Valid	Add with CF sign-extended *imm8* into *r/m32.*
REX.W + 83 /2 *ib*	ADC *r/m64, imm8*	MI	Valid	N.E.	Add with CF sign-extended *imm8* into *r/m64.*
10 /r	ADC *r/m8, r8*	MR	Valid	Valid	Add with carry byte register to *r/m8.*
REX + 10 /r	ADC *r/m8**, *r8**	MR	Valid	N.E.	Add with carry byte register to *r/m64.*
11 /r	ADC *r/m16, r16*	MR	Valid	Valid	Add with carry *r16* to *r/m16.*
11 /r	ADC *r/m32, r32*	MR	Valid	Valid	Add with CF *r32* to *r/m32.*
REX.W + 11 /r	ADC *r/m64, r64*	MR	Valid	N.E.	Add with CF *r64* to *r/m64.*
12 /r	ADC *r8, r/m8*	RM	Valid	Valid	Add with carry *r/m8* to byte register.
REX + 12 /r	ADC *r8**, *r/m8**	RM	Valid	N.E.	Add with carry *r/m64* to byte register.
13 /r	ADC *r16, r/m16*	RM	Valid	Valid	Add with carry *r/m16* to *r16.*
13 /r	ADC *r32, r/m32*	RM	Valid	Valid	Add with CF *r/m32* to *r32.*
REX.W + 13 /r	ADC *r64, r/m64*	RM	Valid	N.E.	Add with CF *r/m64* to *r64.*

NOTES:

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

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
MR	ModRM:r/m (r, w)	ModRM:reg (r)	NA	NA
MI	ModRM:r/m (r, w)	imm8	NA	NA
I	AL/AX/EAX/RAX	imm8	NA	NA

Description

Adds the destination operand (first operand), the source operand (second operand), and the carry (CF) flag and stores the result in the destination operand. The destination operand can be a register or a memory location; the source operand can be an immediate, a register, or a memory location. (However, two memory operands cannot be used in one instruction.) The state of the CF flag represents a carry from a previous addition. When an immediate value is used as an operand, it is sign-extended to the length of the destination operand format.

The ADC instruction does not distinguish between signed or unsigned operands. Instead, the processor evaluates the result for both data types and sets the OF and CF flags to indicate a carry in the signed or unsigned result, respectively. The SF flag indicates the sign of the signed result.

The ADC instruction is usually executed as part of a multibyte or multiword addition in which an ADD instruction is followed by an ADC instruction.

This instruction can be used with a LOCK prefix to allow the instruction to be executed atomically.

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

Operation

```
DEST ← DEST + SRC + CF;
```

Intel C/C++ Compiler Intrinsic Equivalent

ADC: extern unsigned char _addcarry_u8(unsigned char c_in, unsigned char src1, unsigned char src2, unsigned char *sum_out);

ADC: extern unsigned char _addcarry_u16(unsigned char c_in, unsigned short src1, unsigned short src2, unsigned short *sum_out);

ADC: extern unsigned char _addcarry_u32(unsigned char c_in, unsigned int src1, unsigned char int, unsigned int *sum_out);

ADC: extern unsigned char _addcarry_u64(unsigned char c_in, unsigned __int64 src1, unsigned __int64 src2, unsigned __int64 *sum_out);

Flags Affected

The OF, SF, ZF, AF, CF, and PF flags are set according to the result.

Protected Mode Exceptions

#GP(0)	If the destination is located in a non-writable segment.
	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
	If the DS, ES, FS, or GS register is used to access memory and it contains a NULL segment selector.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If the LOCK prefix is used but the destination is not a memory operand.

Real-Address Mode Exceptions

#GP	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS	If a memory operand effective address is outside the SS segment limit.
#UD	If the LOCK prefix is used but the destination is not a memory operand.

Virtual-8086 Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.
#UD	If the LOCK prefix is used but the destination is not a memory operand.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If the LOCK prefix is used but the destination is not a memory operand.

ADCX — Unsigned Integer Addition of Two Operands with Carry Flag

Opcode/ Instruction	Op/ En	64/32bit Mode Support	CPUID Feature Flag	Description
66 0F 38 F6 /r ADCX r32, r/m32	RM	V/V	ADX	Unsigned addition of r32 with CF, r/m32 to r32, writes CF.
66 REX.w 0F 38 F6 /r ADCX r64, r/m64	RM	V/NE	ADX	Unsigned addition of r64 with CF, r/m64 to r64, writes CF.

Instruction Operand Encoding

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

Description

Performs an unsigned addition of the destination operand (first operand), the source operand (second operand) and the carry-flag (CF) and stores the result in the destination operand. The destination operand is a general-purpose register, whereas the source operand can be a general-purpose register or memory location. The state of CF can represent a carry from a previous addition. The instruction sets the CF flag with the carry generated by the unsigned addition of the operands.

The ADCX instruction is executed in the context of multi-precision addition, where we add a series of operands with a carry-chain. At the beginning of a chain of additions, we need to make sure the CF is in a desired initial state. Often, this initial state needs to be 0, which can be achieved with an instruction to zero the CF (e.g. XOR).

This instruction is supported in real mode and virtual-8086 mode. The operand size is always 32 bits if not in 64-bit mode.

In 64-bit mode, the default operation size is 32 bits. Using a REX Prefix in the form of REX.R permits access to additional registers (R8-15). Using REX Prefix in the form of REX.W promotes operation to 64 bits.

ADCX executes normally either inside or outside a transaction region.

Note: ADCX defines the OF flag differently than the ADD/ADC instructions as defined in *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A*.

Operation

```
IF OperandSize is 64-bit
    THEN CF:DEST[63:0] ← DEST[63:0] + SRC[63:0] + CF;
    ELSE CF:DEST[31:0] ← DEST[31:0] + SRC[31:0] + CF;
FI;
```

Flags Affected

CF is updated based on result. OF, SF, ZF, AF and PF flags are unmodified.

Intel C/C++ Compiler Intrinsic Equivalent

unsigned char _addcarryx_u32 (unsigned char c_in, unsigned int src1, unsigned int src2, unsigned int *sum_out);

unsigned char _addcarryx_u64 (unsigned char c_in, unsigned __int64 src1, unsigned __int64 src2, unsigned __int64 *sum_out);

SIMD Floating-Point Exceptions

None

Protected Mode Exceptions

#UD	If the LOCK prefix is used.
	If CPUID.(EAX=07H, ECX=0H):EBX.ADX[bit 19] = 0.
#SS(0)	For an illegal address in the SS segment.

#GP(0)	For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments. If the DS, ES, FS, or GS register is used to access memory and it contains a null segment selector.
#PF(fault-code)	For a page fault.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

Real-Address Mode Exceptions

#UD	If the LOCK prefix is used. If CPUID.(EAX=07H, ECX=0H):EBX.ADX[bit 19] = 0.
#SS(0)	For an illegal address in the SS segment.
#GP(0)	If any part of the operand lies outside the effective address space from 0 to FFFFH.

Virtual-8086 Mode Exceptions

#UD	If the LOCK prefix is used. If CPUID.(EAX=07H, ECX=0H):EBX.ADX[bit 19] = 0.
#SS(0)	For an illegal address in the SS segment.
#GP(0)	If any part of the operand lies outside the effective address space from 0 to FFFFH.
#PF(fault-code)	For a page fault.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#UD	If the LOCK prefix is used. If CPUID.(EAX=07H, ECX=0H):EBX.ADX[bit 19] = 0.
#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form.
#PF(fault-code)	For a page fault.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

ADD—Add

Opcode	Instruction	Op/ En	64-bit Mode	Compat/ Leg Mode	Description
04 *ib*	ADD AL, *imm8*	I	Valid	Valid	Add *imm8* to AL.
05 *iw*	ADD AX, *imm16*	I	Valid	Valid	Add *imm16* to AX.
05 *id*	ADD EAX, *imm32*	I	Valid	Valid	Add *imm32* to EAX.
REX.W + 05 *id*	ADD RAX, *imm32*	I	Valid	N.E.	Add *imm32 sign-extended to 64-bits* to RAX.
80 /0 *ib*	ADD *r/m8, imm8*	MI	Valid	Valid	Add *imm8* to *r/m8.*
REX + 80 /0 *ib*	ADD *r/m8**, *imm8*	MI	Valid	N.E.	Add *sign*-extended *imm8* to *r/m8.*
81 /0 *iw*	ADD *r/m16, imm16*	MI	Valid	Valid	Add *imm16* to *r/m16.*
81 /0 *id*	ADD *r/m32, imm32*	MI	Valid	Valid	Add *imm32* to *r/m32.*
REX.W + 81 /0 *id*	ADD *r/m64, imm32*	MI	Valid	N.E.	Add *imm32 sign-extended to 64-bits* to *r/m64.*
83 /0 *ib*	ADD *r/m16, imm8*	MI	Valid	Valid	Add *sign*-extended *imm8* to *r/m16.*
83 /0 *ib*	ADD *r/m32, imm8*	MI	Valid	Valid	Add *sign*-extended *imm8* to *r/m32.*
REX.W + 83 /0 *ib*	ADD *r/m64, imm8*	MI	Valid	N.E.	Add *sign*-extended *imm8* to *r/m64.*
00 /*r*	ADD *r/m8, r8*	MR	Valid	Valid	Add *r8* to *r/m8.*
REX + 00 /*r*	ADD *r/m8**, *r8**	MR	Valid	N.E.	Add *r8* to *r/m8.*
01 /*r*	ADD *r/m16, r16*	MR	Valid	Valid	Add *r16* to *r/m16.*
01 /*r*	ADD *r/m32, r32*	MR	Valid	Valid	Add r32 to *r/m32.*
REX.W + 01 /*r*	ADD *r/m64, r64*	MR	Valid	N.E.	Add r64 to *r/m64.*
02 /*r*	ADD *r8, r/m8*	RM	Valid	Valid	Add *r/m8* to *r8.*
REX + 02 /*r*	ADD *r8**, *r/m8**	RM	Valid	N.E.	Add *r/m8* to *r8.*
03 /*r*	ADD *r16, r/m16*	RM	Valid	Valid	Add *r/m16* to *r16.*
03 /*r*	ADD *r32, r/m32*	RM	Valid	Valid	Add *r/m32* to *r32.*
REX.W + 03 /*r*	ADD *r64, r/m64*	RM	Valid	N.E.	Add *r/m64* to *r64.*

NOTES:

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

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
MR	ModRM:r/m (r, w)	ModRM:reg (r)	NA	NA
MI	ModRM:r/m (r, w)	imm8	NA	NA
I	AL/AX/EAX/RAX	imm8	NA	NA

Description

Adds the destination operand (first operand) and the source operand (second operand) and then stores the result in the destination operand. The destination operand can be a register or a memory location; the source operand can be an immediate, a register, or a memory location. (However, two memory operands cannot be used in one instruction.) When an immediate value is used as an operand, it is sign-extended to the length of the destination operand format.

The ADD instruction performs integer addition. It evaluates the result for both signed and unsigned integer operands and sets the CF and OF flags to indicate a carry (overflow) in the signed or unsigned result, respectively. The SF flag indicates the sign of the signed result.

This instruction can be used with a LOCK prefix to allow the instruction to be executed atomically.

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

Operation

```
DEST ← DEST + SRC;
```

Flags Affected

The OF, SF, ZF, AF, CF, and PF flags are set according to the result.

Protected Mode Exceptions

#GP(0)	If the destination is located in a non-writable segment.
	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
	If the DS, ES, FS, or GS register is used to access memory and it contains a NULL segment selector.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If the LOCK prefix is used but the destination is not a memory operand.

Real-Address Mode Exceptions

#GP	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS	If a memory operand effective address is outside the SS segment limit.
#UD	If the LOCK prefix is used but the destination is not a memory operand.

Virtual-8086 Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.
#UD	If the LOCK prefix is used but the destination is not a memory operand.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If the LOCK prefix is used but the destination is not a memory operand.

ADDPD—Add Packed Double-Precision Floating-Point Values

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 58 /r ADDPD xmm1, xmm2/m128	A	V/V	SSE2	Add packed double-precision floating-point values from xmm2/mem to xmm1 and store result in xmm1.
VEX.NDS.128.66.0F.WIG 58 /r VADDPD xmm1,xmm2, xmm3/m128	B	V/V	AVX	Add packed double-precision floating-point values from xmm3/mem to xmm2 and store result in xmm1.
VEX.NDS.256.66.0F.WIG 58 /r VADDPD ymm1, ymm2, ymm3/m256	B	V/V	AVX	Add packed double-precision floating-point values from ymm3/mem to ymm2 and store result in ymm1.
EVEX.NDS.128.66.0F.W1 58 /r VADDPD xmm1 {k1}{z}, xmm2, xmm3/m128/m64bcst	C	V/V	AVX512VL AVX512F	Add packed double-precision floating-point values from xmm3/m128/m64bcst to xmm2 and store result in xmm1 with writemask k1.
EVEX.NDS.256.66.0F.W1 58 /r VADDPD ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst	C	V/V	AVX512VL AVX512F	Add packed double-precision floating-point values from ymm3/m256/m64bcst to ymm2 and store result in ymm1 with writemask k1.
EVEX.NDS.512.66.0F.W1 58 /r VADDPD zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst{er}	C	V/V	AVX512F	Add packed double-precision floating-point values from zmm3/m512/m64bcst to zmm2 and store result in zmm1 with writemask k1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	NA	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
B	NA	ModRM:reg (w)	VEX.vvvv	ModRM:r/m (r)	NA
C	Full Vector	ModRM:reg (w)	EVEX.vvvv	ModRM:r/m (r)	NA

Description

Add two, four or eight packed double-precision floating-point values from the first source operand to the second source operand, and stores the packed double-precision floating-point results in the destination operand.

EVEX encoded versions: The first source operand is a ZMM/YMM/XMM register. The second source operand can be a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 64-bit memory location. The destination operand is a ZMM/YMM/XMM register conditionally updated with writemask k1.

VEX.256 encoded version: The first source operand is a YMM register. The second source operand can be a YMM register or a 256-bit memory location. The destination operand is a YMM register. The upper bits (MAXVL-1:256) of the corresponding ZMM register destination are zeroed.

VEX.128 encoded version: the first source operand is a XMM register. The second source operand is an XMM register or 128-bit memory location. The destination operand is an XMM register. The upper bits (MAXVL-1:128) of the corresponding ZMM register destination are zeroed.

128-bit Legacy SSE version: The second source can be an XMM register or an 128-bit memory location. The destination is not distinct from the first source XMM register and the upper Bits (MAXVL-1:128) of the corresponding ZMM register destination are unmodified.

Operation

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

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
IF (VL = 512) AND (EVEX.b = 1)
    THEN
        SET_RM(EVEX.RC);
    ELSE
        SET_RM(MXCSR.RM);
FI;
FOR j ← 0 TO KL-1
    i ← j * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+63:i] ← SRC1[i+63:i] + SRC2[i+63:i]
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE                            ; zeroing-masking
                    DEST[i+63:i] ← 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0
```

VADDPD (EVEX encoded versions) when src2 operand is a memory source

```
(KL, VL) = (2, 128), (4, 256), (8, 512)

FOR j ← 0 TO KL-1
    i ← j * 64
    IF k1[j] OR *no writemask*
        THEN
            IF (EVEX.b = 1)
                THEN
                    DEST[i+63:i] ← SRC1[i+63:i] + SRC2[63:0]
                ELSE
                    DEST[i+63:i] ← SRC1[i+63:i] + SRC2[i+63:i]
            FI;
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE                            ; zeroing-masking
                    DEST[i+63:i] ← 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0
```

VADDPD (VEX.256 encoded version)

```
DEST[63:0] ← SRC1[63:0] + SRC2[63:0]
DEST[127:64] ← SRC1[127:64] + SRC2[127:64]
DEST[191:128] ← SRC1[191:128] + SRC2[191:128]
DEST[255:192] ← SRC1[255:192] + SRC2[255:192]
DEST[MAXVL-1:256] ← 0
```

.

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

ADDPD (128-bit Legacy SSE version)
DEST[63:0] ← DEST[63:0] + SRC[63:0]
DEST[127:64] ← DEST[127:64] + SRC[127:64]
DEST[MAXVL-1:128] (Unmodified)

Intel C/C++ Compiler Intrinsic Equivalent

VADDPD __m512d _mm512_add_pd (__m512d a, __m512d b);
VADDPD __m512d _mm512_mask_add_pd (__m512d s, __mmask8 k, __m512d a, __m512d b);
VADDPD __m512d _mm512_maskz_add_pd (__mmask8 k, __m512d a, __m512d b);
VADDPD __m256d _mm256_mask_add_pd (__m256d s, __mmask8 k, __m256d a, __m256d b);
VADDPD __m256d _mm256_maskz_add_pd (__mmask8 k, __m256d a, __m256d b);
VADDPD __m128d _mm_mask_add_pd (__m128d s, __mmask8 k, __m128d a, __m128d b);
VADDPD __m128d _mm_maskz_add_pd (__mmask8 k, __m128d a, __m128d b);
VADDPD __m512d _mm512_add_round_pd (__m512d a, __m512d b, int);
VADDPD __m512d _mm512_mask_add_round_pd (__m512d s, __mmask8 k, __m512d a, __m512d b, int);
VADDPD __m512d _mm512_maskz_add_round_pd (__mmask8 k, __m512d a, __m512d b, int);
ADDPD __m256d _mm256_add_pd (__m256d a, __m256d b);
ADDPD __m128d _mm_add_pd (__m128d a, __m128d b);

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal

Other Exceptions

VEX-encoded instruction, see Exceptions Type 2.

EVEX-encoded instruction, see Exceptions Type E2.

ADDPS—Add Packed Single-Precision Floating-Point Values

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F 58 /r ADDPS xmm1, xmm2/m128	A	V/V	SSE	Add packed single-precision floating-point values from xmm2/m128 to xmm1 and store result in xmm1.
VEX.NDS.128.0F.WIG 58 /r VADDPS xmm1,xmm2, xmm3/m128	B	V/V	AVX	Add packed single-precision floating-point values from xmm3/m128 to xmm2 and store result in xmm1.
VEX.NDS.256.0F.WIG 58 /r VADDPS ymm1, ymm2, ymm3/m256	B	V/V	AVX	Add packed single-precision floating-point values from ymm3/m256 to ymm2 and store result in ymm1.
EVEX.NDS.128.0F.W0 58 /r VADDPS xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst	C	V/V	AVX512VL AVX512F	Add packed single-precision floating-point values from xmm3/m128/m32bcst to xmm2 and store result in xmm1 with writemask k1.
EVEX.NDS.256.0F.W0 58 /r VADDPS ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst	C	V/V	AVX512VL AVX512F	Add packed single-precision floating-point values from ymm3/m256/m32bcst to ymm2 and store result in ymm1 with writemask k1.
EVEX.NDS.512.0F.W0 58 /r VADDPS zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst {er}	C	V/V	AVX512F	Add packed single-precision floating-point values from zmm3/m512/m32bcst to zmm2 and store result in zmm1 with writemask k1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	NA	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
B	NA	ModRM:reg (w)	VEX.vvvv	ModRM:r/m (r)	NA
C	Full Vector	ModRM:reg (w)	EVEX.vvvv	ModRM:r/m (r)	NA

Description

Add four, eight or sixteen packed single-precision floating-point values from the first source operand with the second source operand, and stores the packed single-precision floating-point results in the destination operand.

EVEX encoded versions: The first source operand is a ZMM/YMM/XMM register. The second source operand can be a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 32-bit memory location. The destination operand is a ZMM/YMM/XMM register conditionally updated with writemask k1.

VEX.256 encoded version: The first source operand is a YMM register. The second source operand can be a YMM register or a 256-bit memory location. The destination operand is a YMM register. The upper bits (MAXVL-1:256) of the corresponding ZMM register destination are zeroed.

VEX.128 encoded version: the first source operand is a XMM register. The second source operand is an XMM register or 128-bit memory location. The destination operand is an XMM register. The upper bits (MAXVL-1:128) of the corresponding ZMM register destination are zeroed.

128-bit Legacy SSE version: The second source can be an XMM register or an 128-bit memory location. The destination is not distinct from the first source XMM register and the upper Bits (MAXVL-1:128) of the corresponding ZMM register destination are unmodified.

Operation

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

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
IF (VL = 512) AND (EVEX.b = 1)
    THEN
        SET_RM(EVEX.RC);
    ELSE
        SET_RM(MXCSR.RM);
FI;
FOR j ← 0 TO KL-1
    i ← j * 32
    IF k1[j] OR *no writemask*
        THEN DEST[i+31:i] ← SRC1[i+31:i] + SRC2[i+31:i]
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE                            ; zeroing-masking
                    DEST[i+31:i] ← 0
            FI
    FI;
ENDFOR;
DEST[MAXVL-1:VL] ← 0
```

VADDPS (EVEX encoded versions) when src2 operand is a memory source

```
(KL, VL) = (4, 128), (8, 256), (16, 512)

FOR j ← 0 TO KL-1
    i ←j * 32
    IF k1[j] OR *no writemask*
        THEN
            IF (EVEX.b = 1)
                THEN
                    DEST[i+31:i] ← SRC1[i+31:i] + SRC2[31:0]
                ELSE
                    DEST[i+31:i] ← SRC1[i+31:i] + SRC2[i+31:i]
            FI;
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE                            ; zeroing-masking
                    DEST[i+31:i] ← 0
            FI
    FI;
ENDFOR;
DEST[MAXVL-1:VL] ← 0
```

VADDPS (VEX.256 encoded version)
DEST[31:0] ← SRC1[31:0] + SRC2[31:0]
DEST[63:32] ← SRC1[63:32] + SRC2[63:32]
DEST[95:64] ← SRC1[95:64] + SRC2[95:64]
DEST[127:96] ← SRC1[127:96] + SRC2[127:96]
DEST[159:128] ← SRC1[159:128] + SRC2[159:128]
DEST[191:160]← SRC1[191:160] + SRC2[191:160]
DEST[223:192] ← SRC1[223:192] + SRC2[223:192]
DEST[255:224] ← SRC1[255:224] + SRC2[255:224].
DEST[MAXVL-1:256] ← 0

VADDPS (VEX.128 encoded version)
DEST[31:0] ← SRC1[31:0] + SRC2[31:0]
DEST[63:32] ← SRC1[63:32] + SRC2[63:32]
DEST[95:64] ← SRC1[95:64] + SRC2[95:64]
DEST[127:96] ← SRC1[127:96] + SRC2[127:96]
DEST[MAXVL-1:128] ← 0

ADDPS (128-bit Legacy SSE version)
DEST[31:0] ← SRC1[31:0] + SRC2[31:0]
DEST[63:32] ← SRC1[63:32] + SRC2[63:32]
DEST[95:64] ← SRC1[95:64] + SRC2[95:64]
DEST[127:96] ← SRC1[127:96] + SRC2[127:96]
DEST[MAXVL-1:128] (Unmodified)

Intel C/C++ Compiler Intrinsic Equivalent

VADDPS __m512 _mm512_add_ps (__m512 a, __m512 b);
VADDPS __m512 _mm512_mask_add_ps (__m512 s, __mmask16 k, __m512 a, __m512 b);
VADDPS __m512 _mm512_maskz_add_ps (__mmask16 k, __m512 a, __m512 b);
VADDPS __m256 _mm256_mask_add_ps (__m256 s, __mmask8 k, __m256 a, __m256 b);
VADDPS __m256 _mm256_maskz_add_ps (__mmask8 k, __m256 a, __m256 b);
VADDPS __m128 _mm_mask_add_ps (__m128d s, __mmask8 k, __m128 a, __m128 b);
VADDPS __m128 _mm_maskz_add_ps (__mmask8 k, __m128 a, __m128 b);
VADDPS __m512 _mm512_add_round_ps (__m512 a, __m512 b, int);
VADDPS __m512 _mm512_mask_add_round_ps (__m512 s, __mmask16 k, __m512 a, __m512 b, int);
VADDPS __m512 _mm512_maskz_add_round_ps (__mmask16 k, __m512 a, __m512 b, int);
ADDPS __m256 _mm256_add_ps (__m256 a, __m256 b);
ADDPS __m128 _mm_add_ps (__m128 a, __m128 b);

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal

Other Exceptions

VEX-encoded instruction, see Exceptions Type 2.

EVEX-encoded instruction, see Exceptions Type E2.

ADDSD—Add Scalar Double-Precision Floating-Point Values

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
F2 0F 58 /r ADDSD xmm1, xmm2/m64	A	V/V	SSE2	Add the low double-precision floating-point value from xmm2/mem to xmm1 and store the result in xmm1.
VEX.NDS.LIG.F2.0F.WIG 58 /r VADDSD xmm1, xmm2, xmm3/m64	B	V/V	AVX	Add the low double-precision floating-point value from xmm3/mem to xmm2 and store the result in xmm1.
EVEX.NDS.LIG.F2.0F.W1 58 /r VADDSD xmm1 {k1}{z}, xmm2, xmm3/m64{er}	C	V/V	AVX512F	Add the low double-precision floating-point value from xmm3/m64 to xmm2 and store the result in xmm1 with writemask k1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	NA	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
B	NA	ModRM:reg (w)	VEX.vvvv	ModRM:r/m (r)	NA
C	Tuple1 Scalar	ModRM:reg (w)	EVEX.vvvv	ModRM:r/m (r)	NA

Description

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

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

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

EVEX and VEX.128 encoded version: The first source operand is encoded by EVEX.vvvv/VEX.vvvv. Bits (127:64) of the XMM register destination are copied from corresponding bits in the first source operand. Bits (MAXVL-1:128) of the destination register are zeroed.

EVEX version: The low quadword element of the destination is updated according to the writemask.

Software should ensure VADDSD is encoded with VEX.L=0. Encoding VADDSD with VEX.L=1 may encounter unpredictable behavior across different processor generations.

Operation

VADDSD (EVEX encoded version)

```
IF (EVEX.b = 1) AND SRC2 *is a register*
    THEN
        SET_RM(EVEX.RC);
    ELSE
        SET_RM(MXCSR.RM);
FI;
IF k1[0] or *no writemask*
    THEN    DEST[63:0] ← SRC1[63:0] + SRC2[63:0]
    ELSE
        IF *merging-masking*                    ; merging-masking
            THEN *DEST[63:0] remains unchanged*
            ELSE                                ; zeroing-masking
                THEN DEST[63:0] ← 0
        FI;
FI;
DEST[127:64] ← SRC1[127:64]
DEST[MAXVL-1:128] ← 0
```

VADDSD (VEX.128 encoded version)

```
DEST[63:0] ←SRC1[63:0] + SRC2[63:0]
DEST[127:64] ←SRC1[127:64]
DEST[MAXVL-1:128] ←0
```

ADDSD (128-bit Legacy SSE version)

```
DEST[63:0] ←DEST[63:0] + SRC[63:0]
DEST[MAXVL-1:64] (Unmodified)
```

Intel C/C++ Compiler Intrinsic Equivalent

VADDSD __m128d _mm_mask_add_sd (__m128d s, __mmask8 k, __m128d a, __m128d b);
VADDSD __m128d _mm_maskz_add_sd (__mmask8 k, __m128d a, __m128d b);
VADDSD __m128d _mm_add_round_sd (__m128d a, __m128d b, int);
VADDSD __m128d _mm_mask_add_round_sd (__m128d s, __mmask8 k, __m128d a, __m128d b, int);
VADDSD __m128d _mm_maskz_add_round_sd (__mmask8 k, __m128d a, __m128d b, int);
ADDSD __m128d _mm_add_sd (__m128d a, __m128d b);

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal

Other Exceptions

VEX-encoded instruction, see Exceptions Type 3.

EVEX-encoded instruction, see Exceptions Type E3.

ADDSS—Add Scalar Single-Precision Floating-Point Values

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
F3 0F 58 /r ADDSS xmm1, xmm2/m32	A	V/V	SSE	Add the low single-precision floating-point value from xmm2/mem to xmm1 and store the result in xmm1.
VEX.NDS.LIG.F3.0F.WIG 58 /r VADDSS xmm1,xmm2, xmm3/m32	B	V/V	AVX	Add the low single-precision floating-point value from xmm3/mem to xmm2 and store the result in xmm1.
EVEX.NDS.LIG.F3.0F.W0 58 /r VADDSS xmm1{k1}{z}, xmm2, xmm3/m32{er}	C	V/V	AVX512F	Add the low single-precision floating-point value from xmm3/m32 to xmm2 and store the result in xmm1with writemask k1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	NA	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
B	NA	ModRM:reg (w)	VEX.vvvv	ModRM:r/m (r)	NA
C	Tuple1 Scalar	ModRM:reg (w)	EVEX.vvvv	ModRM:r/m (r)	NA

Description

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

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

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

EVEX and VEX.128 encoded version: The first source operand is encoded by EVEX.vvvv/VEX.vvvv. Bits (127:32) of the XMM register destination are copied from corresponding bits in the first source operand. Bits (MAXVL-1:128) of the destination register are zeroed.

EVEX version: The low doubleword element of the destination is updated according to the writemask.

Software should ensure VADDSS is encoded with VEX.L=0. Encoding VADDSS with VEX.L=1 may encounter unpredictable behavior across different processor generations.

Operation

VADDSS (EVEX encoded versions)

```
IF (EVEX.b = 1) AND SRC2 *is a register*
    THEN
        SET_RM(EVEX.RC);
    ELSE
        SET_RM(MXCSR.RM);
FI;
IF k1[0] or *no writemask*
    THEN    DEST[31:0] ← SRC1[31:0] + SRC2[31:0]
    ELSE
        IF *merging-masking*                    ; merging-masking
            THEN *DEST[31:0] remains unchanged*
            ELSE                                ; zeroing-masking
                THEN DEST[31:0] ← 0
        FI;
FI;
DEST[127:32] ← SRC1[127:32]
DEST[MAXVL-1:128] ← 0
```

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

```
DEST[31:0] ←SRC1[31:0] + SRC2[31:0]
DEST[127:32] ←SRC1[127:32]
DEST[MAXVL-1:128] ←0
```

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

```
DEST[31:0] ←DEST[31:0] + SRC[31:0]
DEST[MAXVL-1:32] (Unmodified)
```

Intel C/C++ Compiler Intrinsic Equivalent

VADDSS __m128 _mm_mask_add_ss (__m128 s, __mmask8 k, __m128 a, __m128 b);
VADDSS __m128 _mm_maskz_add_ss (__mmask8 k, __m128 a, __m128 b);
VADDSS __m128 _mm_add_round_ss (__m128 a, __m128 b, int);
VADDSS __m128 _mm_mask_add_round_ss (__m128 s, __mmask8 k, __m128 a, __m128 b, int);
VADDSS __m128 _mm_maskz_add_round_ss (__mmask8 k, __m128 a, __m128 b, int);
ADDSS __m128 _mm_add_ss (__m128 a, __m128 b);

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal

Other Exceptions

VEX-encoded instruction, see Exceptions Type 3.

EVEX-encoded instruction, see Exceptions Type E3.

ADDSUBPD—Packed Double-FP Add/Subtract

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

Instruction Operand Encoding

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

Description

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

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

128-bit Legacy SSE version: The second source can be an XMM register or an 128-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (MAXVL-1:128) of the corresponding YMM register destination are unmodified. See Figure 3-3.

VEX.128 encoded version: the first source operand is an XMM register or 128-bit memory location. The destination operand is an XMM register. The upper bits (MAXVL-1:128) of the corresponding YMM register destination are zeroed.

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

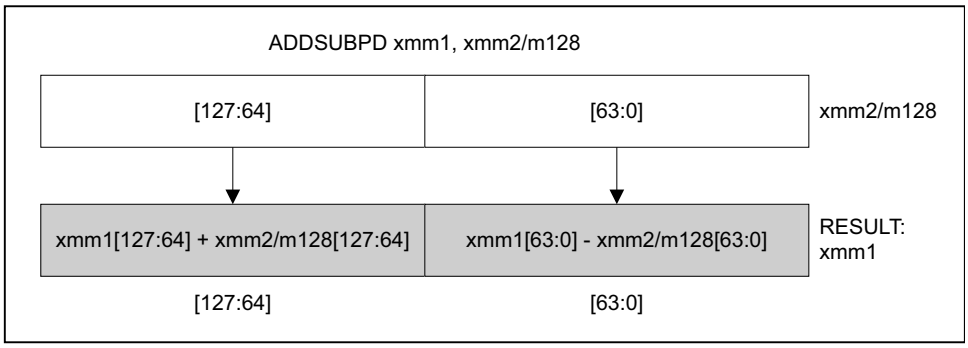


Figure 3-3. ADDSUBPD—Packed Double-FP Add/Subtract

Operation

ADDSUBPD (128-bit Legacy SSE version)

```
DEST[63:0] ← DEST[63:0] - SRC[63:0]
DEST[127:64] ← DEST[127:64] + SRC[127:64]
DEST[MAXVL-1:128] (Unmodified)
```

VADDSUBPD (VEX.128 encoded version)

```
DEST[63:0] ← SRC1[63:0] - SRC2[63:0]
DEST[127:64] ← SRC1[127:64] + SRC2[127:64]
DEST[MAXVL-1:128] ← 0
```

VADDSUBPD (VEX.256 encoded version)

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

Intel C/C++ Compiler Intrinsic Equivalent

ADDSUBPD: __m128d _mm_addsub_pd(__m128d a, __m128d b)

VADDSUBPD: __m256d _mm256_addsub_pd (__m256d a, __m256d b)

Exceptions

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

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal.

Other Exceptions

See Exceptions Type 2.

ADDSUBPS—Packed Single-FP Add/Subtract

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

Instruction Operand Encoding

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

Description

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

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

128-bit Legacy SSE version: The second source can be an XMM register or an 128-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (MAXVL-1:128) of the corresponding YMM register destination are unmodified. See Figure 3-4.

VEX.128 encoded version: the first source operand is an XMM register or 128-bit memory location. The destination operand is an XMM register. The upper bits (MAXVL-1:128) of the corresponding YMM register destination are zeroed.

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

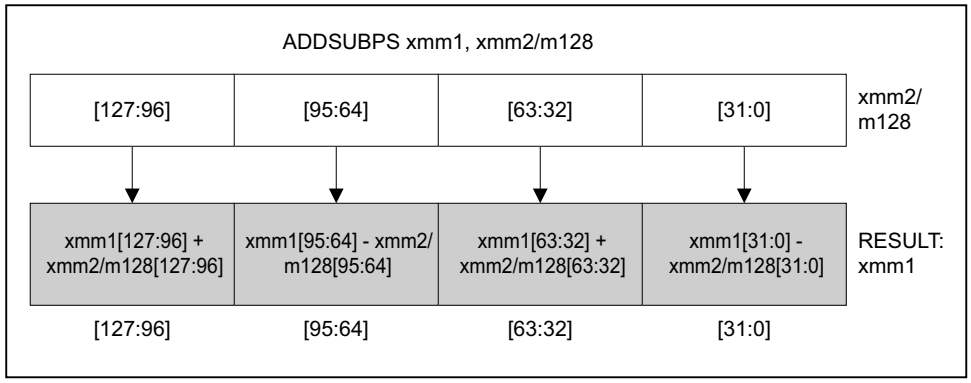


Figure 3-4. ADDSUBPS—Packed Single-FP Add/Subtract

Operation

ADDSUBPS (128-bit Legacy SSE version)

```
DEST[31:0] ← DEST[31:0] - SRC[31:0]
DEST[63:32] ← DEST[63:32] + SRC[63:32]
DEST[95:64] ← DEST[95:64] - SRC[95:64]
DEST[127:96] ← DEST[127:96] + SRC[127:96]
DEST[MAXVL-1:128] (Unmodified)
```

VADDSUBPS (VEX.128 encoded version)

```
DEST[31:0] ← SRC1[31:0] - SRC2[31:0]
DEST[63:32] ← SRC1[63:32] + SRC2[63:32]
DEST[95:64] ← SRC1[95:64] - SRC2[95:64]
DEST[127:96] ← SRC1[127:96] + SRC2[127:96]
DEST[MAXVL-1:128] ← 0
```

VADDSUBPS (VEX.256 encoded version)

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

Intel C/C++ Compiler Intrinsic Equivalent

ADDSUBPS: __m128 _mm_addsub_ps(__m128 a, __m128 b)

VADDSUBPS: __m256 _mm256_addsub_ps (__m256 a, __m256 b)

Exceptions

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

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal.

Other Exceptions

See Exceptions Type 2.

ADOX — Unsigned Integer Addition of Two Operands with Overflow Flag

Opcode/ Instruction	Op/ En	64/32bit Mode Support	CPUID Feature Flag	Description
F3 0F 38 F6 /r ADOX r32, r/m32	RM	V/V	ADX	Unsigned addition of r32 with OF, r/m32 to r32, writes OF.
F3 REX.w 0F 38 F6 /r ADOX r64, r/m64	RM	V/NE	ADX	Unsigned addition of r64 with OF, r/m64 to r64, writes OF.

Instruction Operand Encoding

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

Description

Performs an unsigned addition of the destination operand (first operand), the source operand (second operand) and the overflow-flag (OF) and stores the result in the destination operand. The destination operand is a general-purpose register, whereas the source operand can be a general-purpose register or memory location. The state of OF represents a carry from a previous addition. The instruction sets the OF flag with the carry generated by the unsigned addition of the operands.

The ADOX instruction is executed in the context of multi-precision addition, where we add a series of operands with a carry-chain. At the beginning of a chain of additions, we execute an instruction to zero the OF (e.g. XOR).

This instruction is supported in real mode and virtual-8086 mode. The operand size is always 32 bits if not in 64-bit mode.

In 64-bit mode, the default operation size is 32 bits. Using a REX Prefix in the form of REX.R permits access to additional registers (R8-15). Using REX Prefix in the form of REX.W promotes operation to 64-bits.

ADOX executes normally either inside or outside a transaction region.

Note: ADOX defines the CF and OF flags differently than the ADD/ADC instructions as defined in *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A*.

Operation

```
IF OperandSize is 64-bit
    THEN OF:DEST[63:0] ← DEST[63:0] + SRC[63:0] + OF;
    ELSE OF:DEST[31:0] ← DEST[31:0] + SRC[31:0] + OF;
FI;
```

Flags Affected

OF is updated based on result. CF, SF, ZF, AF and PF flags are unmodified.

Intel C/C++ Compiler Intrinsic Equivalent

unsigned char _addcarryx_u32 (unsigned char c_in, unsigned int src1, unsigned int src2, unsigned int *sum_out);

unsigned char _addcarryx_u64 (unsigned char c_in, unsigned __int64 src1, unsigned __int64 src2, unsigned __int64 *sum_out);

SIMD Floating-Point Exceptions

None

Protected Mode Exceptions

#UD	If the LOCK prefix is used.
	If CPUID.(EAX=07H, ECX=0H):EBX.ADX[bit 19] = 0.
#SS(0)	For an illegal address in the SS segment.
#GP(0)	For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.
	If the DS, ES, FS, or GS register is used to access memory and it contains a null segment selector.
#PF(fault-code)	For a page fault.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

Real-Address Mode Exceptions

#UD	If the LOCK prefix is used.
	If CPUID.(EAX=07H, ECX=0H):EBX.ADX[bit 19] = 0.
#SS(0)	For an illegal address in the SS segment.
#GP(0)	If any part of the operand lies outside the effective address space from 0 to FFFFH.

Virtual-8086 Mode Exceptions

#UD	If the LOCK prefix is used.
	If CPUID.(EAX=07H, ECX=0H):EBX.ADX[bit 19] = 0.
#SS(0)	For an illegal address in the SS segment.
#GP(0)	If any part of the operand lies outside the effective address space from 0 to FFFFH.
#PF(fault-code)	For a page fault.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#UD	If the LOCK prefix is used.
	If CPUID.(EAX=07H, ECX=0H):EBX.ADX[bit 19] = 0.
#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form.
#PF(fault-code)	For a page fault.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

AESDEC—Perform One Round of an AES Decryption Flow

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

Instruction Operand Encoding

Op/En	Operand 1	Operand2	Operand3	Operand4
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
RVM	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA

Description

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

Use the AESDEC instruction for all but the last decryption round. For the last decryption round, use the AESDE-CLAST instruction.

128-bit Legacy SSE version: The first source operand and the destination operand are the same and must be an XMM register. The second source operand can be an XMM register or a 128-bit memory location. Bits (MAXVL-1:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: The first source operand and the destination operand are XMM registers. The second source operand can be an XMM register or a 128-bit memory location. Bits (MAXVL-1:128) of the destination YMM register are zeroed.

Operation

AESDEC

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

VAESDEC

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

Intel C/C++ Compiler Intrinsic Equivalent

(V)AESDEC: __m128i _mm_aesdec (__m128i, __m128i)

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type 4.

AESDECLAST—Perform Last Round of an AES Decryption Flow

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

Instruction Operand Encoding

Op/En	Operand 1	Operand2	Operand3	Operand4
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
RVM	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA

Description

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

128-bit Legacy SSE version: The first source operand and the destination operand are the same and must be an XMM register. The second source operand can be an XMM register or a 128-bit memory location. Bits (MAXVL-1:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: The first source operand and the destination operand are XMM registers. The second source operand can be an XMM register or a 128-bit memory location. Bits (MAXVL-1:128) of the destination YMM register are zeroed.

Operation

AESDECLAST

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

VAESDECLAST

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

Intel C/C++ Compiler Intrinsic Equivalent

(V)AESDECLAST: __m128i _mm_aesdeclast (__m128i, __m128i)

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type 4.

AESENC—Perform One Round of an AES Encryption Flow

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

Instruction Operand Encoding

Op/En	Operand 1	Operand2	Operand3	Operand4
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
RVM	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA

Description

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

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

128-bit Legacy SSE version: The first source operand and the destination operand are the same and must be an XMM register. The second source operand can be an XMM register or a 128-bit memory location. Bits (MAXVL-1:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: The first source operand and the destination operand are XMM registers. The second source operand can be an XMM register or a 128-bit memory location. Bits (MAXVL-1:128) of the destination YMM register are zeroed.

Operation

AESENC

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

VAESENC

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

Intel C/C++ Compiler Intrinsic Equivalent

(V)AESENC: __m128i _mm_aesenc (__m128i, __m128i)

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type 4.

AESENCLAST—Perform Last Round of an AES Encryption Flow

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

Instruction Operand Encoding

Op/En	Operand 1	Operand2	Operand3	Operand4
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
RVM	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA

Description

This instruction performs the last round of an AES encryption flow using a round key from the second source operand, operating on 128-bit data (state) from the first source operand, and store the result in the destination operand.

128-bit Legacy SSE version: The first source operand and the destination operand are the same and must be an XMM register. The second source operand can be an XMM register or a 128-bit memory location. Bits (MAXVL-1:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: The first source operand and the destination operand are XMM registers. The second source operand can be an XMM register or a 128-bit memory location. Bits (MAXVL-1:128) of the destination YMM register are zeroed.

Operation

AESENCLAST

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

VAESENCLAST

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

Intel C/C++ Compiler Intrinsic Equivalent

(V)AESENCLAST: __m128i _mm_aesenclast (__m128i, __m128i)

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type 4.

AESIMC—Perform the AES InvMixColumn Transformation

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

Instruction Operand Encoding

Op/En	Operand 1	Operand2	Operand3	Operand4
RM	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

Description

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

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

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

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

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

Operation

AESIMC

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

VAESIMC

```
DEST[127:0] ← InvMixColumns( SRC );
DEST[MAXVL-1:128] ← 0;
```

Intel C/C++ Compiler Intrinsic Equivalent

(V)AESIMC: __m128i _mm_aesimc (__m128i)

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type 4; additionally

#UD	If VEX.vvvv ≠ 1111B.

AESKEYGENASSIST—AES Round Key Generation Assist

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

Instruction Operand Encoding

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

Description

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

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

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

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

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

Operation

AESKEYGENASSIST

```
X3[31:0] ← SRC [127: 96];
X2[31:0] ← SRC [95: 64];
X1[31:0] ← SRC [63: 32];
X0[31:0] ← SRC [31: 0];
RCON[31:0] ← ZeroExtend(Imm8[7:0]);
DEST[31:0] ← SubWord(X1);
DEST[63:32 ] ← RotWord( SubWord(X1) ) XOR RCON;
DEST[95:64] ← SubWord(X3);
DEST[127:96] ← RotWord( SubWord(X3) ) XOR RCON;
DEST[MAXVL-1:128] (Unmodified)
```

VAESKEYGENASSIST

```
X3[31:0] ← SRC [127: 96];
X2[31:0] ← SRC [95: 64];
X1[31:0] ← SRC [63: 32];
X0[31:0] ← SRC [31: 0];
RCON[31:0] ← ZeroExtend(Imm8[7:0]);
DEST[31:0] ← SubWord(X1);
DEST[63:32 ] ← RotWord( SubWord(X1) ) XOR RCON;
DEST[95:64] ← SubWord(X3);
DEST[127:96] ← RotWord( SubWord(X3) ) XOR RCON;
DEST[MAXVL-1:128] ← 0;
```

Intel C/C++ Compiler Intrinsic Equivalent

(V)AESKEYGENASSIST: __m128i _mm_aeskeygenassist (__m128i, const int)

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type 4; additionally

#UD	If VEX.vvvv ≠ 1111B.

AND—Logical AND

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

NOTES:

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

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
MR	ModRM:r/m (r, w)	ModRM:reg (r)	NA	NA
MI	ModRM:r/m (r, w)	imm8	NA	NA
I	AL/AX/EAX/RAX	imm8	NA	NA

Description

Performs a bitwise AND operation on the destination (first) and source (second) operands and stores the result in the destination operand location. The source operand can be an immediate, a register, or a memory location; the destination operand can be a register or a memory location. (However, two memory operands cannot be used in one instruction.) Each bit of the result is set to 1 if both corresponding bits of the first and second operands are 1; otherwise, it is set to 0.

This instruction can be used with a LOCK prefix to allow the it to be executed atomically.

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

Operation

DEST ← DEST AND SRC;

Flags Affected

The OF and CF flags are cleared; the SF, ZF, and PF flags are set according to the result. The state of the AF flag is undefined.

Protected Mode Exceptions

#GP(0)	If the destination operand points to a non-writable segment.
	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
	If the DS, ES, FS, or GS register contains a NULL segment selector.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If the LOCK prefix is used but the destination is not a memory operand.

Real-Address Mode Exceptions

#GP	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS	If a memory operand effective address is outside the SS segment limit.
#UD	If the LOCK prefix is used but the destination is not a memory operand.

Virtual-8086 Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.
#UD	If the LOCK prefix is used but the destination is not a memory operand.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If the LOCK prefix is used but the destination is not a memory operand.

ANDN — Logical AND NOT

Opcode/Instruction	Op/ En	64/32 -bit Mode	CPUID Feature Flag	Description
VEX.NDS.LZ.0F38.W0 F2 /r ANDN r32a, r32b, r/m32	RVM	V/V	BMI1	Bitwise AND of inverted r32b with r/m32, store result in r32a.
VEX.NDS.LZ. 0F38.W1 F2 /r ANDN r64a, r64b, r/m64	RVM	V/NE	BMI1	Bitwise AND of inverted r64b with r/m64, store result in r64a.

Instruction Operand Encoding

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

Description

Performs a bitwise logical AND of inverted second operand (the first source operand) with the third operand (the second source operand). The result is stored in the first operand (destination operand).

This instruction is not supported in real mode and virtual-8086 mode. The operand size is always 32 bits if not in 64-bit mode. In 64-bit mode operand size 64 requires VEX.W1. VEX.W1 is ignored in non-64-bit modes. An attempt to execute this instruction with VEX.L not equal to 0 will cause #UD.

Operation

```
DEST ← (NOT SRC1) bitwiseAND SRC2;
SF ← DEST[OperandSize -1];
ZF ← (DEST = 0);
```

Flags Affected

SF and ZF are updated based on result. OF and CF flags are cleared. AF and PF flags are undefined.

Intel C/C++ Compiler Intrinsic Equivalent

Auto-generated from high-level language.

SIMD Floating-Point Exceptions

None

Other Exceptions

See Section 2.5.1, “Exception Conditions for VEX-Encoded GPR Instructions”, Table 2-29; additionally

#UD	If VEX.W = 1.

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

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 54 /r ANDPD xmm1, xmm2/m128	A	V/V	SSE2	Return the bitwise logical AND of packed double-precision floating-point values in xmm1 and xmm2/mem.
VEX.NDS.128.66.0F 54 /r VANDPD xmm1, xmm2, xmm3/m128	B	V/V	AVX	Return the bitwise logical AND of packed double-precision floating-point values in xmm2 and xmm3/mem.
VEX.NDS.256.66.0F 54 /r VANDPD ymm1, ymm2, ymm3/m256	B	V/V	AVX	Return the bitwise logical AND of packed double-precision floating-point values in ymm2 and ymm3/mem.
EVEX.NDS.128.66.0F.W1 54 /r VANDPD xmm1 {k1}{z}, xmm2, xmm3/m128/m64bcst	C	V/V	AVX512VL AVX512DQ	Return the bitwise logical AND of packed double-precision floating-point values in xmm2 and xmm3/m128/m64bcst subject to writemask k1.
EVEX.NDS.256.66.0F.W1 54 /r VANDPD ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst	C	V/V	AVX512VL AVX512DQ	Return the bitwise logical AND of packed double-precision floating-point values in ymm2 and ymm3/m256/m64bcst subject to writemask k1.
EVEX.NDS.512.66.0F.W1 54 /r VANDPD zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst	C	V/V	AVX512DQ	Return the bitwise logical AND of packed double-precision floating-point values in zmm2 and zmm3/m512/m64bcst subject to writemask k1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	NA	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
B	NA	ModRM:reg (w)	VEX.vvvv	ModRM:r/m (r)	NA
C	Full Vector	ModRM:reg (w)	EVEX.vvvv	ModRM:r/m (r)	NA

Description

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

EVEX encoded versions: The first source operand is a ZMM/YMM/XMM register. The second source operand can be a ZMM/YMM/XMM register, a 512/256/128-bit memory location, or a 512/256/128-bit vector broadcasted from a 64-bit memory location. The destination operand is a ZMM/YMM/XMM register conditionally updated with writemask k1.

VEX.256 encoded version: The first source operand is a YMM register. The second source operand is a YMM register or a 256-bit memory location. The destination operand is a YMM register. The upper bits (MAXVL-1:256) of the corresponding ZMM register destination are zeroed.

VEX.128 encoded version: The first source operand is an XMM register. The second source operand is an XMM register or 128-bit memory location. The destination operand is an XMM register. The upper bits (MAXVL-1:128) of the corresponding ZMM register destination are zeroed.

128-bit Legacy SSE version: The second source can be an XMM register or an 128-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (MAXVL-1:128) of the corresponding register destination are unmodified.

Operation

VANDPD (EVEX encoded versions)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j ← 0 TO KL-1
    i ← j * 64
    IF k1[j] OR *no writemask*
        THEN
            IF (EVEX.b == 1) AND (SRC2 *is memory*)
                THEN
                    DEST[i+63:i] ← SRC1[i+63:i] BITWISE AND SRC2[63:0]
                ELSE
                    DEST[i+63:i] ← SRC1[i+63:i] BITWISE AND SRC2[i+63:i]
            FI;
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE                                ; zeroing-masking
                    DEST[i+63:i] = 0
            FI;
    FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0
```

VANDPD (VEX.256 encoded version)

```
DEST[63:0] ← SRC1[63:0] BITWISE AND SRC2[63:0]
DEST[127:64] ← SRC1[127:64] BITWISE AND SRC2[127:64]
DEST[191:128] ← SRC1[191:128] BITWISE AND SRC2[191:128]
DEST[255:192] ← SRC1[255:192] BITWISE AND SRC2[255:192]
DEST[MAXVL-1:256] ← 0
```

VANDPD (VEX.128 encoded version)

```
DEST[63:0] ← SRC1[63:0] BITWISE AND SRC2[63:0]
DEST[127:64] ← SRC1[127:64] BITWISE AND SRC2[127:64]
DEST[MAXVL-1:128] ← 0
```

ANDPD (128-bit Legacy SSE version)

```
DEST[63:0] ← DEST[63:0] BITWISE AND SRC[63:0]
DEST[127:64] ← DEST[127:64] BITWISE AND SRC[127:64]
DEST[MAXVL-1:128] (Unmodified)
```

Intel C/C++ Compiler Intrinsic Equivalent

VANDPD __m512d _mm512_and_pd (__m512d a, __m512d b);
VANDPD __m512d _mm512_mask_and_pd (__m512d s, __mmask8 k, __m512d a, __m512d b);
VANDPD __m512d _mm512_maskz_and_pd (__mmask8 k, __m512d a, __m512d b);
VANDPD __m256d _mm256_mask_and_pd (__m256d s, __mmask8 k, __m256d a, __m256d b);
VANDPD __m256d _mm256_maskz_and_pd (__mmask8 k, __m256d a, __m256d b);
VANDPD __m128d _mm_mask_and_pd (__m128d s, __mmask8 k, __m128d a, __m128d b);
VANDPD __m128d _mm_maskz_and_pd (__mmask8 k, __m128d a, __m128d b);
VANDPD __m256d _mm256_and_pd (__m256d a, __m256d b);
ANDPD __m128d _mm_and_pd (__m128d a, __m128d b);

SIMD Floating-Point Exceptions

None

Other Exceptions

VEX-encoded instruction, see Exceptions Type 4.

EVEX-encoded instruction, see Exceptions Type E4.

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

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F 54 /r ANDPS xmm1, xmm2/m128	A	V/V	SSE	Return the bitwise logical AND of packed single-precision floating-point values in xmm1 and xmm2/mem.
VEX.NDS.128.0F 54 /r VANDPS xmm1,xmm2, xmm3/m128	B	V/V	AVX	Return the bitwise logical AND of packed single-precision floating-point values in xmm2 and xmm3/mem.
VEX.NDS.256.0F 54 /r VANDPS ymm1, ymm2, ymm3/m256	B	V/V	AVX	Return the bitwise logical AND of packed single-precision floating-point values in ymm2 and ymm3/mem.
EVEX.NDS.128.0F.W0 54 /r VANDPS xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst	C	V/V	AVX512VL AVX512DQ	Return the bitwise logical AND of packed single-precision floating-point values in xmm2 and xmm3/m128/m32bcst subject to writemask k1.
EVEX.NDS.256.0F.W0 54 /r VANDPS ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst	C	V/V	AVX512VL AVX512DQ	Return the bitwise logical AND of packed single-precision floating-point values in ymm2 and ymm3/m256/m32bcst subject to writemask k1.
EVEX.NDS.512.0F.W0 54 /r VANDPS zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst	C	V/V	AVX512DQ	Return the bitwise logical AND of packed single-precision floating-point values in zmm2 and zmm3/m512/m32bcst subject to writemask k1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	NA	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
B	NA	ModRM:reg (w)	VEX.vvvv	ModRM:r/m (r)	NA
C	Full Vector	ModRM:reg (w)	EVEX.vvvv	ModRM:r/m (r)	NA

Description

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

EVEX encoded versions: The first source operand is a ZMM/YMM/XMM register. The second source operand can be a ZMM/YMM/XMM register, a 512/256/128-bit memory location, or a 512/256/128-bit vector broadcasted from a 32-bit memory location. The destination operand is a ZMM/YMM/XMM register conditionally updated with writemask k1.

VEX.256 encoded version: The first source operand is a YMM register. The second source operand is a YMM register or a 256-bit memory location. The destination operand is a YMM register. The upper bits (MAXVL-1:256) of the corresponding ZMM register destination are zeroed.

VEX.128 encoded version: The first source operand is an XMM register. The second source operand is an XMM register or 128-bit memory location. The destination operand is an XMM register. The upper bits (MAXVL-1:128) of the corresponding ZMM register destination are zeroed.

128-bit Legacy SSE version: The second source can be an XMM register or an 128-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (MAXVL-1:128) of the corresponding ZMM register destination are unmodified.

Operation

VANDPS (EVEX encoded versions)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j ← 0 TO KL-1
    i ← j * 32
    IF k1[j] OR *no writemask*
            IF (EVEX.b == 1) AND (SRC2 *is memory*)
                THEN
                    DEST[i+63:i] ← SRC1[i+31:i] BITWISE AND SRC2[31:0]
                ELSE
                    DEST[i+31:i] ← SRC1[i+31:i] BITWISE AND SRC2[i+31:i]
            FI;
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE                                ; zeroing-masking
                    DEST[i+31:i] ← 0
            FI;
    FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0;
```

VANDPS (VEX.256 encoded version)

```
DEST[31:0] ← SRC1[31:0] BITWISE AND SRC2[31:0]
DEST[63:32] ← SRC1[63:32] BITWISE AND SRC2[63:32]
DEST[95:64] ← SRC1[95:64] BITWISE AND SRC2[95:64]
DEST[127:96] ← SRC1[127:96] BITWISE AND SRC2[127:96]
DEST[159:128] ← SRC1[159:128] BITWISE AND SRC2[159:128]
DEST[191:160] ← SRC1[191:160] BITWISE AND SRC2[191:160]
DEST[223:192] ← SRC1[223:192] BITWISE AND SRC2[223:192]
DEST[255:224] ← SRC1[255:224] BITWISE AND SRC2[255:224].
DEST[MAXVL-1:256] ← 0;
```

VANDPS (VEX.128 encoded version)

```
DEST[31:0] ← SRC1[31:0] BITWISE AND SRC2[31:0]
DEST[63:32] ← SRC1[63:32] BITWISE AND SRC2[63:32]
DEST[95:64] ← SRC1[95:64] BITWISE AND SRC2[95:64]
DEST[127:96] ← SRC1[127:96] BITWISE AND SRC2[127:96]
DEST[MAXVL-1:128] ← 0;
```

ANDPS (128-bit Legacy SSE version)

```
DEST[31:0] ← DEST[31:0] BITWISE AND SRC[31:0]
DEST[63:32] ← DEST[63:32] BITWISE AND SRC[63:32]
DEST[95:64] ← DEST[95:64] BITWISE AND SRC[95:64]
DEST[127:96] ← DEST[127:96] BITWISE AND SRC[127:96]
DEST[MAXVL-1:128] (Unmodified)
```

Intel C/C++ Compiler Intrinsic Equivalent

VANDPS __m512 _mm512_and_ps (__m512 a, __m512 b);
VANDPS __m512 _mm512_mask_and_ps (__m512 s, __mmask16 k, __m512 a, __m512 b);
VANDPS __m512 _mm512_maskz_and_ps (__mmask16 k, __m512 a, __m512 b);
VANDPS __m256 _mm256_mask_and_ps (__m256 s, __mmask8 k, __m256 a, __m256 b);
VANDPS __m256 _mm256_maskz_and_ps (__mmask8 k, __m256 a, __m256 b);
VANDPS __m128 _mm_mask_and_ps (__m128 s, __mmask8 k, __m128 a, __m128 b);
VANDPS __m128 _mm_maskz_and_ps (__mmask8 k, __m128 a, __m128 b);
VANDPS __m256 _mm256_and_ps (__m256 a, __m256 b);
ANDPS __m128 _mm_and_ps (__m128 a, __m128 b);

SIMD Floating-Point Exceptions

None

Other Exceptions

VEX-encoded instruction, see Exceptions Type 4.

EVEX-encoded instruction, see Exceptions Type E4.

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

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 55 /r ANDNPD xmm1, xmm2/m128	A	V/V	SSE2	Return the bitwise logical AND NOT of packed double-precision floating-point values in xmm1 and xmm2/mem.
VEX.NDS.128.66.0F 55 /r VANDNPD xmm1, xmm2, xmm3/m128	B	V/V	AVX	Return the bitwise logical AND NOT of packed double-precision floating-point values in xmm2 and xmm3/mem.
VEX.NDS.256.66.0F 55/r VANDNPD ymm1, ymm2, ymm3/m256	B	V/V	AVX	Return the bitwise logical AND NOT of packed double-precision floating-point values in ymm2 and ymm3/mem.
EVEX.NDS.128.66.0F.W1 55 /r VANDNPD xmm1 {k1}{z}, xmm2, xmm3/m128/m64bcst	C	V/V	AVX512VL AVX512DQ	Return the bitwise logical AND NOT of packed double-precision floating-point values in xmm2 and xmm3/m128/m64bcst subject to writemask k1.
EVEX.NDS.256.66.0F.W1 55 /r VANDNPD ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst	C	V/V	AVX512VL AVX512DQ	Return the bitwise logical AND NOT of packed double-precision floating-point values in ymm2 and ymm3/m256/m64bcst subject to writemask k1.
EVEX.NDS.512.66.0F.W1 55 /r VANDNPD zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst	C	V/V	AVX512DQ	Return the bitwise logical AND NOT of packed double-precision floating-point values in zmm2 and zmm3/m512/m64bcst subject to writemask k1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	NA	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
B	NA	ModRM:reg (w)	VEX.vvvv	ModRM:r/m (r)	NA
C	Full Vector	ModRM:reg (w)	EVEX.vvvv	ModRM:r/m (r)	NA

Description

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

EVEX encoded versions: The first source operand is a ZMM/YMM/XMM register. The second source operand can be a ZMM/YMM/XMM register, a 512/256/128-bit memory location, or a 512/256/128-bit vector broadcasted from a 64-bit memory location. The destination operand is a ZMM/YMM/XMM register conditionally updated with writemask k1.

VEX.256 encoded version: The first source operand is a YMM register. The second source operand is a YMM register or a 256-bit memory location. The destination operand is a YMM register. The upper bits (MAXVL-1:256) of the corresponding ZMM register destination are zeroed.

VEX.128 encoded version: The first source operand is an XMM register. The second source operand is an XMM register or 128-bit memory location. The destination operand is an XMM register. The upper bits (MAXVL-1:128) of the corresponding ZMM register destination are zeroed.

128-bit Legacy SSE version: The second source can be an XMM register or an 128-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (MAXVL-1:128) of the corresponding register destination are unmodified.

Operation

VANDNPD (EVEX encoded versions)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j ← 0 TO KL-1
    i ← j * 64
    IF k1[j] OR *no writemask*
            IF (EVEX.b == 1) AND (SRC2 *is memory*)
                THEN
                    DEST[i+63:i] ← (NOT(SRC1[i+63:i])) BITWISE AND SRC2[63:0]
                ELSE
                    DEST[i+63:i] ← (NOT(SRC1[i+63:i])) BITWISE AND SRC2[i+63:i]
            FI;
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE                                ; zeroing-masking
                    DEST[i+63:i] = 0
            FI;
    FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0
```

VANDNPD (VEX.256 encoded version)

```
DEST[63:0] ← (NOT(SRC1[63:0])) BITWISE AND SRC2[63:0]
DEST[127:64] ← (NOT(SRC1[127:64])) BITWISE AND SRC2[127:64]
DEST[191:128] ← (NOT(SRC1[191:128])) BITWISE AND SRC2[191:128]
DEST[255:192] ← (NOT(SRC1[255:192])) BITWISE AND SRC2[255:192]
DEST[MAXVL-1:256] ← 0
```

VANDNPD (VEX.128 encoded version)

```
DEST[63:0] ← (NOT(SRC1[63:0])) BITWISE AND SRC2[63:0]
DEST[127:64] ← (NOT(SRC1[127:64])) BITWISE AND SRC2[127:64]
DEST[MAXVL-1:128] ← 0
```

ANDNPD (128-bit Legacy SSE version)

```
DEST[63:0] ← (NOT(DEST[63:0])) BITWISE AND SRC[63:0]
DEST[127:64] ← (NOT(DEST[127:64])) BITWISE AND SRC[127:64]
DEST[MAXVL-1:128] (Unmodified)
```

Intel C/C++ Compiler Intrinsic Equivalent

VANDNPD __m512d _mm512_andnot_pd (__m512d a, __m512d b);
VANDNPD __m512d _mm512_mask_andnot_pd (__m512d s, __mmask8 k, __m512d a, __m512d b);
VANDNPD __m512d _mm512_maskz_andnot_pd (__mmask8 k, __m512d a, __m512d b);
VANDNPD __m256d _mm256_mask_andnot_pd (__m256d s, __mmask8 k, __m256d a, __m256d b);
VANDNPD __m256d _mm256_maskz_andnot_pd (__mmask8 k, __m256d a, __m256d b);
VANDNPD __m128d _mm_mask_andnot_pd (__m128d s, __mmask8 k, __m128d a, __m128d b);
VANDNPD __m128d _mm_maskz_andnot_pd (__mmask8 k, __m128d a, __m128d b);
VANDNPD __m256d _mm256_andnot_pd (__m256d a, __m256d b);
ANDNPD __m128d _mm_andnot_pd (__m128d a, __m128d b);

SIMD Floating-Point Exceptions

None

Other Exceptions

VEX-encoded instruction, see Exceptions Type 4.

EVEX-encoded instruction, see Exceptions Type E4.

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

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F 55 /r ANDNPS xmm1, xmm2/m128	A	V/V	SSE	Return the bitwise logical AND NOT of packed single-precision floating-point values in xmm1 and xmm2/mem.
VEX.NDS.128.0F 55 /r VANDNPS xmm1, xmm2, xmm3/m128	B	V/V	AVX	Return the bitwise logical AND NOT of packed single-precision floating-point values in xmm2 and xmm3/mem.
VEX.NDS.256.0F 55 /r VANDNPS ymm1, ymm2, ymm3/m256	B	V/V	AVX	Return the bitwise logical AND NOT of packed single-precision floating-point values in ymm2 and ymm3/mem.
EVEX.NDS.128.0F.W0 55 /r VANDNPS xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst	C	V/V	AVX512VL AVX512DQ	Return the bitwise logical AND of packed single-precision floating-point values in xmm2 and xmm3/m128/m32bcst subject to writemask k1.
EVEX.NDS.256.0F.W0 55 /r VANDNPS ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst	C	V/V	AVX512VL AVX512DQ	Return the bitwise logical AND of packed single-precision floating-point values in ymm2 and ymm3/m256/m32bcst subject to writemask k1.
EVEX.NDS.512.0F.W0 55 /r VANDNPS zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst	C	V/V	AVX512DQ	Return the bitwise logical AND of packed single-precision floating-point values in zmm2 and zmm3/m512/m32bcst subject to writemask k1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	NA	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
B	NA	ModRM:reg (w)	VEX.vvvv	ModRM:r/m (r)	NA
C	Full Vector	ModRM:reg (w)	EVEX.vvvv	ModRM:r/m (r)	NA

Description

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

EVEX encoded versions: The first source operand is a ZMM/YMM/XMM register. The second source operand can be a ZMM/YMM/XMM register, a 512/256/128-bit memory location, or a 512/256/128-bit vector broadcasted from a 32-bit memory location. The destination operand is a ZMM/YMM/XMM register conditionally updated with writemask k1.

VEX.256 encoded version: The first source operand is a YMM register. The second source operand is a YMM register or a 256-bit memory location. The destination operand is a YMM register. The upper bits (MAXVL-1:256) of the corresponding ZMM register destination are zeroed.

VEX.128 encoded version: The first source operand is an XMM register. The second source operand is an XMM register or 128-bit memory location. The destination operand is an XMM register. The upper bits (MAXVL-1:128) of the corresponding ZMM register destination are zeroed.

128-bit Legacy SSE version: The second source can be an XMM register or an 128-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (MAXVL-1:128) of the corresponding ZMM register destination are unmodified.

Operation

VANDNPS (EVEX encoded versions)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j ← 0 TO KL-1
    i ← j * 32
    IF k1[j] OR *no writemask*
            IF (EVEX.b == 1) AND (SRC2 *is memory*)
                THEN
                    DEST[i+31:i] ← (NOT(SRC1[i+31:i])) BITWISE AND SRC2[31:0]
                ELSE
                    DEST[i+31:i] ← (NOT(SRC1[i+31:i])) BITWISE AND SRC2[i+31:i]
            FI;
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE                                ; zeroing-masking
                    DEST[i+31:i] = 0
            FI;
    FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0
```

VANDNPS (VEX.256 encoded version)

```
DEST[31:0] ← (NOT(SRC1[31:0])) BITWISE AND SRC2[31:0]
DEST[63:32] ← (NOT(SRC1[63:32])) BITWISE AND SRC2[63:32]
DEST[95:64] ← (NOT(SRC1[95:64])) BITWISE AND SRC2[95:64]
DEST[127:96] ← (NOT(SRC1[127:96])) BITWISE AND SRC2[127:96]
DEST[159:128] ← (NOT(SRC1[159:128])) BITWISE AND SRC2[159:128]
DEST[191:160] ← (NOT(SRC1[191:160])) BITWISE AND SRC2[191:160]
DEST[223:192] ← (NOT(SRC1[223:192])) BITWISE AND SRC2[223:192]
DEST[255:224] ← (NOT(SRC1[255:224])) BITWISE AND SRC2[255:224].
DEST[MAXVL-1:256] ← 0
```

VANDNPS (VEX.128 encoded version)

```
DEST[31:0] ← (NOT(SRC1[31:0])) BITWISE AND SRC2[31:0]
DEST[63:32] ← (NOT(SRC1[63:32])) BITWISE AND SRC2[63:32]
DEST[95:64] ← (NOT(SRC1[95:64])) BITWISE AND SRC2[95:64]
DEST[127:96] ← (NOT(SRC1[127:96])) BITWISE AND SRC2[127:96]
DEST[MAXVL-1:128] ← 0
```

ANDNPS (128-bit Legacy SSE version)

```
DEST[31:0] ← (NOT(DEST[31:0])) BITWISE AND SRC[31:0]
DEST[63:32] ← (NOT(DEST[63:32])) BITWISE AND SRC[63:32]
DEST[95:64] ← (NOT(DEST[95:64])) BITWISE AND SRC[95:64]
DEST[127:96] ← (NOT(DEST[127:96])) BITWISE AND SRC[127:96]
DEST[MAXVL-1:128] (Unmodified)
```

Intel C/C++ Compiler Intrinsic Equivalent

VANDNPS __m512 _mm512_andnot_ps (__m512 a, __m512 b);
VANDNPS __m512 _mm512_mask_andnot_ps (__m512 s, __mmask16 k, __m512 a, __m512 b);
VANDNPS __m512 _mm512_maskz_andnot_ps (__mmask16 k, __m512 a, __m512 b);
VANDNPS __m256 _mm256_mask_andnot_ps (__m256 s, __mmask8 k, __m256 a, __m256 b);
VANDNPS __m256 _mm256_maskz_andnot_ps (__mmask8 k, __m256 a, __m256 b);
VANDNPS __m128 _mm_mask_andnot_ps (__m128 s, __mmask8 k, __m128 a, __m128 b);
VANDNPS __m128 _mm_maskz_andnot_ps (__mmask8 k, __m128 a, __m128 b);
VANDNPS __m256 _mm256_andnot_ps (__m256 a, __m256 b);
ANDNPS __m128 _mm_andnot_ps (__m128 a, __m128 b);

SIMD Floating-Point Exceptions

None

Other Exceptions

VEX-encoded instruction, see Exceptions Type 4.

EVEX-encoded instruction, see Exceptions Type E4.

ARPL—Adjust RPL Field of Segment Selector

Opcode	Instruction	Op/ En	64-bit Mode	Compat/ Leg Mode	Description
63 /r	ARPL *r/m16, r16*	ZO	N. E.	Valid	Adjust RPL of *r/m16* to not less than RPL of *r16.*

Instruction Operand Encoding

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

Description

Compares the RPL fields of two segment selectors. The first operand (the destination operand) contains one segment selector and the second operand (source operand) contains the other. (The RPL field is located in bits 0 and 1 of each operand.) If the RPL field of the destination operand is less than the RPL field of the source operand, the ZF flag is set and the RPL field of the destination operand is increased to match that of the source operand. Otherwise, the ZF flag is cleared and no change is made to the destination operand. (The destination operand can be a word register or a memory location; the source operand must be a word register.)

The ARPL instruction is provided for use by operating-system procedures (however, it can also be used by applications). It is generally used to adjust the RPL of a segment selector that has been passed to the operating system by an application program to match the privilege level of the application program. Here the segment selector passed to the operating system is placed in the destination operand and segment selector for the application program's code segment is placed in the source operand. (The RPL field in the source operand represents the privilege level of the application program.) Execution of the ARPL instruction then ensures that the RPL of the segment selector received by the operating system is no lower (does not have a higher privilege) than the privilege level of the application program (the segment selector for the application program's code segment can be read from the stack following a procedure call).

This instruction executes as described in compatibility mode and legacy mode. It is not encodable in 64-bit mode.

See "Checking Caller Access Privileges" in Chapter 3, "Protected-Mode Memory Management," of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*, for more information about the use of this instruction.

Operation

```
IF 64-BIT MODE
    THEN
        See MOVSXD;
    ELSE
        IF DEST[RPL] < SRC[RPL]
            THEN
                ZF ← 1;
                DEST[RPL] ← SRC[RPL];
            ELSE
                ZF ← 0;
        FI;
FI;
```

Flags Affected

The ZF flag is set to 1 if the RPL field of the destination operand is less than that of the source operand; otherwise, it is set to 0.

Protected Mode Exceptions

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

Real-Address Mode Exceptions

#UD	The ARPL instruction is not recognized in real-address mode.
	If the LOCK prefix is used.

Virtual-8086 Mode Exceptions

#UD	The ARPL instruction is not recognized in virtual-8086 mode.
	If the LOCK prefix is used.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

Not applicable.

BLENDPD — Blend Packed Double Precision Floating-Point Values

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

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RMI	ModRM:reg (r, w)	ModRM:r/m (r)	imm8	NA
RVMI	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	imm8[3:0]

Description

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

128-bit Legacy SSE version: The second source can be an XMM register or an 128-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (MAXVL-1:128) of the corresponding YMM register destination are unmodified.

VEX.128 encoded version: the first source operand is an XMM register. The second source operand is an XMM register or 128-bit memory location. The destination operand is an XMM register. The upper bits (MAXVL-1:128) of the corresponding YMM register destination are zeroed.

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

Operation

BLENDPD (128-bit Legacy SSE version)

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

VBLENDPD (VEX.128 encoded version)

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

VBLENDPD (VEX.256 encoded version)

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

Intel C/C++ Compiler Intrinsic Equivalent

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

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

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type 4.

BEXTR — Bit Field Extract

Opcode/Instruction	Op/ En	64/32 -bit Mode	CPUID Feature Flag	Description
VEX.NDS.LZ.0F38.W0 F7 /r BEXTR r32a, r/m32, r32b	RMV	V/V	BMI1	Contiguous bitwise extract from r/m32 using r32b as control; store result in r32a.
VEX.NDS.LZ.0F38.W1 F7 /r BEXTR r64a, r/m64, r64b	RMV	V/N.E.	BMI1	Contiguous bitwise extract from r/m64 using r64b as control; store result in r64a

Instruction Operand Encoding

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

Description

Extracts contiguous bits from the first source operand (the second operand) using an index value and length value specified in the second source operand (the third operand). Bit 7:0 of the second source operand specifies the starting bit position of bit extraction. A START value exceeding the operand size will not extract any bits from the second source operand. Bit 15:8 of the second source operand specifies the maximum number of bits (LENGTH) beginning at the START position to extract. Only bit positions up to (OperandSize -1) of the first source operand are extracted. The extracted bits are written to the destination register, starting from the least significant bit. All higher order bits in the destination operand (starting at bit position LENGTH) are zeroed. The destination register is cleared if no bits are extracted.

This instruction is not supported in real mode and virtual-8086 mode. The operand size is always 32 bits if not in 64-bit mode. In 64-bit mode operand size 64 requires VEX.W1. VEX.W1 is ignored in non-64-bit modes. An attempt to execute this instruction with VEX.L not equal to 0 will cause #UD.

Operation

```
START ← SRC2[7:0];
LEN ← SRC2[15:8];
TEMP ← ZERO_EXTEND_TO_512 (SRC1 );
DEST ← ZERO_EXTEND(TEMP[START+LEN -1: START]);
ZF ← (DEST = 0);
```

Flags Affected

ZF is updated based on the result. AF, SF, and PF are undefined. All other flags are cleared.

Intel C/C++ Compiler Intrinsic Equivalent

BEXTR: unsigned __int32 _bextr_u32(unsigned __int32 src, unsigned __int32 start. unsigned __int32 len);

BEXTR: unsigned __int64 _bextr_u64(unsigned __int64 src, unsigned __int32 start. unsigned __int32 len);

SIMD Floating-Point Exceptions

None

Other Exceptions

See Section 2.5.1, “Exception Conditions for VEX-Encoded GPR Instructions”, Table 2-29; additionally

#UD	If VEX.W = 1.

BLENDPS — Blend Packed Single Precision Floating-Point Values

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

Instruction Operand Encoding

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

Description

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

128-bit Legacy SSE version: The second source can be an XMM register or an 128-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (MAXVL-1:128) of the corresponding YMM register destination are unmodified.

VEX.128 encoded version: The first source operand an XMM register. The second source operand is an XMM register or 128-bit memory location. The destination operand is an XMM register. The upper bits (MAXVL-1:128) of the corresponding YMM register destination are zeroed.

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

Operation

BLENDPS (128-bit Legacy SSE version)

```
IF (IMM8[0] = 0) THEN DEST[31:0] ←DEST[31:0]
        ELSE DEST [31:0] ← SRC[31:0] FI
IF (IMM8[1] = 0) THEN DEST[63:32] ← DEST[63:32]
        ELSE DEST [63:32] ← SRC[63:32] FI
IF (IMM8[2] = 0) THEN DEST[95:64] ← DEST[95:64]
        ELSE DEST [95:64] ← SRC[95:64] FI
IF (IMM8[3] = 0) THEN DEST[127:96] ← DEST[127:96]
        ELSE DEST [127:96] ← SRC[127:96] FI
DEST[MAXVL-1:128] (Unmodified)
```

VBLENDPS (VEX.128 encoded version)

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

VBLENDPS (VEX.256 encoded version)

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

Intel C/C++ Compiler Intrinsic Equivalent

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

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

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type 4.

BLENDVPD — Variable Blend Packed Double Precision Floating-Point Values

Opcode/ Instruction	Op/ En	64/32-bit Mode	CPUID Feature Flag	Description
66 0F 38 15 /r BLENDVPD *xmm1, xmm2/m128 , <XMM0>*	RM0	V/V	SSE4_1	Select packed DP FP values from *xmm1* and *xmm2* from mask specified in *XMM0* and store the values in *xmm1*.
VEX.NDS.128.66.0F3A.W0 4B /r /is4 VBLENDVPD xmm1, xmm2, xmm3/m128, xmm4	RVMR	V/V	AVX	Conditionally copy double-precision floating-point values from xmm2 or xmm3/m128 to xmm1, based on mask bits in the mask operand, xmm4.
VEX.NDS.256.66.0F3A.W0 4B /r /is4 VBLENDVPD ymm1, ymm2, ymm3/m256, ymm4	RVMR	V/V	AVX	Conditionally copy double-precision floating-point values from ymm2 or ymm3/m256 to ymm1, based on mask bits in the mask operand, ymm4.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM0	ModRM:reg (r, w)	ModRM:r/m (r)	implicit XMM0	NA
RVMR	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	imm8[7:4]

Description

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

Each quadword element of the destination operand is copied from:

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

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

128-bit Legacy SSE version: The first source operand and the destination operand is the same. Bits (MAXVL-1:128) of the corresponding YMM destination register remain unchanged. The mask register operand is implicitly defined to be the architectural register XMM0. An attempt to execute BLENDVPD with a VEX prefix will cause #UD.

VEX.128 encoded version: The first source operand and the destination operand are XMM registers. The second source operand is an XMM register or 128-bit memory location. The mask operand is the third source register, and encoded in bits[7:4] of the immediate byte(imm8). The bits[3:0] of imm8 are ignored. In 32-bit mode, imm8[7] is ignored. The upper bits (MAXVL-1:128) of the corresponding YMM register (destination register) are zeroed. VEX.W must be 0, otherwise, the instruction will #UD.

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

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

Operation

BLENDVPD (128-bit Legacy SSE version)

```
MASK ← XMM0
IF (MASK[63] = 0) THEN DEST[63:0] ← DEST[63:0]
        ELSE DEST [63:0] ← SRC[63:0] FI
IF (MASK[127] = 0) THEN DEST[127:64] ← DEST[127:64]
        ELSE DEST [127:64] ← SRC[127:64] FI
DEST[MAXVL-1:128] (Unmodified)
```

VBLENDVPD (VEX.128 encoded version)

```
MASK ← SRC3
IF (MASK[63] = 0) THEN DEST[63:0] ← SRC1[63:0]
        ELSE DEST [63:0] ← SRC2[63:0] FI
IF (MASK[127] = 0) THEN DEST[127:64] ← SRC1[127:64]
        ELSE DEST [127:64] ← SRC2[127:64] FI
DEST[MAXVL-1:128] ← 0
```

VBLENDVPD (VEX.256 encoded version)

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

Intel C/C++ Compiler Intrinsic Equivalent

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

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

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

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type 4; additionally

#UD	If VEX.W = 1.

BLENDVPS — Variable Blend Packed Single Precision Floating-Point Values

Opcode/ Instruction	Op/ En	64/32-bit Mode	CPUID Feature Flag	Description
66 0F 38 14 /r BLENDVPS *xmm1, xmm2/m128, <XMM0>*	RM0	V/V	SSE4_1	Select packed single precision floating-point values from *xmm1* and *xmm2/m128* from mask specified in *XMM0* and store the values into *xmm1*.
VEX.NDS.128.66.0F3A.W0 4A /r /is4 VBLENDVPS xmm1, xmm2, xmm3/m128, xmm4	RVMR	V/V	AVX	Conditionally copy single-precision floating-point values from xmm2 or xmm3/m128 to xmm1, based on mask bits in the specified mask operand, xmm4.
VEX.NDS.256.66.0F3A.W0 4A /r /is4 VBLENDVPS ymm1, ymm2, ymm3/m256, ymm4	RVMR	V/V	AVX	Conditionally copy single-precision floating-point values from ymm2 or ymm3/m256 to ymm1, based on mask bits in the specified mask register, ymm4.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM0	ModRM:reg (r, w)	ModRM:r/m (r)	implicit XMM0	NA
RVMR	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	imm8[7:4]

Description

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

Each quadword element of the destination operand is copied from:

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

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

128-bit Legacy SSE version: The first source operand and the destination operand is the same. Bits (MAXVL-1:128) of the corresponding YMM destination register remain unchanged. The mask register operand is implicitly defined to be the architectural register XMM0. An attempt to execute BLENDVPS with a VEX prefix will cause #UD.

VEX.128 encoded version: The first source operand and the destination operand are XMM registers. The second source operand is an XMM register or 128-bit memory location. The mask operand is the third source register, and encoded in bits[7:4] of the immediate byte(imm8). The bits[3:0] of imm8 are ignored. In 32-bit mode, imm8[7] is ignored. The upper bits (MAXVL-1:128) of the corresponding YMM register (destination register) are zeroed. VEX.W must be 0, otherwise, the instruction will #UD.

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

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

Operation

BLENDVPS (128-bit Legacy SSE version)

```
MASK ← XMM0
IF (MASK[31] = 0) THEN DEST[31:0] ← DEST[31:0]
        ELSE DEST [31:0] ← SRC[31:0] FI
IF (MASK[63] = 0) THEN DEST[63:32] ← DEST[63:32]
        ELSE DEST [63:32] ← SRC[63:32] FI
IF (MASK[95] = 0) THEN DEST[95:64] ← DEST[95:64]
        ELSE DEST [95:64] ← SRC[95:64] FI
IF (MASK[127] = 0) THEN DEST[127:96] ← DEST[127:96]
        ELSE DEST [127:96] ← SRC[127:96] FI
DEST[MAXVL-1:128] (Unmodified)
```

VBLENDVPS (VEX.128 encoded version)

```
MASK ← SRC3
IF (MASK[31] = 0) THEN DEST[31:0] ← SRC1[31:0]
        ELSE DEST [31:0] ← SRC2[31:0] FI
IF (MASK[63] = 0) THEN DEST[63:32] ← SRC1[63:32]
        ELSE DEST [63:32] ← SRC2[63:32] FI
IF (MASK[95] = 0) THEN DEST[95:64] ← SRC1[95:64]
        ELSE DEST [95:64] ← SRC2[95:64] FI
IF (MASK[127] = 0) THEN DEST[127:96] ← SRC1[127:96]
        ELSE DEST [127:96] ← SRC2[127:96] FI
DEST[MAXVL-1:128] ← 0
```

VBLENDVPS (VEX.256 encoded version)

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

Intel C/C++ Compiler Intrinsic Equivalent

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

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

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

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type 4; additionally

#UD	If VEX.W = 1.

BLSI — Extract Lowest Set Isolated Bit

Opcode/Instruction	Op/ En	64/32 -bit Mode	CPUID Feature Flag	Description
VEX.NDD.LZ.0F38.W0 F3 /3 BLSI r32, r/m32	VM	V/V	BMI1	Extract lowest set bit from r/m32 and set that bit in r32.
VEX.NDD.LZ.0F38.W1 F3 /3 BLSI r64, r/m64	VM	V/N.E.	BMI1	Extract lowest set bit from r/m64, and set that bit in r64.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
VM	VEX.vvvv (w)	ModRM:r/m (r)	NA	NA

Description

Extracts the lowest set bit from the source operand and set the corresponding bit in the destination register. All other bits in the destination operand are zeroed. If no bits are set in the source operand, BLSI sets all the bits in the destination to 0 and sets ZF and CF.

This instruction is not supported in real mode and virtual-8086 mode. The operand size is always 32 bits if not in 64-bit mode. In 64-bit mode operand size 64 requires VEX.W1. VEX.W1 is ignored in non-64-bit modes. An attempt to execute this instruction with VEX.L not equal to 0 will cause #UD.

Operation

```
temp ← (-SRC) bitwiseAND (SRC);
SF ← temp[OperandSize -1];
ZF ← (temp = 0);
IF SRC = 0
    CF ← 0;
ELSE
    CF ← 1;
FI
DEST ← temp;
```

Flags Affected

ZF and SF are updated based on the result. CF is set if the source is not zero. OF flags are cleared. AF and PF flags are undefined.

Intel C/C++ Compiler Intrinsic Equivalent

BLSI: unsigned __int32 _blsi_u32(unsigned __int32 src);

BLSI: unsigned __int64 _blsi_u64(unsigned __int64 src);

SIMD Floating-Point Exceptions

None

Other Exceptions

See Section 2.5.1, “Exception Conditions for VEX-Encoded GPR Instructions”, Table 2-29; additionally

#UD	If VEX.W = 1.

BLSMSK — Get Mask Up to Lowest Set Bit

Opcode/Instruction	Op/ En	64/32 -bit Mode	CPUID Feature Flag	Description
VEX.NDD.LZ.0F38.W0 F3 /2 BLSMSK r32, r/m32	VM	V/V	BMI1	Set all lower bits in r32 to "1" starting from bit 0 to lowest set bit in r/m32.
VEX.NDD.LZ.0F38.W1 F3 /2 BLSMSK r64, r/m64	VM	V/N.E.	BMI1	Set all lower bits in r64 to "1" starting from bit 0 to lowest set bit in r/m64.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
VM	VEX.vvvv (w)	ModRM:r/m (r)	NA	NA

Description

Sets all the lower bits of the destination operand to "1" up to and including lowest set bit (=1) in the source operand. If source operand is zero, BLSMSK sets all bits of the destination operand to 1 and also sets CF to 1.

This instruction is not supported in real mode and virtual-8086 mode. The operand size is always 32 bits if not in 64-bit mode. In 64-bit mode operand size 64 requires VEX.W1. VEX.W1 is ignored in non-64-bit modes. An attempt to execute this instruction with VEX.L not equal to 0 will cause #UD.

Operation

```
temp ← (SRC-1) XOR (SRC) ;
SF ← temp[OperandSize -1];
ZF ← 0;
IF SRC = 0
    CF ← 1;
ELSE
    CF ← 0;
FI
DEST ← temp;
```

Flags Affected

SF is updated based on the result. CF is set if the source if zero. ZF and OF flags are cleared. AF and PF flag are undefined.

Intel C/C++ Compiler Intrinsic Equivalent

BLSMSK: unsigned __int32 _blsmsk_u32(unsigned __int32 src);

BLSMSK: unsigned __int64 _blsmsk_u64(unsigned __int64 src);

SIMD Floating-Point Exceptions

None

Other Exceptions

See Section 2.5.1, "Exception Conditions for VEX-Encoded GPR Instructions", Table 2-29; additionally

#UD	If VEX.W = 1.

BLSR — Reset Lowest Set Bit

Opcode/Instruction	Op/ En	64/32 -bit Mode	CPUID Feature Flag	Description
VEX.NDD.LZ.0F38.W0 F3 /1 BLSR r32, r/m32	VM	V/V	BMI1	Reset lowest set bit of r/m32, keep all other bits of r/m32 and write result to r32.
VEX.NDD.LZ.0F38.W1 F3 /1 BLSR r64, r/m64	VM	V/N.E.	BMI1	Reset lowest set bit of r/m64, keep all other bits of r/m64 and write result to r64.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
VM	VEX.vvvv (w)	ModRM:r/m (r)	NA	NA

Description

Copies all bits from the source operand to the destination operand and resets (=0) the bit position in the destination operand that corresponds to the lowest set bit of the source operand. If the source operand is zero BLSR sets CF.

This instruction is not supported in real mode and virtual-8086 mode. The operand size is always 32 bits if not in 64-bit mode. In 64-bit mode operand size 64 requires VEX.W1. VEX.W1 is ignored in non-64-bit modes. An attempt to execute this instruction with VEX.L not equal to 0 will cause #UD.

Operation

```
temp ← (SRC-1) bitwiseAND ( SRC );
SF ← temp[OperandSize -1];
ZF ← (temp = 0);
IF SRC = 0
    CF ← 1;
ELSE
    CF ← 0;
FI
DEST ← temp;
```

Flags Affected

ZF and SF flags are updated based on the result. CF is set if the source is zero. OF flag is cleared. AF and PF flags are undefined.

Intel C/C++ Compiler Intrinsic Equivalent

BLSR: unsigned __int32 _blsr_u32(unsigned __int32 src);

BLSR: unsigned __int64 _blsr_u64(unsigned __int64 src);

SIMD Floating-Point Exceptions

None

Other Exceptions

See Section 2.5.1, “Exception Conditions for VEX-Encoded GPR Instructions”, Table 2-29; additionally

#UD	If VEX.W = 1.

BNDCL—Check Lower Bound

Opcode/ Instruction	Op/En	64/32 bit Mode Support	CPUID Feature Flag	Description
F3 0F 1A /r BNDCL bnd, r/m32	RM	NE/V	MPX	Generate a #BR if the address in r/m32 is lower than the lower bound in bnd.LB.
F3 0F 1A /r BNDCL bnd, r/m64	RM	V/NE	MPX	Generate a #BR if the address in r/m64 is lower than the lower bound in bnd.LB.

Instruction Operand Encoding

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

Description

Compare the address in the second operand with the lower bound in bnd. The second operand can be either a register or memory operand. If the address is lower than the lower bound in bnd.LB, it will set BNDSTATUS to 01H and signal a #BR exception.

This instruction does not cause any memory access, and does not read or write any flags.

Operation

BNDCL BND, reg

```
IF reg < BND.LB Then
    BNDSTATUS ← 01H;
    #BR;
FI;
```

BNDCL BND, mem

```
TEMP ← LEA(mem);
IF TEMP < BND.LB Then
    BNDSTATUS ← 01H;
    #BR;
FI;
```

Intel C/C++ Compiler Intrinsic Equivalent

BNDCL void _bnd_chk_ptr_lbounds(const void *q)

Flags Affected

None

Protected Mode Exceptions

#BR	If lower bound check fails.
#UD	If the LOCK prefix is used.
	If ModRM.r/m encodes BND4-BND7 when Intel MPX is enabled.
	If 67H prefix is not used and CS.D=0.
	If 67H prefix is used and CS.D=1.

Real-Address Mode Exceptions

#BR	If lower bound check fails.
#UD	If the LOCK prefix is used.
	If ModRM.r/m encodes BND4-BND7 when Intel MPX is enabled.
	If 16-bit addressing is used.

Virtual-8086 Mode Exceptions

#BR	If lower bound check fails.
#UD	If the LOCK prefix is used.
	If ModRM.r/m encodes BND4-BND7 when Intel MPX is enabled.
	If 16-bit addressing is used.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#UD	If ModRM.r/m and REX encodes BND4-BND15 when Intel MPX is enabled.

Same exceptions as in protected mode.

BNDCU/BNDCN—Check Upper Bound

Opcode/ Instruction	Op/En	64/32 bit Mode Support	CPUID Feature Flag	Description
F2 0F 1A /r BNDCU bnd, r/m32	RM	NE/V	MPX	Generate a #BR if the address in r/m32 is higher than the upper bound in bnd.UB (bnb.UB in 1's complement form).
F2 0F 1A /r BNDCU bnd, r/m64	RM	V/NE	MPX	Generate a #BR if the address in r/m64 is higher than the upper bound in bnd.UB (bnb.UB in 1's complement form).
F2 0F 1B /r BNDCN bnd, r/m32	RM	NE/V	MPX	Generate a #BR if the address in r/m32 is higher than the upper bound in bnd.UB (bnb.UB not in 1's complement form).
F2 0F 1B /r BNDCN bnd, r/m64	RM	V/NE	MPX	Generate a #BR if the address in r/m64 is higher than the upper bound in bnd.UB (bnb.UB not in 1's complement form).

Instruction Operand Encoding

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

Description

Compare the address in the second operand with the upper bound in bnd. The second operand can be either a register or a memory operand. If the address is higher than the upper bound in bnd.UB, it will set BNDSTATUS to 01H and signal a #BR exception.

BNDCU perform 1's complement operation on the upper bound of bnd first before proceeding with address comparison. BNDCN perform address comparison directly using the upper bound in bnd that is already reverted out of 1's complement form.

This instruction does not cause any memory access, and does not read or write any flags.

Effective address computation of m32/64 has identical behavior to LEA

Operation

BNDCU BND, reg

```
IF reg > NOT(BND.UB) Then
    BNDSTATUS ← 01H;
    #BR;
FI;
```

BNDCU BND, mem

```
TEMP ← LEA(mem);
IF TEMP > NOT(BND.UB) Then
    BNDSTATUS ← 01H;
    #BR;
FI;
```

BNDCN BND, reg

```
IF reg > BND.UB Then
    BNDSTATUS ← 01H;
    #BR;
FI;
```

BNDCN BND, mem

```
TEMP ← LEA(mem);
IF TEMP > BND.UB Then
    BNDSTATUS ← 01H;
    #BR;
FI;
```

Intel C/C++ Compiler Intrinsic Equivalent

BNDCU .void _bnd_chk_ptr_ubounds(const void *q)

Flags Affected

None

Protected Mode Exceptions

#BR	If upper bound check fails.
#UD	If the LOCK prefix is used.
	If ModRM.r/m encodes BND4-BND7 when Intel MPX is enabled.
	If 67H prefix is not used and CS.D=0.
	If 67H prefix is used and CS.D=1.

Real-Address Mode Exceptions

#BR	If upper bound check fails.
#UD	If the LOCK prefix is used.
	If ModRM.r/m encodes BND4-BND7 when Intel MPX is enabled.
	If 16-bit addressing is used.

Virtual-8086 Mode Exceptions

#BR	If upper bound check fails.
#UD	If the LOCK prefix is used.
	If ModRM.r/m encodes BND4-BND7 when Intel MPX is enabled.
	If 16-bit addressing is used.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#UD	If ModRM.r/m and REX encodes BND4-BND15 when Intel MPX is enabled.

Same exceptions as in protected mode.

BNDLDX—Load Extended Bounds Using Address Translation

Opcode/ Instruction	Op/En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F 1A /r BNDLDX bnd, mib	RM	V/V	MPX	Load the bounds stored in a bound table entry (BTE) into bnd with address translation using the base of mib and conditional on the index of mib matching the pointer value in the BTE.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3
RM	ModRM:reg (w)	SIB.base (r): Address of pointer SIB.index(r)	NA

Description

BNDLDX uses the linear address constructed from the base register and displacement of the SIB-addressing form of the memory operand (mib) to perform address translation to access a bound table entry and conditionally load the bounds in the BTE to the destination. The destination register is updated with the bounds in the BTE, if the content of the index register of mib matches the pointer value stored in the BTE.

If the pointer value comparison fails, the destination is updated with INIT bounds (lb = 0x0, ub = 0x0) (note: as articulated earlier, the upper bound is represented using 1's complement, therefore, the 0x0 value of upper bound allows for access to full memory).

This instruction does not cause memory access to the linear address of mib nor the effective address referenced by the base, and does not read or write any flags.

Segment overrides apply to the linear address computation with the base of mib, and are used during address translation to generate the address of the bound table entry. By default, the address of the BTE is assumed to be linear address. There are no segmentation checks performed on the base of mib.

The base of mib will not be checked for canonical address violation as it does not access memory.

Any encoding of this instruction that does not specify base or index register will treat those registers as zero (constant). The reg-reg form of this instruction will remain a NOP.

The scale field of the SIB byte has no effect on these instructions and is ignored.

The bound register may be partially updated on memory faults. The order in which memory operands are loaded is implementation specific.

Operation

```
base ← mib.SIB.base ? mib.SIB.base + Disp: 0;
ptr_value ← mib.SIB.index ? mib.SIB.index : 0;

Outside 64-bit mode
A_BDE[31:0] ← (Zero_extend32(base[31:12] « 2) + (BNDCFG[31:12] «12 );
A_BT[31:0] ← LoadFrom(A_BDE );
IF A_BT[0] equal 0 Then
    BNDSTATUS ← A_BDE | 02H;
    #BR;
FI;
A_BTE[31:0] ← (Zero_extend32(base[11:2] « 4) + (A_BT[31:2] « 2 );
Temp_lb[31:0] ← LoadFrom(A_BTE);
Temp_ub[31:0] ← LoadFrom(A_BTE + 4);
Temp_ptr[31:0] ← LoadFrom(A_BTE + 8);
IF Temp_ptr equal ptr_value Then
    BND.LB ← Temp_lb;
    BND.UB ← Temp_ub;
```

```
ELSE
    BND.LB ← 0;
    BND.UB ← 0;
FI;
```

In 64-bit mode

```
A_BDE[63:0] ← (Zero_extend64(base[47+MAWA:20] « 3) + (BNDCFG[63:20] «12 );[1]
A_BT[63:0] ← LoadFrom(A_BDE);
IF A_BT[0] equal 0 Then
    BNDSTATUS ← A_BDE | 02H;
    #BR;
FI;
A_BTE[63:0] ← (Zero_extend64(base[19:3] « 5) + (A_BT[63:3] « 3 );
Temp_lb[63:0] ← LoadFrom(A_BTE);
Temp_ub[63:0] ← LoadFrom(A_BTE + 8);
Temp_ptr[63:0] ← LoadFrom(A_BTE + 16);
IF Temp_ptr equal ptr_value Then
    BND.LB ← Temp_lb;
    BND.UB ← Temp_ub;
ELSE
    BND.LB ← 0;
    BND.UB ← 0;
FI;
```

Intel C/C++ Compiler Intrinsic Equivalent

BNDLDX: Generated by compiler as needed.

Flags Affected

None

Protected Mode Exceptions

#BR	If the bound directory entry is invalid.
#UD	If the LOCK prefix is used.
	If ModRM.r/m encodes BND4-BND7 when Intel MPX is enabled.
	If 67H prefix is not used and CS.D=0.
	If 67H prefix is used and CS.D=1.
#GP(0)	If a destination effective address of the Bound Table entry is outside the DS segment limit.
	If DS register contains a NULL segment selector.
#PF(fault code)	If a page fault occurs.

Real-Address Mode Exceptions

#UD	If the LOCK prefix is used.
	If ModRM.r/m encodes BND4-BND7 when Intel MPX is enabled.
	If 16-bit addressing is used.
#GP(0)	If a destination effective address of the Bound Table entry is outside the DS segment limit.

1. If CPL < 3, the supervisor MAWA (MAWAS) is used; this value is 0. If CPL = 3, the user MAWA (MAWAU) is used; this value is enumerated in CPUID.(EAX=07H,ECX=0H):ECX.MAWAU[bits 21:17]. See Section 17.3.1 of *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*.

Virtual-8086 Mode Exceptions

#UD	If the LOCK prefix is used.
	If ModRM.r/m encodes BND4-BND7 when Intel MPX is enabled.
	If 16-bit addressing is used.
#GP(0)	If a destination effective address of the Bound Table entry is outside the DS segment limit.
#PF(fault code)	If a page fault occurs.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#BR	If the bound directory entry is invalid.
#UD	If ModRM is RIP relative.
	If the LOCK prefix is used.
	If ModRM.r/m and REX encodes BND4-BND15 when Intel MPX is enabled.
#GP(0)	If the memory address (A_BDE or A_BTE) is in a non-canonical form.
#PF(fault code)	If a page fault occurs.

BNDMK—Make Bounds

Opcode/ Instruction	Op/En	64/32 bit Mode Support	CPUID Feature Flag	Description
F3 0F 1B /r BNDMK bnd, m32	RM	NE/V	MPX	Make lower and upper bounds from m32 and store them in bnd.
F3 0F 1B /r BNDMK bnd, m64	RM	V/NE	MPX	Make lower and upper bounds from m64 and store them in bnd.

Instruction Operand Encoding

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

Description

Makes bounds from the second operand and stores the lower and upper bounds in the bound register bnd. The second operand must be a memory operand. The content of the base register from the memory operand is stored in the lower bound bnd.LB. The 1's complement of the effective address of m32/m64 is stored in the upper bound b.UB. Computation of m32/m64 has identical behavior to LEA.

This instruction does not cause any memory access, and does not read or write any flags.

If the instruction did not specify base register, the lower bound will be zero. The reg-reg form of this instruction retains legacy behavior (NOP).

The instruction causes an invalid-opcode exception (#UD) if executed in 64-bit mode with RIP-relative addressing.

Operation

```
BND.LB ← SRCMEM.base;
IF 64-bit mode Then
    BND.UB ← NOT(LEA.64_bits(SRCMEM));
ELSE
    BND.UB ← Zero_Extend.64_bits(NOT(LEA.32_bits(SRCMEM)));
FI;
```

Intel C/C++ Compiler Intrinsic Equivalent

BNDMKvoid * _bnd_set_ptr_bounds(const void * q, size_t size);

Flags Affected

None

Protected Mode Exceptions

#UD	If the LOCK prefix is used.
	If ModRM.r/m encodes BND4-BND7 when Intel MPX is enabled.
	If 67H prefix is not used and CS.D=0.
	If 67H prefix is used and CS.D=1.

Real-Address Mode Exceptions

#UD	If the LOCK prefix is used.
	If ModRM.r/m encodes BND4-BND7 when Intel MPX is enabled.
	If 16-bit addressing is used.

Virtual-8086 Mode Exceptions

#UD	If the LOCK prefix is used. If ModRM.r/m encodes BND4-BND7 when Intel MPX is enabled. If 16-bit addressing is used.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#UD	If the LOCK prefix is used. If ModRM.r/m and REX encodes BND4-BND15 when Intel MPX is enabled. If RIP-relative addressing is used.
#SS(0)	If the memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form.

Same exceptions as in protected mode.

BNDMOV—Move Bounds

Opcode/ Instruction	Op/En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 1A /r BNDMOV bnd1, bnd2/m64	RM	NE/V	MPX	Move lower and upper bound from bnd2/m64 to bound register bnd1.
66 0F 1A /r BNDMOV bnd1, bnd2/m128	RM	V/NE	MPX	Move lower and upper bound from bnd2/m128 to bound register bnd1.
66 0F 1B /r BNDMOV bnd1/m64, bnd2	MR	NE/V	MPX	Move lower and upper bound from bnd2 to bnd1/m64.
66 0F 1B /r BNDMOV bnd1/m128, bnd2	MR	V/NE	MPX	Move lower and upper bound from bnd2 to bound register bnd1/m128.

Instruction Operand Encoding

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

Description

BNDMOV moves a pair of lower and upper bound values from the source operand (the second operand) to the destination (the first operand). Each operation is 128-bit move. The exceptions are same as the MOV instruction. The memory format for loading/store bounds in 64-bit mode is shown in Figure 3-5.

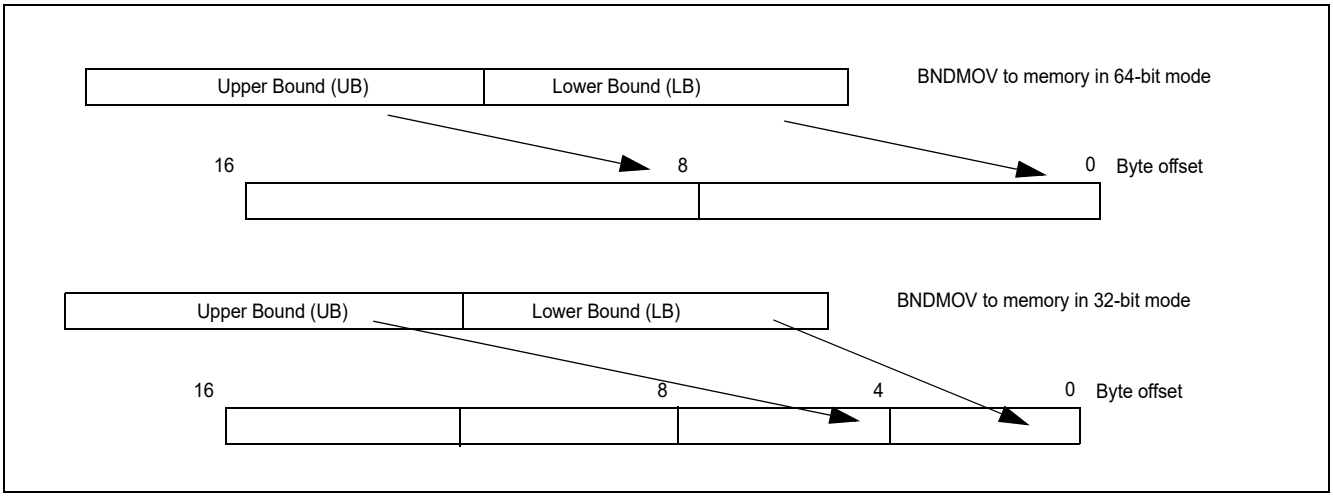


Figure 3-5. Memory Layout of BNDMOV to/from Memory

This instruction does not change flags.

Operation

BNDMOV register to register

DEST.LB ← SRC.LB;
DEST.UB ← SRC.UB;

BNDMOV from memory

```
IF 64-bit mode THEN
        DEST.LB ← LOAD_QWORD(SRC);
        DEST.UB ← LOAD_QWORD(SRC+8);
    ELSE
        DEST.LB ← LOAD_DWORD_ZERO_EXT(SRC);
        DEST.UB ← LOAD_DWORD_ZERO_EXT(SRC+4);
FI;
```

BNDMOV to memory

```
IF 64-bit mode THEN
        DEST[63:0] ← SRC.LB;
        DEST[127:64] ← SRC.UB;
    ELSE
        DEST[31:0] ← SRC.LB;
        DEST[63:32] ← SRC.UB;
FI;
```

Intel C/C++ Compiler Intrinsic Equivalent

BNDMOV void * _bnd_copy_ptr_bounds(const void *q, const void *r)

Flags Affected

None

Protected Mode Exceptions

#UD	If the LOCK prefix is used but the destination is not a memory operand.
	If ModRM.r/m encodes BND4-BND7 when Intel MPX is enabled.
	If 67H prefix is not used and CS.D=0.
	If 67H prefix is used and CS.D=1.
#SS(0)	If the memory operand effective address is outside the SS segment limit.
#GP(0)	If the memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
	If the destination operand points to a non-writable segment
	If the DS, ES, FS, or GS segment register contains a NULL segment selector.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while CPL is 3.
#PF(fault code)	If a page fault occurs.

Real-Address Mode Exceptions

#UD	If the LOCK prefix is used but the destination is not a memory operand.
	If ModRM.r/m encodes BND4-BND7 when Intel MPX is enabled.
	If 16-bit addressing is used.
#GP(0)	If the memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS	If the memory operand effective address is outside the SS segment limit.

Virtual-8086 Mode Exceptions

#UD	If the LOCK prefix is used but the destination is not a memory operand. If ModRM.r/m encodes BND4-BND7 when Intel MPX is enabled. If 16-bit addressing is used.
#GP(0)	If the memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS(0)	If the memory operand effective address is outside the SS segment limit.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while CPL is 3.
#PF(fault code)	If a page fault occurs.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#UD	If the LOCK prefix is used but the destination is not a memory operand. If ModRM.r/m and REX encodes BND4-BND15 when Intel MPX is enabled.
#SS(0)	If the memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while CPL is 3.
#PF(fault code)	If a page fault occurs.

BNDSTX—Store Extended Bounds Using Address Translation

Opcode/ Instruction	Op/En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F 1B /r BNDSTX mib, bnd	MR	V/V	MPX	Store the bounds in bnd and the pointer value in the index register of mib to a bound table entry (BTE) with address translation using the base of mib.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3
MR	SIB.base (r): Address of pointer SIB.index(r)	ModRM:reg (r)	NA

Description

BNDSTX uses the linear address constructed from the displacement and base register of the SIB-addressing form of the memory operand (mib) to perform address translation to store to a bound table entry. The bounds in the source operand bnd are written to the lower and upper bounds in the BTE. The content of the index register of mib is written to the pointer value field in the BTE.

This instruction does not cause memory access to the linear address of mib nor the effective address referenced by the base, and does not read or write any flags.

Segment overrides apply to the linear address computation with the base of mib, and are used during address translation to generate the address of the bound table entry. By default, the address of the BTE is assumed to be linear address. There are no segmentation checks performed on the base of mib.

The base of mib will not be checked for canonical address violation as it does not access memory.

Any encoding of this instruction that does not specify base or index register will treat those registers as zero (constant). The reg-reg form of this instruction will remain a NOP.

The scale field of the SIB byte has no effect on these instructions and is ignored.

The bound register may be partially updated on memory faults. The order in which memory operands are loaded is implementation specific.

Operation

```
base ← mib.SIB.base ? mib.SIB.base + Disp: 0;
ptr_value ← mib.SIB.index ? mib.SIB.index : 0;

Outside 64-bit mode
A_BDE[31:0] ← (Zero_extend32(base[31:12] « 2) + (BNDCFG[31:12] «12 );
A_BT[31:0] ← LoadFrom(A_BDE);
IF A_BT[0] equal 0 Then
    BNDSTATUS ← A_BDE | 02H;
    #BR;
FI;
A_DEST[31:0] ← (Zero_extend32(base[11:2] « 4) + (A_BT[31:2] « 2 ); // address of Bound table entry
A_DEST[8][31:0] ← ptr_value;
A_DEST[0][31:0] ← BND.LB;
A_DEST[4][31:0] ← BND.UB;
```

In 64-bit mode

```
A_BDE[63:0] ← (Zero_extend64(base[47+MAWA:20] « 3) + (BNDCFG[63:20] «12 );[1]
A_BT[63:0] ← LoadFrom(A_BDE);
IF A_BT[0] equal 0 Then
    BNDSTATUS ← A_BDE | 02H;
    #BR;
FI;
A_DEST[63:0] ← (Zero_extend64(base[19:3] « 5) + (A_BT[63:3] « 3 ); // address of Bound table entry
A_DEST[16][63:0] ← ptr_value;
A_DEST[0][63:0] ← BND.LB;
A_DEST[8][63:0] ← BND.UB;
```

Intel C/C++ Compiler Intrinsic Equivalent

BNDSTX: _bnd_store_ptr_bounds(const void **ptr_addr, const void *ptr_val);

Flags Affected

None

Protected Mode Exceptions

#BR	If the bound directory entry is invalid.
#UD	If the LOCK prefix is used.
	If ModRM.r/m encodes BND4-BND7 when Intel MPX is enabled.
	If 67H prefix is not used and CS.D=0.
	If 67H prefix is used and CS.D=1.
#GP(0)	If a destination effective address of the Bound Table entry is outside the DS segment limit.
	If DS register contains a NULL segment selector.
	If the destination operand points to a non-writable segment
#PF(fault code)	If a page fault occurs.

Real-Address Mode Exceptions

#UD	If the LOCK prefix is used.
	If ModRM.r/m encodes BND4-BND7 when Intel MPX is enabled.
	If 16-bit addressing is used.
#GP(0)	If a destination effective address of the Bound Table entry is outside the DS segment limit.

Virtual-8086 Mode Exceptions

#UD	If the LOCK prefix is used.
	If ModRM.r/m encodes BND4-BND7 when Intel MPX is enabled.
	If 16-bit addressing is used.
#GP(0)	If a destination effective address of the Bound Table entry is outside the DS segment limit.
#PF(fault code)	If a page fault occurs.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

1. If CPL < 3, the supervisor MAWA (MAWAS) is used; this value is 0. If CPL = 3, the user MAWA (MAWAU) is used; this value is enumerated in CPUID.(EAX=07H,ECX=0H):ECX.MAWAU[bits 21:17]. See Section 17.3.1 of *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*.

64-Bit Mode Exceptions

#BR	If the bound directory entry is invalid.
#UD	If ModRM is RIP relative.
	If the LOCK prefix is used.
	If ModRM.r/m and REX encodes BND4-BND15 when Intel MPX is enabled.
#GP(0)	If the memory address (A_BDE or A_BTE) is in a non-canonical form.
	If the destination operand points to a non-writable segment
#PF(fault code)	If a page fault occurs.

BOUND—Check Array Index Against Bounds

Opcode	Instruction	Op/ En	64-bit Mode	Compat/ Leg Mode	Description
62 /r	BOUND *r16, m16&16*	RM	Invalid	Valid	Check if *r16* (array index) is within bounds specified by *m16&16.*
62 /r	BOUND *r32, m32&32*	RM	Invalid	Valid	Check if *r32* (array index) is within bounds specified by *m32&32.*

Instruction Operand Encoding

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

Description

BOUND determines if the first operand (array index) is within the bounds of an array specified the second operand (bounds operand). The array index is a signed integer located in a register. The bounds operand is a memory location that contains a pair of signed doubleword-integers (when the operand-size attribute is 32) or a pair of signed word-integers (when the operand-size attribute is 16). The first doubleword (or word) is the lower bound of the array and the second doubleword (or word) is the upper bound of the array. The array index must be greater than or equal to the lower bound and less than or equal to the upper bound plus the operand size in bytes. If the index is not within bounds, a BOUND range exceeded exception (#BR) is signaled. When this exception is generated, the saved return instruction pointer points to the BOUND instruction.

The bounds limit data structure (two words or doublewords containing the lower and upper limits of the array) is usually placed just before the array itself, making the limits addressable via a constant offset from the beginning of the array. Because the address of the array already will be present in a register, this practice avoids extra bus cycles to obtain the effective address of the array bounds.

This instruction executes as described in compatibility mode and legacy mode. It is not valid in 64-bit mode.

Operation

```
IF 64bit Mode
    THEN
        #UD;
    ELSE
        IF (ArrayIndex < LowerBound OR ArrayIndex > UpperBound) THEN
        (* Below lower bound or above upper bound *)
            IF <equation for PL enabled> THEN BNDSTATUS ← 0
            #BR;
        FI;
FI;
```

Flags Affected

None.

Protected Mode Exceptions

#BR	If the bounds test fails.
#UD	If second operand is not a memory location.
	If the LOCK prefix is used.
#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
	If the DS, ES, FS, or GS register contains a NULL segment selector.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

Real-Address Mode Exceptions

#BR	If the bounds test fails.
#UD	If second operand is not a memory location.
	If the LOCK prefix is used.
#GP	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS	If a memory operand effective address is outside the SS segment limit.

Virtual-8086 Mode Exceptions

#BR	If the bounds test fails.
#UD	If second operand is not a memory location.
	If the LOCK prefix is used.
#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#UD	If in 64-bit mode.

BSF—Bit Scan Forward

Opcode	Instruction	Op/ En	64-bit Mode	Compat/ Leg Mode	Description
0F BC /r	BSF *r16, r/m16*	RM	Valid	Valid	Bit scan forward on *r/m16.*
0F BC /r	BSF *r32, r/m32*	RM	Valid	Valid	Bit scan forward on *r/m32.*
REX.W + 0F BC /r	BSF *r64, r/m64*	RM	Valid	N.E.	Bit scan forward on *r/m64.*

Instruction Operand Encoding

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

Description

Searches the source operand (second operand) for the least significant set bit (1 bit). If a least significant 1 bit is found, its bit index is stored in the destination operand (first operand). The source operand can be a register or a memory location; the destination operand is a register. The bit index is an unsigned offset from bit 0 of the source operand. If the content of the source operand is 0, the content of the destination operand is undefined.

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

Operation

```
IF SRC = 0
    THEN
        ZF ← 1;
        DEST is undefined;
    ELSE
        ZF ← 0;
        temp ← 0;
        WHILE Bit(SRC, temp) = 0
        DO
            temp ← temp + 1;
        OD;
        DEST ← temp;
FI;
```

Flags Affected

The ZF flag is set to 1 if all the source operand is 0; otherwise, the ZF flag is cleared. The CF, OF, SF, AF, and PF, flags are undefined.

Protected Mode Exceptions

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

Real-Address Mode Exceptions

#GP	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS	If a memory operand effective address is outside the SS segment limit.
#UD	If the LOCK prefix is used.

Virtual-8086 Mode Exceptions

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

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If the LOCK prefix is used.

BSR—Bit Scan Reverse

Opcode	Instruction	Op/ En	64-bit Mode	Compat/ Leg Mode	Description
0F BD /r	BSR *r16, r/m16*	RM	Valid	Valid	Bit scan reverse on *r/m16.*
0F BD /r	BSR *r32, r/m32*	RM	Valid	Valid	Bit scan reverse on *r/m32.*
REX.W + 0F BD /r	BSR *r64, r/m64*	RM	Valid	N.E.	Bit scan reverse on *r/m64.*

Instruction Operand Encoding

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

Description

Searches the source operand (second operand) for the most significant set bit (1 bit). If a most significant 1 bit is found, its bit index is stored in the destination operand (first operand). The source operand can be a register or a memory location; the destination operand is a register. The bit index is an unsigned offset from bit 0 of the source operand. If the content source operand is 0, the content of the destination operand is undefined.

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

Operation

```
IF SRC = 0
    THEN
        ZF ← 1;
        DEST is undefined;
    ELSE
        ZF ← 0;
        temp ← OperandSize - 1;
        WHILE Bit(SRC, temp) = 0
        DO
            temp ← temp - 1;
        OD;
        DEST ← temp;
FI;
```

Flags Affected

The ZF flag is set to 1 if all the source operand is 0; otherwise, the ZF flag is cleared. The CF, OF, SF, AF, and PF, flags are undefined.

Protected Mode Exceptions

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

Real-Address Mode Exceptions

#GP	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS	If a memory operand effective address is outside the SS segment limit.
#UD	If the LOCK prefix is used.

Virtual-8086 Mode Exceptions

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

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If the LOCK prefix is used.

BSWAP—Byte Swap

Opcode	Instruction	Op/ En	64-bit Mode	Compat/ Leg Mode	Description
0F C8+*rd*	BSWAP *r32*	O	Valid*	Valid	Reverses the byte order of a 32-bit register.
REX.W + 0F C8+*rd*	BSWAP *r64*	O	Valid	N.E.	Reverses the byte order of a 64-bit register.

NOTES:

* See IA-32 Architecture Compatibility section below.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
O	opcode + rd (r, w)	NA	NA	NA

Description

Reverses the byte order of a 32-bit or 64-bit (destination) register. This instruction is provided for converting little-endian values to big-endian format and vice versa. To swap bytes in a word value (16-bit register), use the XCHG instruction. When the BSWAP instruction references a 16-bit register, the result is undefined.

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

IA-32 Architecture Legacy Compatibility

The BSWAP instruction is not supported on IA-32 processors earlier than the Intel486™ processor family. For compatibility with this instruction, software should include functionally equivalent code for execution on Intel processors earlier than the Intel486 processor family.

Operation

```
TEMP ← DEST
IF 64-bit mode AND OperandSize = 64
    THEN
        DEST[7:0] ← TEMP[63:56];
        DEST[15:8] ← TEMP[55:48];
        DEST[23:16] ← TEMP[47:40];
        DEST[31:24] ← TEMP[39:32];
        DEST[39:32] ← TEMP[31:24];
        DEST[47:40] ← TEMP[23:16];
        DEST[55:48] ← TEMP[15:8];
        DEST[63:56] ← TEMP[7:0];
    ELSE
        DEST[7:0] ← TEMP[31:24];
        DEST[15:8] ← TEMP[23:16];
        DEST[23:16] ← TEMP[15:8];
        DEST[31:24] ← TEMP[7:0];
FI;
```

Flags Affected

None.

Exceptions (All Operating Modes)

#UD	If the LOCK prefix is used.

BT—Bit Test

Opcode	Instruction	Op/ En	64-bit Mode	Compat/ Leg Mode	Description
0F A3 /r	BT *r/m16, r16*	MR	Valid	Valid	Store selected bit in CF flag.
0F A3 /r	BT *r/m32, r32*	MR	Valid	Valid	Store selected bit in CF flag.
REX.W + 0F A3 /r	BT *r/m64, r64*	MR	Valid	N.E.	Store selected bit in CF flag.
0F BA /4 *ib*	BT *r/m16, imm8*	MI	Valid	Valid	Store selected bit in CF flag.
0F BA /4 *ib*	BT *r/m32, imm8*	MI	Valid	Valid	Store selected bit in CF flag.
REX.W + 0F BA /4 *ib*	BT *r/m64, imm8*	MI	Valid	N.E.	Store selected bit in CF flag.

Instruction Operand Encoding

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

Description

Selects the bit in a bit string (specified with the first operand, called the bit base) at the bit-position designated by the bit offset (specified by the second operand) and stores the value of the bit in the CF flag. The bit base operand can be a register or a memory location; the bit offset operand can be a register or an immediate value:

- If the bit base operand specifies a register, the instruction takes the modulo 16, 32, or 64 of the bit offset operand (modulo size depends on the mode and register size; 64-bit operands are available only in 64-bit mode).
- If the bit base operand specifies a memory location, the operand represents the address of the byte in memory that contains the bit base (bit 0 of the specified byte) of the bit string. The range of the bit position that can be referenced by the offset operand depends on the operand size.

See also: **Bit(BitBase, BitOffset)** on page 3-11.

Some assemblers support immediate bit offsets larger than 31 by using the immediate bit offset field in combination with the displacement field of the memory operand. In this case, the low-order 3 or 5 bits (3 for 16-bit operands, 5 for 32-bit operands) of the immediate bit offset are stored in the immediate bit offset field, and the high-order bits are shifted and combined with the byte displacement in the addressing mode by the assembler. The processor will ignore the high order bits if they are not zero.

When accessing a bit in memory, the processor may access 4 bytes starting from the memory address for a 32-bit operand size, using by the following relationship:

```
Effective Address + (4 * (BitOffset DIV 32))
```

Or, it may access 2 bytes starting from the memory address for a 16-bit operand, using this relationship:

```
Effective Address + (2 * (BitOffset DIV 16))
```

It may do so even when only a single byte needs to be accessed to reach the given bit. When using this bit addressing mechanism, software should avoid referencing areas of memory close to address space holes. In particular, it should avoid references to memory-mapped I/O registers. Instead, software should use the MOV instructions to load from or store to these addresses, and use the register form of these instructions to manipulate the data.

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

Operation

```
CF ← Bit(BitBase, BitOffset);
```

Flags Affected

The CF flag contains the value of the selected bit. The ZF flag is unaffected. The OF, SF, AF, and PF flags are undefined.

Protected Mode Exceptions

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

Real-Address Mode Exceptions

#GP	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS	If a memory operand effective address is outside the SS segment limit.
#UD	If the LOCK prefix is used.

Virtual-8086 Mode Exceptions

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

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If the LOCK prefix is used.

BTC—Bit Test and Complement

Opcode	Instruction	Op/ En	64-bit Mode	Compat/ Leg Mode	Description
0F BB /r	BTC *r/m16, r16*	MR	Valid	Valid	Store selected bit in CF flag and complement.
0F BB /r	BTC *r/m32, r32*	MR	Valid	Valid	Store selected bit in CF flag and complement.
REX.W + 0F BB /r	BTC *r/m64, r64*	MR	Valid	N.E.	Store selected bit in CF flag and complement.
0F BA /7 *ib*	BTC *r/m16, imm8*	MI	Valid	Valid	Store selected bit in CF flag and complement.
0F BA /7 *ib*	BTC *r/m32, imm8*	MI	Valid	Valid	Store selected bit in CF flag and complement.
REX.W + 0F BA /7 *ib*	BTC *r/m64, imm8*	MI	Valid	N.E.	Store selected bit in CF flag and complement.

Instruction Operand Encoding

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

Description

Selects the bit in a bit string (specified with the first operand, called the bit base) at the bit-position designated by the bit offset operand (second operand), stores the value of the bit in the CF flag, and complements the selected bit in the bit string. The bit base operand can be a register or a memory location; the bit offset operand can be a register or an immediate value:

- If the bit base operand specifies a register, the instruction takes the modulo 16, 32, or 64 of the bit offset operand (modulo size depends on the mode and register size; 64-bit operands are available only in 64-bit mode). This allows any bit position to be selected.
- If the bit base operand specifies a memory location, the operand represents the address of the byte in memory that contains the bit base (bit 0 of the specified byte) of the bit string. The range of the bit position that can be referenced by the offset operand depends on the operand size.

See also: **Bit(BitBase, BitOffset)** on page 3-11.

Some assemblers support immediate bit offsets larger than 31 by using the immediate bit offset field in combination with the displacement field of the memory operand. See "BT—Bit Test" in this chapter for more information on this addressing mechanism.

This instruction can be used with a LOCK prefix to allow the instruction to be executed atomically.

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

Operation

```
CF ← Bit(BitBase, BitOffset);
Bit(BitBase, BitOffset) ← NOT Bit(BitBase, BitOffset);
```

Flags Affected

The CF flag contains the value of the selected bit before it is complemented. The ZF flag is unaffected. The OF, SF, AF, and PF flags are undefined.

Protected Mode Exceptions

#GP(0)	If the destination operand points to a non-writable segment.
	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
	If the DS, ES, FS, or GS register contains a NULL segment selector.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If the LOCK prefix is used but the destination is not a memory operand.

Real-Address Mode Exceptions

#GP	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS	If a memory operand effective address is outside the SS segment limit.
#UD	If the LOCK prefix is used but the destination is not a memory operand.

Virtual-8086 Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.
#UD	If the LOCK prefix is used but the destination is not a memory operand.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If the LOCK prefix is used but the destination is not a memory operand.

BTR—Bit Test and Reset

Opcode	Instruction	Op/ En	64-bit Mode	Compat/ Leg Mode	Description
0F B3 /r	BTR *r/m16, r16*	MR	Valid	Valid	Store selected bit in CF flag and clear.
0F B3 /r	BTR *r/m32, r32*	MR	Valid	Valid	Store selected bit in CF flag and clear.
REX.W + 0F B3 /r	BTR *r/m64, r64*	MR	Valid	N.E.	Store selected bit in CF flag and clear.
0F BA /6 *ib*	BTR *r/m16, imm8*	MI	Valid	Valid	Store selected bit in CF flag and clear.
0F BA /6 *ib*	BTR *r/m32, imm8*	MI	Valid	Valid	Store selected bit in CF flag and clear.
REX.W + 0F BA /6 *ib*	BTR *r/m64, imm8*	MI	Valid	N.E.	Store selected bit in CF flag and clear.

Instruction Operand Encoding

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

Description

Selects the bit in a bit string (specified with the first operand, called the bit base) at the bit-position designated by the bit offset operand (second operand), stores the value of the bit in the CF flag, and clears the selected bit in the bit string to 0. The bit base operand can be a register or a memory location; the bit offset operand can be a register or an immediate value:

- If the bit base operand specifies a register, the instruction takes the modulo 16, 32, or 64 of the bit offset operand (modulo size depends on the mode and register size; 64-bit operands are available only in 64-bit mode). This allows any bit position to be selected.
- If the bit base operand specifies a memory location, the operand represents the address of the byte in memory that contains the bit base (bit 0 of the specified byte) of the bit string. The range of the bit position that can be referenced by the offset operand depends on the operand size.

See also: **Bit(BitBase, BitOffset)** on page 3-11.

Some assemblers support immediate bit offsets larger than 31 by using the immediate bit offset field in combination with the displacement field of the memory operand. See "BT—Bit Test" in this chapter for more information on this addressing mechanism.

This instruction can be used with a LOCK prefix to allow the instruction to be executed atomically.

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

Operation

```
CF ← Bit(BitBase, BitOffset);
Bit(BitBase, BitOffset) ← 0;
```

Flags Affected

The CF flag contains the value of the selected bit before it is cleared. The ZF flag is unaffected. The OF, SF, AF, and PF flags are undefined.

Protected Mode Exceptions

#GP(0)	If the destination operand points to a non-writable segment.
	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
	If the DS, ES, FS, or GS register contains a NULL segment selector.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If the LOCK prefix is used but the destination is not a memory operand.

Real-Address Mode Exceptions

#GP	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS	If a memory operand effective address is outside the SS segment limit.
#UD	If the LOCK prefix is used but the destination is not a memory operand.

Virtual-8086 Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.
#UD	If the LOCK prefix is used but the destination is not a memory operand.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If the LOCK prefix is used but the destination is not a memory operand.

BTS—Bit Test and Set

Opcode	Instruction	Op/ En	64-bit Mode	Compat/ Leg Mode	Description
0F AB /r	BTS *r/m16, r16*	MR	Valid	Valid	Store selected bit in CF flag and set.
0F AB /r	BTS *r/m32, r32*	MR	Valid	Valid	Store selected bit in CF flag and set.
REX.W + 0F AB /r	BTS *r/m64, r64*	MR	Valid	N.E.	Store selected bit in CF flag and set.
0F BA /5 *ib*	BTS *r/m16, imm8*	MI	Valid	Valid	Store selected bit in CF flag and set.
0F BA /5 *ib*	BTS *r/m32, imm8*	MI	Valid	Valid	Store selected bit in CF flag and set.
REX.W + 0F BA /5 *ib*	BTS *r/m64, imm8*	MI	Valid	N.E.	Store selected bit in CF flag and set.

Instruction Operand Encoding

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

Description

Selects the bit in a bit string (specified with the first operand, called the bit base) at the bit-position designated by the bit offset operand (second operand), stores the value of the bit in the CF flag, and sets the selected bit in the bit string to 1. The bit base operand can be a register or a memory location; the bit offset operand can be a register or an immediate value:

- If the bit base operand specifies a register, the instruction takes the modulo 16, 32, or 64 of the bit offset operand (modulo size depends on the mode and register size; 64-bit operands are available only in 64-bit mode). This allows any bit position to be selected.
- If the bit base operand specifies a memory location, the operand represents the address of the byte in memory that contains the bit base (bit 0 of the specified byte) of the bit string. The range of the bit position that can be referenced by the offset operand depends on the operand size.

See also: **Bit(BitBase, BitOffset)** on page 3-11.

Some assemblers support immediate bit offsets larger than 31 by using the immediate bit offset field in combination with the displacement field of the memory operand. See "BT—Bit Test" in this chapter for more information on this addressing mechanism.

This instruction can be used with a LOCK prefix to allow the instruction to be executed atomically.

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

Operation

```
CF ← Bit(BitBase, BitOffset);
Bit(BitBase, BitOffset) ← 1;
```

Flags Affected

The CF flag contains the value of the selected bit before it is set. The ZF flag is unaffected. The OF, SF, AF, and PF flags are undefined.

Protected Mode Exceptions

#GP(0)	If the destination operand points to a non-writable segment.
	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
	If the DS, ES, FS, or GS register contains a NULL segment selector.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If the LOCK prefix is used but the destination is not a memory operand.

Real-Address Mode Exceptions

#GP	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS	If a memory operand effective address is outside the SS segment limit.
#UD	If the LOCK prefix is used but the destination is not a memory operand.

Virtual-8086 Mode Exceptions

#GP	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.
#UD	If the LOCK prefix is used but the destination is not a memory operand.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If the LOCK prefix is used but the destination is not a memory operand.

BZHI — Zero High Bits Starting with Specified Bit Position

Opcode/Instruction	Op/ En	64/32 -bit Mode	CPUID Feature Flag	Description
VEX.NDS.LZ.0F38.W0 F5 /r BZHI r32a, r/m32, r32b	RMV	V/V	BMI2	Zero bits in r/m32 starting with the position in r32b, write result to r32a.
VEX.NDS.LZ.0F38.W1 F5 /r BZHI r64a, r/m64, r64b	RMV	V/N.E.	BMI2	Zero bits in r/m64 starting with the position in r64b, write result to r64a.

Instruction Operand Encoding

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

Description

BZHI copies the bits of the first source operand (the second operand) into the destination operand (the first operand) and clears the higher bits in the destination according to the INDEX value specified by the second source operand (the third operand). The INDEX is specified by bits 7:0 of the second source operand. The INDEX value is saturated at the value of OperandSize -1. CF is set, if the number contained in the 8 low bits of the third operand is greater than OperandSize -1.

This instruction is not supported in real mode and virtual-8086 mode. The operand size is always 32 bits if not in 64-bit mode. In 64-bit mode operand size 64 requires VEX.W1. VEX.W1 is ignored in non-64-bit modes. An attempt to execute this instruction with VEX.L not equal to 0 will cause #UD.

Operation

```
N ← SRC2[7:0]
DEST ← SRC1
IF (N < OperandSize)
    DEST[OperandSize-1:N] ← 0
FI
IF (N > OperandSize - 1)
    CF ← 1
ELSE
    CF ← 0
FI
```

Flags Affected

ZF, CF and SF flags are updated based on the result. OF flag is cleared. AF and PF flags are undefined.

Intel C/C++ Compiler Intrinsic Equivalent

BZHI: unsigned __int32 _bzhi_u32(unsigned __int32 src, unsigned __int32 index);

BZHI: unsigned __int64 _bzhi_u64(unsigned __int64 src, unsigned __int32 index);

SIMD Floating-Point Exceptions

None

Other Exceptions

See Section 2.5.1, “Exception Conditions for VEX-Encoded GPR Instructions”, Table 2-29; additionally

#UD	If VEX.W = 1.

CALL—Call Procedure

Opcode	Instruction	Op/ En	64-bit Mode	Compat/ Leg Mode	Description
E8 *cw*	CALL *rel16*	M	N.S.	Valid	Call near, relative, displacement relative to next instruction.
E8 *cd*	CALL *rel32*	M	Valid	Valid	Call near, relative, displacement relative to next instruction. 32-bit displacement sign extended to 64-bits in 64-bit mode.
FF /2	CALL *r/m16*	M	N.E.	Valid	Call near, absolute indirect, address given in *r/m16.*
FF /2	CALL *r/m32*	M	N.E.	Valid	Call near, absolute indirect, address given in *r/m32.*
FF /2	CALL *r/m64*	M	Valid	N.E.	Call near, absolute indirect, address given in *r/m*64.
9A *cd*	CALL *ptr16:16*	D	Invalid	Valid	Call far, absolute, address given in operand.
9A *cp*	CALL *ptr16:32*	D	Invalid	Valid	Call far, absolute, address given in operand.
FF /3	CALL *m16:16*	M	Valid	Valid	Call far, absolute indirect address given in *m16:16.* In 32-bit mode: if selector points to a gate, then RIP = 32-bit zero extended displacement taken from gate; else RIP = zero extended 16-bit offset from far pointer referenced in the instruction.
FF /3	CALL *m16:32*	M	Valid	Valid	In 64-bit mode: If selector points to a gate, then RIP = 64-bit displacement taken from gate; else RIP = zero extended 32-bit offset from far pointer referenced in the instruction.
REX.W + FF /3	CALL *m16:64*	M	Valid	N.E.	In 64-bit mode: If selector points to a gate, then RIP = 64-bit displacement taken from gate; else RIP = 64-bit offset from far pointer referenced in the instruction.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
D	Offset	NA	NA	NA
M	ModRM:r/m (r)	NA	NA	NA

Description

Saves procedure linking information on the stack and branches to the called procedure specified using the target operand. The target operand specifies the address of the first instruction in the called procedure. The operand can be an immediate value, a general-purpose register, or a memory location.

This instruction can be used to execute four types of calls:

- **Near Call** — A call to a procedure in the current code segment (the segment currently pointed to by the CS register), sometimes referred to as an intra-segment call.
- **Far Call** — A call to a procedure located in a different segment than the current code segment, sometimes referred to as an inter-segment call.
- **Inter-privilege-level far call** — A far call to a procedure in a segment at a different privilege level than that of the currently executing program or procedure.
- **Task switch** — A call to a procedure located in a different task.

The latter two call types (inter-privilege-level call and task switch) can only be executed in protected mode. See "Calling Procedures Using Call and RET" in Chapter 6 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*, for additional information on near, far, and inter-privilege-level calls. See Chapter 7, "Task Management," in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*, for information on performing task switches with the CALL instruction.

Near Call. When executing a near call, the processor pushes the value of the EIP register (which contains the offset of the instruction following the CALL instruction) on the stack (for use later as a return-instruction pointer). The processor then branches to the address in the current code segment specified by the target operand. The target operand specifies either an absolute offset in the code segment (an offset from the base of the code segment) or a relative offset (a signed displacement relative to the current value of the instruction pointer in the EIP register; this value points to the instruction following the CALL instruction). The CS register is not changed on near calls.

For a near call absolute, an absolute offset is specified indirectly in a general-purpose register or a memory location (*r/m16*, *r/m32, or r/m64*). The operand-size attribute determines the size of the target operand (16, 32 or 64 bits). When in 64-bit mode, the operand size for near call (and all near branches) is forced to 64-bits. Absolute offsets are loaded directly into the EIP(RIP) register. If the operand size attribute is 16, the upper two bytes of the EIP register are cleared, resulting in a maximum instruction pointer size of 16 bits. When accessing an absolute offset indirectly using the stack pointer [ESP] as the base register, the base value used is the value of the ESP before the instruction executes.

A relative offset (*rel16* or *rel32*) is generally specified as a label in assembly code. But at the machine code level, it is encoded as a signed, 16- or 32-bit immediate value. This value is added to the value in the EIP(RIP) register. In 64-bit mode the relative offset is always a 32-bit immediate value which is sign extended to 64-bits before it is added to the value in the RIP register for the target calculation. As with absolute offsets, the operand-size attribute determines the size of the target operand (16, 32, or 64 bits). In 64-bit mode the target operand will always be 64-bits because the operand size is forced to 64-bits for near branches.

Far Calls in Real-Address or Virtual-8086 Mode. When executing a far call in real- address or virtual-8086 mode, the processor pushes the current value of both the CS and EIP registers on the stack for use as a return-instruction pointer. The processor then performs a "far branch" to the code segment and offset specified with the target operand for the called procedure. The target operand specifies an absolute far address either directly with a pointer (*ptr16:16* or *ptr16:32*) or indirectly with a memory location (*m16:16* or *m16:32*). With the pointer method, the segment and offset of the called procedure is encoded in the instruction using a 4-byte (16-bit operand size) or 6-byte (32-bit operand size) far address immediate. With the indirect method, the target operand specifies a memory location that contains a 4-byte (16-bit operand size) or 6-byte (32-bit operand size) far address. The operand-size attribute determines the size of the offset (16 or 32 bits) in the far address. The far address is loaded directly into the CS and EIP registers. If the operand-size attribute is 16, the upper two bytes of the EIP register are cleared.

Far Calls in Protected Mode. When the processor is operating in protected mode, the CALL instruction can be used to perform the following types of far calls:

- Far call to the same privilege level
- Far call to a different privilege level (inter-privilege level call)
- Task switch (far call to another task)

In protected mode, the processor always uses the segment selector part of the far address to access the corresponding descriptor in the GDT or LDT. The descriptor type (code segment, call gate, task gate, or TSS) and access rights determine the type of call operation to be performed.

If the selected descriptor is for a code segment, a far call to a code segment at the same privilege level is performed. (If the selected code segment is at a different privilege level and the code segment is non-conforming, a general-protection exception is generated.) A far call to the same privilege level in protected mode is very similar to one carried out in real-address or virtual-8086 mode. The target operand specifies an absolute far address either directly with a pointer (*ptr16:16* or *ptr16:32*) or indirectly with a memory location (*m16:16* or *m16:32*). The operand- size attribute determines the size of the offset (16 or 32 bits) in the far address. The new code segment selector and its descriptor are loaded into CS register; the offset from the instruction is loaded into the EIP register.

A call gate (described in the next paragraph) can also be used to perform a far call to a code segment at the same privilege level. Using this mechanism provides an extra level of indirection and is the preferred method of making calls between 16-bit and 32-bit code segments.

When executing an inter-privilege-level far call, the code segment for the procedure being called must be accessed through a call gate. The segment selector specified by the target operand identifies the call gate. The target operand can specify the call gate segment selector either directly with a pointer (*ptr16:16* or *ptr16:32*) or indirectly with a memory location (*m16:16* or *m16:32*). The processor obtains the segment selector for the new code segment and the new instruction pointer (offset) from the call gate descriptor. (The offset from the target operand is ignored when a call gate is used.)

On inter-privilege-level calls, the processor switches to the stack for the privilege level of the called procedure. The segment selector for the new stack segment is specified in the TSS for the currently running task. The branch to the new code segment occurs after the stack switch. (Note that when using a call gate to perform a far call to a segment at the same privilege level, no stack switch occurs.) On the new stack, the processor pushes the segment selector and stack pointer for the calling procedure's stack, an optional set of parameters from the calling procedures stack, and the segment selector and instruction pointer for the calling procedure's code segment. (A value in the call gate descriptor determines how many parameters to copy to the new stack.) Finally, the processor branches to the address of the procedure being called within the new code segment.

Executing a task switch with the CALL instruction is similar to executing a call through a call gate. The target operand specifies the segment selector of the task gate for the new task activated by the switch (the offset in the target operand is ignored). The task gate in turn points to the TSS for the new task, which contains the segment selectors for the task's code and stack segments. Note that the TSS also contains the EIP value for the next instruction that was to be executed before the calling task was suspended. This instruction pointer value is loaded into the EIP register to re-start the calling task.

The CALL instruction can also specify the segment selector of the TSS directly, which eliminates the indirection of the task gate. See Chapter 7, "Task Management," in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*, for information on the mechanics of a task switch.

When you execute at task switch with a CALL instruction, the nested task flag (NT) is set in the EFLAGS register and the new TSS's previous task link field is loaded with the old task's TSS selector. Code is expected to suspend this nested task by executing an IRET instruction which, because the NT flag is set, automatically uses the previous task link to return to the calling task. (See "Task Linking" in Chapter 7 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*, for information on nested tasks.) Switching tasks with the CALL instruction differs in this regard from JMP instruction. JMP does not set the NT flag and therefore does not expect an IRET instruction to suspend the task.

Mixing 16-Bit and 32-Bit Calls. When making far calls between 16-bit and 32-bit code segments, use a call gate. If the far call is from a 32-bit code segment to a 16-bit code segment, the call should be made from the first 64 KBytes of the 32-bit code segment. This is because the operand-size attribute of the instruction is set to 16, so only a 16-bit return address offset can be saved. Also, the call should be made using a 16-bit call gate so that 16-bit values can be pushed on the stack. See Chapter 21, "Mixing 16-Bit and 32-Bit Code," in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3B*, for more information.

Far Calls in Compatibility Mode. When the processor is operating in compatibility mode, the CALL instruction can be used to perform the following types of far calls:

- Far call to the same privilege level, remaining in compatibility mode
- Far call to the same privilege level, transitioning to 64-bit mode
- Far call to a different privilege level (inter-privilege level call), transitioning to 64-bit mode

Note that a CALL instruction can not be used to cause a task switch in compatibility mode since task switches are not supported in IA-32e mode.

In compatibility mode, the processor always uses the segment selector part of the far address to access the corresponding descriptor in the GDT or LDT. The descriptor type (code segment, call gate) and access rights determine the type of call operation to be performed.

If the selected descriptor is for a code segment, a far call to a code segment at the same privilege level is performed. (If the selected code segment is at a different privilege level and the code segment is non-conforming, a general-protection exception is generated.) A far call to the same privilege level in compatibility mode is very similar to one carried out in protected mode. The target operand specifies an absolute far address either directly with a pointer (*ptr16:16* or *ptr16:32*) or indirectly with a memory location (*m16:16* or *m16:32*). The operand-size attribute determines the size of the offset (16 or 32 bits) in the far address. The new code segment selector and its descriptor are loaded into CS register and the offset from the instruction is loaded into the EIP register. The difference is that 64-bit mode may be entered. This specified by the L bit in the new code segment descriptor.

Note that a 64-bit call gate (described in the next paragraph) can also be used to perform a far call to a code segment at the same privilege level. However, using this mechanism requires that the target code segment descriptor have the L bit set, causing an entry to 64-bit mode.

When executing an inter-privilege-level far call, the code segment for the procedure being called must be accessed through a 64-bit call gate. The segment selector specified by the target operand identifies the call gate. The target

operand can specify the call gate segment selector either directly with a pointer (*ptr16:16* or *ptr16:32*) or indirectly with a memory location (*m16:16* or *m16:32*). The processor obtains the segment selector for the new code segment and the new instruction pointer (offset) from the 16-byte call gate descriptor. (The offset from the target operand is ignored when a call gate is used.)

On inter-privilege-level calls, the processor switches to the stack for the privilege level of the called procedure. The segment selector for the new stack segment is set to NULL. The new stack pointer is specified in the TSS for the currently running task. The branch to the new code segment occurs after the stack switch. (Note that when using a call gate to perform a far call to a segment at the same privilege level, an implicit stack switch occurs as a result of entering 64-bit mode. The SS selector is unchanged, but stack segment accesses use a segment base of 0x0, the limit is ignored, and the default stack size is 64-bits. The full value of RSP is used for the offset, of which the upper 32-bits are undefined.) On the new stack, the processor pushes the segment selector and stack pointer for the calling procedure's stack and the segment selector and instruction pointer for the calling procedure's code segment. (Parameter copy is not supported in IA-32e mode.) Finally, the processor branches to the address of the procedure being called within the new code segment.

Near/(Far) Calls in 64-bit Mode. When the processor is operating in 64-bit mode, the CALL instruction can be used to perform the following types of far calls:

- Far call to the same privilege level, transitioning to compatibility mode
- Far call to the same privilege level, remaining in 64-bit mode
- Far call to a different privilege level (inter-privilege level call), remaining in 64-bit mode

Note that in this mode the CALL instruction can not be used to cause a task switch in 64-bit mode since task switches are not supported in IA-32e mode.

In 64-bit mode, the processor always uses the segment selector part of the far address to access the corresponding descriptor in the GDT or LDT. The descriptor type (code segment, call gate) and access rights determine the type of call operation to be performed.

If the selected descriptor is for a code segment, a far call to a code segment at the same privilege level is performed. (If the selected code segment is at a different privilege level and the code segment is non-conforming, a general-protection exception is generated.) A far call to the same privilege level in 64-bit mode is very similar to one carried out in compatibility mode. The target operand specifies an absolute far address indirectly with a memory location (*m16:16*, *m16:32* or *m16:64*). The form of CALL with a direct specification of absolute far address is not defined in 64-bit mode. The operand-size attribute determines the size of the offset (16, 32, or 64 bits) in the far address. The new code segment selector and its descriptor are loaded into the CS register; the offset from the instruction is loaded into the EIP register. The new code segment may specify entry either into compatibility or 64-bit mode, based on the L bit value.

A 64-bit call gate (described in the next paragraph) can also be used to perform a far call to a code segment at the same privilege level. However, using this mechanism requires that the target code segment descriptor have the L bit set.

When executing an inter-privilege-level far call, the code segment for the procedure being called must be accessed through a 64-bit call gate. The segment selector specified by the target operand identifies the call gate. The target operand can only specify the call gate segment selector indirectly with a memory location (*m16:16*, *m16:32* or *m16:64*). The processor obtains the segment selector for the new code segment and the new instruction pointer (offset) from the 16-byte call gate descriptor. (The offset from the target operand is ignored when a call gate is used.)

On inter-privilege-level calls, the processor switches to the stack for the privilege level of the called procedure. The segment selector for the new stack segment is set to NULL. The new stack pointer is specified in the TSS for the currently running task. The branch to the new code segment occurs after the stack switch.

Note that when using a call gate to perform a far call to a segment at the same privilege level, an implicit stack switch occurs as a result of entering 64-bit mode. The SS selector is unchanged, but stack segment accesses use a segment base of 0x0, the limit is ignored, and the default stack size is 64-bits. (The full value of RSP is used for the offset.) On the new stack, the processor pushes the segment selector and stack pointer for the calling procedure's stack and the segment selector and instruction pointer for the calling procedure's code segment. (Parameter copy is not supported in IA-32e mode.) Finally, the processor branches to the address of the procedure being called within the new code segment.

Operation

```
IF near call
    THEN IF near relative call
        THEN
            IF OperandSize = 64
                THEN
                    tempDEST ← SignExtend(DEST); (* DEST is rel32 *)
                    tempRIP ← RIP + tempDEST;
                    IF stack not large enough for a 8-byte return address
                        THEN #SS(0); FI;
                    Push(RIP);
                    RIP ← tempRIP;
            FI;
            IF OperandSize = 32
                THEN
                    tempEIP ← EIP + DEST; (* DEST is rel32 *)
                    IF tempEIP is not within code segment limit THEN #GP(0); FI;
                    IF stack not large enough for a 4-byte return address
                        THEN #SS(0); FI;
                    Push(EIP);
                    EIP ← tempEIP;
            FI;
            IF OperandSize = 16
                THEN
                    tempEIP ← (EIP + DEST) AND 0000FFFFH; (* DEST is rel16 *)
                    IF tempEIP is not within code segment limit THEN #GP(0); FI;
                    IF stack not large enough for a 2-byte return address
                        THEN #SS(0); FI;
                    Push(IP);
                    EIP ← tempEIP;
            FI;
        ELSE (* Near absolute call *)
            IF OperandSize = 64
                THEN
                    tempRIP ← DEST; (* DEST is r/m64 *)
                    IF stack not large enough for a 8-byte return address
                        THEN #SS(0); FI;
                    Push(RIP);
                    RIP ← tempRIP;
            FI;
            IF OperandSize = 32
                THEN
                    tempEIP ← DEST; (* DEST is r/m32 *)
                    IF tempEIP is not within code segment limit THEN #GP(0); FI;
                    IF stack not large enough for a 4-byte return address
                        THEN #SS(0); FI;
                    Push(EIP);
                    EIP ← tempEIP;
            FI;
            IF OperandSize = 16
                THEN
                    tempEIP ← DEST AND 0000FFFFH; (* DEST is r/m16 *)
                    IF tempEIP is not within code segment limit THEN #GP(0); FI;
```

```
                    IF stack not large enough for a 2-byte return address
                        THEN #SS(0); FI;
                    Push(IP);
                    EIP ← tempEIP;
            FI;
    FI;rel/abs
FI; near

IF far call and (PE = 0 or (PE = 1 and VM = 1)) (* Real-address or virtual-8086 mode *)
    THEN
        IF OperandSize = 32
            THEN
                IF stack not large enough for a 6-byte return address
                    THEN #SS(0); FI;
                IF DEST[31:16] is not zero THEN #GP(0); FI;
                Push(CS); (* Padded with 16 high-order bits *)
                Push(EIP);
                CS ← DEST[47:32]; (* DEST is ptr16:32 or [m16:32] *)
                EIP ← DEST[31:0]; (* DEST is ptr16:32 or [m16:32] *)
            ELSE (* OperandSize = 16 *)
                IF stack not large enough for a 4-byte return address
                    THEN #SS(0); FI;
                Push(CS);
                Push(IP);
                CS ← DEST[31:16]; (* DEST is ptr16:16 or [m16:16] *)
                EIP ← DEST[15:0]; (* DEST is ptr16:16 or [m16:16]; clear upper 16 bits *)
        FI;
FI;

IF far call and (PE = 1 and VM = 0) (* Protected mode or IA-32e Mode, not virtual-8086 mode*)
    THEN
        IF segment selector in target operand NULL
            THEN #GP(0); FI;
        IF segment selector index not within descriptor table limits
            THEN #GP(new code segment selector); FI;
        Read type and access rights of selected segment descriptor;
        IF IA32_EFER.LMA = 0
            THEN
                IF segment type is not a conforming or nonconforming code segment, call
                gate, task gate, or TSS
                    THEN #GP(segment selector); FI;
            ELSE
                IF segment type is not a conforming or nonconforming code segment or
                64-bit call gate,
                    THEN #GP(segment selector); FI;
        FI;
        Depending on type and access rights:
            GO TO CONFORMING-CODE-SEGMENT;
            GO TO NONCONFORMING-CODE-SEGMENT;
            GO TO CALL-GATE;
            GO TO TASK-GATE;
            GO TO TASK-STATE-SEGMENT;
FI;
```

```
CONFORMING-CODE-SEGMENT:
    IF L bit = 1 and D bit = 1 and IA32_EFER.LMA = 1
        THEN GP(new code segment selector); FI;
    IF DPL > CPL
        THEN #GP(new code segment selector); FI;
    IF segment not present
        THEN #NP(new code segment selector); FI;
    IF stack not large enough for return address
        THEN #SS(0); FI;
    tempEIP ← DEST(Offset);
    IF target mode = Compatibility mode
        THEN tempEIP ← tempEIP AND 00000000_FFFFFFFFH; FI;
    IF OperandSize = 16
        THEN
            tempEIP ← tempEIP AND 0000FFFFH; FI; (* Clear upper 16 bits *)
    IF (EFER.LMA = 0 or target mode = Compatibility mode) and (tempEIP outside new code
    segment limit)
        THEN #GP(0); FI;
    IF tempEIP is non-canonical
        THEN #GP(0); FI;
    IF OperandSize = 32
        THEN
            Push(CS); (* Padded with 16 high-order bits *)
            Push(EIP);
            CS ← DEST(CodeSegmentSelector);
            (* Segment descriptor information also loaded *)
            CS(RPL) ← CPL;
            EIP ← tempEIP;
        ELSE
            IF OperandSize = 16
                THEN
                    Push(CS);
                    Push(IP);
                    CS ← DEST(CodeSegmentSelector);
                    (* Segment descriptor information also loaded *)
                    CS(RPL) ← CPL;
                    EIP ← tempEIP;
                ELSE (* OperandSize = 64 *)
                    Push(CS); (* Padded with 48 high-order bits *)
                    Push(RIP);
                    CS ← DEST(CodeSegmentSelector);
                    (* Segment descriptor information also loaded *)
                    CS(RPL) ← CPL;
                    RIP ← tempEIP;
            FI;
    FI;
END;

NONCONFORMING-CODE-SEGMENT:
    IF L-Bit = 1 and D-BIT = 1 and IA32_EFER.LMA = 1
        THEN GP(new code segment selector); FI;
    IF (RPL > CPL) or (DPL ≠ CPL)
        THEN #GP(new code segment selector); FI;
    IF segment not present
```

```
        THEN #NP(new code segment selector); FI;
    IF stack not large enough for return address
        THEN #SS(0); FI;
    tempEIP ← DEST(Offset);
    IF target mode = Compatibility mode
        THEN tempEIP ← tempEIP AND 00000000_FFFFFFFFH; FI;
    IF OperandSize = 16
        THEN tempEIP ← tempEIP AND 0000FFFFH; FI; (* Clear upper 16 bits *)
    IF (EFER.LMA = 0 or target mode = Compatibility mode) and (tempEIP outside new code
    segment limit)
        THEN #GP(0); FI;
    IF tempEIP is non-canonical
        THEN #GP(0); FI;
    IF OperandSize = 32
        THEN
            Push(CS); (* Padded with 16 high-order bits *)
            Push(EIP);
            CS ← DEST(CodeSegmentSelector);
            (* Segment descriptor information also loaded *)
            CS(RPL) ← CPL;
            EIP ← tempEIP;
        ELSE
            IF OperandSize = 16
                THEN
                    Push(CS);
                    Push(IP);
                    CS ← DEST(CodeSegmentSelector);
                    (* Segment descriptor information also loaded *)
                    CS(RPL) ← CPL;
                    EIP ← tempEIP;
                ELSE (* OperandSize = 64 *)
                    Push(CS); (* Padded with 48 high-order bits *)
                    Push(RIP);
                    CS ← DEST(CodeSegmentSelector);
                    (* Segment descriptor information also loaded *)
                    CS(RPL) ← CPL;
                    RIP ← tempEIP;
            FI;
    FI;
END;

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

```
    IF IA32_EFER.LMA = 1 AND (call-gate code-segment descriptor is
    not a 64-bit code segment or call-gate code-segment descriptor has both L-bit and D-bit set)
        THEN #GP(call-gate code-segment selector); FI;
    IF call-gate code segment not present
        THEN #NP(call-gate code-segment selector); FI;
    IF call-gate code segment is non-conforming and DPL < CPL
        THEN go to MORE-PRIVILEGE;
        ELSE go to SAME-PRIVILEGE;
    FI;
END;

MORE-PRIVILEGE:
    IF current TSS is 32-bit
        THEN
            TSSstackAddress ← (new code-segment DPL * 8) + 4;
            IF (TSSstackAddress + 5) > current TSS limit
                THEN #TS(current TSS selector); FI;
            NewSS ← 2 bytes loaded from (TSS base + TSSstackAddress + 4);
            NewESP ← 4 bytes loaded from (TSS base + TSSstackAddress);
        ELSE
            IF current TSS is 16-bit
                THEN
                    TSSstackAddress ← (new code-segment DPL * 4) + 2
                    IF (TSSstackAddress + 3) > current TSS limit
                        THEN #TS(current TSS selector); FI;
                    NewSS ← 2 bytes loaded from (TSS base + TSSstackAddress + 2);
                    NewESP ← 2 bytes loaded from (TSS base + TSSstackAddress);
                ELSE (* current TSS is 64-bit *)
                    TSSstackAddress ← (new code-segment DPL * 8) + 4;
                    IF (TSSstackAddress + 7) > current TSS limit
                        THEN #TS(current TSS selector); FI;
                    NewSS ← new code-segment DPL; (* NULL selector with RPL = new CPL *)
                    NewRSP ← 8 bytes loaded from (current TSS base + TSSstackAddress);
            FI;
    FI;
    IF IA32_EFER.LMA = 0 and NewSS is NULL
        THEN #TS(NewSS); FI;
    Read new stack-segment descriptor;
    IF IA32_EFER.LMA = 0 and (NewSS RPL ≠ new code-segment DPL
    or new stack-segment DPL ≠ new code-segment DPL or new stack segment is not a
    writable data segment)
        THEN #TS(NewSS); FI
    IF IA32_EFER.LMA = 0 and new stack segment not present
        THEN #SS(NewSS); FI;
    IF CallGateSize = 32
        THEN
            IF new stack does not have room for parameters plus 16 bytes
                THEN #SS(NewSS); FI;
            IF CallGate(InstructionPointer) not within new code-segment limit
                THEN #GP(0); FI;
            SS ← newSS; (* Segment descriptor information also loaded *)
            ESP ← newESP;
            CS:EIP ← CallGate(CS:InstructionPointer);
            (* Segment descriptor information also loaded *)
```

```
            Push(oldSS:oldESP); (* From calling procedure *)
            temp ← parameter count from call gate, masked to 5 bits;
            Push(parameters from calling procedure's stack, temp)
            Push(oldCS:oldEIP); (* Return address to calling procedure *)
        ELSE
            IF CallGateSize = 16
                THEN
                    IF new stack does not have room for parameters plus 8 bytes
                        THEN #SS(NewSS); FI;
                    IF (CallGate(InstructionPointer) AND FFFFH) not in new code-segment limit
                        THEN #GP(0); FI;
                    SS ← newSS; (* Segment descriptor information also loaded *)
                    ESP ← newESP;
                    CS:IP ← CallGate(CS:InstructionPointer);
                    (* Segment descriptor information also loaded *)
                    Push(oldSS:oldESP); (* From calling procedure *)
                    temp ← parameter count from call gate, masked to 5 bits;
                    Push(parameters from calling procedure's stack, temp)
                    Push(oldCS:oldEIP); (* Return address to calling procedure *)
                ELSE (* CallGateSize = 64 *)
                    IF pushing 32 bytes on the stack would use a non-canonical address
                        THEN #SS(NewSS); FI;
                    IF (CallGate(InstructionPointer) is non-canonical)
                        THEN #GP(0); FI;
                    SS ← NewSS; (* NewSS is NULL)
                    RSP ← NewESP;
                    CS:IP ← CallGate(CS:InstructionPointer);
                    (* Segment descriptor information also loaded *)
                    Push(oldSS:oldESP); (* From calling procedure *)
                    Push(oldCS:oldEIP); (* Return address to calling procedure *)
            FI;
    FI;
    CPL ← CodeSegment(DPL)
    CS(RPL) ← CPL
END;

SAME-PRIVILEGE:
    IF CallGateSize = 32
        THEN
            IF stack does not have room for 8 bytes
                THEN #SS(0); FI;
            IF CallGate(InstructionPointer) not within code segment limit
                THEN #GP(0); FI;
            CS:EIP ← CallGate(CS:EIP) (* Segment descriptor information also loaded *)
            Push(oldCS:oldEIP); (* Return address to calling procedure *)
        ELSE
            If CallGateSize = 16
                THEN
                    IF stack does not have room for 4 bytes
                        THEN #SS(0); FI;
                    IF CallGate(InstructionPointer) not within code segment limit
                        THEN #GP(0); FI;
                    CS:IP ← CallGate(CS:instruction pointer);
                    (* Segment descriptor information also loaded *)
```

```
                    Push(oldCS:oldIP); (* Return address to calling procedure *)
                ELSE (* CallGateSize = 64)
                    IF pushing 16 bytes on the stack touches non-canonical addresses
                        THEN #SS(0); FI;
                    IF RIP non-canonical
                        THEN #GP(0); FI;
                    CS:IP ← CallGate(CS:instruction pointer);
                    (* Segment descriptor information also loaded *)
                    Push(oldCS:oldIP); (* Return address to calling procedure *)
            FI;
    FI;
    CS(RPL) ← CPL
END;

TASK-GATE:
    IF task gate DPL < CPL or RPL
        THEN #GP(task gate selector); FI;
    IF task gate not present
        THEN #NP(task gate selector); FI;
    Read the TSS segment selector in the task-gate descriptor;
    IF TSS segment selector local/global bit is set to local
    or index not within GDT limits
        THEN #GP(TSS selector); FI;
    Access TSS descriptor in GDT;
    IF descriptor is not a TSS segment
        THEN #GP(TSS selector); FI;
    IF TSS descriptor specifies that the TSS is busy
        THEN #GP(TSS selector); FI;
    IF TSS not present
        THEN #NP(TSS selector); FI;
    SWITCH-TASKS (with nesting) to TSS;
    IF EIP not within code segment limit
        THEN #GP(0); FI;
END;

TASK-STATE-SEGMENT:
    IF TSS DPL < CPL or RPL
    or TSS descriptor indicates TSS not available
        THEN #GP(TSS selector); FI;
    IF TSS is not present
        THEN #NP(TSS selector); FI;
    SWITCH-TASKS (with nesting) to TSS;
    IF EIP not within code segment limit
        THEN #GP(0); FI;
END;
```

Flags Affected

All flags are affected if a task switch occurs; no flags are affected if a task switch does not occur.

Protected Mode Exceptions

#GP(0)	If the target offset in destination operand is beyond the new code segment limit.
	If the segment selector in the destination operand is NULL.
	If the code segment selector in the gate is NULL.
	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
	If the DS, ES, FS, or GS register is used to access memory and it contains a NULL segment selector.
#GP(selector)	If a code segment or gate or TSS selector index is outside descriptor table limits.
	If the segment descriptor pointed to by the segment selector in the destination operand is not for a conforming-code segment, nonconforming-code segment, call gate, task gate, or task state segment.
	If the DPL for a nonconforming-code segment is not equal to the CPL or the RPL for the segment's segment selector is greater than the CPL.
	If the DPL for a conforming-code segment is greater than the CPL.
	If the DPL from a call-gate, task-gate, or TSS segment descriptor is less than the CPL or than the RPL of the call-gate, task-gate, or TSS's segment selector.
	If the segment descriptor for a segment selector from a call gate does not indicate it is a code segment.
	If the segment selector from a call gate is beyond the descriptor table limits.
	If the DPL for a code-segment obtained from a call gate is greater than the CPL.
	If the segment selector for a TSS has its local/global bit set for local.
	If a TSS segment descriptor specifies that the TSS is busy or not available.
#SS(0)	If pushing the return address, parameters, or stack segment pointer onto the stack exceeds the bounds of the stack segment, when no stack switch occurs.
	If a memory operand effective address is outside the SS segment limit.
#SS(selector)	If pushing the return address, parameters, or stack segment pointer onto the stack exceeds the bounds of the stack segment, when a stack switch occurs.
	If the SS register is being loaded as part of a stack switch and the segment pointed to is marked not present.
	If stack segment does not have room for the return address, parameters, or stack segment pointer, when stack switch occurs.
#NP(selector)	If a code segment, data segment, stack segment, call gate, task gate, or TSS is not present.
#TS(selector)	If the new stack segment selector and ESP are beyond the end of the TSS.
	If the new stack segment selector is NULL.
	If the RPL of the new stack segment selector in the TSS is not equal to the DPL of the code segment being accessed.
	If DPL of the stack segment descriptor for the new stack segment is not equal to the DPL of the code segment descriptor.
	If the new stack segment is not a writable data segment.
	If segment-selector index for stack segment is outside descriptor table limits.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If the LOCK prefix is used.

Real-Address Mode Exceptions

#GP	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
	If the target offset is beyond the code segment limit.
#UD	If the LOCK prefix is used.

Virtual-8086 Mode Exceptions

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

Compatibility Mode Exceptions

Same exceptions as in protected mode.

#GP(selector)	If a memory address accessed by the selector is in non-canonical space.
#GP(0)	If the target offset in the destination operand is non-canonical.

64-Bit Mode Exceptions

#GP(0)	If a memory address is non-canonical.
	If target offset in destination operand is non-canonical.
	If the segment selector in the destination operand is NULL.
	If the code segment selector in the 64-bit gate is NULL.
#GP(selector)	If code segment or 64-bit call gate is outside descriptor table limits.
	If code segment or 64-bit call gate overlaps non-canonical space.
	If the segment descriptor pointed to by the segment selector in the destination operand is not for a conforming-code segment, nonconforming-code segment, or 64-bit call gate.
	If the segment descriptor pointed to by the segment selector in the destination operand is a code segment and has both the D-bit and the L- bit set.
	If the DPL for a nonconforming-code segment is not equal to the CPL, or the RPL for the segment's segment selector is greater than the CPL.
	If the DPL for a conforming-code segment is greater than the CPL.
	If the DPL from a 64-bit call-gate is less than the CPL or than the RPL of the 64-bit call-gate.
	If the upper type field of a 64-bit call gate is not 0x0.
	If the segment selector from a 64-bit call gate is beyond the descriptor table limits.
	If the DPL for a code-segment obtained from a 64-bit call gate is greater than the CPL.
	If the code segment descriptor pointed to by the selector in the 64-bit gate doesn't have the L-bit set and the D-bit clear.
	If the segment descriptor for a segment selector from the 64-bit call gate does not indicate it is a code segment.
#SS(0)	If pushing the return offset or CS selector onto the stack exceeds the bounds of the stack segment when no stack switch occurs.
	If a memory operand effective address is outside the SS segment limit.
	If the stack address is in a non-canonical form.
#SS(selector)	If pushing the old values of SS selector, stack pointer, EFLAGS, CS selector, offset, or error code onto the stack violates the canonical boundary when a stack switch occurs.
#NP(selector)	If a code segment or 64-bit call gate is not present.
#TS(selector)	If the load of the new RSP exceeds the limit of the TSS.
#UD	(64-bit mode only) If a far call is direct to an absolute address in memory.
	If the LOCK prefix is used.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

CBW/CWDE/CDQE—Convert Byte to Word/Convert Word to Doubleword/Convert Doubleword to Quadword

Opcode	Instruction	Op/ En	64-bit Mode	Compat/ Leg Mode	Description
98	CBW	ZO	Valid	Valid	AX ← sign-extend of AL.
98	CWDE	ZO	Valid	Valid	EAX ← sign-extend of AX.
REX.W + 98	CDQE	ZO	Valid	N.E.	RAX ← sign-extend of EAX.

Instruction Operand Encoding

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

Description

Double the size of the source operand by means of sign extension. The CBW (convert byte to word) instruction copies the sign (bit 7) in the source operand into every bit in the AH register. The CWDE (convert word to double-word) instruction copies the sign (bit 15) of the word in the AX register into the high 16 bits of the EAX register.

CBW and CWDE reference the same opcode. The CBW instruction is intended for use when the operand-size attribute is 16; CWDE is intended for use when the operand-size attribute is 32. Some assemblers may force the operand size. Others may treat these two mnemonics as synonyms (CBW/CWDE) and use the setting of the operand-size attribute to determine the size of values to be converted.

In 64-bit mode, the default operation size is the size of the destination register. Use of the REX.W prefix promotes this instruction (CDQE when promoted) to operate on 64-bit operands. In which case, CDQE copies the sign (bit 31) of the doubleword in the EAX register into the high 32 bits of RAX.

Operation

```
IF OperandSize = 16 (* Instruction = CBW *)
    THEN
        AX ← SignExtend(AL);
    ELSE IF (OperandSize = 32, Instruction = CWDE)
        EAX ← SignExtend(AX); FI;
    ELSE (* 64-Bit Mode, OperandSize = 64, Instruction = CDQE*)
        RAX ← SignExtend(EAX);
FI;
```

Flags Affected

None.

Exceptions (All Operating Modes)

#UD	If the LOCK prefix is used.

CLAC—Clear AC Flag in EFLAGS Register

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F 01 CA CLAC	ZO	V/V	SMAP	Clear the AC flag in the EFLAGS register.

Instruction Operand Encoding

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

Description

Clears the AC flag bit in EFLAGS register. This disables any alignment checking of user-mode data accesses. If the SMAP bit is set in the CR4 register, this disallows explicit supervisor-mode data accesses to user-mode pages.

This instruction's operation is the same in non-64-bit modes and 64-bit mode. Attempts to execute CLAC when CPL > 0 cause #UD.

Operation

```
EFLAGS.AC ← 0;
```

Flags Affected

AC cleared. Other flags are unaffected.

Protected Mode Exceptions

#UD	If the LOCK prefix is used.
	If the CPL > 0.
	If CPUID.(EAX=07H, ECX=0H):EBX.SMAP[bit 20] = 0.

Real-Address Mode Exceptions

#UD	If the LOCK prefix is used.
	If CPUID.(EAX=07H, ECX=0H):EBX.SMAP[bit 20] = 0.

Virtual-8086 Mode Exceptions

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

Compatibility Mode Exceptions

#UD	If the LOCK prefix is used.
	If the CPL > 0.
	If CPUID.(EAX=07H, ECX=0H):EBX.SMAP[bit 20] = 0.

64-Bit Mode Exceptions

#UD	If the LOCK prefix is used.
	If the CPL > 0.
	If CPUID.(EAX=07H, ECX=0H):EBX.SMAP[bit 20] = 0.

CLC—Clear Carry Flag

Opcode	Instruction	Op/ En	64-bit Mode	Compat/ Leg Mode	Description
F8	CLC	ZO	Valid	Valid	Clear CF flag.

Instruction Operand Encoding

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

Description

Clears the CF flag in the EFLAGS register. Operation is the same in all modes.

Operation

CF ← 0;

Flags Affected

The CF flag is set to 0. The OF, ZF, SF, AF, and PF flags are unaffected.

Exceptions (All Operating Modes)

#UD	If the LOCK prefix is used.

CLD—Clear Direction Flag

Opcode	Instruction	Op/ En	64-bit Mode	Compat/ Leg Mode	Description
FC	CLD	ZO	Valid	Valid	Clear DF flag.

Instruction Operand Encoding

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

Description

Clears the DF flag in the EFLAGS register. When the DF flag is set to 0, string operations increment the index registers (ESI and/or EDI). Operation is the same in all modes.

Operation

DF ← 0;

Flags Affected

The DF flag is set to 0. The CF, OF, ZF, SF, AF, and PF flags are unaffected.

Exceptions (All Operating Modes)

#UD	If the LOCK prefix is used.

CLFLUSH—Flush Cache Line

Opcode	Instruction	Op/ En	64-bit Mode	Compat/ Leg Mode	Description
NP 0F AE /7	CLFLUSH *m8*	M	Valid	Valid	Flushes cache line containing *m8*.

Instruction Operand Encoding

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

Description

Invalidates from every level of the cache hierarchy in the cache coherence domain the cache line that contains the linear address specified with the memory operand. If that cache line contains modified data at any level of the cache hierarchy, that data is written back to memory. The source operand is a byte memory location.

The availability of CLFLUSH is indicated by the presence of the CPUID feature flag CLFSH (CPUID.01H:EDX[bit 19]). The aligned cache line size affected is also indicated with the CPUID instruction (bits 8 through 15 of the EBX register when the initial value in the EAX register is 1).

The memory attribute of the page containing the affected line has no effect on the behavior of this instruction. It should be noted that processors are free to speculatively fetch and cache data from system memory regions assigned a memory-type allowing for speculative reads (such as, the WB, WC, and WT memory types). PREFETCH*h* instructions can be used to provide the processor with hints for this speculative behavior. Because this speculative fetching can occur at any time and is not tied to instruction execution, the CLFLUSH instruction is not ordered with respect to PREFETCH*h* instructions or any of the speculative fetching mechanisms (that is, data can be speculatively loaded into a cache line just before, during, or after the execution of a CLFLUSH instruction that references the cache line).

Executions of the CLFLUSH instruction are ordered with respect to each other and with respect to writes, locked read-modify-write instructions, fence instructions, and executions of CLFLUSHOPT to the same cache line.[1] They are not ordered with respect to executions of CLFLUSHOPT to different cache lines.

The CLFLUSH instruction can be used at all privilege levels and is subject to all permission checking and faults associated with a byte load (and in addition, a CLFLUSH instruction is allowed to flush a linear address in an execute-only segment). Like a load, the CLFLUSH instruction sets the A bit but not the D bit in the page tables.

In some implementations, the CLFLUSH instruction may always cause transactional abort with Transactional Synchronization Extensions (TSX). The CLFLUSH instruction is not expected to be commonly used inside typical transactional regions. However, programmers must not rely on CLFLUSH instruction to force a transactional abort, since whether they cause transactional abort is implementation dependent.

The CLFLUSH instruction was introduced with the SSE2 extensions; however, because it has its own CPUID feature flag, it can be implemented in IA-32 processors that do not include the SSE2 extensions. Also, detecting the presence of the SSE2 extensions with the CPUID instruction does not guarantee that the CLFLUSH instruction is implemented in the processor.

CLFLUSH operation is the same in non-64-bit modes and 64-bit mode.

Operation

```
Flush_Cache_Line(SRC);
```

Intel C/C++ Compiler Intrinsic Equivalents

CLFLUSH: void _mm_clflush(void const *p)

1. Earlier versions of this manual specified that executions of the CLFLUSH instruction were ordered only by the MFENCE instruction. All processors implementing the CLFLUSH instruction also order it relative to the other operations enumerated above.

Protected Mode Exceptions

#GP(0)	For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.
#SS(0)	For an illegal address in the SS segment.
#PF(fault-code)	For a page fault.
#UD	If CPUID.01H:EDX.CLFSH[bit 19] = 0.
	If the LOCK prefix is used.

Real-Address Mode Exceptions

#GP	If any part of the operand lies outside the effective address space from 0 to FFFFH.
#UD	If CPUID.01H:EDX.CLFSH[bit 19] = 0.
	If the LOCK prefix is used.

Virtual-8086 Mode Exceptions

Same exceptions as in real address mode.

#PF(fault-code)	For a page fault.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form.
#PF(fault-code)	For a page fault.
#UD	If CPUID.01H:EDX.CLFSH[bit 19] = 0.
	If the LOCK prefix is used.

CLFLUSHOPT—Flush Cache Line Optimized

Opcode	Instruction	Op/ En	64-bit Mode	Compat/ Leg Mode	Description
66 0F AE /7	CLFLUSHOPT *m8*	M	Valid	Valid	Flushes cache line containing *m8*.

Instruction Operand Encoding

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

Description

Invalidates from every level of the cache hierarchy in the cache coherence domain the cache line that contains the linear address specified with the memory operand. If that cache line contains modified data at any level of the cache hierarchy, that data is written back to memory. The source operand is a byte memory location.

The availability of CLFLUSHOPT is indicated by the presence of the CPUID feature flag CLFLUSHOPT (CPUID.(EAX=7,ECX=0):EBX[bit 23]). The aligned cache line size affected is also indicated with the CPUID instruction (bits 8 through 15 of the EBX register when the initial value in the EAX register is 1).

The memory attribute of the page containing the affected line has no effect on the behavior of this instruction. It should be noted that processors are free to speculatively fetch and cache data from system memory regions assigned a memory-type allowing for speculative reads (such as, the WB, WC, and WT memory types). PREFETCH*h* instructions can be used to provide the processor with hints for this speculative behavior. Because this speculative fetching can occur at any time and is not tied to instruction execution, the CLFLUSH instruction is not ordered with respect to PREFETCH*h* instructions or any of the speculative fetching mechanisms (that is, data can be speculatively loaded into a cache line just before, during, or after the execution of a CLFLUSH instruction that references the cache line).

Executions of the CLFLUSHOPT instruction are ordered with respect to fence instructions and to locked read-modify-write instructions; they are also ordered with respect to the following accesses to the cache line being invalidated: writes, executions of CLFLUSH, and executions of CLFLUSHOPT. They are not ordered with respect to writes, executions of CLFLUSH, or executions of CLFLUSHOPT that access other cache lines; to enforce ordering with such an operation, software can insert an SFENCE instruction between CFLUSHOPT and that operation.

The CLFLUSHOPT instruction can be used at all privilege levels and is subject to all permission checking and faults associated with a byte load (and in addition, a CLFLUSHOPT instruction is allowed to flush a linear address in an execute-only segment). Like a load, the CLFLUSHOPT instruction sets the A bit but not the D bit in the page tables.

In some implementations, the CLFLUSHOPT instruction may always cause transactional abort with Transactional Synchronization Extensions (TSX). The CLFLUSHOPT instruction is not expected to be commonly used inside typical transactional regions. However, programmers must not rely on CLFLUSHOPT instruction to force a transactional abort, since whether they cause transactional abort is implementation dependent.

CLFLUSHOPT operation is the same in non-64-bit modes and 64-bit mode.

Operation

```
Flush_Cache_Line_Optimized(SRC);
```

Intel C/C++ Compiler Intrinsic Equivalents

CLFLUSHOPT:void _mm_clflushopt(void const *p)

Protected Mode Exceptions

#GP(0)	For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.
#SS(0)	For an illegal address in the SS segment.
#PF(fault-code)	For a page fault.
#UD	If CPUID.(EAX=7,ECX=0):EBX.CLFLUSHOPT[bit 23] = 0.
	If the LOCK prefix is used.
	If an instruction prefix F2H or F3H is used.

Real-Address Mode Exceptions

#GP	If any part of the operand lies outside the effective address space from 0 to FFFFH.
#UD	If CPUID.(EAX=7,ECX=0):EBX.CLFLUSHOPT[bit 23] = 0.
	If the LOCK prefix is used.
	If an instruction prefix F2H or F3H is used.

Virtual-8086 Mode Exceptions

Same exceptions as in real address mode.

#PF(fault-code)	For a page fault.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form.
#PF(fault-code)	For a page fault.
#UD	If CPUID.(EAX=7,ECX=0):EBX.CLFLUSHOPT[bit 23] = 0.
	If the LOCK prefix is used.
	If an instruction prefix F2H or F3H is used.

CLI — Clear Interrupt Flag

Opcode	Instruction	Op/ En	64-bit Mode	Compat/ Leg Mode	Description
FA	CLI	ZO	Valid	Valid	Clear interrupt flag; interrupts disabled when interrupt flag cleared.

Instruction Operand Encoding

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

Description

In most cases, CLI clears the IF flag in the EFLAGS register and no other flags are affected. Clearing the IF flag causes the processor to ignore maskable external interrupts. The IF flag and the CLI and STI instruction have no effect on the generation of exceptions and NMI interrupts.

Operation is different in two modes defined as follows:

- **PVI mode** (protected-mode virtual interrupts): CR0.PE = 1, EFLAGS.VM = 0, CPL = 3, and CR4.PVI = 1;
- **VME mode** (virtual-8086 mode extensions): CR0.PE = 1, EFLAGS.VM = 1, and CR4.VME = 1.

If IOPL < 3 and either VME mode or PVI mode is active, CLI clears the VIF flag in the EFLAGS register, leaving IF unaffected.

Table 3-7 indicates the action of the CLI instruction depending on the processor operating mode, IOPL, and CPL.

Table 3-7. Decision Table for CLI Results

Mode	IOPL	CLI Result
Real-address	X[1]	IF = 0
Protected, not PVI[2]	≥ CPL	IF = 0
	< CPL	#GP fault
Protected, PVI[3]	3	IF = 0
	0–2	VIF = 0
Virtual-8086, not VME[3]	3	IF = 0
	0–2	#GP fault
Virtual-8086, VME[3]	3	IF = 0
	0–2	VIF = 0

NOTES:

1. X = This setting has no effect on instruction operation.
2. For this table, "protected mode" applies whenever CR0.PE = 1 and EFLAGS.VM = 0; it includes compatibility mode and 64-bit mode.
3. PVI mode and virtual-8086 mode each imply CPL = 3.

Operation

```
IF CR0.PE = 0
    THEN IF ← 0; (* Reset Interrupt Flag *)
    ELSE
        IF IOPL ≥ CPL  (* CPL = 3 if EFLAGS.VM = 1 *)
            THEN IF ← 0; (* Reset Interrupt Flag *)
            ELSE
                IF VME mode OR PVI mode
                    THEN VIF ← 0; (* Reset Virtual Interrupt Flag *)
                    ELSE #GP(0);
                FI;
        FI;
FI;
```

Flags Affected

Either the IF flag or the VIF flag is cleared to 0. Other flags are unaffected.

Protected Mode Exceptions

#GP(0)	If CPL is greater than IOPL and PVI mode is not active.
	If CPL is greater than IOPL and less than 3.
#UD	If the LOCK prefix is used.

Real-Address Mode Exceptions

#UD	If the LOCK prefix is used.

Virtual-8086 Mode Exceptions

#GP(0)	If IOPL is less than 3 and VME mode is not active.
#UD	If the LOCK prefix is used.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

Same exceptions as in protected mode.

CLTS—Clear Task-Switched Flag in CR0

Opcode	Instruction	Op/ En	64-bit Mode	Compat/ Leg Mode	Description
0F 06	CLTS	ZO	Valid	Valid	Clears TS flag in CR0.

Instruction Operand Encoding

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

Description

Clears the task-switched (TS) flag in the CR0 register. This instruction is intended for use in operating-system procedures. It is a privileged instruction that can only be executed at a CPL of 0. It is allowed to be executed in real-address mode to allow initialization for protected mode.

The processor sets the TS flag every time a task switch occurs. The flag is used to synchronize the saving of FPU context in multitasking applications. See the description of the TS flag in the section titled "Control Registers" in Chapter 2 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*, for more information about this flag.

CLTS operation is the same in non-64-bit modes and 64-bit mode.

See Chapter 25, "VMX Non-Root Operation," of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3C*, for more information about the behavior of this instruction in VMX non-root operation.

Operation

```
CR0.TS[bit 3] ← 0;
```

Flags Affected

The TS flag in CR0 register is cleared.

Protected Mode Exceptions

#GP(0)	If the current privilege level is not 0.
#UD	If the LOCK prefix is used.

Real-Address Mode Exceptions

#UD	If the LOCK prefix is used.

Virtual-8086 Mode Exceptions

#GP(0)	CLTS is not recognized in virtual-8086 mode.
#UD	If the LOCK prefix is used.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#GP(0)	If the CPL is greater than 0.
#UD	If the LOCK prefix is used.

CLWB—Cache Line Write Back

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F AE /6 CLWB m8	M	V/V	CLWB	Writes back modified cache line containing m8, and may retain the line in cache hierarchy in non-modified state.

Instruction Operand Encoding[1]

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

Description

Writes back to memory the cache line (if modified) that contains the linear address specified with the memory operand from any level of the cache hierarchy in the cache coherence domain. The line may be retained in the cache hierarchy in non-modified state. Retaining the line in the cache hierarchy is a performance optimization (treated as a hint by hardware) to reduce the possibility of cache miss on a subsequent access. Hardware may choose to retain the line at any of the levels in the cache hierarchy, and in some cases, may invalidate the line from the cache hierarchy. The source operand is a byte memory location.

The availability of CLWB instruction is indicated by the presence of the CPUID feature flag CLWB (bit 24 of the EBX register, see "CPUID — CPU Identification" in this chapter). The aligned cache line size affected is also indicated with the CPUID instruction (bits 8 through 15 of the EBX register when the initial value in the EAX register is 1).

The memory attribute of the page containing the affected line has no effect on the behavior of this instruction. It should be noted that processors are free to speculatively fetch and cache data from system memory regions that are assigned a memory-type allowing for speculative reads (such as, the WB, WC, and WT memory types). PREFETCHh instructions can be used to provide the processor with hints for this speculative behavior. Because this speculative fetching can occur at any time and is not tied to instruction execution, the CLWB instruction is not ordered with respect to PREFETCHh instructions or any of the speculative fetching mechanisms (that is, data can be speculatively loaded into a cache line just before, during, or after the execution of a CLWB instruction that references the cache line).

CLWB instruction is ordered only by store-fencing operations. For example, software can use an SFENCE, MFENCE, XCHG, or LOCK-prefixed instructions to ensure that previous stores are included in the write-back. CLWB instruction need not be ordered by another CLWB or CLFLUSHOPT instruction. CLWB is implicitly ordered with older stores executed by the logical processor to the same address.

For usages that require only writing back modified data from cache lines to memory (do not require the line to be invalidated), and expect to subsequently access the data, software is recommended to use CLWB (with appropriate fencing) instead of CLFLUSH or CLFLUSHOPT for improved performance.

The CLWB instruction can be used at all privilege levels and is subject to all permission checking and faults associated with a byte load. Like a load, the CLWB instruction sets the accessed flag but not the dirty flag in the page tables.

In some implementations, the CLWB instruction may always cause transactional abort with Transactional Synchronization Extensions (TSX). CLWB instruction is not expected to be commonly used inside typical transactional regions. However, programmers must not rely on CLWB instruction to force a transactional abort, since whether they cause transactional abort is implementation dependent.

Operation

```
Cache_Line_Write_Back(m8);
```

Flags Affected

None.

1. ModRM.MOD != 011B

C/C++ Compiler Intrinsic Equivalent

CLWB void _mm_clwb(void const *p);

Protected Mode Exceptions

#UD	If the LOCK prefix is used.
	If CPUID.(EAX=07H, ECX=0H):EBX.CLWB[bit 24] = 0.
#GP(0)	For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.
#SS(0)	For an illegal address in the SS segment.
#PF(fault-code)	For a page fault.

Real-Address Mode Exceptions

#UD	If the LOCK prefix is used.
	If CPUID.(EAX=07H, ECX=0H):EBX.CLWB[bit 24] = 0.
#GP	If any part of the operand lies outside the effective address space from 0 to FFFFH.

Virtual-8086 Mode Exceptions

Same exceptions as in real address mode.

#PF(fault-code)	For a page fault.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#UD	If the LOCK prefix is used.
	If CPUID.(EAX=07H, ECX=0H):EBX.CLWB[bit 24] = 0.
#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form.
#PF(fault-code)	For a page fault.

CMC—Complement Carry Flag

Opcode	Instruction	Op/ En	64-bit Mode	Compat/ Leg Mode	Description
F5	CMC	ZO	Valid	Valid	Complement CF flag.

Instruction Operand Encoding

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

Description

Complements the CF flag in the EFLAGS register. CMC operation is the same in non-64-bit modes and 64-bit mode.

Operation

EFLAGS.CF[bit 0]← NOT EFLAGS.CF[bit 0];

Flags Affected

The CF flag contains the complement of its original value. The OF, ZF, SF, AF, and PF flags are unaffected.

Exceptions (All Operating Modes)

#UD	If the LOCK prefix is used.

CMOV*cc*—Conditional Move

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
0F 47 */r*	CMOVA *r16, r/m16*	RM	Valid	Valid	Move if above (CF=0 and ZF=0).
0F 47 */r*	CMOVA *r32, r/m32*	RM	Valid	Valid	Move if above (CF=0 and ZF=0).
REX.W + 0F 47 */r*	CMOVA *r64, r/m64*	RM	Valid	N.E.	Move if above (CF=0 and ZF=0).
0F 43 */r*	CMOVAE *r16, r/m16*	RM	Valid	Valid	Move if above or equal (CF=0).
0F 43 */r*	CMOVAE *r32, r/m32*	RM	Valid	Valid	Move if above or equal (CF=0).
REX.W + 0F 43 */r*	CMOVAE *r64, r/m64*	RM	Valid	N.E.	Move if above or equal (CF=0).
0F 42 */r*	CMOVB *r16, r/m16*	RM	Valid	Valid	Move if below (CF=1).
0F 42 */r*	CMOVB *r32, r/m32*	RM	Valid	Valid	Move if below (CF=1).
REX.W + 0F 42 */r*	CMOVB *r64, r/m64*	RM	Valid	N.E.	Move if below (CF=1).
0F 46 */r*	CMOVBE *r16, r/m16*	RM	Valid	Valid	Move if below or equal (CF=1 or ZF=1).
0F 46 */r*	CMOVBE *r32, r/m32*	RM	Valid	Valid	Move if below or equal (CF=1 or ZF=1).
REX.W + 0F 46 */r*	CMOVBE *r64, r/m64*	RM	Valid	N.E.	Move if below or equal (CF=1 or ZF=1).
0F 42 */r*	CMOVC *r16, r/m16*	RM	Valid	Valid	Move if carry (CF=1).
0F 42 */r*	CMOVC *r32, r/m32*	RM	Valid	Valid	Move if carry (CF=1).
REX.W + 0F 42 */r*	CMOVC *r64, r/m64*	RM	Valid	N.E.	Move if carry (CF=1).
0F 44 */r*	CMOVE *r16, r/m16*	RM	Valid	Valid	Move if equal (ZF=1).
0F 44 */r*	CMOVE *r32, r/m32*	RM	Valid	Valid	Move if equal (ZF=1).
REX.W + 0F 44 */r*	CMOVE *r64, r/m64*	RM	Valid	N.E.	Move if equal (ZF=1).
0F 4F */r*	CMOVG *r16, r/m16*	RM	Valid	Valid	Move if greater (ZF=0 and SF=OF).
0F 4F */r*	CMOVG *r32, r/m32*	RM	Valid	Valid	Move if greater (ZF=0 and SF=OF).
REX.W + 0F 4F */r*	CMOVG *r64, r/m64*	RM	V/N.E.	NA	Move if greater (ZF=0 and SF=OF).
0F 4D */r*	CMOVGE *r16, r/m16*	RM	Valid	Valid	Move if greater or equal (SF=OF).
0F 4D */r*	CMOVGE *r32, r/m32*	RM	Valid	Valid	Move if greater or equal (SF=OF).
REX.W + 0F 4D */r*	CMOVGE *r64, r/m64*	RM	Valid	N.E.	Move if greater or equal (SF=OF).
0F 4C */r*	CMOVL *r16, r/m16*	RM	Valid	Valid	Move if less (SF≠ OF).
0F 4C */r*	CMOVL *r32, r/m32*	RM	Valid	Valid	Move if less (SF≠ OF).
REX.W + 0F 4C */r*	CMOVL *r64, r/m64*	RM	Valid	N.E.	Move if less (SF≠ OF).
0F 4E */r*	CMOVLE *r16, r/m16*	RM	Valid	Valid	Move if less or equal (ZF=1 or SF≠ OF).
0F 4E */r*	CMOVLE *r32, r/m32*	RM	Valid	Valid	Move if less or equal (ZF=1 or SF≠ OF).
REX.W + 0F 4E */r*	CMOVLE *r64, r/m64*	RM	Valid	N.E.	Move if less or equal (ZF=1 or SF≠ OF).
0F 46 */r*	CMOVNA *r16, r/m16*	RM	Valid	Valid	Move if not above (CF=1 or ZF=1).
0F 46 */r*	CMOVNA *r32, r/m32*	RM	Valid	Valid	Move if not above (CF=1 or ZF=1).
REX.W + 0F 46 */r*	CMOVNA *r64, r/m64*	RM	Valid	N.E.	Move if not above (CF=1 or ZF=1).
0F 42 */r*	CMOVNAE *r16, r/m16*	RM	Valid	Valid	Move if not above or equal (CF=1).
0F 42 */r*	CMOVNAE *r32, r/m32*	RM	Valid	Valid	Move if not above or equal (CF=1).
REX.W + 0F 42 */r*	CMOVNAE *r64, r/m64*	RM	Valid	N.E.	Move if not above or equal (CF=1).
0F 43 */r*	CMOVNB *r16, r/m16*	RM	Valid	Valid	Move if not below (CF=0).
0F 43 */r*	CMOVNB *r32, r/m32*	RM	Valid	Valid	Move if not below (CF=0).
REX.W + 0F 43 */r*	CMOVNB *r64, r/m64*	RM	Valid	N.E.	Move if not below (CF=0).
0F 47 */r*	CMOVNBE *r16, r/m16*	RM	Valid	Valid	Move if not below or equal (CF=0 and ZF=0).

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
0F 47 /r	CMOVNBE *r32, r/m32*	RM	Valid	Valid	Move if not below or equal (CF=0 and ZF=0).
REX.W + 0F 47 /r	CMOVNBE *r64, r/m64*	RM	Valid	N.E.	Move if not below or equal (CF=0 and ZF=0).
0F 43 /r	CMOVNC *r16, r/m16*	RM	Valid	Valid	Move if not carry (CF=0).
0F 43 /r	CMOVNC *r32, r/m32*	RM	Valid	Valid	Move if not carry (CF=0).
REX.W + 0F 43 /r	CMOVNC *r64, r/m64*	RM	Valid	N.E.	Move if not carry (CF=0).
0F 45 /r	CMOVNE *r16, r/m16*	RM	Valid	Valid	Move if not equal (ZF=0).
0F 45 /r	CMOVNE *r32, r/m32*	RM	Valid	Valid	Move if not equal (ZF=0).
REX.W + 0F 45 /r	CMOVNE *r64, r/m64*	RM	Valid	N.E.	Move if not equal (ZF=0).
0F 4E /r	CMOVNG *r16, r/m16*	RM	Valid	Valid	Move if not greater (ZF=1 or SF≠ OF).
0F 4E /r	CMOVNG *r32, r/m32*	RM	Valid	Valid	Move if not greater (ZF=1 or SF≠ OF).
REX.W + 0F 4E /r	CMOVNG *r64, r/m64*	RM	Valid	N.E.	Move if not greater (ZF=1 or SF≠ OF).
0F 4C /r	CMOVNGE *r16, r/m16*	RM	Valid	Valid	Move if not greater or equal (SF≠ OF).
0F 4C /r	CMOVNGE *r32, r/m32*	RM	Valid	Valid	Move if not greater or equal (SF≠ OF).
REX.W + 0F 4C /r	CMOVNGE *r64, r/m64*	RM	Valid	N.E.	Move if not greater or equal (SF≠ OF).
0F 4D /r	CMOVNL *r16, r/m16*	RM	Valid	Valid	Move if not less (SF=OF).
0F 4D /r	CMOVNL *r32, r/m32*	RM	Valid	Valid	Move if not less (SF=OF).
REX.W + 0F 4D /r	CMOVNL *r64, r/m64*	RM	Valid	N.E.	Move if not less (SF=OF).
0F 4F /r	CMOVNLE *r16, r/m16*	RM	Valid	Valid	Move if not less or equal (ZF=0 and SF=OF).
0F 4F /r	CMOVNLE *r32, r/m32*	RM	Valid	Valid	Move if not less or equal (ZF=0 and SF=OF).
REX.W + 0F 4F /r	CMOVNLE *r64, r/m64*	RM	Valid	N.E.	Move if not less or equal (ZF=0 and SF=OF).
0F 41 /r	CMOVNO *r16, r/m16*	RM	Valid	Valid	Move if not overflow (OF=0).
0F 41 /r	CMOVNO *r32, r/m32*	RM	Valid	Valid	Move if not overflow (OF=0).
REX.W + 0F 41 /r	CMOVNO *r64, r/m64*	RM	Valid	N.E.	Move if not overflow (OF=0).
0F 4B /r	CMOVNP *r16, r/m16*	RM	Valid	Valid	Move if not parity (PF=0).
0F 4B /r	CMOVNP *r32, r/m32*	RM	Valid	Valid	Move if not parity (PF=0).
REX.W + 0F 4B /r	CMOVNP *r64, r/m64*	RM	Valid	N.E.	Move if not parity (PF=0).
0F 49 /r	CMOVNS *r16, r/m16*	RM	Valid	Valid	Move if not sign (SF=0).
0F 49 /r	CMOVNS *r32, r/m32*	RM	Valid	Valid	Move if not sign (SF=0).
REX.W + 0F 49 /r	CMOVNS *r64, r/m64*	RM	Valid	N.E.	Move if not sign (SF=0).
0F 45 /r	CMOVNZ *r16, r/m16*	RM	Valid	Valid	Move if not zero (ZF=0).
0F 45 /r	CMOVNZ *r32, r/m32*	RM	Valid	Valid	Move if not zero (ZF=0).
REX.W + 0F 45 /r	CMOVNZ *r64, r/m64*	RM	Valid	N.E.	Move if not zero (ZF=0).
0F 40 /r	CMOVO *r16, r/m16*	RM	Valid	Valid	Move if overflow (OF=1).
0F 40 /r	CMOVO *r32, r/m32*	RM	Valid	Valid	Move if overflow (OF=1).
REX.W + 0F 40 /r	CMOVO *r64, r/m64*	RM	Valid	N.E.	Move if overflow (OF=1).
0F 4A /r	CMOVP *r16, r/m16*	RM	Valid	Valid	Move if parity (PF=1).
0F 4A /r	CMOVP *r32, r/m32*	RM	Valid	Valid	Move if parity (PF=1).
REX.W + 0F 4A /r	CMOVP *r64, r/m64*	RM	Valid	N.E.	Move if parity (PF=1).
0F 4A /r	CMOVPE *r16, r/m16*	RM	Valid	Valid	Move if parity even (PF=1).
0F 4A /r	CMOVPE *r32, r/m32*	RM	Valid	Valid	Move if parity even (PF=1).
REX.W + 0F 4A /r	CMOVPE *r64, r/m64*	RM	Valid	N.E.	Move if parity even (PF=1).

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
0F 4B /r	CMOVPO *r16, r/m16*	RM	Valid	Valid	Move if parity odd (PF=0).
0F 4B /r	CMOVPO *r32, r/m32*	RM	Valid	Valid	Move if parity odd (PF=0).
REX.W + 0F 4B /r	CMOVPO *r64, r/m64*	RM	Valid	N.E.	Move if parity odd (PF=0).
0F 48 /r	CMOVS *r16, r/m16*	RM	Valid	Valid	Move if sign (SF=1).
0F 48 /r	CMOVS *r32, r/m32*	RM	Valid	Valid	Move if sign (SF=1).
REX.W + 0F 48 /r	CMOVS *r64, r/m64*	RM	Valid	N.E.	Move if sign (SF=1).
0F 44 /r	CMOVZ *r16, r/m16*	RM	Valid	Valid	Move if zero (ZF=1).
0F 44 /r	CMOVZ *r32, r/m32*	RM	Valid	Valid	Move if zero (ZF=1).
REX.W + 0F 44 /r	CMOVZ *r64, r/m64*	RM	Valid	N.E.	Move if zero (ZF=1).

Instruction Operand Encoding

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

Description

The CMOV*cc* instructions check the state of one or more of the status flags in the EFLAGS register (CF, OF, PF, SF, and ZF) and perform a move operation if the flags are in a specified state (or condition). A condition code (*cc*) is associated with each instruction to indicate the condition being tested for. If the condition is not satisfied, a move is not performed and execution continues with the instruction following the CMOV*cc* instruction.

These instructions can move 16-bit, 32-bit or 64-bit values from memory to a general-purpose register or from one general-purpose register to another. Conditional moves of 8-bit register operands are not supported.

The condition for each CMOV*cc* mnemonic is given in the description column of the above table. The terms "less" and "greater" are used for comparisons of signed integers and the terms "above" and "below" are used for unsigned integers.

Because a particular state of the status flags can sometimes be interpreted in two ways, two mnemonics are defined for some opcodes. For example, the CMOVA (conditional move if above) instruction and the CMOVNBE (conditional move if not below or equal) instruction are alternate mnemonics for the opcode 0F 47H.

The CMOV*cc* instructions were introduced in P6 family processors; however, these instructions may not be supported by all IA-32 processors. Software can determine if the CMOV*cc* instructions are supported by checking the processor's feature information with the CPUID instruction (see "CPUID—CPU Identification" in this chapter).

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

Operation

```
temp ← SRC

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

Flags Affected

None.

Protected Mode Exceptions

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

Real-Address Mode Exceptions

#GP	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS	If a memory operand effective address is outside the SS segment limit.
#UD	If the LOCK prefix is used.

Virtual-8086 Mode Exceptions

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

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If the LOCK prefix is used.

CMP—Compare Two Operands

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
3C *ib*	CMP AL, *imm8*	I	Valid	Valid	Compare *imm8* with AL.
3D *iw*	CMP AX, *imm16*	I	Valid	Valid	Compare *imm16* with AX.
3D *id*	CMP EAX, *imm32*	I	Valid	Valid	Compare *imm32* with EAX.
REX.W + 3D *id*	CMP RAX, *imm32*	I	Valid	N.E.	Compare *imm32 sign-extended to 64-bits* with RAX.
80 /7 *ib*	CMP *r/m8, imm8*	MI	Valid	Valid	Compare *imm8* with *r/m8.*
REX + 80 /7 *ib*	CMP *r/m8**, *imm8*	MI	Valid	N.E.	Compare *imm8* with *r/m8.*
81 /7 *iw*	CMP *r/m16, imm16*	MI	Valid	Valid	Compare *imm16* with *r/m16.*
81 /7 *id*	CMP *r/m32, imm32*	MI	Valid	Valid	Compare *imm32* with *r/m32.*
REX.W + 81 /7 *id*	CMP *r/m64, imm32*	MI	Valid	N.E.	Compare *imm32 sign-extended to 64-bits* with *r/m64.*
83 /7 *ib*	CMP *r/m16, imm8*	MI	Valid	Valid	Compare *imm8* with *r/m16.*
83 /7 *ib*	CMP *r/m32, imm8*	MI	Valid	Valid	Compare *imm8* with *r/m32.*
REX.W + 83 /7 *ib*	CMP *r/m64, imm8*	MI	Valid	N.E.	Compare *imm8* with *r/m64.*
38 /*r*	CMP *r/m8, r8*	MR	Valid	Valid	Compare *r8* with *r/m8.*
REX + 38 /*r*	CMP *r/m8**, *r8**	MR	Valid	N.E.	Compare *r8* with *r/m8.*
39 /*r*	CMP *r/m16, r16*	MR	Valid	Valid	Compare *r16* with *r/m16.*
39 /*r*	CMP *r/m32, r32*	MR	Valid	Valid	Compare *r32* with *r/m32.*
REX.W + 39 /*r*	CMP *r/m64,r64*	MR	Valid	N.E.	Compare *r64* with *r/m64.*
3A /*r*	CMP *r8, r/m8*	RM	Valid	Valid	Compare *r/m8* with *r8.*
REX + 3A /*r*	CMP *r8**, *r/m8**	RM	Valid	N.E.	Compare *r/m8 with r8.*
3B /*r*	CMP *r16, r/m16*	RM	Valid	Valid	Compare *r/m16* with *r16.*
3B /*r*	CMP *r32, r/m32*	RM	Valid	Valid	Compare *r/m32* with *r32.*
REX.W + 3B /*r*	CMP *r64, r/m64*	RM	Valid	N.E.	Compare *r/m64* with *r64.*

NOTES:

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

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r)	ModRM:r/m (r)	NA	NA
MR	ModRM:r/m (r)	ModRM:reg (r)	NA	NA
MI	ModRM:r/m (r)	imm8	NA	NA
I	AL/AX/EAX/RAX (r)	imm8	NA	NA

Description

Compares the first source operand with the second source operand and sets the status flags in the EFLAGS register according to the results. The comparison is performed by subtracting the second operand from the first operand and then setting the status flags in the same manner as the SUB instruction. When an immediate value is used as an operand, it is sign-extended to the length of the first operand.

The condition codes used by the J*cc*, CMOV*cc*, and SET*cc* instructions are based on the results of a CMP instruction. Appendix B, "EFLAGS Condition Codes," in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*, shows the relationship of the status flags and the condition codes.

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

Operation

```
temp ← SRC1 – SignExtend(SRC2);
ModifyStatusFlags; (* Modify status flags in the same manner as the SUB instruction*)
```

Flags Affected

The CF, OF, SF, ZF, AF, and PF flags are set according to the result.

Protected Mode Exceptions

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

Real-Address Mode Exceptions

#GP	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS	If a memory operand effective address is outside the SS segment limit.

Virtual-8086 Mode Exceptions

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

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If the LOCK prefix is used.

CMPPD—Compare Packed Double-Precision Floating-Point Values

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F C2 /r ib CMPPD xmm1, xmm2/m128, imm8	A	V/V	SSE2	Compare packed double-precision floating-point values in xmm2/m128 and xmm1 using bits 2:0 of imm8 as a comparison predicate.
VEX.NDS.128.66.0F.WIG C2 /r ib VCMPPD xmm1, xmm2, xmm3/m128, imm8	B	V/V	AVX	Compare packed double-precision floating-point values in xmm3/m128 and xmm2 using bits 4:0 of imm8 as a comparison predicate.
VEX.NDS.256.66.0F.WIG C2 /r ib VCMPPD ymm1, ymm2, ymm3/m256, imm8	B	V/V	AVX	Compare packed double-precision floating-point values in ymm3/m256 and ymm2 using bits 4:0 of imm8 as a comparison predicate.
EVEX.NDS.128.66.0F.W1 C2 /r ib VCMPPD k1 {k2}, xmm2, xmm3/m128/m64bcst, imm8	C	V/V	AVX512VL AVX512F	Compare packed double-precision floating-point values in xmm3/m128/m64bcst and xmm2 using bits 4:0 of imm8 as a comparison predicate with writemask k2 and leave the result in mask register k1.
EVEX.NDS.256.66.0F.W1 C2 /r ib VCMPPD k1 {k2}, ymm2, ymm3/m256/m64bcst, imm8	C	V/V	AVX512VL AVX512F	Compare packed double-precision floating-point values in ymm3/m256/m64bcst and ymm2 using bits 4:0 of imm8 as a comparison predicate with writemask k2 and leave the result in mask register k1.
EVEX.NDS.512.66.0F.W1 C2 /r ib VCMPPD k1 {k2}, zmm2, zmm3/m512/m64bcst{sae}, imm8	C	V/V	AVX512F	Compare packed double-precision floating-point values in zmm3/m512/m64bcst and zmm2 using bits 4:0 of imm8 as a comparison predicate with writemask k2 and leave the result in mask register k1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	NA	ModRM:reg (r, w)	ModRM:r/m (r)	Imm8	NA
B	NA	ModRM:reg (w)	VEX.vvvv	ModRM:r/m (r)	Imm8
C	Full Vector	ModRM:reg (w)	EVEX.vvvv	ModRM:r/m (r)	Imm8

Description

Performs a SIMD compare of the packed double-precision floating-point values in the second source operand and the first source operand and returns the results of the comparison to the destination operand. The comparison predicate operand (immediate byte) specifies the type of comparison performed on each pair of packed values in the two source operands.

EVEX encoded versions: The first source operand (second operand) is a ZMM/YMM/XMM register. The second source operand can be a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 64-bit memory location. The destination operand (first operand) is an opmask register. Comparison results are written to the destination operand under the writemask k2. Each comparison result is a single mask bit of 1 (comparison true) or 0 (comparison false).

VEX.256 encoded version: The first source operand (second operand) is a YMM register. The second source operand (third operand) can be a YMM register or a 256-bit memory location. The destination operand (first operand) is a YMM register. Four comparisons are performed with results written to the destination operand. The result of each comparison is a quadword mask of all 1s (comparison true) or all 0s (comparison false).

128-bit Legacy SSE version: The first source and destination operand (first operand) is an XMM register. The second source operand (second operand) can be an XMM register or 128-bit memory location. Bits (MAXVL-1:128) of the corresponding ZMM destination register remain unchanged. Two comparisons are performed with results written to bits 127:0 of the destination operand. The result of each comparison is a quadword mask of all 1s (comparison true) or all 0s (comparison false).

VEX.128 encoded version: The first source operand (second operand) is an XMM register. The second source operand (third operand) can be an XMM register or a 128-bit memory location. Bits (MAXVL-1:128) of the destination ZMM register are zeroed. Two comparisons are performed with results written to bits 127:0 of the destination operand.

The comparison predicate operand is an 8-bit immediate:

- For instructions encoded using the VEX or EVEX prefix, bits 4:0 define the type of comparison to be performed (see Table 3-1). Bits 5 through 7 of the immediate are reserved.
- For instruction encodings that do not use VEX prefix, bits 2:0 define the type of comparison to be made (see the first 8 rows of Table 3-1). Bits 3 through 7 of the immediate are reserved.

Table 3-1. Comparison Predicate for CMPPD and CMPPS Instructions

Predicate	imm8 Value	Description	Result: A Is 1st Operand, B Is 2nd Operand				Signals #IA on QNAN
			A >B	A < B	A = B	Unordered[1]	
EQ_OQ (EQ)	0H	Equal (ordered, non-signaling)	False	False	True	False	No
LT_OS (LT)	1H	Less-than (ordered, signaling)	False	True	False	False	Yes
LE_OS (LE)	2H	Less-than-or-equal (ordered, signaling)	False	True	True	False	Yes
UNORD_Q (UNORD)	3H	Unordered (non-signaling)	False	False	False	True	No
NEQ_UQ (NEQ)	4H	Not-equal (unordered, non-signaling)	True	True	False	True	No
NLT_US (NLT)	5H	Not-less-than (unordered, signaling)	True	False	True	True	Yes
NLE_US (NLE)	6H	Not-less-than-or-equal (unordered, signaling)	True	False	False	True	Yes
ORD_Q (ORD)	7H	Ordered (non-signaling)	True	True	True	False	No
EQ_UQ	8H	Equal (unordered, non-signaling)	False	False	True	True	No
NGE_US (NGE)	9H	Not-greater-than-or-equal (unordered, signaling)	False	True	False	True	Yes
NGT_US (NGT)	AH	Not-greater-than (unordered, signaling)	False	True	True	True	Yes
FALSE_OQ(FALSE)	BH	False (ordered, non-signaling)	False	False	False	False	No
NEQ_OQ	CH	Not-equal (ordered, non-signaling)	True	True	False	False	No
GE_OS (GE)	DH	Greater-than-or-equal (ordered, signaling)	True	False	True	False	Yes
GT_OS (GT)	EH	Greater-than (ordered, signaling)	True	False	False	False	Yes
TRUE_UQ(TRUE)	FH	True (unordered, non-signaling)	True	True	True	True	No
EQ_OS	10H	Equal (ordered, signaling)	False	False	True	False	Yes
LT_OQ	11H	Less-than (ordered, nonsignaling)	False	True	False	False	No
LE_OQ	12H	Less-than-or-equal (ordered, nonsignaling)	False	True	True	False	No
UNORD_S	13H	Unordered (signaling)	False	False	False	True	Yes
NEQ_US	14H	Not-equal (unordered, signaling)	True	True	False	True	Yes
NLT_UQ	15H	Not-less-than (unordered, nonsignaling)	True	False	True	True	No
NLE_UQ	16H	Not-less-than-or-equal (unordered, nonsignaling)	True	False	False	True	No
ORD_S	17H	Ordered (signaling)	True	True	True	False	Yes
EQ_US	18H	Equal (unordered, signaling)	False	False	True	True	Yes
NGE_UQ	19H	Not-greater-than-or-equal (unordered, non-signaling)	False	True	False	True	No

Table 3-1. Comparison Predicate for CMPPD and CMPPS Instructions (Contd.)

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

NOTES:

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

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

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

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

Compilers and assemblers may implement the following two-operand pseudo-ops in addition to the three-operand CMPPD instruction, for processors with "CPUID.1H:ECX.AVX =0". See Table 3-2. Compiler should treat reserved Imm8 values as illegal syntax.

Table 3-2. Pseudo-Op and CMPPD Implementation

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

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

Processors with "CPUID.1H:ECX.AVX =1" implement the full complement of 32 predicates shown in Table 3-3, software emulation is no longer needed. Compilers and assemblers may implement the following three-operand pseudo-ops in addition to the four-operand VCMPPD instruction. See Table 3-3, where the notations of reg1 reg2, and reg3 represent either XMM registers or YMM registers. Compiler should treat reserved Imm8 values as illegal

syntax. Alternately, intrinsics can map the pseudo-ops to pre-defined constants to support a simpler intrinsic interface. Compilers and assemblers may implement three-operand pseudo-ops for EVEX encoded VCMPPD instructions in a similar fashion by extending the syntax listed in Table 3-3.

Table 3-3. Pseudo-Op and VCMPPD Implementation

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

Operation

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

VCMPPD (EVEX encoded versions)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j ← 0 TO KL-1
    i ← j * 64
    IF k2[j] OR *no writemask*
        THEN
            IF (EVEX.b = 1) AND (SRC2 *is memory*)
                THEN
                    CMP ← SRC1[i+63:i] OP5 SRC2[63:0]
                ELSE
                    CMP ← SRC1[i+63:i] OP5 SRC2[i+63:i]
            FI;
            IF CMP = TRUE
                THEN DEST[j] ← 1;
                ELSE DEST[j] ← 0; FI;
        ELSE    DEST[j] ← 0             ; zeroing-masking only
    FI;
ENDFOR
DEST[MAX_KL-1:KL] ← 0
```

VCMPPD (VEX.256 encoded version)

```
CMP0 ← SRC1[63:0] OP5 SRC2[63:0];
CMP1 ← SRC1[127:64] OP5 SRC2[127:64];
CMP2 ← SRC1[191:128] OP5 SRC2[191:128];
CMP3 ← SRC1[255:192] OP5 SRC2[255:192];
IF CMP0 = TRUE
    THEN DEST[63:0] ← FFFFFFFFFFFFFFFFH;
    ELSE DEST[63:0] ← 0000000000000000H; FI;
IF CMP1 = TRUE
    THEN DEST[127:64] ← FFFFFFFFFFFFFFFFH;
    ELSE DEST[127:64] ← 0000000000000000H; FI;
IF CMP2 = TRUE
    THEN DEST[191:128] ← FFFFFFFFFFFFFFFFH;
    ELSE DEST[191:128] ← 0000000000000000H; FI;
IF CMP3 = TRUE
    THEN DEST[255:192] ← FFFFFFFFFFFFFFFFH;
    ELSE DEST[255:192] ← 0000000000000000H; FI;
DEST[MAXVL-1:256] ← 0
```

VCMPPD (VEX.128 encoded version)

```
CMP0 ← SRC1[63:0] OP5 SRC2[63:0];
CMP1 ← SRC1[127:64] OP5 SRC2[127:64];
IF CMP0 = TRUE
    THEN DEST[63:0] ← FFFFFFFFFFFFFFFFH;
    ELSE DEST[63:0] ← 0000000000000000H; FI;
IF CMP1 = TRUE
    THEN DEST[127:64] ← FFFFFFFFFFFFFFFFH;
    ELSE DEST[127:64] ← 0000000000000000H; FI;
DEST[MAXVL-1:128] ← 0
```

CMPPD (128-bit Legacy SSE version)

```
CMP0 ← SRC1[63:0] OP3 SRC2[63:0];
CMP1 ← SRC1[127:64] OP3 SRC2[127:64];
IF CMP0 = TRUE
    THEN DEST[63:0] ← FFFFFFFFFFFFFFFFH;
    ELSE DEST[63:0] ← 0000000000000000H; FI;
IF CMP1 = TRUE
    THEN DEST[127:64] ← FFFFFFFFFFFFFFFFH;
    ELSE DEST[127:64] ← 0000000000000000H; FI;
DEST[MAXVL-1:128] (Unmodified)
```

Intel C/C++ Compiler Intrinsic Equivalent

VCMPPD __mmask8 _mm512_cmp_pd_mask(__m512d a, __m512d b, int imm);
VCMPPD __mmask8 _mm512_cmp_round_pd_mask(__m512d a, __m512d b, int imm, int sae);
VCMPPD __mmask8 _mm512_mask_cmp_pd_mask(__mmask8 k1, __m512d a, __m512d b, int imm);
VCMPPD __mmask8 _mm512_mask_cmp_round_pd_mask(__mmask8 k1, __m512d a, __m512d b, int imm, int sae);
VCMPPD __mmask8 _mm256_cmp_pd_mask(__m256d a, __m256d b, int imm);
VCMPPD __mmask8 _mm256_mask_cmp_pd_mask(__mmask8 k1, __m256d a, __m256d b, int imm);
VCMPPD __mmask8 _mm_cmp_pd_mask(__m128d a, __m128d b, int imm);
VCMPPD __mmask8 _mm_mask_cmp_pd_mask(__mmask8 k1, __m128d a, __m128d b, int imm);
VCMPPD __m256 _mm256_cmp_pd(__m256d a, __m256d b, int imm)
(V)CMPPD __m128 _mm_cmp_pd(__m128d a, __m128d b, int imm)

SIMD Floating-Point Exceptions

Invalid if SNaN operand and invalid if QNaN and predicate as listed in Table 3-1.

Denormal

Other Exceptions

VEX-encoded instructions, see Exceptions Type 2.

EVEX-encoded instructions, see Exceptions Type E2.

CMPPS—Compare Packed Single-Precision Floating-Point Values

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F C2 /r ib CMPPS xmm1, xmm2/m128, imm8	A	V/V	SSE	Compare packed single-precision floating-point values in xmm2/m128 and xmm1 using bits 2:0 of imm8 as a comparison predicate.
VEX.NDS.128.0F.WIG C2 /r ib VCMPPS xmm1, xmm2, xmm3/m128, imm8	B	V/V	AVX	Compare packed single-precision floating-point values in xmm3/m128 and xmm2 using bits 4:0 of imm8 as a comparison predicate.
VEX.NDS.256.0F.WIG C2 /r ib VCMPPS ymm1, ymm2, ymm3/m256, imm8	B	V/V	AVX	Compare packed single-precision floating-point values in ymm3/m256 and ymm2 using bits 4:0 of imm8 as a comparison predicate.
EVEX.NDS.128.0F.W0 C2 /r ib VCMPPS k1 {k2}, xmm2, xmm3/m128/m32bcst, imm8	C	V/V	AVX512VL AVX512F	Compare packed single-precision floating-point values in xmm3/m128/m32bcst and xmm2 using bits 4:0 of imm8 as a comparison predicate with writemask k2 and leave the result in mask register k1.
EVEX.NDS.256.0F.W0 C2 /r ib VCMPPS k1 {k2}, ymm2, ymm3/m256/m32bcst, imm8	C	V/V	AVX512VL AVX512F	Compare packed single-precision floating-point values in ymm3/m256/m32bcst and ymm2 using bits 4:0 of imm8 as a comparison predicate with writemask k2 and leave the result in mask register k1.
EVEX.NDS.512.0F.W0 C2 /r ib VCMPPS k1 {k2}, zmm2, zmm3/m512/m32bcst{sae}, imm8	C	V/V	AVX512F	Compare packed single-precision floating-point values in zmm3/m512/m32bcst and zmm2 using bits 4:0 of imm8 as a comparison predicate with writemask k2 and leave the result in mask register k1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	NA	ModRM:reg (r, w)	ModRM:r/m (r)	Imm8	NA
B	NA	ModRM:reg (w)	VEX.vvvv	ModRM:r/m (r)	Imm8
C	Full Vector	ModRM:reg (w)	EVEX.vvvv	ModRM:r/m (r)	Imm8

Description

Performs a SIMD compare of the packed single-precision floating-point values in the second source operand and the first source operand and returns the results of the comparison to the destination operand. The comparison predicate operand (immediate byte) specifies the type of comparison performed on each of the pairs of packed values.

EVEX encoded versions: The first source operand (second operand) is a ZMM/YMM/XMM register. The second source operand can be a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 32-bit memory location. The destination operand (first operand) is an opmask register. Comparison results are written to the destination operand under the writemask k2. Each comparison result is a single mask bit of 1 (comparison true) or 0 (comparison false).

VEX.256 encoded version: The first source operand (second operand) is a YMM register. The second source operand (third operand) can be a YMM register or a 256-bit memory location. The destination operand (first operand) is a YMM register. Eight comparisons are performed with results written to the destination operand. The result of each comparison is a doubleword mask of all 1s (comparison true) or all 0s (comparison false).

128-bit Legacy SSE version: The first source and destination operand (first operand) is an XMM register. The second source operand (second operand) can be an XMM register or 128-bit memory location. Bits (MAXVL-1:128) of the corresponding ZMM destination register remain unchanged. Four comparisons are performed with results written to bits 127:0 of the destination operand. The result of each comparison is a doubleword mask of all 1s (comparison true) or all 0s (comparison false).

VEX.128 encoded version: The first source operand (second operand) is an XMM register. The second source operand (third operand) can be an XMM register or a 128-bit memory location. Bits (MAXVL-1:128) of the destination ZMM register are zeroed. Four comparisons are performed with results written to bits 127:0 of the destination operand.

The comparison predicate operand is an 8-bit immediate:

- For instructions encoded using the VEX prefix and EVEX prefix, bits 4:0 define the type of comparison to be performed (see Table 3-1). Bits 5 through 7 of the immediate are reserved.
- For instruction encodings that do not use VEX prefix, bits 2:0 define the type of comparison to be made (see the first 8 rows of Table 3-1). Bits 3 through 7 of the immediate are reserved.

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

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

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

Compilers and assemblers may implement the following two-operand pseudo-ops in addition to the three-operand CMPPS instruction, for processors with "CPUID.1H:ECX.AVX =0". See Table 3-4. Compiler should treat reserved Imm8 values as illegal syntax.

Table 3-4. Pseudo-Op and CMPPS Implementation

Pseudo-Op	CMPPS Implementation
CMPEQPS x*mm1, xmm2*	CMPPS *xmm1, xmm2, 0*
CMPLTPS x*mm1, xmm2*	CMPPS *xmm1, xmm2, 1*
CMPLEPS x*mm1, xmm2*	CMPPS *xmm1, xmm2, 2*
CMPUNORDPS x*mm1, xmm2*	CMPPS *xmm1, xmm2, 3*
CMPNEQPS x*mm1, xmm2*	CMPPS *xmm1, xmm2, 4*
CMPNLTPS x*mm1, xmm2*	CMPPS *xmm1, xmm2, 5*
CMPNLEPS x*mm1, xmm2*	CMPPS *xmm1, xmm2, 6*
CMPORDPS x*mm1, xmm2*	CMPPS *xmm1, xmm2, 7*

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

Processors with "CPUID.1H:ECX.AVX =1" implement the full complement of 32 predicates shown in Table 3-5, software emulation is no longer needed. Compilers and assemblers may implement the following three-operand pseudo-ops in addition to the four-operand VCMPPS instruction. See Table 3-5, where the notation of reg1 and reg2 represent either XMM registers or YMM registers. Compiler should treat reserved Imm8 values as illegal syntax. Alternately, intrinsics can map the pseudo-ops to pre-defined constants to support a simpler intrinsic interface. Compilers and assemblers may implement three-operand pseudo-ops for EVEX encoded VCMPPS instructions in a similar fashion by extending the syntax listed in Table 3-5.

Table 3-5. Pseudo-Op and VCMPPS Implementation

Pseudo-Op	CMPPS Implementation
VCMPEQPS *reg1, reg2, reg3*	VCMPPS *reg1, reg2, reg3, 0*
VCMPLTPS *reg1, reg2, reg3*	VCMPPS *reg1, reg2, reg3, 1*
VCMPLEPS *reg1, reg2, reg3*	VCMPPS *reg1, reg2, reg3, 2*
VCMPUNORDPS *reg1, reg2, reg3*	VCMPPS *reg1, reg2, reg3, 3*
VCMPNEQPS *reg1, reg2, reg3*	VCMPPS *reg1, reg2, reg3, 4*
VCMPNLTPS *reg1, reg2, reg3*	VCMPPS *reg1, reg2, reg3, 5*
VCMPNLEPS *reg1, reg2, reg3*	VCMPPS *reg1, reg2, reg3, 6*
VCMPORDPS *reg1, reg2, reg3*	VCMPPS *reg1, reg2, reg3, 7*
VCMPEQ_UQPS *reg1, reg2, reg3*	VCMPPS *reg1, reg2, reg3, 8*
VCMPNGEPS *reg1, reg2, reg3*	VCMPPS *reg1, reg2, reg3, 9*
VCMPNGTPS *reg1, reg2, reg3*	VCMPPS *reg1, reg2, reg3, 0AH*
VCMPFALSEPS *reg1, reg2, reg3*	VCMPPS *reg1, reg2, reg3, 0BH*
VCMPNEQ_OQPS *reg1, reg2, reg3*	VCMPPS *reg1, reg2, reg3, 0CH*
VCMPGEPS *reg1, reg2, reg3*	VCMPPS *reg1, reg2, reg3, 0DH*
VCMPGTPS *reg1, reg2, reg3*	VCMPPS *reg1, reg2, reg3, 0EH*
VCMPTRUEPS *reg1, reg2, reg3*	VCMPPS *reg1, reg2, reg3, 0FH*
VCMPEQ_OSPS *reg1, reg2, reg3*	VCMPPS *reg1, reg2, reg3, 10H*
VCMPLT_OQPS *reg1, reg2, reg3*	VCMPPS *reg1, reg2, reg3, 11H*
VCMPLE_OQPS *reg1, reg2, reg3*	VCMPPS *reg1, reg2, reg3, 12H*
VCMPUNORD_SPS *reg1, reg2, reg3*	VCMPPS *reg1, reg2, reg3, 13H*
VCMPNEQ_USPS *reg1, reg2, reg3*	VCMPPS *reg1, reg2, reg3, 14H*
VCMPNLT_UQPS *reg1, reg2, reg3*	VCMPPS *reg1, reg2, reg3, 15H*
VCMPNLE_UQPS *reg1, reg2, reg3*	VCMPPS *reg1, reg2, reg3, 16H*
VCMPORD_SPS *reg1, reg2, reg3*	VCMPPS *reg1, reg2, reg3, 17H*
VCMPEQ_USPS *reg1, reg2, reg3*	VCMPPS *reg1, reg2, reg3, 18H*
VCMPNGE_UQPS *reg1, reg2, reg3*	VCMPPS *reg1, reg2, reg3, 19H*
VCMPNGT_UQPS *reg1, reg2, reg3*	VCMPPS *reg1, reg2, reg3, 1AH*
VCMPFALSE_OSPS *reg1, reg2, reg3*	VCMPPS *reg1, reg2, reg3, 1BH*
VCMPNEQ_OSPS *reg1, reg2, reg3*	VCMPPS *reg1, reg2, reg3, 1CH*
VCMPGE_OQPS *reg1, reg2, reg3*	VCMPPS *reg1, reg2, reg3, 1DH*
VCMPGT_OQPS *reg1, reg2, reg3*	VCMPPS *reg1, reg2, reg3, 1EH*
VCMPTRUE_USPS *reg1, reg2, reg3*	VCMPPS *reg1, reg2, reg3, 1FH*

Operation

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

VCMPPS (EVEX encoded versions)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j ← 0 TO KL-1
    i ← j * 32
    IF k2[j] OR *no writemask*
        THEN
            IF (EVEX.b = 1) AND (SRC2 *is memory*)
                THEN
                    CMP ← SRC1[i+31:i] OP5 SRC2[31:0]
                ELSE
                    CMP ← SRC1[i+31:i] OP5 SRC2[i+31:i]
            FI;
            IF CMP = TRUE
                THEN DEST[j] ← 1;
                ELSE DEST[j] ← 0; FI;
        ELSE    DEST[j] ← 0                    ; zeroing-masking onlyFI;
    FI;
ENDFOR
DEST[MAX_KL-1:KL] ← 0
```

VCMPPS (VEX.256 encoded version)

```
CMP0 ← SRC1[31:0] OP5 SRC2[31:0];
CMP1 ← SRC1[63:32] OP5 SRC2[63:32];
CMP2 ← SRC1[95:64] OP5 SRC2[95:64];
CMP3 ← SRC1[127:96] OP5 SRC2[127:96];
CMP4 ← SRC1[159:128] OP5 SRC2[159:128];
CMP5 ← SRC1[191:160] OP5 SRC2[191:160];
CMP6 ← SRC1[223:192] OP5 SRC2[223:192];
CMP7 ← SRC1[255:224] OP5 SRC2[255:224];
IF CMP0 = TRUE
    THEN DEST[31:0] ←FFFFFFFFH;
    ELSE DEST[31:0] ← 000000000H; FI;
IF CMP1 = TRUE
    THEN DEST[63:32] ← FFFFFFFFH;
    ELSE DEST[63:32] ←000000000H; FI;
IF CMP2 = TRUE
    THEN DEST[95:64] ← FFFFFFFFH;
    ELSE DEST[95:64] ← 000000000H; FI;
IF CMP3 = TRUE
    THEN DEST[127:96] ← FFFFFFFFH;
    ELSE DEST[127:96] ← 000000000H; FI;
IF CMP4 = TRUE
    THEN DEST[159:128] ← FFFFFFFFH;
    ELSE DEST[159:128] ← 000000000H; FI;
IF CMP5 = TRUE
    THEN DEST[191:160] ← FFFFFFFFH;
    ELSE DEST[191:160] ← 000000000H; FI;
IF CMP6 = TRUE
    THEN DEST[223:192] ← FFFFFFFFH;
    ELSE DEST[223:192] ←000000000H; FI;
IF CMP7 = TRUE
    THEN DEST[255:224] ← FFFFFFFFH;
    ELSE DEST[255:224] ← 000000000H; FI;
DEST[MAXVL-1:256] ← 0
```

VCMPPS (VEX.128 encoded version)

```
CMP0 ← SRC1[31:0] OP5 SRC2[31:0];
CMP1 ← SRC1[63:32] OP5 SRC2[63:32];
CMP2 ← SRC1[95:64] OP5 SRC2[95:64];
CMP3 ← SRC1[127:96] OP5 SRC2[127:96];
IF CMP0 = TRUE
    THEN DEST[31:0] ←FFFFFFFFH;
    ELSE DEST[31:0] ← 000000000H; FI;
IF CMP1 = TRUE
    THEN DEST[63:32] ← FFFFFFFFH;
    ELSE DEST[63:32] ← 000000000H; FI;
IF CMP2 = TRUE
    THEN DEST[95:64] ← FFFFFFFFH;
    ELSE DEST[95:64] ← 000000000H; FI;
IF CMP3 = TRUE
    THEN DEST[127:96] ← FFFFFFFFH;
    ELSE DEST[127:96] ←000000000H; FI;
DEST[MAXVL-1:128] ← 0
```

CMPPS (128-bit Legacy SSE version)

```
CMP0 ← SRC1[31:0] OP3 SRC2[31:0];
CMP1 ← SRC1[63:32] OP3 SRC2[63:32];
CMP2 ← SRC1[95:64] OP3 SRC2[95:64];
CMP3 ← SRC1[127:96] OP3 SRC2[127:96];
IF CMP0 = TRUE
    THEN DEST[31:0] ←FFFFFFFFH;
    ELSE DEST[31:0] ← 000000000H; FI;
IF CMP1 = TRUE
    THEN DEST[63:32] ← FFFFFFFFH;
    ELSE DEST[63:32] ← 000000000H; FI;
IF CMP2 = TRUE
    THEN DEST[95:64] ← FFFFFFFFH;
    ELSE DEST[95:64] ← 000000000H; FI;
IF CMP3 = TRUE
    THEN DEST[127:96] ← FFFFFFFFH;
    ELSE DEST[127:96] ←000000000H; FI;
DEST[MAXVL-1:128] (Unmodified)
```

Intel C/C++ Compiler Intrinsic Equivalent

VCMPPS __mmask16 _mm512_cmp_ps_mask(__m512 a, __m512 b, int imm);
VCMPPS __mmask16 _mm512_cmp_round_ps_mask(__m512 a, __m512 b, int imm, int sae);
VCMPPS __mmask16 _mm512_mask_cmp_ps_mask(__mmask16 k1, __m512 a, __m512 b, int imm);
VCMPPS __mmask16 _mm512_mask_cmp_round_ps_mask(__mmask16 k1, __m512 a, __m512 b, int imm, int sae);
VCMPPD __mmask8 _mm256_cmp_ps_mask(__m256 a, __m256 b, int imm);
VCMPPS __mmask8 _mm256_mask_cmp_ps_mask(__mmask8 k1, __m256 a, __m256 b, int imm);
VCMPPS __mmask8 _mm_cmp_ps_mask(__m128 a, __m128 b, int imm);
VCMPPS __mmask8 _mm_mask_cmp_ps_mask(__mmask8 k1, __m128 a, __m128 b, int imm);
VCMPPS __m256 _mm256_cmp_ps(__m256 a, __m256 b, int imm)
CMPPS __m128 _mm_cmp_ps(__m128 a, __m128 b, int imm)

SIMD Floating-Point Exceptions

Invalid if SNaN operand and invalid if QNaN and predicate as listed in Table 3-1.

Denormal

Other Exceptions

VEX-encoded instructions, see Exceptions Type 2.
EVEX-encoded instructions, see Exceptions Type E2.

CMPS/CMPSB/CMPSW/CMPSD/CMPSQ—Compare String Operands

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
A6	CMPS *m8, m8*	ZO	Valid	Valid	For legacy mode, compare byte at address DS:(E)SI with byte at address ES:(E)DI; For 64-bit mode compare byte at address (R|E)SI to byte at address (R|E)DI. The status flags are set accordingly.
A7	CMPS *m16, m16*	ZO	Valid	Valid	For legacy mode, compare word at address DS:(E)SI with word at address ES:(E)DI; For 64-bit mode compare word at address (R|E)SI with word at address (R|E)DI. The status flags are set accordingly.
A7	CMPS *m32, m32*	ZO	Valid	Valid	For legacy mode, compare dword at address DS:(E)SI at dword at address ES:(E)DI; For 64-bit mode compare dword at address (R|E)SI at dword at address (R|E)DI. The status flags are set accordingly.
REX.W + A7	CMPS *m64, m64*	ZO	Valid	N.E.	Compares quadword at address (R|E)SI with quadword at address (R|E)DI and sets the status flags accordingly.
A6	CMPSB	ZO	Valid	Valid	For legacy mode, compare byte at address DS:(E)SI with byte at address ES:(E)DI; For 64-bit mode compare byte at address (R|E)SI with byte at address (R|E)DI. The status flags are set accordingly.
A7	CMPSW	ZO	Valid	Valid	For legacy mode, compare word at address DS:(E)SI with word at address ES:(E)DI; For 64-bit mode compare word at address (R|E)SI with word at address (R|E)DI. The status flags are set accordingly.
A7	CMPSD	ZO	Valid	Valid	For legacy mode, compare dword at address DS:(E)SI with dword at address ES:(E)DI; For 64-bit mode compare dword at address (R|E)SI with dword at address (R|E)DI. The status flags are set accordingly.
REX.W + A7	CMPSQ	ZO	Valid	N.E.	Compares quadword at address (R|E)SI with quadword at address (R|E)DI and sets the status flags accordingly.

Instruction Operand Encoding

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

Description

Compares the byte, word, doubleword, or quadword specified with the first source operand with the byte, word, doubleword, or quadword specified with the second source operand and sets the status flags in the EFLAGS register according to the results.

Both source operands are located in memory. The address of the first source operand is read from DS:SI, DS:ESI or RSI (depending on the address-size attribute of the instruction is 16, 32, or 64, respectively). The address of the second source operand is read from ES:DI, ES:EDI or RDI (again depending on the address-size attribute of the instruction is 16, 32, or 64). The DS segment may be overridden with a segment override prefix, but the ES segment cannot be overridden.

At the assembly-code level, two forms of this instruction are allowed: the "explicit-operands" form and the "no-operands" form. The explicit-operands form (specified with the CMPS mnemonic) allows the two source operands to be specified explicitly. Here, the source operands should be symbols that indicate the size and location of the source values. This explicit-operand form is provided to allow documentation. However, note that the documentation provided by this form can be misleading. That is, the source operand symbols must specify the correct type (size) of the operands (bytes, words, or doublewords, quadwords), but they do not have to specify the correct loca-

tion. Locations of the source operands are always specified by the DS:(E)SI (or RSI) and ES:(E)DI (or RDI) registers, which must be loaded correctly before the compare string instruction is executed.

The no-operands form provides "short forms" of the byte, word, and doubleword versions of the CMPS instructions. Here also the DS:(E)SI (or RSI) and ES:(E)DI (or RDI) registers are assumed by the processor to specify the location of the source operands. The size of the source operands is selected with the mnemonic: CMPSB (byte comparison), CMPSW (word comparison), CMPSD (doubleword comparison), or CMPSQ (quadword comparison using REX.W).

After the comparison, the (E/R)SI and (E/R)DI registers increment or decrement automatically according to the setting of the DF flag in the EFLAGS register. (If the DF flag is 0, the (E/R)SI and (E/R)DI register increment; if the DF flag is 1, the registers decrement.) The registers increment or decrement by 1 for byte operations, by 2 for word operations, 4 for doubleword operations. If operand size is 64, RSI and RDI registers increment by 8 for quadword operations.

The CMPS, CMPSB, CMPSW, CMPSD, and CMPSQ instructions can be preceded by the REP prefix for block comparisons. More often, however, these instructions will be used in a LOOP construct that takes some action based on the setting of the status flags before the next comparison is made. See "REP/REPE/REPZ /REPNE/REPNZ—Repeat String Operation Prefix" in Chapter 4 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2B*, for a description of the REP prefix.

In 64-bit mode, the instruction's default address size is 64 bits, 32 bit address size is supported using the prefix 67H. Use of the REX.W prefix promotes doubleword operation to 64 bits (see CMPSQ). See the summary chart at the beginning of this section for encoding data and limits.

Operation

```
temp ← SRC1 - SRC2;
SetStatusFlags(temp);

IF (64-Bit Mode)
    THEN
        IF (Byte comparison)
        THEN IF DF = 0
            THEN
                (R|E)SI ← (R|E)SI + 1;
                (R|E)DI ← (R|E)DI + 1;
            ELSE
                (R|E)SI ← (R|E)SI – 1;
                (R|E)DI ← (R|E)DI – 1;
            FI;
        ELSE IF (Word comparison)
            THEN IF DF = 0
                THEN
                    (R|E)SI ← (R|E)SI + 2;
                    (R|E)DI ← (R|E)DI + 2;
                ELSE
                    (R|E)SI ← (R|E)SI – 2;
                    (R|E)DI ← (R|E)DI – 2;
                FI;
        ELSE IF (Doubleword comparison)
            THEN IF DF = 0
                THEN
                    (R|E)SI ← (R|E)SI + 4;
                    (R|E)DI ← (R|E)DI + 4;
                ELSE
                    (R|E)SI ← (R|E)SI – 4;
                    (R|E)DI ← (R|E)DI – 4;
                FI;
```

```
        ELSE (* Quadword comparison *)
            THEN IF DF = 0
                (R|E)SI ← (R|E)SI + 8;
                (R|E)DI ← (R|E)DI + 8;
            ELSE
                (R|E)SI ← (R|E)SI – 8;
                (R|E)DI ← (R|E)DI – 8;
            FI;
        FI;
    ELSE (* Non-64-bit Mode *)
        IF (byte comparison)
        THEN IF DF = 0
            THEN
                (E)SI ← (E)SI + 1;
                (E)DI ← (E)DI + 1;
            ELSE
                (E)SI ← (E)SI – 1;
                (E)DI ← (E)DI – 1;
            FI;
        ELSE IF (Word comparison)
            THEN IF DF = 0
                (E)SI ← (E)SI + 2;
                (E)DI ← (E)DI + 2;
            ELSE
                (E)SI ← (E)SI – 2;
                (E)DI ← (E)DI – 2;
            FI;
        ELSE (* Doubleword comparison *)
            THEN IF DF = 0
                (E)SI ← (E)SI + 4;
                (E)DI ← (E)DI + 4;
            ELSE
                (E)SI ← (E)SI – 4;
                (E)DI ← (E)DI – 4;
            FI;
        FI;
FI;
```

Flags Affected

The CF, OF, SF, ZF, AF, and PF flags are set according to the temporary result of the comparison.

Protected Mode Exceptions

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

Real-Address Mode Exceptions

#GP	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS	If a memory operand effective address is outside the SS segment limit.
#UD	If the LOCK prefix is used.

Virtual-8086 Mode Exceptions

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

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If the LOCK prefix is used.

CMPSD—Compare Scalar Double-Precision Floating-Point Value

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
F2 0F C2 /r ib CMPSD xmm1, xmm2/m64, imm8	A	V/V	SSE2	Compare low double-precision floating-point value in xmm2/m64 and xmm1 using bits 2:0 of imm8 as comparison predicate.
VEX.NDS.LIG.F2.0F.WIG C2 /r ib VCMPSD xmm1, xmm2, xmm3/m64, imm8	B	V/V	AVX	Compare low double-precision floating-point value in xmm3/m64 and xmm2 using bits 4:0 of imm8 as comparison predicate.
EVEX.NDS.LIG.F2.0F.W1 C2 /r ib VCMPSD k1 {k2}, xmm2, xmm3/m64{sae}, imm8	C	V/V	AVX512F	Compare low double-precision floating-point value in xmm3/m64 and xmm2 using bits 4:0 of imm8 as comparison predicate with writemask k2 and leave the result in mask register k1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	NA	ModRM:reg (r, w)	ModRM:r/m (r)	Imm8	NA
B	NA	ModRM:reg (w)	VEX.vvvv	ModRM:r/m (r)	Imm8
C	Tuple1 Scalar	ModRM:reg (w)	EVEX.vvvv	ModRM:r/m (r)	Imm8

Description

Compares the low double-precision floating-point values in the second source operand and the first source operand and returns the results in of the comparison to the destination operand. The comparison predicate operand (immediate operand) specifies the type of comparison performed.

128-bit Legacy SSE version: The first source and destination operand (first operand) is an XMM register. The second source operand (second operand) can be an XMM register or 64-bit memory location. Bits (MAXVL-1:64) of the corresponding YMM destination register remain unchanged. The comparison result is a quadword mask of all 1s (comparison true) or all 0s (comparison false).

VEX.128 encoded version: The first source operand (second operand) is an XMM register. The second source operand (third operand) can be an XMM register or a 64-bit memory location. The result is stored in the low quadword of the destination operand; the high quadword is filled with the contents of the high quadword of the first source operand. Bits (MAXVL-1:128) of the destination ZMM register are zeroed. The comparison result is a quadword mask of all 1s (comparison true) or all 0s (comparison false).

EVEX encoded version: The first source operand (second operand) is an XMM register. The second source operand can be a XMM register or a 64-bit memory location. The destination operand (first operand) is an opmask register. The comparison result is a single mask bit of 1 (comparison true) or 0 (comparison false), written to the destination starting from the LSB according to the writemask k2. Bits (MAX_KL-1:128) of the destination register are cleared.

The comparison predicate operand is an 8-bit immediate:

- For instructions encoded using the VEX prefix, bits 4:0 define the type of comparison to be performed (see Table 3-1). Bits 5 through 7 of the immediate are reserved.
- For instruction encodings that do not use VEX prefix, bits 2:0 define the type of comparison to be made (see the first 8 rows of Table 3-1). Bits 3 through 7 of the immediate are reserved.

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

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

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

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

Compilers and assemblers may implement the following two-operand pseudo-ops in addition to the three-operand CMPSD instruction, for processors with "CPUID.1H:ECX.AVX =0". See Table 3-6. Compiler should treat reserved Imm8 values as illegal syntax.

Table 3-6. Pseudo-Op and CMPSD Implementation

Pseudo-Op	CMPSD Implementation
CMPEQSD x*mm1, xmm2*	CMPSD x*mm1, xmm2, 0*
CMPLTSD x*mm1, xmm2*	CMPSD x*mm1, xmm2, 1*
CMPLESD x*mm1, xmm2*	CMPSD x*mm1, xmm2, 2*
CMPUNORDSD x*mm1, xmm2*	CMPSD x*mm1, xmm2, 3*
CMPNEQSD x*mm1, xmm2*	CMPSD x*mm1, xmm2, 4*
CMPNLTSD x*mm1, xmm2*	CMPSD x*mm1, xmm2, 5*
CMPNLESD x*mm1, xmm2*	CMPSD x*mm1, xmm2, 6*
CMPORDSD x*mm1, xmm2*	CMPSD x*mm1, xmm2, 7*

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

Processors with "CPUID.1H:ECX.AVX =1" implement the full complement of 32 predicates shown in Table 3-7, software emulation is no longer needed. Compilers and assemblers may implement the following three-operand pseudo-ops in addition to the four-operand VCMPSD instruction. See Table 3-7, where the notations of reg1 reg2, and reg3 represent either XMM registers or YMM registers. Compiler should treat reserved Imm8 values as illegal syntax. Alternately, intrinsics can map the pseudo-ops to pre-defined constants to support a simpler intrinsic interface. Compilers and assemblers may implement three-operand pseudo-ops for EVEX encoded VCMPSD instructions in a similar fashion by extending the syntax listed in Table 3-7.

Table 3-7. Pseudo-Op and VCMPSD Implementation

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

Table 3-7. Pseudo-Op and VCMPSD Implementation

Pseudo-Op	CMPSD Implementation
VCMPGTSD *reg1, reg2, reg3*	VCMPSD *reg1, reg2, reg3, 0EH*
VCMPTRUESD *reg1, reg2, reg3*	VCMPSD *reg1, reg2, reg3, 0FH*
VCMPEQ_OSSD *reg1, reg2, reg3*	VCMPSD *reg1, reg2, reg3, 10H*
VCMPLT_OQSD *reg1, reg2, reg3*	VCMPSD *reg1, reg2, reg3, 11H*
VCMPLE_OQSD *reg1, reg2, reg3*	VCMPSD *reg1, reg2, reg3, 12H*
VCMPUNORD_SSD *reg1, reg2, reg3*	VCMPSD *reg1, reg2, reg3, 13H*
VCMPNEQ_USSD *reg1, reg2, reg3*	VCMPSD *reg1, reg2, reg3, 14H*
VCMPNLT_UQSD *reg1, reg2, reg3*	VCMPSD *reg1, reg2, reg3, 15H*
VCMPNLE_UQSD *reg1, reg2, reg3*	VCMPSD *reg1, reg2, reg3, 16H*
VCMPORD_SSD *reg1, reg2, reg3*	VCMPSD *reg1, reg2, reg3, 17H*
VCMPEQ_USSD *reg1, reg2, reg3*	VCMPSD *reg1, reg2, reg3, 18H*
VCMPNGE_UQSD *reg1, reg2, reg3*	VCMPSD *reg1, reg2, reg3, 19H*
VCMPNGT_UQSD *reg1, reg2, reg3*	VCMPSD *reg1, reg2, reg3, 1AH*
VCMPFALSE_OSSD *reg1, reg2, reg3*	VCMPSD *reg1, reg2, reg3, 1BH*
VCMPNEQ_OSSD *reg1, reg2, reg3*	VCMPSD *reg1, reg2, reg3, 1CH*
VCMPGE_OQSD *reg1, reg2, reg3*	VCMPSD *reg1, reg2, reg3, 1DH*
VCMPGT_OQSD *reg1, reg2, reg3*	VCMPSD *reg1, reg2, reg3, 1EH*
VCMPTRUE_USSD *reg1, reg2, reg3*	VCMPSD *reg1, reg2, reg3, 1FH*

Software should ensure VCMPSD is encoded with VEX.L=0. Encoding VCMPSD with VEX.L=1 may encounter unpredictable behavior across different processor generations.

Operation

```
CASE (COMPARISON PREDICATE) OF
    0: OP3 ←EQ_OQ; OP5 ←EQ_OQ;
    1: OP3 ←LT_OS; OP5 ←LT_OS;
    2: OP3 ←LE_OS; OP5 ←LE_OS;
    3: OP3 ←UNORD_Q; OP5 ←UNORD_Q;
    4: OP3 ←NEQ_UQ; OP5 ←NEQ_UQ;
    5: OP3 ←NLT_US; OP5 ←NLT_US;
    6: OP3 ←NLE_US; OP5 ←NLE_US;
    7: OP3 ←ORD_Q; OP5 ←ORD_Q;
    8: OP5 ←EQ_UQ;
    9: OP5 ←NGE_US;
    10: OP5 ←NGT_US;
    11: OP5 ←FALSE_OQ;
    12: OP5 ←NEQ_OQ;
    13: OP5 ←GE_OS;
    14: OP5 ←GT_OS;
    15: OP5 ←TRUE_UQ;
    16: OP5 ←EQ_OS;
    17: OP5 ←LT_OQ;
    18: OP5 ←LE_OQ;
    19: OP5 ←UNORD_S;
    20: OP5 ←NEQ_US;
    21: OP5 ←NLT_UQ;
```

```
    22: OP5 ←NLE_UQ;
    23: OP5 ←ORD_S;
    24: OP5 ←EQ_US;
    25: OP5 ←NGE_UQ;
    26: OP5 ←NGT_UQ;
    27: OP5 ←FALSE_OS;
    28: OP5 ←NEQ_OS;
    29: OP5 ←GE_OQ;
    30: OP5 ←GT_OQ;
    31: OP5 ←TRUE_US;
    DEFAULT: Reserved
ESAC;
```

VCMPSD (EVEX encoded version)

```
CMP0 ← SRC1[63:0] OP5 SRC2[63:0];

IF k2[0] or *no writemask*
    THEN    IF CMP0 = TRUE
                THEN DEST[0] ← 1;
                ELSE DEST[0] ← 0; FI;
    ELSE    DEST[0] ← 0                 ; zeroing-masking only
FI;
DEST[MAX_KL-1:1] ← 0
```

CMPSD (128-bit Legacy SSE version)

```
CMP0 ←DEST[63:0] OP3 SRC[63:0];
IF CMP0 = TRUE
THEN DEST[63:0] ←FFFFFFFFFFFFFFFFH;
ELSE DEST[63:0] ←0000000000000000H; FI;
DEST[MAXVL-1:64] (Unmodified)
```

VCMPSD (VEX.128 encoded version)

```
CMP0 ←SRC1[63:0] OP5 SRC2[63:0];
IF CMP0 = TRUE
THEN DEST[63:0] ←FFFFFFFFFFFFFFFFH;
ELSE DEST[63:0] ←0000000000000000H; FI;
DEST[127:64] ←SRC1[127:64]
DEST[MAXVL-1:128] ←0
```

Intel C/C++ Compiler Intrinsic Equivalent

VCMPSD __mmask8 _mm_cmp_sd_mask(__m128d a, __m128d b, int imm);
VCMPSD __mmask8 _mm_cmp_round_sd_mask(__m128d a, __m128d b, int imm, int sae);
VCMPSD __mmask8 _mm_mask_cmp_sd_mask(__mmask8 k1, __m128d a, __m128d b, int imm);
VCMPSD __mmask8 _mm_mask_cmp_round_sd_mask(__mmask8 k1, __m128d a, __m128d b, int imm, int sae);
(V)CMPSD __m128d _mm_cmp_sd(__m128d a, __m128d b, const int imm)

SIMD Floating-Point Exceptions

Invalid if SNaN operand, Invalid if QNaN and predicate as listed in Table 3-1 Denormal.

Other Exceptions

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

CMPSS—Compare Scalar Single-Precision Floating-Point Value

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
F3 0F C2 /r ib CMPSS xmm1, xmm2/m32, imm8	A	V/V	SSE	Compare low single-precision floating-point value in xmm2/m32 and xmm1 using bits 2:0 of imm8 as comparison predicate.
VEX.NDS.LIG.F3.0F.WIG C2 /r ib VCMPSS xmm1, xmm2, xmm3/m32, imm8	B	V/V	AVX	Compare low single-precision floating-point value in xmm3/m32 and xmm2 using bits 4:0 of imm8 as comparison predicate.
EVEX.NDS.LIG.F3.0F.W0 C2 /r ib VCMPSS k1 {k2}, xmm2, xmm3/m32{sae}, imm8	C	V/V	AVX512F	Compare low single-precision floating-point value in xmm3/m32 and xmm2 using bits 4:0 of imm8 as comparison predicate with writemask k2 and leave the result in mask register k1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	NA	ModRM:reg (r, w)	ModRM:r/m (r)	Imm8	NA
B	NA	ModRM:reg (w)	VEX.vvvv	ModRM:r/m (r)	Imm8
C	Tuple1 Scalar	ModRM:reg (w)	EVEX.vvvv	ModRM:r/m (r)	Imm8

Description

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

128-bit Legacy SSE version: The first source and destination operand (first operand) is an XMM register. The second source operand (second operand) can be an XMM register or 32-bit memory location. Bits (MAXVL-1:32) of the corresponding YMM destination register remain unchanged. The comparison result is a doubleword mask of all 1s (comparison true) or all 0s (comparison false).

VEX.128 encoded version: The first source operand (second operand) is an XMM register. The second source operand (third operand) can be an XMM register or a 32-bit memory location. The result is stored in the low 32 bits of the destination operand; bits 128:32 of the destination operand are copied from the first source operand. Bits (MAXVL-1:128) of the destination ZMM register are zeroed. The comparison result is a doubleword mask of all 1s (comparison true) or all 0s (comparison false).

EVEX encoded version: The first source operand (second operand) is an XMM register. The second source operand can be a XMM register or a 32-bit memory location. The destination operand (first operand) is an opmask register. The comparison result is a single mask bit of 1 (comparison true) or 0 (comparison false), written to the destination starting from the LSB according to the writemask k2. Bits (MAX_KL-1:128) of the destination register are cleared.

The comparison predicate operand is an 8-bit immediate:

- For instructions encoded using the VEX prefix, bits 4:0 define the type of comparison to be performed (see Table 3-1). Bits 5 through 7 of the immediate are reserved.
- For instruction encodings that do not use VEX prefix, bits 2:0 define the type of comparison to be made (see the first 8 rows of Table 3-1). Bits 3 through 7 of the immediate are reserved.

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

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

Note that processors with "CPUID.1H:ECX.AVX =0" do not implement the "greater-than", "greater-than-or-equal", "not-greater than", and "not-greater-than-or-equal relations" predicates. These comparisons can be made either

by using the inverse relationship (that is, use the "not-less-than-or-equal" to make a "greater-than" comparison) or by using software emulation. When using software emulation, the program must swap the operands (copying registers when necessary to protect the data that will now be in the destination), and then perform the compare using a different predicate. The predicate to be used for these emulations is listed in the first 8 rows of Table 3-7 (*Intel 64 and IA-32 Architectures Software Developer's Manual Volume 2A*) under the heading Emulation.

Compilers and assemblers may implement the following two-operand pseudo-ops in addition to the three-operand CMPSS instruction, for processors with "CPUID.1H:ECX.AVX =0". See Table 3-8. Compiler should treat reserved Imm8 values as illegal syntax.

Table 3-8. Pseudo-Op and CMPSS Implementation

Pseudo-Op	CMPSS Implementation
CMPEQSS x*mm1, xmm2*	CMPSS x*mm1, xmm2, 0*
CMPLTSS x*mm1, xmm2*	CMPSS x*mm1, xmm2, 1*
CMPLESS x*mm1, xmm2*	CMPSS x*mm1, xmm2, 2*
CMPUNORDSS x*mm1, xmm2*	CMPSS x*mm1, xmm2, 3*
CMPNEQSS x*mm1, xmm2*	CMPSS x*mm1, xmm2, 4*
CMPNLTSS x*mm1, xmm2*	CMPSS x*mm1, xmm2, 5*
CMPNLESS x*mm1, xmm2*	CMPSS x*mm1, xmm2, 6*
CMPORDSS x*mm1, xmm2*	CMPSS x*mm1, xmm2, 7*

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

Processors with "CPUID.1H:ECX.AVX =1" implement the full complement of 32 predicates shown in Table 3-7, software emulation is no longer needed. Compilers and assemblers may implement the following three-operand pseudo-ops in addition to the four-operand VCMPSS instruction. See Table 3-9, where the notations of reg1 reg2, and reg3 represent either XMM registers or YMM registers. Compiler should treat reserved Imm8 values as illegal syntax. Alternately, intrinsics can map the pseudo-ops to pre-defined constants to support a simpler intrinsic interface. Compilers and assemblers may implement three-operand pseudo-ops for EVEX encoded VCMPSS instructions in a similar fashion by extending the syntax listed in Table 3-9.

Table 3-9. Pseudo-Op and VCMPSS Implementation

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

Table 3-9. Pseudo-Op and VCMPSS Implementation

Pseudo-Op	CMPSS Implementation
VCMPGTSS *reg1, reg2, reg3*	VCMPSS *reg1, reg2, reg3, 0EH*
VCMPTRUESS *reg1, reg2, reg3*	VCMPSS *reg1, reg2, reg3, 0FH*
VCMPEQ_OSSS *reg1, reg2, reg3*	VCMPSS *reg1, reg2, reg3, 10H*
VCMPLT_OQSS *reg1, reg2, reg3*	VCMPSS *reg1, reg2, reg3, 11H*
VCMPLE_OQSS *reg1, reg2, reg3*	VCMPSS *reg1, reg2, reg3, 12H*
VCMPUNORD_SSS *reg1, reg2, reg3*	VCMPSS *reg1, reg2, reg3, 13H*
VCMPNEQ_USSS *reg1, reg2, reg3*	VCMPSS *reg1, reg2, reg3, 14H*
VCMPNLT_UQSS *reg1, reg2, reg3*	VCMPSS *reg1, reg2, reg3, 15H*
VCMPNLE_UQSS *reg1, reg2, reg3*	VCMPSS *reg1, reg2, reg3, 16H*
VCMPORD_SSS *reg1, reg2, reg3*	VCMPSS *reg1, reg2, reg3, 17H*
VCMPEQ_USSS *reg1, reg2, reg3*	VCMPSS *reg1, reg2, reg3, 18H*
VCMPNGE_UQSS *reg1, reg2, reg3*	VCMPSS *reg1, reg2, reg3, 19H*
VCMPNGT_UQSS *reg1, reg2, reg3*	VCMPSS *reg1, reg2, reg3, 1AH*
VCMPFALSE_OSSS *reg1, reg2, reg3*	VCMPSS *reg1, reg2, reg3, 1BH*
VCMPNEQ_OSSS *reg1, reg2, reg3*	VCMPSS *reg1, reg2, reg3, 1CH*
VCMPGE_OQSS *reg1, reg2, reg3*	VCMPSS *reg1, reg2, reg3, 1DH*
VCMPGT_OQSS *reg1, reg2, reg3*	VCMPSS *reg1, reg2, reg3, 1EH*
VCMPTRUE_USSS *reg1, reg2, reg3*	VCMPSS *reg1, reg2, reg3, 1FH*

Software should ensure VCMPSS is encoded with VEX.L=0. Encoding VCMPSS with VEX.L=1 may encounter unpredictable behavior across different processor generations.

Operation

```
CASE (COMPARISON PREDICATE) OF
    0: OP3 ←EQ_OQ; OP5 ←EQ_OQ;
    1: OP3 ←LT_OS; OP5 ←LT_OS;
    2: OP3 ←LE_OS; OP5 ←LE_OS;
    3: OP3 ←UNORD_Q; OP5 ←UNORD_Q;
    4: OP3 ←NEQ_UQ; OP5 ←NEQ_UQ;
    5: OP3 ←NLT_US; OP5 ←NLT_US;
    6: OP3 ←NLE_US; OP5 ←NLE_US;
    7: OP3 ←ORD_Q; OP5 ←ORD_Q;
    8: OP5 ←EQ_UQ;
    9: OP5 ←NGE_US;
    10: OP5 ←NGT_US;
    11: OP5 ←FALSE_OQ;
    12: OP5 ←NEQ_OQ;
    13: OP5 ←GE_OS;
    14: OP5 ←GT_OS;
    15: OP5 ←TRUE_UQ;
    16: OP5 ←EQ_OS;
    17: OP5 ←LT_OQ;
    18: OP5 ←LE_OQ;
    19: OP5 ←UNORD_S;
    20: OP5 ←NEQ_US;
    21: OP5 ←NLT_UQ;
```

```
    22: OP5 ←NLE_UQ;
    23: OP5 ←ORD_S;
    24: OP5 ←EQ_US;
    25: OP5 ←NGE_UQ;
    26: OP5 ←NGT_UQ;
    27: OP5 ←FALSE_OS;
    28: OP5 ←NEQ_OS;
    29: OP5 ←GE_OQ;
    30: OP5 ←GT_OQ;
    31: OP5 ←TRUE_US;
    DEFAULT: Reserved
ESAC;
```

VCMPSS (EVEX encoded version)

```
CMP0 ← SRC1[31:0] OP5 SRC2[31:0];

IF k2[0] or *no writemask*
    THEN    IF CMP0 = TRUE
                THEN DEST[0] ← 1;
                ELSE DEST[0] ← 0; FI;
    ELSE    DEST[0] ← 0                 ; zeroing-masking only
FI;
DEST[MAX_KL-1:1] ← 0
```

CMPSS (128-bit Legacy SSE version)

```
CMP0 ←DEST[31:0] OP3 SRC[31:0];
IF CMP0 = TRUE
THEN DEST[31:0] ←FFFFFFFFH;
ELSE DEST[31:0] ←00000000H; FI;
DEST[MAXVL-1:32] (Unmodified)
```

VCMPSS (VEX.128 encoded version)

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

Intel C/C++ Compiler Intrinsic Equivalent

VCMPSS __mmask8 _mm_cmp_ss_mask(__m128 a, __m128 b, int imm);
VCMPSS __mmask8 _mm_cmp_round_ss_mask(__m128 a, __m128 b, int imm, int sae);
VCMPSS __mmask8 _mm_mask_cmp_ss_mask(__mmask8 k1, __m128 a, __m128 b, int imm);
VCMPSS __mmask8 _mm_mask_cmp_round_ss_mask(__mmask8 k1, __m128 a, __m128 b, int imm, int sae);
(V)CMPSS __m128 _mm_cmp_ss(__m128 a, __m128 b, const int imm)

SIMD Floating-Point Exceptions

Invalid if SNaN operand, Invalid if QNaN and predicate as listed in Table 3-1, Denormal.

Other Exceptions

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

CMPXCHG—Compare and Exchange

Opcode/ Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
0F B0/*r* CMPXCHG *r/m8, r8*	MR	Valid	Valid*	Compare AL with *r/m8*. If equal, ZF is set and *r8* is loaded into *r/m8*. Else, clear ZF and load *r/m8* into AL.
REX + 0F B0/*r* CMPXCHG *r/m8**,*r8*	MR	Valid	N.E.	Compare AL with *r/m8*. If equal, ZF is set and *r8* is loaded into *r/m8*. Else, clear ZF and load *r/m8* into AL.
0F B1/*r* CMPXCHG *r/m16, r16*	MR	Valid	Valid*	Compare AX with *r/m16*. If equal, ZF is set and *r16* is loaded into *r/m16*. Else, clear ZF and load *r/m16* into AX.
0F B1/*r* CMPXCHG *r/m32, r32*	MR	Valid	Valid*	Compare EAX with *r/m32*. If equal, ZF is set and *r32* is loaded into *r/m32*. Else, clear ZF and load *r/m32* into EAX.
REX.W + 0F B1/*r* CMPXCHG *r/m64, r64*	MR	Valid	N.E.	Compare RAX with *r/m64*. If equal, ZF is set and *r64* is loaded into *r/m64*. Else, clear ZF and load *r/m64* into RAX.

NOTES:

* See the IA-32 Architecture Compatibility section below.

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

Instruction Operand Encoding

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

Description

Compares the value in the AL, AX, EAX, or RAX register with the first operand (destination operand). If the two values are equal, the second operand (source operand) is loaded into the destination operand. Otherwise, the destination operand is loaded into the AL, AX, EAX or RAX register. RAX register is available only in 64-bit mode.

This instruction can be used with a LOCK prefix to allow the instruction to be executed atomically. To simplify the interface to the processor's bus, the destination operand receives a write cycle without regard to the result of the comparison. The destination operand is written back if the comparison fails; otherwise, the source operand is written into the destination. (The processor never produces a locked read without also producing a locked write.)

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

IA-32 Architecture Compatibility

This instruction is not supported on Intel processors earlier than the Intel486 processors.

Operation

```
(* Accumulator = AL, AX, EAX, or RAX depending on whether a byte, word, doubleword, or quadword comparison is being performed *)
TEMP ← DEST
IF accumulator = TEMP
    THEN
        ZF ← 1;
        DEST ← SRC;
    ELSE
        ZF ← 0;
        accumulator ← TEMP;
        DEST ← TEMP;
FI;
```

Flags Affected

The ZF flag is set if the values in the destination operand and register AL, AX, or EAX are equal; otherwise it is cleared. The CF, PF, AF, SF, and OF flags are set according to the results of the comparison operation.

Protected Mode Exceptions

#GP(0)	If the destination is located in a non-writable segment.
	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
	If the DS, ES, FS, or GS register contains a NULL segment selector.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If the LOCK prefix is used but the destination is not a memory operand.

Real-Address Mode Exceptions

#GP	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS	If a memory operand effective address is outside the SS segment limit.
#UD	If the LOCK prefix is used but the destination is not a memory operand.

Virtual-8086 Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.
#UD	If the LOCK prefix is used but the destination is not a memory operand.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If the LOCK prefix is used but the destination is not a memory operand.

CMPXCHG8B/CMPXCHG16B—Compare and Exchange Bytes

Opcode/ Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
0F C7 /1 *m64* CMPXCHG8B *m64*	M	Valid	Valid*	Compare EDX:EAX with *m64*. If equal, set ZF and load ECX:EBX into *m64*. Else, clear ZF and load *m64* into EDX:EAX.
REX.W + 0F C7 /1 *m128* CMPXCHG16B *m128*	M	Valid	N.E.	Compare RDX:RAX with *m128*. If equal, set ZF and load RCX:RBX into *m128*. Else, clear ZF and load *m128* into RDX:RAX.

NOTES:
*See IA-32 Architecture Compatibility section below.

Instruction Operand Encoding

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

Description

Compares the 64-bit value in EDX:EAX (or 128-bit value in RDX:RAX if operand size is 128 bits) with the operand (destination operand). If the values are equal, the 64-bit value in ECX:EBX (or 128-bit value in RCX:RBX) is stored in the destination operand. Otherwise, the value in the destination operand is loaded into EDX:EAX (or RDX:RAX). The destination operand is an 8-byte memory location (or 16-byte memory location if operand size is 128 bits). For the EDX:EAX and ECX:EBX register pairs, EDX and ECX contain the high-order 32 bits and EAX and EBX contain the low-order 32 bits of a 64-bit value. For the RDX:RAX and RCX:RBX register pairs, RDX and RCX contain the high-order 64 bits and RAX and RBX contain the low-order 64bits of a 128-bit value.

This instruction can be used with a LOCK prefix to allow the instruction to be executed atomically. To simplify the interface to the processor's bus, the destination operand receives a write cycle without regard to the result of the comparison. The destination operand is written back if the comparison fails; otherwise, the source operand is written into the destination. (The processor never produces a locked read without also producing a locked write.)

In 64-bit mode, default operation size is 64 bits. Use of the REX.W prefix promotes operation to 128 bits. Note that CMPXCHG16B requires that the destination (memory) operand be 16-byte aligned. See the summary chart at the beginning of this section for encoding data and limits. For information on the CPUID flag that indicates CMPXCHG16B, see page 3-208.

IA-32 Architecture Compatibility

This instruction encoding is not supported on Intel processors earlier than the Pentium processors.

Operation

```
IF (64-Bit Mode and OperandSize = 64)
    THEN
        TEMP128 ← DEST
        IF (RDX:RAX = TEMP128)
            THEN
                ZF ← 1;
                DEST ← RCX:RBX;
            ELSE
                ZF ← 0;
                RDX:RAX ← TEMP128;
                DEST ← TEMP128;
                FI;
        FI
    ELSE
        TEMP64 ← DEST;
        IF (EDX:EAX = TEMP64)
            THEN
                ZF ← 1;
                DEST ← ECX:EBX;
            ELSE
                ZF ← 0;
                EDX:EAX ← TEMP64;
                DEST ← TEMP64;
                FI;
        FI;
FI;
```

Flags Affected

The ZF flag is set if the destination operand and EDX:EAX are equal; otherwise it is cleared. The CF, PF, AF, SF, and OF flags are unaffected.

Protected Mode Exceptions

#UD	If the destination is not a memory operand.
#GP(0)	If the destination is located in a non-writable segment.
	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
	If the DS, ES, FS, or GS register contains a NULL segment selector.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

Real-Address Mode Exceptions

#UD	If the destination operand is not a memory location.
#GP	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS	If a memory operand effective address is outside the SS segment limit.

Virtual-8086 Mode Exceptions

#UD	If the destination operand is not a memory location.
#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form. If memory operand for CMPXCHG16B is not aligned on a 16-byte boundary. If CPUID.01H:ECX.CMPXCHG16B[bit 13] = 0.
#UD	If the destination operand is not a memory location.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

COMISD—Compare Scalar Ordered Double-Precision Floating-Point Values and Set EFLAGS

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 2F /r COMISD xmm1, xmm2/m64	A	V/V	SSE2	Compare low double-precision floating-point values in xmm1 and xmm2/mem64 and set the EFLAGS flags accordingly.
VEX.LIG.66.0F.WIG 2F /r VCOMISD xmm1, xmm2/m64	A	V/V	AVX	Compare low double-precision floating-point values in xmm1 and xmm2/mem64 and set the EFLAGS flags accordingly.
EVEX.LIG.66.0F.W1 2F /r VCOMISD xmm1, xmm2/m64{sae}	B	V/V	AVX512F	Compare low double-precision floating-point values in xmm1 and xmm2/mem64 and set the EFLAGS flags accordingly.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	NA	ModRM:reg (w)	ModRM:r/m (r)	NA	NA
B	Tuple1 Scalar	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

Description

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

Operand 1 is an XMM register; operand 2 can be an XMM register or a 64 bit memory

location. The COMISD instruction differs from the UCOMISD instruction in that it signals a SIMD floating-point invalid operation exception (#I) when a source operand is either a QNaN or SNaN. The UCOMISD instruction signals an invalid numeric exception only if a source operand is an SNaN.

The EFLAGS register is not updated if an unmasked SIMD floating-point exception is generated.

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

Software should ensure VCOMISD is encoded with VEX.L=0. Encoding VCOMISD with VEX.L=1 may encounter unpredictable behavior across different processor generations.

Operation

COMISD (all versions)

```
RESULT ← OrderedCompare(DEST[63:0] <> SRC[63:0]) {
(* Set EFLAGS *) CASE (RESULT) OF
    UNORDERED: ZF,PF,CF ← 111;
    GREATER_THAN: ZF,PF,CF ← 000;
    LESS_THAN: ZF,PF,CF ← 001;
    EQUAL: ZF,PF,CF ← 100;
ESAC;
OF, AF, SF ←0; }
```

Intel C/C++ Compiler Intrinsic Equivalent

VCOMISD int _mm_comi_round_sd(__m128d a, __m128d b, int imm, int sae);
VCOMISD int _mm_comieq_sd (__m128d a, __m128d b)
VCOMISD int _mm_comilt_sd (__m128d a, __m128d b)
VCOMISD int _mm_comile_sd (__m128d a, __m128d b)
VCOMISD int _mm_comigt_sd (__m128d a, __m128d b)
VCOMISD int _mm_comige_sd (__m128d a, __m128d b)
VCOMISD int _mm_comineq_sd (__m128d a, __m128d b)

SIMD Floating-Point Exceptions

Invalid (if SNaN or QNaN operands), Denormal.

Other Exceptions

VEX-encoded instructions, see Exceptions Type 3;

EVEX-encoded instructions, see Exceptions Type E3NF.

#UD	If VEX.vvvv != 1111B or EVEX.vvvv != 1111B.

COMISS—Compare Scalar Ordered Single-Precision Floating-Point Values and Set EFLAGS

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F 2F /r COMISS xmm1, xmm2/m32	A	V/V	SSE	Compare low single-precision floating-point values in xmm1 and xmm2/mem32 and set the EFLAGS flags accordingly.
VEX.LIG.0F.WIG 2F /r VCOMISS xmm1, xmm2/m32	A	V/V	AVX	Compare low single-precision floating-point values in xmm1 and xmm2/mem32 and set the EFLAGS flags accordingly.
EVEX.LIG.0F.W0 2F /r VCOMISS xmm1, xmm2/m32{sae}	B	V/V	AVX512F	Compare low single-precision floating-point values in xmm1 and xmm2/mem32 and set the EFLAGS flags accordingly.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	NA	ModRM:reg (w)	ModRM:r/m (r)	NA	NA
B	Tuple1 Scalar	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

Description

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

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

The COMISS instruction differs from the UCOMISS instruction in that it signals a SIMD floating-point invalid operation exception (#I) when a source operand is either a QNaN or SNaN. The UCOMISS instruction signals an invalid numeric exception only if a source operand is an SNaN.

The EFLAGS register is not updated if an unmasked SIMD floating-point exception is generated.

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

Software should ensure VCOMISS is encoded with VEX.L=0. Encoding VCOMISS with VEX.L=1 may encounter unpredictable behavior across different processor generations.

Operation

COMISS (all versions)

```
RESULT ← OrderedCompare(DEST[31:0] <> SRC[31:0]) {
(* Set EFLAGS *) CASE (RESULT) OF
    UNORDERED: ZF,PF,CF ← 111;
    GREATER_THAN: ZF,PF,CF ← 000;
    LESS_THAN: ZF,PF,CF ← 001;
    EQUAL: ZF,PF,CF ← 100;
ESAC;
OF, AF, SF ← 0; }
```

Intel C/C++ Compiler Intrinsic Equivalent

VCOMISS int _mm_comi_round_ss(__m128 a, __m128 b, int imm, int sae);
VCOMISS int _mm_comieq_ss (__m128 a, __m128 b)
VCOMISS int _mm_comilt_ss (__m128 a, __m128 b)
VCOMISS int _mm_comile_ss (__m128 a, __m128 b)
VCOMISS int _mm_comigt_ss (__m128 a, __m128 b)
VCOMISS int _mm_comige_ss (__m128 a, __m128 b)
VCOMISS int _mm_comineq_ss (__m128 a, __m128 b)

SIMD Floating-Point Exceptions

Invalid (if SNaN or QNaN operands), Denormal.

Other Exceptions

VEX-encoded instructions, see Exceptions Type 3;

EVEX-encoded instructions, see Exceptions Type E3NF.

#UD	If VEX.vvvv != 1111B or EVEX.vvvv != 1111B.

CPUID—CPU Identification

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

Instruction Operand Encoding

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

Description

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

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

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

Table 3-8 shows information returned, depending on the initial value loaded into the EAX register.

Two types of information are returned: basic and extended function information. If a value entered for CPUID.EAX is higher than the maximum input value for basic or extended function for that processor then the data for the highest basic information leaf is returned. For example, using the Intel Core i7 processor, the following is true:

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

If a value entered for CPUID.EAX is less than or equal to the maximum input value and the leaf is not supported on that processor then 0 is returned in all the registers.

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

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

See also:

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

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

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

Table 3-8. Information Returned by CPUID Instruction

Initial EAX Value	Information Provided about the Processor	
	Basic CPUID Information	
0H	EAX	Maximum Input Value for Basic CPUID Information.
	EBX	"Genu"
	ECX	"ntel"
	EDX	"inel"
01H	EAX	Version Information: Type, Family, Model, and Stepping ID (see Figure 3-6).
	EBX	Bits 07 - 00: Brand Index. Bits 15 - 08: CLFLUSH line size (Value ∗ 8 = cache line size in bytes; used also by CLFLUSHOPT). Bits 23 - 16: Maximum number of addressable IDs for logical processors in this physical package*. Bits 31 - 24: Initial APIC ID.
	ECX	Feature Information (see Figure 3-7 and Table 3-10).
	EDX	Feature Information (see Figure 3-8 and Table 3-11).
		NOTES: * The nearest power-of-2 integer that is not smaller than EBX[23:16] is the number of unique initial APIC IDs reserved for addressing different logical processors in a physical package. This field is only valid if CPUID.1.EDX.HTT[bit 28]= 1.
02H	EAX	Cache and TLB Information (see Table 3-12).
	EBX	Cache and TLB Information.
	ECX	Cache and TLB Information.
	EDX	Cache and TLB Information.
03H	EAX	Reserved.
	EBX	Reserved.
	ECX	Bits 00 - 31 of 96 bit processor serial number. (Available in Pentium III processor only; otherwise, the value in this register is reserved.)
	EDX	Bits 32 - 63 of 96 bit processor serial number. (Available in Pentium III processor only; otherwise, the value in this register is reserved.)
		NOTES: Processor serial number (PSN) is not supported in the Pentium 4 processor or later. On all models, use the PSN flag (returned using CPUID) to check for PSN support before accessing the feature.
CPUID leaves above 2 and below 80000000H are visible only when IA32_MISC_ENABLE[bit 22] has its default value of 0.		
	Deterministic Cache Parameters Leaf	
04H		**NOTES:** Leaf 04H output depends on the initial value in ECX.* See also: "INPUT EAX = 04H: Returns Deterministic Cache Parameters for Each Level" on page 216.
	EAX	Bits 04 - 00: Cache Type Field. 0 = Null - No more caches. 1 = Data Cache. 2 = Instruction Cache. 3 = Unified Cache. 4-31 = Reserved.

Table 3-8. Information Returned by CPUID Instruction (Contd.)

Initial EAX Value	Information Provided about the Processor	
		Bits 07 - 05: Cache Level (starts at 1). Bit 08: Self Initializing cache level (does not need SW initialization). Bit 09: Fully Associative cache. Bits 13 - 10: Reserved. Bits 25 - 14: Maximum number of addressable IDs for logical processors sharing this cache**, ***. Bits 31 - 26: Maximum number of addressable IDs for processor cores in the physical package**, ****, *****.
	EBX	Bits 11 - 00: L = System Coherency Line Size**. Bits 21 - 12: P = Physical Line partitions**. Bits 31 - 22: W = Ways of associativity**.
	ECX	Bits 31-00: S = Number of Sets**.
	EDX	Bit 00: Write-Back Invalidate/Invalidate. 0 = WBINVD/INVD from threads sharing this cache acts upon lower level caches for threads sharing this cache. 1 = WBINVD/INVD is not guaranteed to act upon lower level caches of non-originating threads sharing this cache. Bit 01: Cache Inclusiveness. 0 = Cache is not inclusive of lower cache levels. 1 = Cache is inclusive of lower cache levels. Bit 02: Complex Cache Indexing. 0 = Direct mapped cache. 1 = A complex function is used to index the cache, potentially using all address bits. Bits 31 - 03: Reserved = 0. **NOTES:** * If ECX contains an invalid sub leaf index, EAX/EBX/ECX/EDX return 0. Sub-leaf index n+1 is invalid if sub-leaf n returns EAX[4:0] as 0. ** Add one to the return value to get the result. ***The nearest power-of-2 integer that is not smaller than (1 + EAX[25:14]) is the number of unique initial APIC IDs reserved for addressing different logical processors sharing this cache. **** The nearest power-of-2 integer that is not smaller than (1 + EAX[31:26]) is the number of unique Core_IDs reserved for addressing different processor cores in a physical package. Core ID is a subset of bits of the initial APIC ID. ***** The returned value is constant for valid initial values in ECX. Valid ECX values start from 0.
	MONITOR/MWAIT Leaf	
05H	EAX	Bits 15 - 00: Smallest monitor-line size in bytes (default is processor's monitor granularity). Bits 31 - 16: Reserved = 0.
	EBX	Bits 15 - 00: Largest monitor-line size in bytes (default is processor's monitor granularity). Bits 31 - 16: Reserved = 0.
	ECX	Bit 00: Enumeration of Monitor-Mwait extensions (beyond EAX and EBX registers) supported. Bit 01: Supports treating interrupts as break-event for MWAIT, even when interrupts disabled. Bits 31 - 02: Reserved.

Table 3-8. Information Returned by CPUID Instruction (Contd.)

Initial EAX Value	Information Provided about the Processor	
	EDX	Bits 03 - 00: Number of C0* sub C-states supported using MWAIT. Bits 07 - 04: Number of C1* sub C-states supported using MWAIT. Bits 11 - 08: Number of C2* sub C-states supported using MWAIT. Bits 15 - 12: Number of C3* sub C-states supported using MWAIT. Bits 19 - 16: Number of C4* sub C-states supported using MWAIT. Bits 23 - 20: Number of C5* sub C-states supported using MWAIT. Bits 27 - 24: Number of C6* sub C-states supported using MWAIT. Bits 31 - 28: Number of C7* sub C-states supported using MWAIT. **NOTE:** * The definition of C0 through C7 states for MWAIT extension are processor-specific C-states, not ACPI C-states.
Thermal and Power Management Leaf		
06H	EAX	Bit 00: Digital temperature sensor is supported if set. Bit 01: Intel Turbo Boost Technology available (see description of IA32_MISC_ENABLE[38]). Bit 02: ARAT. APIC-Timer-always-running feature is supported if set. Bit 03: Reserved. Bit 04: PLN. Power limit notification controls are supported if set. Bit 05: ECMD. Clock modulation duty cycle extension is supported if set. Bit 06: PTM. Package thermal management is supported if set. Bit 07: HWP. HWP base registers (IA32_PM_ENABLE[bit 0], IA32_HWP_CAPABILITIES, IA32_HWP_REQUEST, IA32_HWP_STATUS) are supported if set. Bit 08: HWP_Notification. IA32_HWP_INTERRUPT MSR is supported if set. Bit 09: HWP_Activity_Window. IA32_HWP_REQUEST[bits 41:32] is supported if set. Bit 10: HWP_Energy_Performance_Preference. IA32_HWP_REQUEST[bits 31:24] is supported if set. Bit 11: HWP_Package_Level_Request. IA32_HWP_REQUEST_PKG MSR is supported if set. Bit 12: Reserved. Bit 13: HDC. HDC base registers IA32_PKG_HDC_CTL, IA32_PM_CTL1, IA32_THREAD_STALL MSRs are supported if set. Bit 14: Intel® Turbo Boost Max Technology 3.0 available. Bits 31 - 15: Reserved.
	EBX	Bits 03 - 00: Number of Interrupt Thresholds in Digital Thermal Sensor. Bits 31 - 04: Reserved.
	ECX	Bit 00: Hardware Coordination Feedback Capability (Presence of IA32_MPERF and IA32_APERF). The capability to provide a measure of delivered processor performance (since last reset of the counters), as a percentage of the expected processor performance when running at the TSC frequency. Bits 02 - 01: Reserved = 0. Bit 03: The processor supports performance-energy bias preference if CPUID.06H:ECX.SETBH[bit 3] is set and it also implies the presence of a new architectural MSR called IA32_ENERGY_PERF_BIAS (1B0H). Bits 31 - 04: Reserved = 0.
	EDX	Reserved = 0.

Table 3-8. Information Returned by CPUID Instruction (Contd.)

Initial EAX Value	Information Provided about the Processor
Structured Extended Feature Flags Enumeration Leaf (Output depends on ECX input value)	
07H	Sub-leaf 0 (Input ECX = 0). *
	EAX: Bits 31 - 00: Reports the maximum input value for supported leaf 7 sub-leaves.
	EBX: Bit 00: FSGSBASE. Supports RDFSBASE/RDGSBASE/WRFSBASE/WRGSBASE if 1. Bit 01: IA32_TSC_ADJUST MSR is supported if 1. Bit 02: SGX. Supports Intel® Software Guard Extensions (Intel® SGX Extensions) if 1. Bit 03: BMI1. Bit 04: HLE. Bit 05: AVX2. Bit 06: FDP_EXCPTN_ONLY. x87 FPU Data Pointer updated only on x87 exceptions if 1. Bit 07: SMEP. Supports Supervisor-Mode Execution Prevention if 1. Bit 08: BMI2. Bit 09: Supports Enhanced REP MOVSB/STOSB if 1. Bit 10: INVPCID. If 1, supports INVPCID instruction for system software that manages process-context identifiers. Bit 11: RTM. Bit 12: RDT-M. Supports Intel® Resource Director Technology (Intel® RDT) Monitoring capability if 1. Bit 13: Deprecates FPU CS and FPU DS values if 1. Bit 14: MPX. Supports Intel® Memory Protection Extensions if 1. Bit 15: RDT-A. Supports Intel® Resource Director Technology (Intel® RDT) Allocation capability if 1. Bits 17:16: Reserved. Bit 18: RDSEED. Bit 19: ADX. Bit 20: SMAP. Supports Supervisor-Mode Access Prevention (and the CLAC/STAC instructions) if 1. Bits 22 - 21: Reserved. Bit 23: CLFLUSHOPT. Bit 24: CLWB. Bit 25: Intel Processor Trace. Bits 28 - 26: Reserved. Bit 29: SHA. supports Intel® Secure Hash Algorithm Extensions (Intel® SHA Extensions) if 1. Bits 31 - 30: Reserved.
	ECX: Bit 00: PREFETCHWT1. Bit 01: Reserved. Bit 02: UMIP. Supports user-mode instruction prevention if 1. Bit 03: PKU. Supports protection keys for user-mode pages if 1. Bit 04: OSPKE. If 1, OS has set CR4.PKE to enable protection keys (and the RDPKRU/WRPKRU instructions). Bits 16 - 5: Reserved. Bits 21 - 17: The value of MAWAU used by the BNDLDX and BNDSTX instructions in 64-bit mode. Bit 22: RDPID. Supports Read Processor ID if 1. Bits 29 - 23: Reserved. Bit 30: SGX_LC. Supports SGX Launch Configuration if 1. Bit 31: Reserved.
	EDX: Reserved.
	NOTE: * If ECX contains an invalid sub-leaf index, EAX/EBX/ECX/EDX return 0. Sub-leaf index n is invalid if n exceeds the value that sub-leaf 0 returns in EAX.

Table 3-8. Information Returned by CPUID Instruction (Contd.)

Initial EAX Value	Information Provided about the Processor	
	Direct Cache Access Information Leaf	
09H	EAX	Value of bits [31:0] of IA32_PLATFORM_DCA_CAP MSR (address 1F8H).
	EBX	Reserved.
	ECX	Reserved.
	EDX	Reserved.
	Architectural Performance Monitoring Leaf	
0AH	EAX	Bits 07 - 00: Version ID of architectural performance monitoring. Bits 15 - 08: Number of general-purpose performance monitoring counter per logical processor. Bits 23 - 16: Bit width of general-purpose, performance monitoring counter. Bits 31 - 24: Length of EBX bit vector to enumerate architectural performance monitoring events.
	EBX	Bit 00: Core cycle event not available if 1. Bit 01: Instruction retired event not available if 1. Bit 02: Reference cycles event not available if 1. Bit 03: Last-level cache reference event not available if 1. Bit 04: Last-level cache misses event not available if 1. Bit 05: Branch instruction retired event not available if 1. Bit 06: Branch mispredict retired event not available if 1. Bits 31 - 07: Reserved = 0.
	ECX	Reserved = 0.
	EDX	Bits 04 - 00: Number of fixed-function performance counters (if Version ID > 1). Bits 12 - 05: Bit width of fixed-function performance counters (if Version ID > 1). Bits 14 - 13: Reserved = 0. Bit 15: AnyThread deprecation. Bits 31 - 16: Reserved = 0.
	Extended Topology Enumeration Leaf	
0BH		**NOTES:** Most of Leaf 0BH output depends on the initial value in ECX. The EDX output of leaf 0BH is always valid and does not vary with input value in ECX. Output value in ECX[7:0] always equals input value in ECX[7:0]. For sub-leaves that return an invalid level-type of 0 in ECX[15:8]; EAX and EBX will return 0. If an input value n in ECX returns the invalid level-type of 0 in ECX[15:8], other input values with ECX > n also return 0 in ECX[15:8].
	EAX	Bits 04 - 00: Number of bits to shift right on x2APIC ID to get a unique topology ID of the next level type*. All logical processors with the same next level ID share current level. Bits 31 - 05: Reserved.
	EBX	Bits 15 - 00: Number of logical processors at this level type. The number reflects configuration as shipped by Intel**. Bits 31- 16: Reserved.
	ECX	Bits 07 - 00: Level number. Same value in ECX input. Bits 15 - 08: Level type***. Bits 31 - 16: Reserved.
	EDX	Bits 31- 00: x2APIC ID the current logical processor.
		NOTES: * Software should use this field (EAX[4:0]) to enumerate processor topology of the system.

Table 3-8. Information Returned by CPUID Instruction (Contd.)

Initial EAX Value	Information Provided about the Processor	
		** Software must not use EBX[15:0] to enumerate processor topology of the system. This value in this field (EBX[15:0]) is only intended for display/diagnostic purposes. The actual number of logical processors available to BIOS/OS/Applications may be different from the value of EBX[15:0], depending on software and platform hardware configurations. *** The value of the "level type" field is not related to level numbers in any way, higher "level type" values do not mean higher levels. Level type field has the following encoding: 0: Invalid. 1: SMT. 2: Core. 3-255: Reserved.
	Processor Extended State Enumeration Main Leaf (EAX = 0DH, ECX = 0)	
0DH	**NOTES:** Leaf 0DH main leaf (ECX = 0).	
	EAX	Bits 31 - 00: Reports the supported bits of the lower 32 bits of XCR0. XCR0[n] can be set to 1 only if EAX[n] is 1. Bit 00: x87 state. Bit 01: SSE state. Bit 02: AVX state. Bits 04 - 03: MPX state. Bits 07 - 05: AVX-512 state. Bit 08: Used for IA32_XSS. Bit 09: PKRU state. Bits 31 - 10: Reserved.
	EBX	Bits 31 - 00: Maximum size (bytes, from the beginning of the XSAVE/XRSTOR save area) required by enabled features in XCR0. May be different than ECX if some features at the end of the XSAVE save area are not enabled.
	ECX	Bit 31 - 00: Maximum size (bytes, from the beginning of the XSAVE/XRSTOR save area) of the XSAVE/XRSTOR save area required by all supported features in the processor, i.e., all the valid bit fields in XCR0.
	EDX	Bit 31 - 00: Reports the supported bits of the upper 32 bits of XCR0. XCR0[n+32] can be set to 1 only if EDX[n] is 1. Bits 31 - 00: Reserved.
	Processor Extended State Enumeration Sub-leaf (EAX = 0DH, ECX = 1)	
0DH	EAX	Bit 00: XSAVEOPT is available. Bit 01: Supports XSAVEC and the compacted form of XRSTOR if set. Bit 02: Supports XGETBV with ECX = 1 if set. Bit 03: Supports XSAVES/XRSTORS and IA32_XSS if set. Bits 31 - 04: Reserved.
	EBX	Bits 31 - 00: The size in bytes of the XSAVE area containing all states enabled by XCR0 | IA32_XSS.
	ECX	Bits 31 - 00: Reports the supported bits of the lower 32 bits of the IA32_XSS MSR. IA32_XSS[n] can be set to 1 only if ECX[n] is 1. Bits 07 - 00: Used for XCR0. Bit 08: PT state. Bit 09: Used for XCR0. Bits 31 - 10: Reserved.
	EDX	Bits 31 - 00: Reports the supported bits of the upper 32 bits of the IA32_XSS MSR. IA32_XSS[n+32] can be set to 1 only if EDX[n] is 1. Bits 31 - 00: Reserved.

Table 3-8. Information Returned by CPUID Instruction (Contd.)

Initial EAX Value	Information Provided about the Processor	
	Processor Extended State Enumeration Sub-leaves (EAX = 0DH, ECX = n, n > 1)	
0DH		**NOTES:** Leaf 0DH output depends on the initial value in ECX. Each sub-leaf index (starting at position 2) is supported if it corresponds to a supported bit in either the XCR0 register or the IA32_XSS MSR. * If ECX contains an invalid sub-leaf index, EAX/EBX/ECX/EDX return 0. Sub-leaf n (0 ≤ n ≤ 31) is invalid if sub-leaf 0 returns 0 in EAX[n] and sub-leaf 1 returns 0 in ECX[n]. Sub-leaf n (32 ≤ n ≤ 63) is invalid if sub-leaf 0 returns 0 in EDX[n-32] and sub-leaf 1 returns 0 in EDX[n-32].
	EAX	Bits 31 - 0: The size in bytes (from the offset specified in EBX) of the save area for an extended state feature associated with a valid sub-leaf index, *n*.
	EBX	Bits 31 - 0: The offset in bytes of this extended state component's save area from the beginning of the XSAVE/XRSTOR area. This field reports 0 if the sub-leaf index, n, does not map to a valid bit in the XCR0 register*.
	ECX	Bit 00 is set if the bit n (corresponding to the sub-leaf index) is supported in the IA32_XSS MSR; it is clear if bit n is instead supported in XCR0. Bit 01 is set if, when the compacted format of an XSAVE area is used, this extended state component located on the next 64-byte boundary following the preceding state component (otherwise, it is located immediately following the preceding state component). Bits 31 - 02 are reserved. This field reports 0 if the sub-leaf index, n, is invalid*.
	EDX	This field reports 0 if the sub-leaf index, *n*, is invalid*; otherwise it is reserved.
	Intel Resource Director Technology (Intel RDT) Monitoring Enumeration Sub-leaf (EAX = 0FH, ECX = 0)	
0FH		**NOTES:** Leaf 0FH output depends on the initial value in ECX. Sub-leaf index 0 reports valid resource type starting at bit position 1 of EDX.
	EAX	Reserved.
	EBX	Bits 31 - 00: Maximum range (zero-based) of RMID within this physical processor of all types.
	ECX	Reserved.
	EDX	Bit 00: Reserved. Bit 01: Supports L3 Cache Intel RDT Monitoring if 1. Bits 31 - 02: Reserved.
	L3 Cache Intel RDT Monitoring Capability Enumeration Sub-leaf (EAX = 0FH, ECX = 1)	
0FH		**NOTES:** Leaf 0FH output depends on the initial value in ECX.
	EAX	Reserved.
	EBX	Bits 31 - 00: Conversion factor from reported IA32_QM_CTR value to occupancy metric (bytes).
	ECX	Maximum range (zero-based) of RMID of this resource type.
	EDX	Bit 00: Supports L3 occupancy monitoring if 1. Bit 01: Supports L3 Total Bandwidth monitoring if 1. Bit 02: Supports L3 Local Bandwidth monitoring if 1. Bits 31 - 03: Reserved.

Table 3-8. Information Returned by CPUID Instruction (Contd.)

Initial EAX Value	Information Provided about the Processor
Intel Resource Director Technology (Intel RDT) Allocation Enumeration Sub-leaf (EAX = 10H, ECX = 0)	
10H	**NOTES:** Leaf 10H output depends on the initial value in ECX. Sub-leaf index 0 reports valid resource identification (ResID) starting at bit position 1 of EBX. EAX: Reserved. EBX: Bit 00: Reserved. Bit 01: Supports L3 Cache Allocation Technology if 1. Bit 02: Supports L2 Cache Allocation Technology if 1. Bit 03: Supports Memory Bandwidth Allocation if 1. Bits 31 - 04: Reserved. ECX: Reserved. EDX: Reserved.
L3 Cache Allocation Technology Enumeration Sub-leaf (EAX = 10H, ECX = ResID =1)	
10H	**NOTES:** Leaf 10H output depends on the initial value in ECX. EAX: Bits 04 - 00: Length of the capacity bit mask for the corresponding ResID using minus-one notation. Bits 31 - 05: Reserved. EBX: Bits 31 - 00: Bit-granular map of isolation/contention of allocation units. ECX: Bits 01- 00: Reserved. Bit 02: Code and Data Prioritization Technology supported if 1. Bits 31 - 03: Reserved. EDX: Bits 15 - 00: Highest COS number supported for this ResID. Bits 31 - 16: Reserved.
L2 Cache Allocation Technology Enumeration Sub-leaf (EAX = 10H, ECX = ResID =2)	
10H	**NOTES:** Leaf 10H output depends on the initial value in ECX. EAX: Bits 04 - 00: Length of the capacity bit mask for the corresponding ResID using minus-one notation. Bits 31 - 05: Reserved. EBX: Bits 31 - 00: Bit-granular map of isolation/contention of allocation units. ECX: Bits 31 - 00: Reserved. EDX: Bits 15 - 00: Highest COS number supported for this ResID. Bits 31 - 16: Reserved.
Memory Bandwidth Allocation Enumeration Sub-leaf (EAX = 10H, ECX = ResID =3)	
10H	**NOTES:** Leaf 10H output depends on the initial value in ECX. EAX: Bits 11 - 00: Reports the maximum MBA throttling value supported for the corresponding ResID using minus-one notation. Bits 31 - 12: Reserved. EBX: Bits 31 - 00: Reserved. ECX: Bits 01 - 00: Reserved. Bit 02: Reports whether the response of the delay values is linear. Bits 31 - 03: Reserved.

Table 3-8. Information Returned by CPUID Instruction (Contd.)

Initial EAX Value	Information Provided about the Processor	
	EDX	Bits 15 - 00: Highest COS number supported for this ResID. Bits 31 - 16: Reserved.
	Intel SGX Capability Enumeration Leaf, sub-leaf 0 (EAX = 12H, ECX = 0)	
12H		**NOTES:** Leaf 12H sub-leaf 0 (ECX = 0) is supported if CPUID.(EAX=07H, ECX=0H):EBX[SGX] = 1.
	EAX	Bit 00: SGX1. If 1, Indicates Intel SGX supports the collection of SGX1 leaf functions. Bit 01: SGX2. If 1, Indicates Intel SGX supports the collection of SGX2 leaf functions. Bit 31 - 02: Reserved.
	EBX	Bit 31 - 00: MISCSELECT. Bit vector of supported extended SGX features.
	ECX	Bit 31 - 00: Reserved.
	EDX	Bit 07 - 00: MaxEnclaveSize_Not64. The maximum supported enclave size in non-64-bit mode is 2^(EDX[7:0]). Bit 15 - 08: MaxEnclaveSize_64. The maximum supported enclave size in 64-bit mode is 2^(EDX[15:8]). Bits 31 - 16: Reserved.
	Intel SGX Attributes Enumeration Leaf, sub-leaf 1 (EAX = 12H, ECX = 1)	
12H		**NOTES:** Leaf 12H sub-leaf 1 (ECX = 1) is supported if CPUID.(EAX=07H, ECX=0H):EBX[SGX] = 1.
	EAX	Bit 31 - 00: Reports the valid bits of SECS.ATTRIBUTES[31:0] that software can set with ECREATE.
	EBX	Bit 31 - 00: Reports the valid bits of SECS.ATTRIBUTES[63:32] that software can set with ECREATE.
	ECX	Bit 31 - 00: Reports the valid bits of SECS.ATTRIBUTES[95:64] that software can set with ECREATE.
	EDX	Bit 31 - 00: Reports the valid bits of SECS.ATTRIBUTES[127:96] that software can set with ECREATE.
	Intel SGX EPC Enumeration Leaf, sub-leaves (EAX = 12H, ECX = 2 or higher)	
12H		**NOTES:** Leaf 12H sub-leaf 2 or higher (ECX >= 2) is supported if CPUID.(EAX=07H, ECX=0H):EBX[SGX] = 1. For sub-leaves (ECX = 2 or higher), definition of EDX,ECX,EBX,EAX[31:4] depends on the sub-leaf type listed below.
	EAX	Bit 03 - 00: Sub-leaf Type 0000b: Indicates this sub-leaf is invalid. 0001b: This sub-leaf enumerates an EPC section. EBX:EAX and EDX:ECX provide information on the Enclave Page Cache (EPC) section. All other type encodings are reserved.
	Type	0000b. This sub-leaf is invalid. EDX:ECX:EBX:EAX return 0.

Table 3-8. Information Returned by CPUID Instruction (Contd.)

Initial EAX Value	Information Provided about the Processor	
	Type	0001b. This sub-leaf enumerates an EPC sections with EDX:ECX, EBX:EAX defined as follows. EAX[11:04]: Reserved (enumerate 0). EAX[31:12]: Bits 31:12 of the physical address of the base of the EPC section. EBX[19:00]: Bits 51:32 of the physical address of the base of the EPC section. EBX[31:20]: Reserved. ECX[03:00]: EPC section property encoding defined as follows: If EAX[3:0] 0000b, then all bits of the EDX:ECX pair are enumerated as 0. If EAX[3:0] 0001b, then this section has confidentiality and integrity protection. All other encodings are reserved. ECX[11:04]: Reserved (enumerate 0). ECX[31:12]: Bits 31:12 of the size of the corresponding EPC section within the Processor Reserved Memory. EDX[19:00]: Bits 51:32 of the size of the corresponding EPC section within the Processor Reserved Memory. EDX[31:20]: Reserved.
	Intel Processor Trace Enumeration Main Leaf (EAX = 14H, ECX = 0)	
14H		**NOTES:** Leaf 14H main leaf (ECX = 0).
	EAX	Bits 31 - 00: Reports the maximum sub-leaf supported in leaf 14H.
	EBX	Bit 00: If 1, indicates that IA32_RTIT_CTL.CR3Filter can be set to 1, and that IA32_RTIT_CR3_MATCH MSR can be accessed. Bit 01: If 1, indicates support of Configurable PSB and Cycle-Accurate Mode. Bit 02: If 1, indicates support of IP Filtering, TraceStop filtering, and preservation of Intel PT MSRs across warm reset. Bit 03: If 1, indicates support of MTC timing packet and suppression of COFI-based packets. Bit 04: If 1, indicates support of PTWRITE. Writes can set IA32_RTIT_CTL[12] (PTWEn) and IA32_RTIT_CTL[5] (FUPonPTW), and PTWRITE can generate packets. Bit 05: If 1, indicates support of Power Event Trace. Writes can set IA32_RTIT_CTL[4] (PwrEvtEn), enabling Power Event Trace packet generation. Bit 31 - 06: Reserved.
	ECX	Bit 00: If 1, Tracing can be enabled with IA32_RTIT_CTL.ToPA = 1, hence utilizing the ToPA output scheme; IA32_RTIT_OUTPUT_BASE and IA32_RTIT_OUTPUT_MASK_PTRS MSRs can be accessed. Bit 01: If 1, ToPA tables can hold any number of output entries, up to the maximum allowed by the MaskOrTableOffset field of IA32_RTIT_OUTPUT_MASK_PTRS. Bit 02: If 1, indicates support of Single-Range Output scheme. Bit 03: If 1, indicates support of output to Trace Transport subsystem. Bit 30 - 04: Reserved. Bit 31: If 1, generated packets which contain IP payloads have LIP values, which include the CS base component.
	EDX	Bits 31 - 00: Reserved.
	Intel Processor Trace Enumeration Sub-leaf (EAX = 14H, ECX = 1)	
14H	EAX	Bits 02 - 00: Number of configurable Address Ranges for filtering. Bits 15 - 03: Reserved. Bits 31 - 16: Bitmap of supported MTC period encodings.
	EBX	Bits 15 - 00: Bitmap of supported Cycle Threshold value encodings. Bit 31 - 16: Bitmap of supported Configurable PSB frequency encodings.
	ECX	Bits 31 - 00: Reserved.

Table 3-8. Information Returned by CPUID Instruction (Contd.)

Initial EAX Value	Information Provided about the Processor	
	EDX	Bits 31 - 00: Reserved.
	Time Stamp Counter and Nominal Core Crystal Clock Information Leaf	
15H		**NOTES:** If EBX[31:0] is 0, the TSC/"core crystal clock" ratio is not enumerated. EBX[31:0]/EAX[31:0] indicates the ratio of the TSC frequency and the core crystal clock frequency. If ECX is 0, the nominal core crystal clock frequency is not enumerated. "TSC frequency" = "core crystal clock frequency" * EBX/EAX. The core crystal clock may differ from the reference clock, bus clock, or core clock frequencies.
	EAX	Bits 31 - 00: An unsigned integer which is the denominator of the TSC/"core crystal clock" ratio.
	EBX	Bits 31 - 00: An unsigned integer which is the numerator of the TSC/"core crystal clock" ratio.
	ECX	Bits 31 - 00: An unsigned integer which is the nominal frequency of the core crystal clock in Hz.
	EDX	Bits 31 - 00: Reserved = 0.
	Processor Frequency Information Leaf	
16H	EAX	Bits 15 - 00: Processor Base Frequency (in MHz). Bits 31 - 16: Reserved =0.
	EBX	Bits 15 - 00: Maximum Frequency (in MHz). Bits 31 - 16: Reserved = 0.
	ECX	Bits 15 - 00: Bus (Reference) Frequency (in MHz). Bits 31 - 16: Reserved = 0.
	EDX	Reserved.
		NOTES: * Data is returned from this interface in accordance with the processor's specification and does not reflect actual values. Suitable use of this data includes the display of processor information in like manner to the processor brand string and for determining the appropriate range to use when displaying processor information e.g. frequency history graphs. The returned information should not be used for any other purpose as the returned information does not accurately correlate to information / counters returned by other processor interfaces. While a processor may support the Processor Frequency Information leaf, fields that return a value of zero are not supported.
	System-On-Chip Vendor Attribute Enumeration Main Leaf (EAX = 17H, ECX = 0)	
17H		**NOTES:** Leaf 17H main leaf (ECX = 0). Leaf 17H output depends on the initial value in ECX. Leaf 17H sub-leaves 1 through 3 reports SOC Vendor Brand String. Leaf 17H is valid if MaxSOCID_Index >= 3. Leaf 17H sub-leaves 4 and above are reserved.
	EAX	Bits 31 - 00: MaxSOCID_Index. Reports the maximum input value of supported sub-leaf in leaf 17H.
	EBX	Bits 15 - 00: SOC Vendor ID. Bit 16: IsVendorScheme. If 1, the SOC Vendor ID field is assigned via an industry standard enumeration scheme. Otherwise, the SOC Vendor ID field is assigned by Intel. Bits 31 - 17: Reserved = 0.
	ECX	Bits 31 - 00: Project ID. A unique number an SOC vendor assigns to its SOC projects.
	EDX	Bits 31 - 00: Stepping ID. A unique number within an SOC project that an SOC vendor assigns.

Table 3-8. Information Returned by CPUID Instruction (Contd.)

Initial EAX Value	Information Provided about the Processor	
System-On-Chip Vendor Attribute Enumeration Sub-leaf (EAX = 17H, ECX = 1..3)		
17H	EAX	Bit 31 - 00: SOC Vendor Brand String. UTF-8 encoded string.
	EBX	Bit 31 - 00: SOC Vendor Brand String. UTF-8 encoded string.
	ECX	Bit 31 - 00: SOC Vendor Brand String. UTF-8 encoded string.
	EDX	Bit 31 - 00: SOC Vendor Brand String. UTF-8 encoded string.
		NOTES: Leaf 17H output depends on the initial value in ECX. SOC Vendor Brand String is a UTF-8 encoded string padded with trailing bytes of 00H. The complete SOC Vendor Brand String is constructed by concatenating in ascending order of EAX:EBX:ECX:EDX and from the sub-leaf 1 fragment towards sub-leaf 3.
System-On-Chip Vendor Attribute Enumeration Sub-leaves (EAX = 17H, ECX > MaxSOCID_Index)		
17H		**NOTES:** Leaf 17H output depends on the initial value in ECX.
	EAX	Bits 31 - 00: Reserved = 0.
	EBX	Bits 31 - 00: Reserved = 0.
	ECX	Bits 31 - 00: Reserved = 0.
	EDX	Bits 31 - 00: Reserved = 0.
Deterministic Address Translation Parameters Main Leaf (EAX = 18H, ECX = 0)		
18H		**NOTES:** Each sub-leaf enumerates a different address translation structure. If ECX contains an invalid sub-leaf index, EAX/EBX/ECX/EDX return 0. Sub-leaf index n is invalid if n exceeds the value that sub-leaf 0 returns in EAX. A sub-leaf index is also invalid if EDX[4:0] returns 0. Valid sub-leaves do not need to be contiguous or in any particular order. A valid sub-leaf may be in a higher input ECX value than an invalid sub-leaf or than a valid sub-leaf of a higher or lower-level structure. * Some unified TLBs will allow a single TLB entry to satisfy data read/write and instruction fetches. Others will require separate entries (e.g., one loaded on data read/write and another loaded on an instruction fetch) . Please see the *Intel® 64 and IA-32 Architectures Optimization Reference Manual* for details of a particular product. ** Add one to the return value to get the result.
	EAX	Bits 31 - 00: Reports the maximum input value of supported sub-leaf in leaf 18H.
	EBX	Bit 00: 4K page size entries supported by this structure. Bit 01: 2MB page size entries supported by this structure. Bit 02: 4MB page size entries supported by this structure. Bit 03: 1 GB page size entries supported by this structure. Bits 07 - 04: Reserved. Bits 10 - 08: Partitioning (0: Soft partitioning between the logical processors sharing this structure). Bits 15 - 11: Reserved. Bits 31 - 16: W = Ways of associativity.
	ECX	Bits 31 - 00: S = Number of Sets.

Table 3-8. Information Returned by CPUID Instruction (Contd.)

Initial EAX Value	Information Provided about the Processor	
	EDX	Bits 04 - 00: Translation cache type field. 00000b: Null (indicates this sub-leaf is not valid). 00001b: Data TLB. 00010b: Instruction TLB. 00011b: Unified TLB*. All other encodings are reserved. Bits 07 - 05: Translation cache level (starts at 1). Bit 08: Fully associative structure. Bits 13 - 09: Reserved. Bits 25- 14: Maximum number of addressable IDs for logical processors sharing this translation cache** Bits 31 - 26: Reserved.
	Deterministic Address Translation Parameters Sub-leaf (EAX = 18H, ECX ≥ 1)	
18H		**NOTES:** Each sub-leaf enumerates a different address translation structure. If ECX contains an invalid sub-leaf index, EAX/EBX/ECX/EDX return 0. Sub-leaf index n is invalid if n exceeds the value that sub-leaf 0 returns in EAX. A sub-leaf index is also invalid if EDX[4:0] returns 0. Valid sub-leaves do not need to be contiguous or in any particular order. A valid sub-leaf may be in a higher input ECX value than an invalid sub-leaf or than a valid sub-leaf of a higher or lower-level structure. * Some unified TLBs will allow a single TLB entry to satisfy data read/write and instruction fetches. Others will require separate entries (e.g., one loaded on data read/write and another loaded on an instruction fetch) . Please see the *Intel® 64 and IA-32 Architectures Optimization Reference Manual* for details of a particular product. ** Add one to the return value to get the result.
	EAX	Bits 31 - 00: Reserved.
	EBX	Bit 00: 4K page size entries supported by this structure. Bit 01: 2MB page size entries supported by this structure. Bit 02: 4MB page size entries supported by this structure. Bit 03: 1 GB page size entries supported by this structure. Bits 07 - 04: Reserved. Bits 10 - 08: Partitioning (0: Soft partitioning between the logical processors sharing this structure). Bits 15 - 11: Reserved. Bits 31 - 16: W = Ways of associativity.
	ECX	Bits 31 - 00: S = Number of Sets.
	EDX	Bits 04 - 00: Translation cache type field. 0000b: Null (indicates this sub-leaf is not valid). 0001b: Data TLB. 0010b: Instruction TLB. 0011b: Unified TLB*. All other encodings are reserved. Bits 07 - 05: Translation cache level (starts at 1). Bit 08: Fully associative structure. Bits 13 - 09: Reserved. Bits 25- 14: Maximum number of addressable IDs for logical processors sharing this translation cache** Bits 31 - 26: Reserved.
	Unimplemented CPUID Leaf Functions	
40000000H - 4FFFFFFFH		Invalid. No existing or future CPU will return processor identification or feature information if the initial EAX value is in the range 40000000H to 4FFFFFFFH.

Table 3-8. Information Returned by CPUID Instruction (Contd.)

Initial EAX Value	Information Provided about the Processor	
	Extended Function CPUID Information	
80000000H	EAX	Maximum Input Value for Extended Function CPUID Information.
	EBX	Reserved.
	ECX	Reserved.
	EDX	Reserved.
80000001H	EAX	Extended Processor Signature and Feature Bits.
	EBX	Reserved.
	ECX	Bit 00: LAHF/SAHF available in 64-bit mode. Bits 04 - 01: Reserved. Bit 05: LZCNT. Bits 07 - 06: Reserved. Bit 08: PREFETCHW. Bits 31 - 09: Reserved.
	EDX	Bits 10 - 00: Reserved. Bit 11: SYSCALL/SYSRET available in 64-bit mode. Bits 19 - 12: Reserved = 0. Bit 20: Execute Disable Bit available. Bits 25 - 21: Reserved = 0. Bit 26: 1-GByte pages are available if 1. Bit 27: RDTSCP and IA32_TSC_AUX are available if 1. Bit 28: Reserved = 0. Bit 29: Intel® 64 Architecture available if 1. Bits 31 - 30: Reserved = 0.
80000002H	EAX EBX ECX EDX	Processor Brand String. Processor Brand String Continued. Processor Brand String Continued. Processor Brand String Continued.
80000003H	EAX EBX ECX EDX	Processor Brand String Continued. Processor Brand String Continued. Processor Brand String Continued. Processor Brand String Continued.
80000004H	EAX EBX ECX EDX	Processor Brand String Continued. Processor Brand String Continued. Processor Brand String Continued. Processor Brand String Continued.
80000005H	EAX EBX ECX EDX	Reserved = 0. Reserved = 0. Reserved = 0. Reserved = 0.
80000006H	EAX EBX	Reserved = 0. Reserved = 0.
	ECX	Bits 07 - 00: Cache Line size in bytes. Bits 11 - 08: Reserved. Bits 15 - 12: L2 Associativity field *. Bits 31 - 16: Cache size in 1K units.
	EDX	Reserved = 0.

Table 3-8. Information Returned by CPUID Instruction (Contd.)

Initial EAX Value	Information Provided about the Processor	
		NOTES: * L2 associativity field encodings: 00H - Disabled. 01H - Direct mapped. 02H - 2-way. 04H - 4-way. 06H - 8-way. 08H - 16-way. 0FH - Fully associative.
80000007H	EAX EBX ECX EDX	Reserved = 0. Reserved = 0. Reserved = 0. Bits 07 - 00: Reserved = 0. Bit 08: Invariant TSC available if 1. Bits 31 - 09: Reserved = 0.
80000008H	EAX EBX ECX EDX	Linear/Physical Address size. Bits 07 - 00: #Physical Address Bits*. Bits 15 - 08: #Linear Address Bits. Bits 31 - 16: Reserved = 0. Reserved = 0. Reserved = 0. Reserved = 0. **NOTES:** * If CPUID.80000008H:EAX[7:0] is supported, the maximum physical address number supported should come from this field.

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

When CPUID executes with EAX set to 0, the processor returns the highest value the CPUID recognizes for returning basic processor information. The value is returned in the EAX register and is processor specific.

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

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

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

When CPUID executes with EAX set to 80000000H, the processor returns the highest value the processor recognizes for returning extended processor information. The value is returned in the EAX register and is processor specific.

IA32_BIOS_SIGN_ID Returns Microcode Update Signature

For processors that support the microcode update facility, the IA32_BIOS_SIGN_ID MSR is loaded with the update signature whenever CPUID executes. The signature is returned in the upper DWORD. For details, see Chapter 9 in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*.

INPUT EAX = 01H: Returns Model, Family, Stepping Information

When CPUID executes with EAX set to 01H, version information is returned in EAX (see Figure 3-6). For example: model, family, and processor type for the Intel Xeon processor 5100 series is as follows:

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

See Table 3-9 for available processor type values. Stepping IDs are provided as needed.

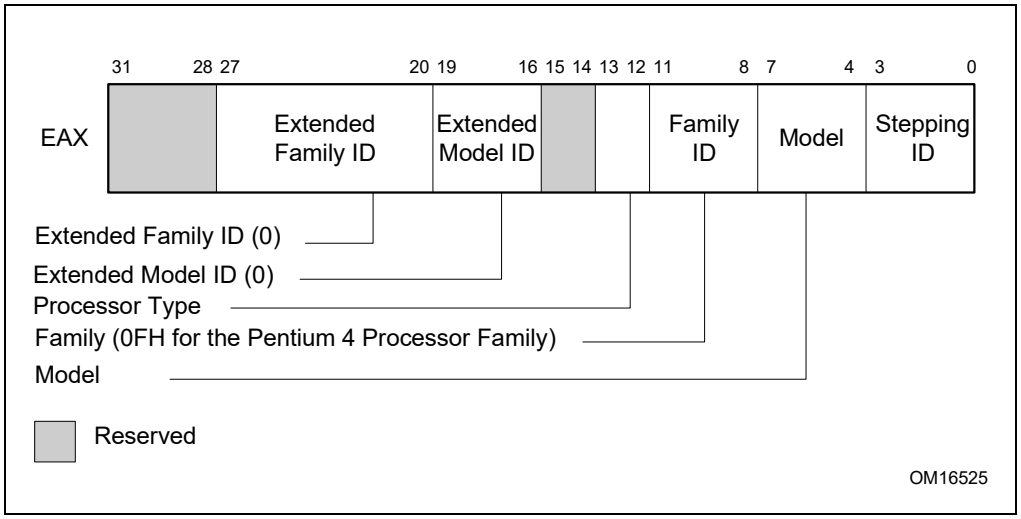


Figure 3-6. Version Information Returned by CPUID in EAX

Table 3-9. Processor Type Field

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

NOTE

See Chapter 19 in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*, for information on identifying earlier IA-32 processors.

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

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

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

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

INPUT EAX = 01H: Returns Additional Information in EBX

When CPUID executes with EAX set to 01H, additional information is returned to the EBX register:

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

INPUT EAX = 01H: Returns Feature Information in ECX and EDX

When CPUID executes with EAX set to 01H, feature information is returned in ECX and EDX.

- Figure 3-7 and Table 3-10 show encodings for ECX.
- Figure 3-8 and Table 3-11 show encodings for EDX.

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

NOTE

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

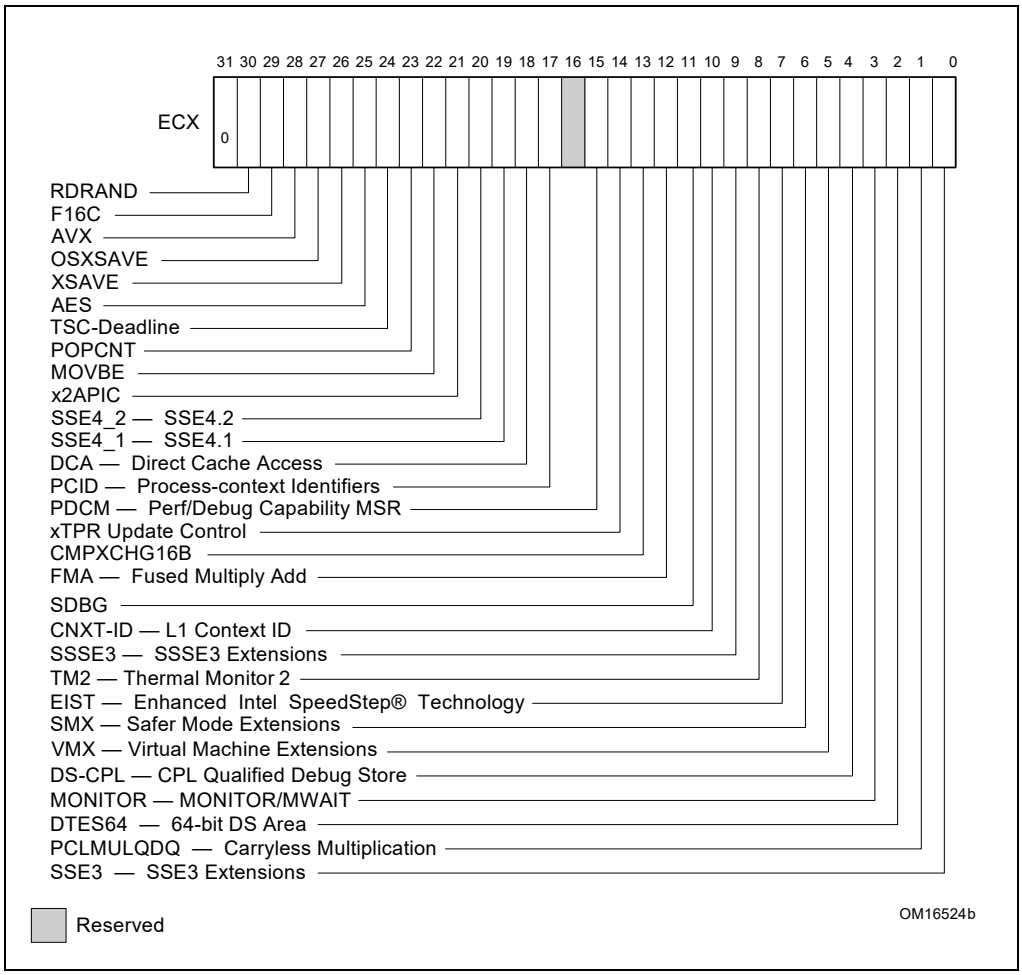


Figure 3-7. Feature Information Returned in the ECX Register

Table 3-10. Feature Information Returned in the ECX Register

Bit #	Mnemonic	Description
0	SSE3	**Streaming SIMD Extensions 3 (SSE3).** A value of 1 indicates the processor supports this technology.
1	PCLMULQDQ	**PCLMULQDQ.** A value of 1 indicates the processor supports the PCLMULQDQ instruction.
2	DTES64	**64-bit DS Area.** A value of 1 indicates the processor supports DS area using 64-bit layout.
3	MONITOR	**MONITOR/MWAIT.** A value of 1 indicates the processor supports this feature.
4	DS-CPL	**CPL Qualified Debug Store.** A value of 1 indicates the processor supports the extensions to the Debug Store feature to allow for branch message storage qualified by CPL.
5	VMX	**Virtual Machine Extensions.** A value of 1 indicates that the processor supports this technology.
6	SMX	**Safer Mode Extensions.** A value of 1 indicates that the processor supports this technology. See Chapter 6, “Safer Mode Extensions Reference”.
7	EIST	**Enhanced Intel SpeedStep® technology.** A value of 1 indicates that the processor supports this technology.
8	TM2	**Thermal Monitor 2.** A value of 1 indicates whether the processor supports this technology.
9	SSSE3	A value of 1 indicates the presence of the Supplemental Streaming SIMD Extensions 3 (SSSE3). A value of 0 indicates the instruction extensions are not present in the processor.

Table 3-10. Feature Information Returned in the ECX Register (Contd.)

Bit #	Mnemonic	Description
10	CNXT-ID	**L1 Context ID.** A value of 1 indicates the L1 data cache mode can be set to either adaptive mode or shared mode. A value of 0 indicates this feature is not supported. See definition of the IA32_MISC_ENABLE MSR Bit 24 (L1 Data Cache Context Mode) for details.
11	SDBG	A value of 1 indicates the processor supports IA32_DEBUG_INTERFACE MSR for silicon debug.
12	FMA	A value of 1 indicates the processor supports FMA extensions using YMM state.
13	CMPXCHG16B	**CMPXCHG16B Available.** A value of 1 indicates that the feature is available. See the "CMPXCHG8B/CMPXCHG16B—Compare and Exchange Bytes" section in this chapter for a description.
14	xTPR Update Control	**xTPR Update Control.** A value of 1 indicates that the processor supports changing IA32_MISC_ENABLE[bit 23].
15	PDCM	**Perfmon and Debug Capability:** A value of 1 indicates the processor supports the performance and debug feature indication MSR IA32_PERF_CAPABILITIES.
16	Reserved	Reserved
17	PCID	**Process-context identifiers.** A value of 1 indicates that the processor supports PCIDs and that software may set CR4.PCIDE to 1.
18	DCA	A value of 1 indicates the processor supports the ability to prefetch data from a memory mapped device.
19	SSE4.1	A value of 1 indicates that the processor supports SSE4.1.
20	SSE4.2	A value of 1 indicates that the processor supports SSE4.2.
21	x2APIC	A value of 1 indicates that the processor supports x2APIC feature.
22	MOVBE	A value of 1 indicates that the processor supports MOVBE instruction.
23	POPCNT	A value of 1 indicates that the processor supports the POPCNT instruction.
24	TSC-Deadline	A value of 1 indicates that the processor's local APIC timer supports one-shot operation using a TSC deadline value.
25	AESNI	A value of 1 indicates that the processor supports the AESNI instruction extensions.
26	XSAVE	A value of 1 indicates that the processor supports the XSAVE/XRSTOR processor extended states feature, the XSETBV/XGETBV instructions, and XCR0.
27	OSXSAVE	A value of 1 indicates that the OS has set CR4.OSXSAVE[bit 18] to enable XSETBV/XGETBV instructions to access XCR0 and to support processor extended state management using XSAVE/XRSTOR.
28	AVX	A value of 1 indicates the processor supports the AVX instruction extensions.
29	F16C	A value of 1 indicates that processor supports 16-bit floating-point conversion instructions.
30	RDRAND	A value of 1 indicates that processor supports RDRAND instruction.
31	Not Used	Always returns 0.

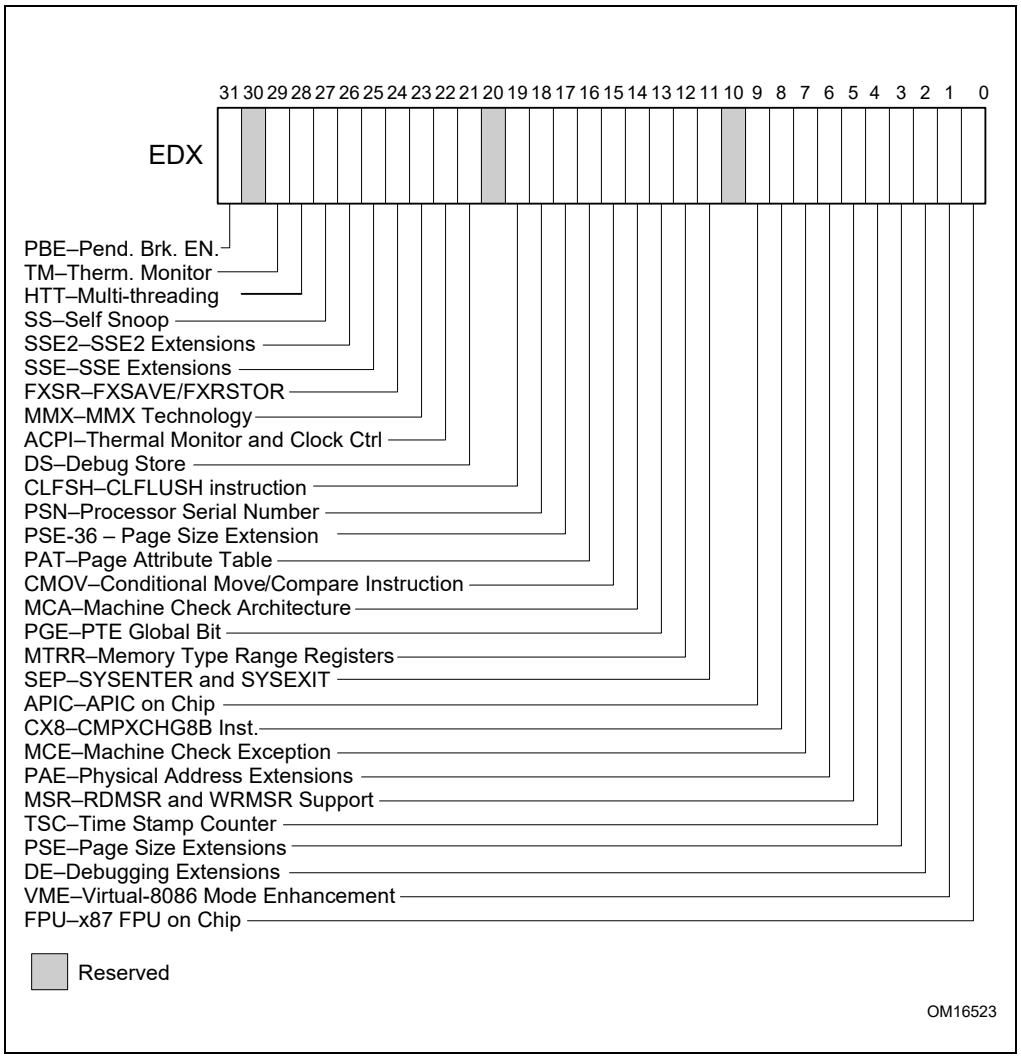


Figure 3-8. Feature Information Returned in the EDX Register

Table 3-11. More on Feature Information Returned in the EDX Register

Bit #	Mnemonic	Description
0	FPU	**Floating Point Unit On-Chip.** The processor contains an x87 FPU.
1	VME	**Virtual 8086 Mode Enhancements.** Virtual 8086 mode enhancements, including CR4.VME for controlling the feature, CR4.PVI for protected mode virtual interrupts, software interrupt indirection, expansion of the TSS with the software indirection bitmap, and EFLAGS.VIF and EFLAGS.VIP flags.
2	DE	**Debugging Extensions.** Support for I/O breakpoints, including CR4.DE for controlling the feature, and optional trapping of accesses to DR4 and DR5.
3	PSE	**Page Size Extension.** Large pages of size 4 MByte are supported, including CR4.PSE for controlling the feature, the defined dirty bit in PDE (Page Directory Entries), optional reserved bit trapping in CR3, PDEs, and PTEs.
4	TSC	**Time Stamp Counter.** The RDTSC instruction is supported, including CR4.TSD for controlling privilege.
5	MSR	**Model Specific Registers RDMSR and WRMSR Instructions.** The RDMSR and WRMSR instructions are supported. Some of the MSRs are implementation dependent.
6	PAE	**Physical Address Extension.** Physical addresses greater than 32 bits are supported: extended page table entry formats, an extra level in the page translation tables is defined, 2-MByte pages are supported instead of 4 Mbyte pages if PAE bit is 1.
7	MCE	**Machine Check Exception.** Exception 18 is defined for Machine Checks, including CR4.MCE for controlling the feature. This feature does not define the model-specific implementations of machine-check error logging, reporting, and processor shutdowns. Machine Check exception handlers may have to depend on processor version to do model specific processing of the exception, or test for the presence of the Machine Check feature.
8	CX8	**CMPXCHG8B Instruction.** The compare-and-exchange 8 bytes (64 bits) instruction is supported (implicitly locked and atomic).
9	APIC	**APIC On-Chip.** The processor contains an Advanced Programmable Interrupt Controller (APIC), responding to memory mapped commands in the physical address range FFFE0000H to FFFE0FFFH (by default - some processors permit the APIC to be relocated).
10	Reserved	Reserved
11	SEP	**SYSENTER and SYSEXIT Instructions.** The SYSENTER and SYSEXIT and associated MSRs are supported.
12	MTRR	**Memory Type Range Registers.** MTRRs are supported. The MTRRcap MSR contains feature bits that describe what memory types are supported, how many variable MTRRs are supported, and whether fixed MTRRs are supported.
13	PGE	**Page Global Bit.** The global bit is supported in paging-structure entries that map a page, indicating TLB entries that are common to different processes and need not be flushed. The CR4.PGE bit controls this feature.
14	MCA	**Machine Check Architecture.** A value of 1 indicates the Machine Check Architecture of reporting machine errors is supported. The MCG_CAP MSR contains feature bits describing how many banks of error reporting MSRs are supported.
15	CMOV	**Conditional Move Instructions.** The conditional move instruction CMOV is supported. In addition, if x87 FPU is present as indicated by the CPUID.FPU feature bit, then the FCOMI and FCMOV instructions are supported
16	PAT	**Page Attribute Table.** Page Attribute Table is supported. This feature augments the Memory Type Range Registers (MTRRs), allowing an operating system to specify attributes of memory accessed through a linear address on a 4KB granularity.
17	PSE-36	**36-Bit Page Size Extension.** 4-MByte pages addressing physical memory beyond 4 GBytes are supported with 32-bit paging. This feature indicates that upper bits of the physical address of a 4-MByte page are encoded in bits 20:13 of the page-directory entry. Such physical addresses are limited by MAXPHYADDR and may be up to 40 bits in size.
18	PSN	**Processor Serial Number.** The processor supports the 96-bit processor identification number feature and the feature is enabled.
19	CLFSH	**CLFLUSH Instruction.** CLFLUSH Instruction is supported.
20	Reserved	Reserved

Table 3-11. More on Feature Information Returned in the EDX Register (Contd.)

Bit #	Mnemonic	Description
21	DS	**Debug Store.** The processor supports the ability to write debug information into a memory resident buffer. This feature is used by the branch trace store (BTS) and processor event-based sampling (PEBS) facilities (see Chapter 23, "Introduction to Virtual-Machine Extensions," in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3C*).
22	ACPI	**Thermal Monitor and Software Controlled Clock Facilities.** The processor implements internal MSRs that allow processor temperature to be monitored and processor performance to be modulated in predefined duty cycles under software control.
23	MMX	**Intel MMX Technology.** The processor supports the Intel MMX technology.
24	FXSR	**FXSAVE and FXRSTOR Instructions.** The FXSAVE and FXRSTOR instructions are supported for fast save and restore of the floating point context. Presence of this bit also indicates that CR4.OSFXSR is available for an operating system to indicate that it supports the FXSAVE and FXRSTOR instructions.
25	SSE	**SSE.** The processor supports the SSE extensions.
26	SSE2	**SSE2.** The processor supports the SSE2 extensions.
27	SS	**Self Snoop.** The processor supports the management of conflicting memory types by performing a snoop of its own cache structure for transactions issued to the bus.
28	HTT	**Max APIC IDs reserved field is Valid.** A value of 0 for HTT indicates there is only a single logical processor in the package and software should assume only a single APIC ID is reserved. A value of 1 for HTT indicates the value in CPUID.1.EBX[23:16] (the Maximum number of addressable IDs for logical processors in this package) is valid for the package.
29	TM	**Thermal Monitor.** The processor implements the thermal monitor automatic thermal control circuitry (TCC).
30	Reserved	Reserved
31	PBE	**Pending Break Enable.** The processor supports the use of the FERR#/PBE# pin when the processor is in the stop-clock state (STPCLK# is asserted) to signal the processor that an interrupt is pending and that the processor should return to normal operation to handle the interrupt. Bit 10 (PBE enable) in the IA32_MISC_ENABLE MSR enables this capability.

INPUT EAX = 02H: TLB/Cache/Prefetch Information Returned in EAX, EBX, ECX, EDX

When CPUID executes with EAX set to 02H, the processor returns information about the processor's internal TLBs, cache and prefetch hardware in the EAX, EBX, ECX, and EDX registers. The information is reported in encoded form and fall into the following categories:

- The least-significant byte in register EAX (register AL) will always return 01H. Software should ignore this value and not interpret it as an informational descriptor.
- The most significant bit (bit 31) of each register indicates whether the register contains valid information (set to 0) or is reserved (set to 1).
- If a register contains valid information, the information is contained in 1 byte descriptors. There are four types of encoding values for the byte descriptor, the encoding type is noted in the second column of Table 3-12. Table 3-12 lists the encoding of these descriptors. Note that the order of descriptors in the EAX, EBX, ECX, and EDX registers is not defined; that is, specific bytes are not designated to contain descriptors for specific cache, prefetch, or TLB types. The descriptors may appear in any order. Note also a processor may report a general descriptor type (FFH) and not report any byte descriptor of "cache type" via CPUID leaf 2.

Table 3-12. Encoding of CPUID Leaf 2 Descriptors

Value	Type	Description
00H	General	Null descriptor, this byte contains no information
01H	TLB	Instruction TLB: 4 KByte pages, 4-way set associative, 32 entries
02H	TLB	Instruction TLB: 4 MByte pages, fully associative, 2 entries
03H	TLB	Data TLB: 4 KByte pages, 4-way set associative, 64 entries
04H	TLB	Data TLB: 4 MByte pages, 4-way set associative, 8 entries
05H	TLB	Data TLB1: 4 MByte pages, 4-way set associative, 32 entries
06H	Cache	1st-level instruction cache: 8 KBytes, 4-way set associative, 32 byte line size
08H	Cache	1st-level instruction cache: 16 KBytes, 4-way set associative, 32 byte line size
09H	Cache	1st-level instruction cache: 32KBytes, 4-way set associative, 64 byte line size
0AH	Cache	1st-level data cache: 8 KBytes, 2-way set associative, 32 byte line size
0BH	TLB	Instruction TLB: 4 MByte pages, 4-way set associative, 4 entries
0CH	Cache	1st-level data cache: 16 KBytes, 4-way set associative, 32 byte line size
0DH	Cache	1st-level data cache: 16 KBytes, 4-way set associative, 64 byte line size
0EH	Cache	1st-level data cache: 24 KBytes, 6-way set associative, 64 byte line size
1DH	Cache	2nd-level cache: 128 KBytes, 2-way set associative, 64 byte line size
21H	Cache	2nd-level cache: 256 KBytes, 8-way set associative, 64 byte line size
22H	Cache	3rd-level cache: 512 KBytes, 4-way set associative, 64 byte line size, 2 lines per sector
23H	Cache	3rd-level cache: 1 MBytes, 8-way set associative, 64 byte line size, 2 lines per sector
24H	Cache	2nd-level cache: 1 MBytes, 16-way set associative, 64 byte line size
25H	Cache	3rd-level cache: 2 MBytes, 8-way set associative, 64 byte line size, 2 lines per sector
29H	Cache	3rd-level cache: 4 MBytes, 8-way set associative, 64 byte line size, 2 lines per sector
2CH	Cache	1st-level data cache: 32 KBytes, 8-way set associative, 64 byte line size
30H	Cache	1st-level instruction cache: 32 KBytes, 8-way set associative, 64 byte line size
40H	Cache	No 2nd-level cache or, if processor contains a valid 2nd-level cache, no 3rd-level cache
41H	Cache	2nd-level cache: 128 KBytes, 4-way set associative, 32 byte line size
42H	Cache	2nd-level cache: 256 KBytes, 4-way set associative, 32 byte line size
43H	Cache	2nd-level cache: 512 KBytes, 4-way set associative, 32 byte line size
44H	Cache	2nd-level cache: 1 MByte, 4-way set associative, 32 byte line size
45H	Cache	2nd-level cache: 2 MByte, 4-way set associative, 32 byte line size
46H	Cache	3rd-level cache: 4 MByte, 4-way set associative, 64 byte line size
47H	Cache	3rd-level cache: 8 MByte, 8-way set associative, 64 byte line size
48H	Cache	2nd-level cache: 3MByte, 12-way set associative, 64 byte line size
49H	Cache	3rd-level cache: 4MB, 16-way set associative, 64-byte line size (Intel Xeon processor MP, Family 0FH, Model 06H); 2nd-level cache: 4 MByte, 16-way set associative, 64 byte line size
4AH	Cache	3rd-level cache: 6MByte, 12-way set associative, 64 byte line size
4BH	Cache	3rd-level cache: 8MByte, 16-way set associative, 64 byte line size
4CH	Cache	3rd-level cache: 12MByte, 12-way set associative, 64 byte line size
4DH	Cache	3rd-level cache: 16MByte, 16-way set associative, 64 byte line size
4EH	Cache	2nd-level cache: 6MByte, 24-way set associative, 64 byte line size
4FH	TLB	Instruction TLB: 4 KByte pages, 32 entries

Table 3-12. Encoding of CPUID Leaf 2 Descriptors (Contd.)

Value	Type	Description
50H	TLB	Instruction TLB: 4 KByte and 2-MByte or 4-MByte pages, 64 entries
51H	TLB	Instruction TLB: 4 KByte and 2-MByte or 4-MByte pages, 128 entries
52H	TLB	Instruction TLB: 4 KByte and 2-MByte or 4-MByte pages, 256 entries
55H	TLB	Instruction TLB: 2-MByte or 4-MByte pages, fully associative, 7 entries
56H	TLB	Data TLB0: 4 MByte pages, 4-way set associative, 16 entries
57H	TLB	Data TLB0: 4 KByte pages, 4-way associative, 16 entries
59H	TLB	Data TLB0: 4 KByte pages, fully associative, 16 entries
5AH	TLB	Data TLB0: 2 MByte or 4 MByte pages, 4-way set associative, 32 entries
5BH	TLB	Data TLB: 4 KByte and 4 MByte pages, 64 entries
5CH	TLB	Data TLB: 4 KByte and 4 MByte pages,128 entries
5DH	TLB	Data TLB: 4 KByte and 4 MByte pages,256 entries
60H	Cache	1st-level data cache: 16 KByte, 8-way set associative, 64 byte line size
61H	TLB	Instruction TLB: 4 KByte pages, fully associative, 48 entries
63H	TLB	Data TLB: 2 MByte or 4 MByte pages, 4-way set associative, 32 entries and a separate array with 1 GByte pages, 4-way set associative, 4 entries
64H	TLB	Data TLB: 4 KByte pages, 4-way set associative, 512 entries
66H	Cache	1st-level data cache: 8 KByte, 4-way set associative, 64 byte line size
67H	Cache	1st-level data cache: 16 KByte, 4-way set associative, 64 byte line size
68H	Cache	1st-level data cache: 32 KByte, 4-way set associative, 64 byte line size
6AH	Cache	uTLB: 4 KByte pages, 8-way set associative, 64 entries
6BH	Cache	DTLB: 4 KByte pages, 8-way set associative, 256 entries
6CH	Cache	DTLB: 2M/4M pages, 8-way set associative, 128 entries
6DH	Cache	DTLB: 1 GByte pages, fully associative, 16 entries
70H	Cache	Trace cache: 12 K-µop, 8-way set associative
71H	Cache	Trace cache: 16 K-µop, 8-way set associative
72H	Cache	Trace cache: 32 K-µop, 8-way set associative
76H	TLB	Instruction TLB: 2M/4M pages, fully associative, 8 entries
78H	Cache	2nd-level cache: 1 MByte, 4-way set associative, 64byte line size
79H	Cache	2nd-level cache: 128 KByte, 8-way set associative, 64 byte line size, 2 lines per sector
7AH	Cache	2nd-level cache: 256 KByte, 8-way set associative, 64 byte line size, 2 lines per sector
7BH	Cache	2nd-level cache: 512 KByte, 8-way set associative, 64 byte line size, 2 lines per sector
7CH	Cache	2nd-level cache: 1 MByte, 8-way set associative, 64 byte line size, 2 lines per sector
7DH	Cache	2nd-level cache: 2 MByte, 8-way set associative, 64byte line size
7FH	Cache	2nd-level cache: 512 KByte, 2-way set associative, 64-byte line size
80H	Cache	2nd-level cache: 512 KByte, 8-way set associative, 64-byte line size
82H	Cache	2nd-level cache: 256 KByte, 8-way set associative, 32 byte line size
83H	Cache	2nd-level cache: 512 KByte, 8-way set associative, 32 byte line size
84H	Cache	2nd-level cache: 1 MByte, 8-way set associative, 32 byte line size
85H	Cache	2nd-level cache: 2 MByte, 8-way set associative, 32 byte line size
86H	Cache	2nd-level cache: 512 KByte, 4-way set associative, 64 byte line size
87H	Cache	2nd-level cache: 1 MByte, 8-way set associative, 64 byte line size

Table 3-12. Encoding of CPUID Leaf 2 Descriptors (Contd.)

Value	Type	Description
A0H	DTLB	DTLB: 4k pages, fully associative, 32 entries
B0H	TLB	Instruction TLB: 4 KByte pages, 4-way set associative, 128 entries
B1H	TLB	Instruction TLB: 2M pages, 4-way, 8 entries or 4M pages, 4-way, 4 entries
B2H	TLB	Instruction TLB: 4KByte pages, 4-way set associative, 64 entries
B3H	TLB	Data TLB: 4 KByte pages, 4-way set associative, 128 entries
B4H	TLB	Data TLB1: 4 KByte pages, 4-way associative, 256 entries
B5H	TLB	Instruction TLB: 4KByte pages, 8-way set associative, 64 entries
B6H	TLB	Instruction TLB: 4KByte pages, 8-way set associative, 128 entries
BAH	TLB	Data TLB1: 4 KByte pages, 4-way associative, 64 entries
C0H	TLB	Data TLB: 4 KByte and 4 MByte pages, 4-way associative, 8 entries
C1H	STLB	Shared 2nd-Level TLB: 4 KByte/2MByte pages, 8-way associative, 1024 entries
C2H	DTLB	DTLB: 4 KByte/2 MByte pages, 4-way associative, 16 entries
C3H	STLB	Shared 2nd-Level TLB: 4 KByte /2 MByte pages, 6-way associative, 1536 entries. Also 1GBbyte pages, 4-way, 16 entries.
C4H	DTLB	DTLB: 2M/4M Byte pages, 4-way associative, 32 entries
CAH	STLB	Shared 2nd-Level TLB: 4 KByte pages, 4-way associative, 512 entries
D0H	Cache	3rd-level cache: 512 KByte, 4-way set associative, 64 byte line size
D1H	Cache	3rd-level cache: 1 MByte, 4-way set associative, 64 byte line size
D2H	Cache	3rd-level cache: 2 MByte, 4-way set associative, 64 byte line size
D6H	Cache	3rd-level cache: 1 MByte, 8-way set associative, 64 byte line size
D7H	Cache	3rd-level cache: 2 MByte, 8-way set associative, 64 byte line size
D8H	Cache	3rd-level cache: 4 MByte, 8-way set associative, 64 byte line size
DCH	Cache	3rd-level cache: 1.5 MByte, 12-way set associative, 64 byte line size
DDH	Cache	3rd-level cache: 3 MByte, 12-way set associative, 64 byte line size
DEH	Cache	3rd-level cache: 6 MByte, 12-way set associative, 64 byte line size
E2H	Cache	3rd-level cache: 2 MByte, 16-way set associative, 64 byte line size
E3H	Cache	3rd-level cache: 4 MByte, 16-way set associative, 64 byte line size
E4H	Cache	3rd-level cache: 8 MByte, 16-way set associative, 64 byte line size
EAH	Cache	3rd-level cache: 12MByte, 24-way set associative, 64 byte line size
EBH	Cache	3rd-level cache: 18MByte, 24-way set associative, 64 byte line size
ECH	Cache	3rd-level cache: 24MByte, 24-way set associative, 64 byte line size
F0H	Prefetch	64-Byte prefetching
F1H	Prefetch	128-Byte prefetching
FEH	General	CPUID leaf 2 does not report TLB descriptor information; use CPUID leaf 18H to query TLB and other address translation parameters.
FFH	General	CPUID leaf 2 does not report cache descriptor information, use CPUID leaf 4 to query cache parameters

Example 3-1. Example of Cache and TLB Interpretation

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

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

Which means:

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

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

When CPUID executes with EAX set to 04H and ECX contains an index value, the processor returns encoded data that describe a set of deterministic cache parameters (for the cache level associated with the input in ECX). Valid index values start from 0.

Software can enumerate the deterministic cache parameters for each level of the cache hierarchy starting with an index value of 0, until the parameters report the value associated with the cache type field is 0. The architecturally defined fields reported by deterministic cache parameters are documented in Table 3-8.

This Cache Size in Bytes

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

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

The CPUID leaf 04H also reports data that can be used to derive the topology of processor cores in a physical package. This information is constant for all valid index values. Software can query the raw data reported by executing CPUID with EAX=04H and ECX=0 and use it as part of the topology enumeration algorithm described in Chapter 8, "Multiple-Processor Management," in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*.

INPUT EAX = 05H: Returns MONITOR and MWAIT Features

When CPUID executes with EAX set to 05H, the processor returns information about features available to MONITOR/MWAIT instructions. The MONITOR instruction is used for address-range monitoring in conjunction with MWAIT instruction. The MWAIT instruction optionally provides additional extensions for advanced power management. See Table 3-8.

INPUT EAX = 06H: Returns Thermal and Power Management Features

When CPUID executes with EAX set to 06H, the processor returns information about thermal and power management features. See Table 3-8.

INPUT EAX = 07H: Returns Structured Extended Feature Enumeration Information

When CPUID executes with EAX set to 07H and ECX = 0, the processor returns information about the maximum input value for sub-leaves that contain extended feature flags. See Table 3-8.

When CPUID executes with EAX set to 07H and the input value of ECX is invalid (see leaf 07H entry in Table 3-8), the processor returns 0 in EAX/EBX/ECX/EDX. In subleaf 0, EAX returns the maximum input value of the highest leaf 7 sub-leaf, and EBX, ECX & EDX contain information of extended feature flags.

INPUT EAX = 09H: Returns Direct Cache Access Information

When CPUID executes with EAX set to 09H, the processor returns information about Direct Cache Access capabilities. See Table 3-8.

INPUT EAX = 0AH: Returns Architectural Performance Monitoring Features

When CPUID executes with EAX set to 0AH, the processor returns information about support for architectural performance monitoring capabilities. Architectural performance monitoring is supported if the version ID (see Table 3-8) is greater than Pn 0. See Table 3-8.

For each version of architectural performance monitoring capability, software must enumerate this leaf to discover the programming facilities and the architectural performance events available in the processor. The details are described in Chapter 23, "Introduction to Virtual-Machine Extensions," in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3C*.

INPUT EAX = 0BH: Returns Extended Topology Information

When CPUID executes with EAX set to 0BH, the processor returns information about extended topology enumeration data. Software must detect the presence of CPUID leaf 0BH by verifying (a) the highest leaf index supported by CPUID is >= 0BH, and (b) CPUID.0BH:EBX[15:0] reports a non-zero value. See Table 3-8.

INPUT EAX = 0DH: Returns Processor Extended States Enumeration Information

When CPUID executes with EAX set to 0DH and ECX = 0, the processor returns information about the bit-vector representation of all processor state extensions that are supported in the processor and storage size requirements of the XSAVE/XRSTOR area. See Table 3-8.

When CPUID executes with EAX set to 0DH and ECX = n (n > 1, and is a valid sub-leaf index), the processor returns information about the size and offset of each processor extended state save area within the XSAVE/XRSTOR area. See Table 3-8. Software can use the forward-extendable technique depicted below to query the valid sub-leaves and obtain size and offset information for each processor extended state save area:

```
For i = 2 to 62 // sub-leaf 1 is reserved
    IF (CPUID.(EAX=0DH, ECX=0):VECTOR[i] = 1 ) // VECTOR is the 64-bit value of EDX:EAX
        Execute CPUID.(EAX=0DH, ECX = i) to examine size and offset for sub-leaf i;
    FI;
```

INPUT EAX = 0FH: Returns Intel Resource Director Technology (Intel RDT) Monitoring Enumeration Information

When CPUID executes with EAX set to 0FH and ECX = 0, the processor returns information about the bit-vector representation of QoS monitoring resource types that are supported in the processor and maximum range of RMID values the processor can use to monitor of any supported resource types. Each bit, starting from bit 1, corresponds to a specific resource type if the bit is set. The bit position corresponds to the sub-leaf index (or ResID) that software must use to query QoS monitoring capability available for that type. See Table 3-8.

When CPUID executes with EAX set to 0FH and ECX = n (n >= 1, and is a valid ResID), the processor returns information software can use to program IA32_PQR_ASSOC, IA32_QM_EVTSEL MSRs before reading QoS data from the IA32_QM_CTR MSR.

INPUT EAX = 10H: Returns Intel Resource Director Technology (Intel RDT) Allocation Enumeration Information

When CPUID executes with EAX set to 10H and ECX = 0, the processor returns information about the bit-vector representation of QoS Enforcement resource types that are supported in the processor. Each bit, starting from bit

1, corresponds to a specific resource type if the bit is set. The bit position corresponds to the sub-leaf index (or ResID) that software must use to query QoS enforcement capability available for that type. See Table 3-8.

When CPUID executes with EAX set to 10H and ECX = n (n >= 1, and is a valid ResID), the processor returns information about available classes of service and range of QoS mask MSRs that software can use to configure each class of services using capability bit masks in the QoS Mask registers, IA32_resourceType_Mask_n.

INPUT EAX = 12H: Returns Intel SGX Enumeration Information

When CPUID executes with EAX set to 12H and ECX = 0H, the processor returns information about Intel SGX capabilities. See Table 3-8.

When CPUID executes with EAX set to 12H and ECX = 1H, the processor returns information about Intel SGX attributes. See Table 3-8.

When CPUID executes with EAX set to 12H and ECX = n (n > 1), the processor returns information about Intel SGX Enclave Page Cache. See Table 3-8.

INPUT EAX = 14H: Returns Intel Processor Trace Enumeration Information

When CPUID executes with EAX set to 14H and ECX = 0H, the processor returns information about Intel Processor Trace extensions. See Table 3-8.

When CPUID executes with EAX set to 14H and ECX = n (n > 0 and less than the number of non-zero bits in CPUID.(EAX=14H, ECX= 0H).EAX), the processor returns information about packet generation in Intel Processor Trace. See Table 3-8.

INPUT EAX = 15H: Returns Time Stamp Counter and Nominal Core Crystal Clock Information

When CPUID executes with EAX set to 15H and ECX = 0H, the processor returns information about Time Stamp Counter and Core Crystal Clock. See Table 3-8.

INPUT EAX = 16H: Returns Processor Frequency Information

When CPUID executes with EAX set to 16H, the processor returns information about Processor Frequency Information. See Table 3-8.

INPUT EAX = 17H: Returns System-On-Chip Information

When CPUID executes with EAX set to 17H, the processor returns information about the System-On-Chip Vendor Attribute Enumeration. See Table 3-8.

INPUT EAX = 18H: Returns Deterministic Address Translation Parameters Information

When CPUID executes with EAX set to 18H, the processor returns information about the Deterministic Address Translation Parameters. See Table 3-8.

METHODS FOR RETURNING BRANDING INFORMATION

Use the following techniques to access branding information:

1. Processor brand string method.
2. Processor brand index; this method uses a software supplied brand string table.

These two methods are discussed in the following sections. For methods that are available in early processors, see Section: "Identification of Earlier IA-32 Processors" in Chapter 19 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*.

The Processor Brand String Method

Figure 3-9 describes the algorithm used for detection of the brand string. Processor brand identification software should execute this algorithm on all Intel 64 and IA-32 processors.

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

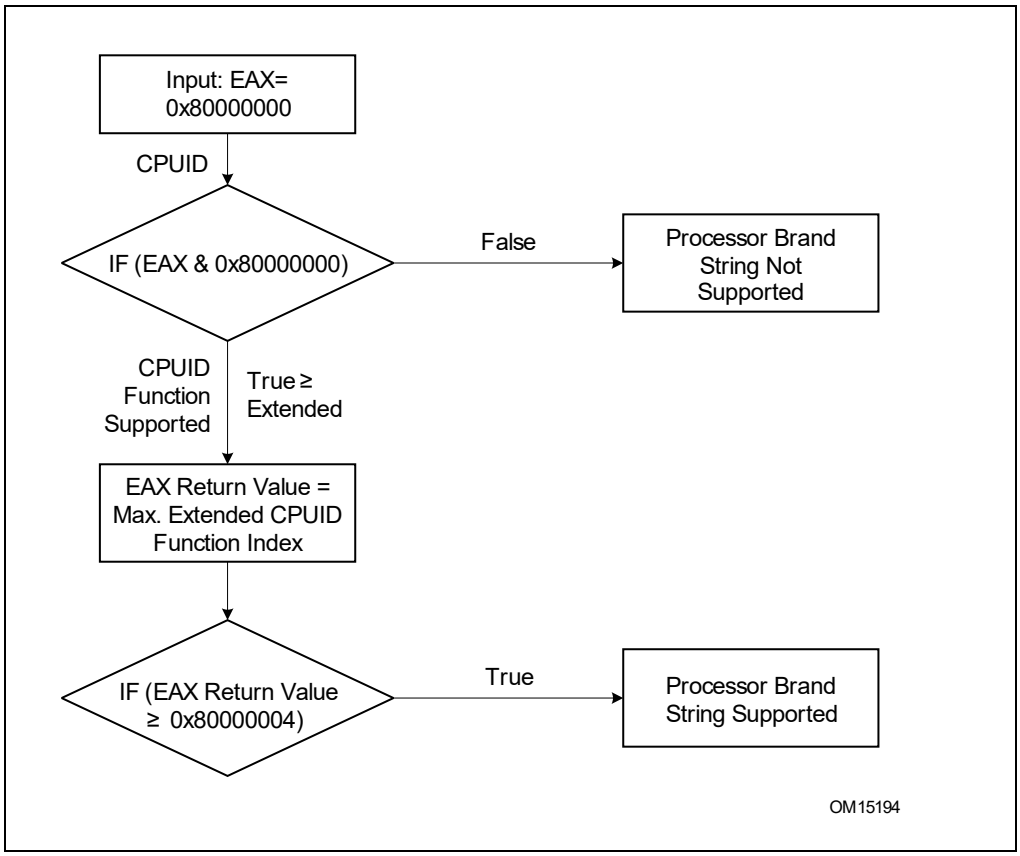


Figure 3-9. Determination of Support for the Processor Brand String

How Brand Strings Work

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

Table 3-13 shows the brand string that is returned by the first processor in the Pentium 4 processor family.

Table 3-13. Processor Brand String Returned with Pentium 4 Processor

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

Extracting the Processor Frequency from Brand Strings

Figure 3-10 provides an algorithm which software can use to extract the Processor Base frequency from the processor brand string.

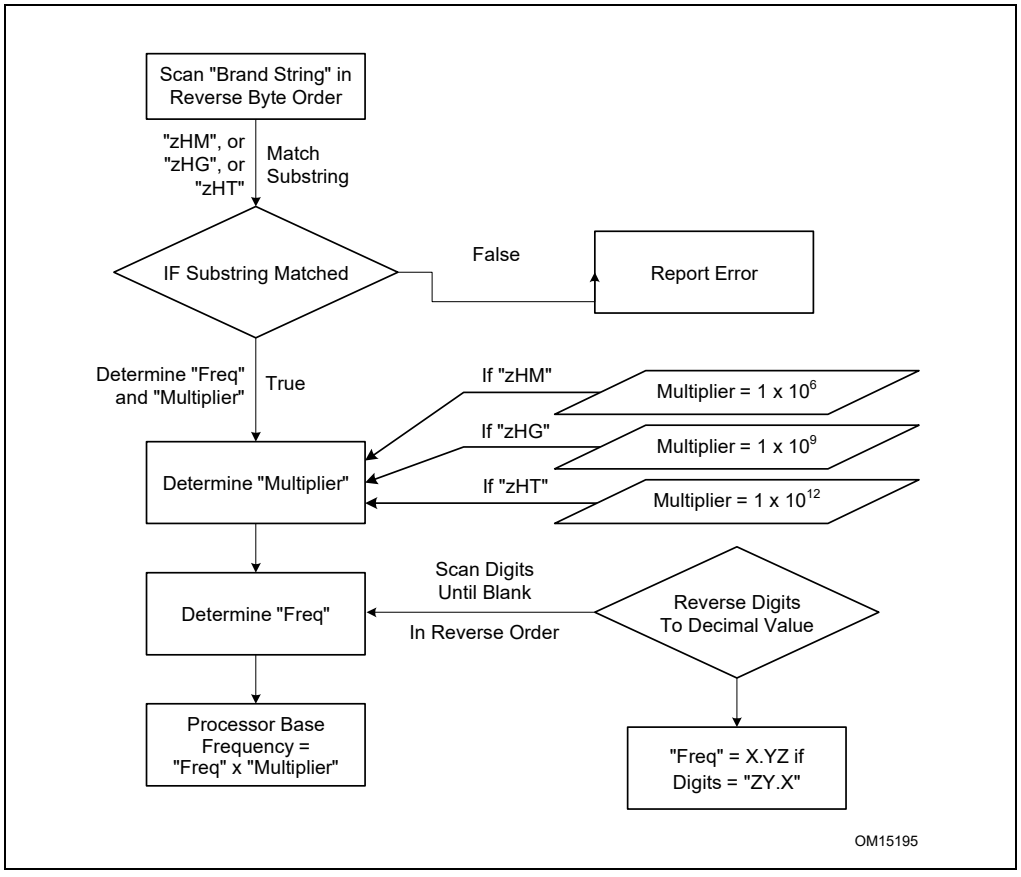


Figure 3-10. Algorithm for Extracting Processor Frequency

The Processor Brand Index Method

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

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

Table 3-14 shows brand indices that have identification strings associated with them.

Table 3-14. Mapping of Brand Indices; and Intel 64 and IA-32 Processor Brand Strings

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

NOTES:

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

IA-32 Architecture Compatibility

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

Operation

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

CASE (EAX) OF
    EAX = 0:
        EAX ← Highest basic function input value understood by CPUID;
        EBX ← Vendor identification string;
        EDX ← Vendor identification string;
        ECX ← Vendor identification string;
    BREAK;
    EAX = 1H:
        EAX[3:0] ← Stepping ID;
        EAX[7:4] ← Model;
        EAX[11:8] ← Family;
        EAX[13:12] ← Processor type;
        EAX[15:14] ← Reserved;
        EAX[19:16] ← Extended Model;
        EAX[27:20] ← Extended Family;
        EAX[31:28] ← Reserved;
        EBX[7:0] ← Brand Index; (* Reserved if the value is zero. *)
        EBX[15:8] ← CLFLUSH Line Size;
        EBX[16:23] ← Reserved; (* Number of threads enabled = 2 if MT enable fuse set. *)
        EBX[24:31] ← Initial APIC ID;
        ECX ← Feature flags; (* See Figure 3-7. *)
        EDX ← Feature flags; (* See Figure 3-8. *)
    BREAK;
    EAX = 2H:
        EAX ← Cache and TLB information;
        EBX ← Cache and TLB information;
        ECX ← Cache and TLB information;
        EDX ← Cache and TLB information;
    BREAK;
    EAX = 3H:
        EAX ← Reserved;
        EBX ← Reserved;
        ECX ← ProcessorSerialNumber[31:0];
        (* Pentium III processors only, otherwise reserved. *)
        EDX ← ProcessorSerialNumber[63:32];
        (* Pentium III processors only, otherwise reserved. *
    BREAK
    EAX = 4H:
        EAX ← Deterministic Cache Parameters Leaf; (* See Table 3-8. *)
        EBX ← Deterministic Cache Parameters Leaf;
        ECX ← Deterministic Cache Parameters Leaf;
        EDX ← Deterministic Cache Parameters Leaf;
    BREAK;
    EAX = 5H:
        EAX ← MONITOR/MWAIT Leaf; (* See Table 3-8. *)
        EBX ← MONITOR/MWAIT Leaf;
        ECX ← MONITOR/MWAIT Leaf;
        EDX ← MONITOR/MWAIT Leaf;
    BREAK;
```

```
EAX = 6H:
    EAX ← Thermal and Power Management Leaf; (* See Table 3-8. *)
    EBX ← Thermal and Power Management Leaf;
    ECX ← Thermal and Power Management Leaf;
    EDX ← Thermal and Power Management Leaf;
BREAK;
EAX = 7H:
    EAX ← Structured Extended Feature Flags Enumeration Leaf; (* See Table 3-8. *)
    EBX ← Structured Extended Feature Flags Enumeration Leaf;
    ECX ← Structured Extended Feature Flags Enumeration Leaf;
    EDX ← Structured Extended Feature Flags Enumeration Leaf;
BREAK;
EAX = 8H:
    EAX ← Reserved = 0;
    EBX ← Reserved = 0;
    ECX ← Reserved = 0;
    EDX ← Reserved = 0;
BREAK;
EAX = 9H:
    EAX ← Direct Cache Access Information Leaf; (* See Table 3-8. *)
    EBX ← Direct Cache Access Information Leaf;
    ECX ← Direct Cache Access Information Leaf;
    EDX ← Direct Cache Access Information Leaf;
BREAK;
EAX = AH:
    EAX ← Architectural Performance Monitoring Leaf; (* See Table 3-8. *)
    EBX ← Architectural Performance Monitoring Leaf;
    ECX ← Architectural Performance Monitoring Leaf;
    EDX ← Architectural Performance Monitoring Leaf;
    BREAK
EAX = BH:
    EAX ← Extended Topology Enumeration Leaf; (* See Table 3-8. *)
    EBX ← Extended Topology Enumeration Leaf;
    ECX ← Extended Topology Enumeration Leaf;
    EDX ← Extended Topology Enumeration Leaf;
BREAK;
EAX = CH:
    EAX ← Reserved = 0;
    EBX ← Reserved = 0;
    ECX ← Reserved = 0;
    EDX ← Reserved = 0;
BREAK;
EAX = DH:
    EAX ← Processor Extended State Enumeration Leaf; (* See Table 3-8. *)
    EBX ← Processor Extended State Enumeration Leaf;
    ECX ← Processor Extended State Enumeration Leaf;
    EDX ← Processor Extended State Enumeration Leaf;
BREAK;
EAX = EH:
    EAX ← Reserved = 0;
    EBX ← Reserved = 0;
    ECX ← Reserved = 0;
    EDX ← Reserved = 0;
BREAK;
```

```
    EAX = FH:
        EAX ← Intel Resource Director Technology Monitoring Enumeration Leaf; (* See Table 3-8. *)
        EBX ← Intel Resource Director Technology Monitoring Enumeration Leaf;
        ECX ← Intel Resource Director Technology Monitoring Enumeration Leaf;
        EDX ← Intel Resource Director Technology Monitoring Enumeration Leaf;
    BREAK;
    EAX = 10H:
        EAX ← Intel Resource Director Technology Allocation Enumeration Leaf; (* See Table 3-8. *)
        EBX ← Intel Resource Director Technology Allocation Enumeration Leaf;
        ECX ← Intel Resource Director Technology Allocation Enumeration Leaf;
        EDX ← Intel Resource Director Technology Allocation Enumeration Leaf;
    BREAK;
        EAX = 12H:
        EAX ← Intel SGX Enumeration Leaf; (* See Table 3-8. *)
        EBX ← Intel SGX Enumeration Leaf;
        ECX ← Intel SGX Enumeration Leaf;
        EDX ← Intel SGX Enumeration Leaf;
    BREAK;
    EAX = 14H:
        EAX ← Intel Processor Trace Enumeration Leaf; (* See Table 3-8. *)
        EBX ← Intel Processor Trace Enumeration Leaf;
        ECX ← Intel Processor Trace Enumeration Leaf;
        EDX ← Intel Processor Trace Enumeration Leaf;
    BREAK;
    EAX = 15H:
        EAX ← Time Stamp Counter and Nominal Core Crystal Clock Information Leaf; (* See Table 3-8. *)
        EBX ← Time Stamp Counter and Nominal Core Crystal Clock Information Leaf;
        ECX ← Time Stamp Counter and Nominal Core Crystal Clock Information Leaf;
        EDX ← Time Stamp Counter and Nominal Core Crystal Clock Information Leaf;
    BREAK;
    EAX = 16H:
        EAX ← Processor Frequency Information Enumeration Leaf; (* See Table 3-8. *)
        EBX ← Processor Frequency Information Enumeration Leaf;
        ECX ← Processor Frequency Information Enumeration Leaf;
        EDX ← Processor Frequency Information Enumeration Leaf;
    BREAK;
    EAX = 17H:
        EAX ← System-On-Chip Vendor Attribute Enumeration Leaf; (* See Table 3-8. *)
        EBX ← System-On-Chip Vendor Attribute Enumeration Leaf;
        ECX ← System-On-Chip Vendor Attribute Enumeration Leaf;
        EDX ← System-On-Chip Vendor Attribute Enumeration Leaf;
    BREAK;
        EAX = 18H:
        EAX ← Deterministic Address Translation Parameters Enumeration Leaf; (* See Table 3-8. *)
        EBX ← Deterministic Address Translation Parameters Enumeration Leaf;
        ECX ←Deterministic Address Translation Parameters Enumeration Leaf;
        EDX ← Deterministic Address Translation Parameters Enumeration Leaf;
    BREAK;
    EAX = 80000000H:
        EAX ← Highest extended function input value understood by CPUID;
        EBX ← Reserved;
        ECX ← Reserved;
        EDX ← Reserved;
    BREAK;
```

```
EAX = 80000001H:
    EAX ← Reserved;
    EBX ← Reserved;
    ECX ← Extended Feature Bits (* See Table 3-8.*);
    EDX ← Extended Feature Bits (* See Table 3-8. *);
BREAK;
EAX = 80000002H:
    EAX ← Processor Brand String;
    EBX ← Processor Brand String, continued;
    ECX ← Processor Brand String, continued;
    EDX ← Processor Brand String, continued;
BREAK;
EAX = 80000003H:
    EAX ← Processor Brand String, continued;
    EBX ← Processor Brand String, continued;
    ECX ← Processor Brand String, continued;
    EDX ← Processor Brand String, continued;
BREAK;
EAX = 80000004H:
    EAX ← Processor Brand String, continued;
    EBX ← Processor Brand String, continued;
    ECX ← Processor Brand String, continued;
    EDX ← Processor Brand String, continued;
BREAK;
EAX = 80000005H:
    EAX ← Reserved = 0;
    EBX ← Reserved = 0;
    ECX ← Reserved = 0;
    EDX ← Reserved = 0;
BREAK;
EAX = 80000006H:
    EAX ← Reserved = 0;
    EBX ← Reserved = 0;
    ECX ← Cache information;
    EDX ← Reserved = 0;
BREAK;
EAX = 80000007H:
    EAX ← Reserved = 0;
    EBX ← Reserved = 0;
    ECX ← Reserved = 0;
    EDX ← Reserved = Misc Feature Flags;
BREAK;
EAX = 80000008H:
    EAX ← Reserved = Physical Address Size Information;
    EBX ← Reserved = Virtual Address Size Information;
    ECX ← Reserved = 0;
    EDX ← Reserved = 0;
BREAK;
EAX >= 40000000H and EAX <= 4FFFFFFFH:
DEFAULT: (* EAX = Value outside of recognized range for CPUID. *)
    (* If the highest basic information leaf data depend on ECX input value, ECX is honored.*)
    EAX ← Reserved; (* Information returned for highest basic information leaf. *)
    EBX ← Reserved; (* Information returned for highest basic information leaf. *)
    ECX ← Reserved; (* Information returned for highest basic information leaf. *)
```

```
        EDX ← Reserved; (* Information returned for highest basic information leaf. *)
    BREAK;
ESAC;
```

Flags Affected

None.

Exceptions (All Operating Modes)

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

CRC32 — Accumulate CRC32 Value

Opcode/ Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
F2 0F 38 F0 */r* CRC32 *r32, r/m8*	RM	Valid	Valid	Accumulate CRC32 on *r/m8*.
F2 REX 0F 38 F0 */r* CRC32 *r32, r/m8**	RM	Valid	N.E.	Accumulate CRC32 on *r/m8*.
F2 0F 38 F1 */r* CRC32 *r32, r/m16*	RM	Valid	Valid	Accumulate CRC32 on *r/m16*.
F2 0F 38 F1 */r* CRC32 *r32, r/m32*	RM	Valid	Valid	Accumulate CRC32 on *r/m32*.
F2 REX.W 0F 38 F0 */r* CRC32 *r64, r/m8*	RM	Valid	N.E.	Accumulate CRC32 on *r/m8*.
F2 REX.W 0F 38 F1 */r* CRC32 *r64, r/m64*	RM	Valid	N.E.	Accumulate CRC32 on *r/m64*.

NOTES:

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

Instruction Operand Encoding

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

Description

Starting with an initial value in the first operand (destination operand), accumulates a CRC32 (polynomial 11EDC6F41H) value for the second operand (source operand) and stores the result in the destination operand. The source operand can be a register or a memory location. The destination operand must be an r32 or r64 register. If the destination is an r64 register, then the 32-bit result is stored in the least significant double word and 00000000H is stored in the most significant double word of the r64 register.

The initial value supplied in the destination operand is a double word integer stored in the r32 register or the least significant double word of the r64 register. To incrementally accumulate a CRC32 value, software retains the result of the previous CRC32 operation in the destination operand, then executes the CRC32 instruction again with new input data in the source operand. Data contained in the source operand is processed in reflected bit order. This means that the most significant bit of the source operand is treated as the least significant bit of the quotient, and so on, for all the bits of the source operand. Likewise, the result of the CRC operation is stored in the destination operand in reflected bit order. This means that the most significant bit of the resulting CRC (bit 31) is stored in the least significant bit of the destination operand (bit 0), and so on, for all the bits of the CRC.

Operation

Notes:

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

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

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

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

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

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

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

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

```
TEMP1[7-0] ← BIT_REFLECT8(SRC[7-0])
TEMP2[31-0] ← BIT_REFLECT32 (DEST[31-0])
TEMP3[39-0] ← TEMP1[7-0] « 32
TEMP4[39-0] ← TEMP2[31-0] « 8
TEMP5[39-0] ← TEMP3[39-0] XOR TEMP4[39-0]
TEMP6[31-0] ← TEMP5[39-0] MOD2 11EDC6F41H
DEST[31-0] ← BIT_REFLECT (TEMP6[31-0])
DEST[63-32] ← 00000000H
```

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

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

Flags Affected

None

Intel C/C++ Compiler Intrinsic Equivalent

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

SIMD Floating Point Exceptions

None

Protected Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS or GS segments.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF (fault-code)	For a page fault.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If CPUID.01H:ECX.SSE4_2 [Bit 20] = 0.
	If LOCK prefix is used.

Real-Address Mode Exceptions

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

Virtual 8086 Mode Exceptions

#GP(0)	If any part of the operand lies outside of the effective address space from 0 to 0FFFFH.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF (fault-code)	For a page fault.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.
#UD	If CPUID.01H:ECX.SSE4_2 [Bit 20] = 0.
	If LOCK prefix is used.

Compatibility Mode Exceptions

Same exceptions as in Protected Mode.

64-Bit Mode Exceptions

#GP(0)	If the memory address is in a non-canonical form.
#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#PF (fault-code)	For a page fault.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If CPUID.01H:ECX.SSE4_2 [Bit 20] = 0.
	If LOCK prefix is used.

CVTDQ2PD—Convert Packed Doubleword Integers to Packed Double-Precision Floating-Point Values

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
F3 0F E6 /r CVTDQ2PD xmm1, xmm2/m64	A	V/V	SSE2	Convert two packed signed doubleword integers from xmm2/mem to two packed double-precision floating-point values in xmm1.
VEX.128.F3.0F.WIG E6 /r VCVTDQ2PD xmm1, xmm2/m64	A	V/V	AVX	Convert two packed signed doubleword integers from xmm2/mem to two packed double-precision floating-point values in xmm1.
VEX.256.F3.0F.WIG E6 /r VCVTDQ2PD ymm1, xmm2/m128	A	V/V	AVX	Convert four packed signed doubleword integers from xmm2/mem to four packed double-precision floating-point values in ymm1.
EVEX.128.F3.0F.W0 E6 /r VCVTDQ2PD xmm1 {k1}{z}, xmm2/m128/m32bcst	B	V/V	AVX512VL AVX512F	Convert 2 packed signed doubleword integers from xmm2/m128/m32bcst to eight packed double-precision floating-point values in xmm1 with writemask k1.
EVEX.256.F3.0F.W0 E6 /r VCVTDQ2PD ymm1 {k1}{z}, xmm2/m128/m32bcst	B	V/V	AVX512VL AVX512F	Convert 4 packed signed doubleword integers from xmm2/m128/m32bcst to 4 packed double-precision floating-point values in ymm1 with writemask k1.
EVEX.512.F3.0F.W0 E6 /r VCVTDQ2PD zmm1 {k1}{z}, ymm2/m256/m32bcst	B	V/V	AVX512F	Convert eight packed signed doubleword integers from ymm2/m256/m32bcst to eight packed double-precision floating-point values in zmm1 with writemask k1.

Instruction Operand Encoding

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

Description

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

EVEX encoded versions: The source operand can be a YMM/XMM/XMM (low 64 bits) register, a 256/128/64-bit memory location or a 256/128/64-bit vector broadcasted from a 32-bit memory location. The destination operand is a ZMM/YMM/XMM register conditionally updated with writemask k1. Attempt to encode this instruction with EVEX embedded rounding is ignored.

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

VEX.128 encoded version: The source operand is an XMM register or 64- bit memory location. The destination operand is a XMM register. The upper Bits (MAXVL-1:128) of the corresponding ZMM register destination are zeroed.

128-bit Legacy SSE version: The source operand is an XMM register or 64- bit memory location. The destination operand is an XMM register. The upper Bits (MAXVL-1:128) of the corresponding ZMM register destination are unmodified.

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

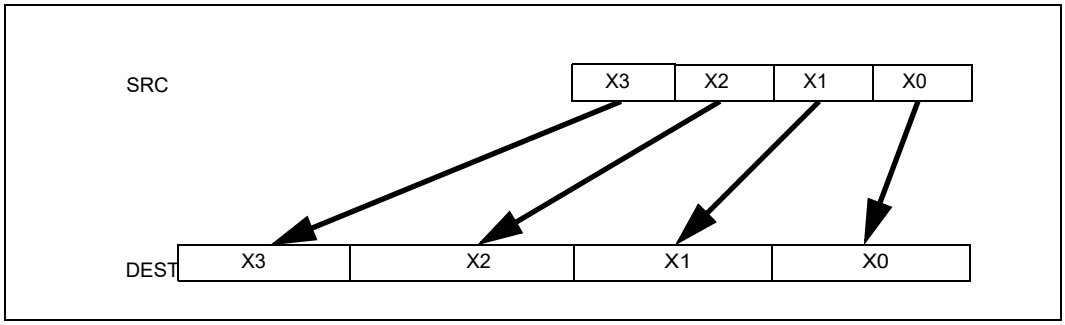


Figure 3-11. CVTDQ2PD (VEX.256 encoded version)

Operation

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

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j ← 0 TO KL-1
    i ← j * 64
    k ← j * 32
    IF k1[j] OR *no writemask*
        THEN DEST[i+63:i] ←
            Convert_Integer_To_Double_Precision_Floating_Point(SRC[k+31:k])
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE                                ; zeroing-masking
                    DEST[i+63:i] ← 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0
```

VCVTDQ2PD (EVEX encoded versions) when src operand is a memory source
(KL, VL) = (2, 128), (4, 256), (8, 512)

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

VCVTDQ2PD (VEX.256 encoded version)

```
DEST[63:0] ← Convert_Integer_To_Double_Precision_Floating_Point(SRC[31:0])
DEST[127:64] ← Convert_Integer_To_Double_Precision_Floating_Point(SRC[63:32])
DEST[191:128] ← Convert_Integer_To_Double_Precision_Floating_Point(SRC[95:64])
DEST[255:192] ← Convert_Integer_To_Double_Precision_Floating_Point(SRC[127:96)
DEST[MAXVL-1:256] ← 0
```

VCVTDQ2PD (VEX.128 encoded version)

```
DEST[63:0] ← Convert_Integer_To_Double_Precision_Floating_Point(SRC[31:0])
DEST[127:64] ← Convert_Integer_To_Double_Precision_Floating_Point(SRC[63:32])
DEST[MAXVL-1:128] ← 0
```

CVTDQ2PD (128-bit Legacy SSE version)

```
DEST[63:0] ← Convert_Integer_To_Double_Precision_Floating_Point(SRC[31:0])
DEST[127:64] ← Convert_Integer_To_Double_Precision_Floating_Point(SRC[63:32])
DEST[MAXVL-1:128] (unmodified)
```

Intel C/C++ Compiler Intrinsic Equivalent

VCVTDQ2PD __m512d _mm512_cvtepi32_pd(__m256i a);
VCVTDQ2PD __m512d _mm512_mask_cvtepi32_pd(__m512d s, __mmask8 k, __m256i a);
VCVTDQ2PD __m512d _mm512_maskz_cvtepi32_pd(__mmask8 k, __m256i a);
VCVTDQ2PD __m256d _mm256_cvtepi32_pd (__m128i src);
VCVTDQ2PD __m256d _mm256_mask_cvtepi32_pd(__m256d s, __mmask8 k, __m256i a);
VCVTDQ2PD __m256d _mm256_maskz_cvtepi32_pd(__mmask8 k, __m256i a);
VCVTDQ2PD __m128d _mm_mask_cvtepi32_pd(__m128d s, __mmask8 k, __m128i a);
VCVTDQ2PD __m128d _mm_maskz_cvtepi32_pd(__mmask8 k, __m128i a);
CVTDQ2PD __m128d _mm_cvtepi32_pd (__m128i src)

Other Exceptions

VEX-encoded instructions, see Exceptions Type 5;

EVEX-encoded instructions, see Exceptions Type E5.

#UD	If VEX.vvvv != 1111B or EVEX.vvvv != 1111B.

CVTDQ2PS—Convert Packed Doubleword Integers to Packed Single-Precision Floating-Point Values

Opcode Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F 5B /r CVTDQ2PS xmm1, xmm2/m128	A	V/V	SSE2	Convert four packed signed doubleword integers from xmm2/mem to four packed single-precision floating-point values in xmm1.
VEX.128.0F.WIG 5B /r VCVTDQ2PS xmm1, xmm2/m128	A	V/V	AVX	Convert four packed signed doubleword integers from xmm2/mem to four packed single-precision floating-point values in xmm1.
VEX.256.0F.WIG 5B /r VCVTDQ2PS ymm1, ymm2/m256	A	V/V	AVX	Convert eight packed signed doubleword integers from ymm2/mem to eight packed single-precision floating-point values in ymm1.
EVEX.128.0F.W0 5B /r VCVTDQ2PS xmm1 {k1}{z}, xmm2/m128/m32bcst	B	V/V	AVX512VL AVX512F	Convert four packed signed doubleword integers from xmm2/m128/m32bcst to four packed single-precision floating-point values in xmm1with writemask k1.
EVEX.256.0F.W0 5B /r VCVTDQ2PS ymm1 {k1}{z}, ymm2/m256/m32bcst	B	V/V	AVX512VL AVX512F	Convert eight packed signed doubleword integers from ymm2/m256/m32bcst to eight packed single-precision floating-point values in ymm1with writemask k1.
EVEX.512.0F.W0 5B /r VCVTDQ2PS zmm1 {k1}{z}, zmm2/m512/m32bcst{er}	B	V/V	AVX512F	Convert sixteen packed signed doubleword integers from zmm2/m512/m32bcst to sixteen packed single-precision floating-point values in zmm1with writemask k1.

Instruction Operand Encoding

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

Description

Converts four, eight or sixteen packed signed doubleword integers in the source operand to four, eight or sixteen packed single-precision floating-point values in the destination operand.

EVEX encoded versions: The source operand can be a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 32-bit memory location. The destination operand is a ZMM/YMM/XMM register conditionally updated with writemask k1.

VEX.256 encoded version: The source operand is a YMM register or 256- bit memory location. The destination operand is a YMM register. Bits (MAXVL-1:256) of the corresponding register destination are zeroed.

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

128-bit Legacy SSE version: The source operand is an XMM register or 128- bit memory location. The destination operand is an XMM register. The upper Bits (MAXVL-1:128) of the corresponding register destination are unmodified.

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

Operation

VCVTDQ2PS (EVEX encoded versions) when SRC operand is a register

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
IF (VL = 512) AND (EVEX.b = 1)
    THEN
        SET_RM(EVEX.RC);            ; refer to Table 2-4 in the Intel® Architecture Instruction Set Extensions Programming Reference
    ELSE
        SET_RM(MXCSR.RM);           ; refer to Table 2-4 in the Intel® Architecture Instruction Set Extensions Programming Reference
FI;

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

VCVTDQ2PS (EVEX encoded versions) when SRC operand is a memory source

```
(KL, VL) = (4, 128), (8, 256), (16, 512)

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

VCVTDQ2PS (VEX.256 encoded version)
DEST[31:0] ← Convert_Integer_To_Single_Precision_Floating_Point(SRC[31:0])
DEST[63:32] ← Convert_Integer_To_Single_Precision_Floating_Point(SRC[63:32])
DEST[95:64] ← Convert_Integer_To_Single_Precision_Floating_Point(SRC[95:64])
DEST[127:96] ← Convert_Integer_To_Single_Precision_Floating_Point(SRC[127:96)
DEST[159:128] ← Convert_Integer_To_Single_Precision_Floating_Point(SRC[159:128])
DEST[191:160] ← Convert_Integer_To_Single_Precision_Floating_Point(SRC[191:160])
DEST[223:192] ← Convert_Integer_To_Single_Precision_Floating_Point(SRC[223:192])
DEST[255:224] ← Convert_Integer_To_Single_Precision_Floating_Point(SRC[255:224)
DEST[MAXVL-1:256] ← 0

VCVTDQ2PS (VEX.128 encoded version)
DEST[31:0] ← Convert_Integer_To_Single_Precision_Floating_Point(SRC[31:0])
DEST[63:32] ← Convert_Integer_To_Single_Precision_Floating_Point(SRC[63:32])
DEST[95:64] ← Convert_Integer_To_Single_Precision_Floating_Point(SRC[95:64])
DEST[127:96] ← Convert_Integer_To_Single_Precision_Floating_Point(SRC[127z:96)
DEST[MAXVL-1:128] ← 0

CVTDQ2PS (128-bit Legacy SSE version)
DEST[31:0] ← Convert_Integer_To_Single_Precision_Floating_Point(SRC[31:0])
DEST[63:32] ← Convert_Integer_To_Single_Precision_Floating_Point(SRC[63:32])
DEST[95:64] ← Convert_Integer_To_Single_Precision_Floating_Point(SRC[95:64])
DEST[127:96] ← Convert_Integer_To_Single_Precision_Floating_Point(SRC[127z:96)
DEST[MAXVL-1:128] (unmodified)

Intel C/C++ Compiler Intrinsic Equivalent

VCVTDQ2PS __m512 _mm512_cvtepi32_ps(__m512i a);
VCVTDQ2PS __m512 _mm512_mask_cvtepi32_ps(__m512 s, __mmask16 k, __m512i a);
VCVTDQ2PS __m512 _mm512_maskz_cvtepi32_ps(__mmask16 k, __m512i a);
VCVTDQ2PS __m512 _mm512_cvt_roundepi32_ps(__m512i a, int r);
VCVTDQ2PS __m512 _mm512_mask_cvt_roundepi_ps(__m512 s, __mmask16 k, __m512i a, int r);
VCVTDQ2PS __m512 _mm512_maskz_cvt_roundepi32_ps(__mmask16 k, __m512i a, int r);
VCVTDQ2PS __m256 _mm256_mask_cvtepi32_ps(__m256 s, __mmask8 k, __m256i a);
VCVTDQ2PS __m256 _mm256_maskz_cvtepi32_ps(__mmask8 k, __m256i a);
VCVTDQ2PS __m128 _mm_mask_cvtepi32_ps(__m128 s, __mmask8 k, __m128i a);
VCVTDQ2PS __m128 _mm_maskz_cvtepi32_ps(__mmask8 k, __m128i a);
CVTDQ2PS __m256 _mm256_cvtepi32_ps (__m256i src)
CVTDQ2PS __m128 _mm_cvtepi32_ps (__m128i src)

SIMD Floating-Point Exceptions

Precision

Other Exceptions

VEX-encoded instructions, see Exceptions Type 2;

EVEX-encoded instructions, see Exceptions Type E2.

#UD	If VEX.vvvv != 1111B or EVEX.vvvv != 1111B.

CVTPD2DQ—Convert Packed Double-Precision Floating-Point Values to Packed Doubleword Integers

Opcode Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
F2 0F E6 /r CVTPD2DQ xmm1, xmm2/m128	A	V/V	SSE2	Convert two packed double-precision floating-point values in xmm2/mem to two signed doubleword integers in xmm1.
VEX.128.F2.0F.WIG E6 /r VCVTPD2DQ xmm1, xmm2/m128	A	V/V	AVX	Convert two packed double-precision floating-point values in xmm2/mem to two signed doubleword integers in xmm1.
VEX.256.F2.0F.WIG E6 /r VCVTPD2DQ xmm1, ymm2/m256	A	V/V	AVX	Convert four packed double-precision floating-point values in ymm2/mem to four signed doubleword integers in xmm1.
EVEX.128.F2.0F.W1 E6 /r VCVTPD2DQ xmm1 {k1}{z}, xmm2/m128/m64bcst	B	V/V	AVX512VL AVX512F	Convert two packed double-precision floating-point values in xmm2/m128/m64bcst to two signed doubleword integers in xmm1 subject to writemask k1.
EVEX.256.F2.0F.W1 E6 /r VCVTPD2DQ xmm1 {k1}{z}, ymm2/m256/m64bcst	B	V/V	AVX512VL AVX512F	Convert four packed double-precision floating-point values in ymm2/m256/m64bcst to four signed doubleword integers in xmm1 subject to writemask k1.
EVEX.512.F2.0F.W1 E6 /r VCVTPD2DQ ymm1 {k1}{z}, zmm2/m512/m64bcst{er}	B	V/V	AVX512F	Convert eight packed double-precision floating-point values in zmm2/m512/m64bcst to eight signed doubleword integers in ymm1 subject to writemask k1.

Instruction Operand Encoding

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

Description

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

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

EVEX encoded versions: The source operand is a ZMM/YMM/XMM register, a 512-bit memory location, or a 512-bit vector broadcasted from a 64-bit memory location. The destination operand is a ZMM/YMM/XMM register conditionally updated with writemask k1. The upper bits (MAXVL-1:256/128/64) of the corresponding destination are zeroed.

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

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

128-bit Legacy SSE version: The source operand is an XMM register or 128- bit memory location. The destination operand is an XMM register. Bits[127:64] of the destination XMM register are zeroed. However, the upper bits (MAXVL-1:128) of the corresponding ZMM register destination are unmodified.

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

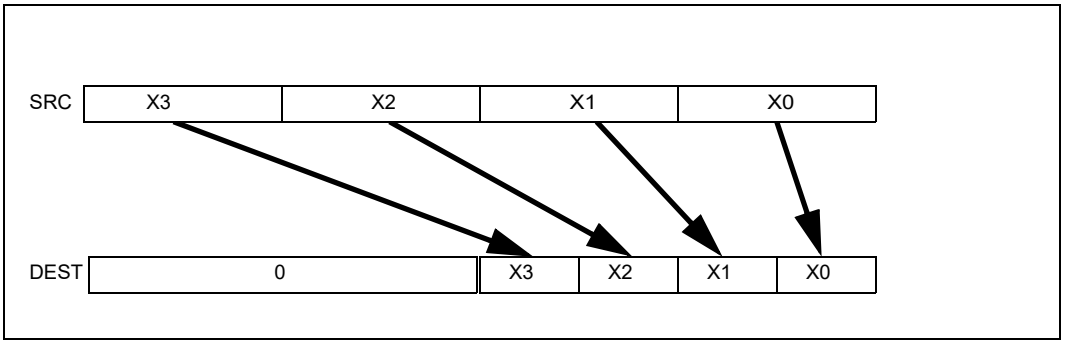


Figure 3-12. VCVTPD2DQ (VEX.256 encoded version)

Operation

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

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
IF (VL = 512) AND (EVEX.b = 1)
    THEN
        SET_RM(EVEX.RC);
    ELSE
        SET_RM(MXCSR.RM);
FI;

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

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

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j ← 0 TO KL-1
    i ← j * 32
    k ← j * 64
    IF k1[j] OR *no writemask*
        THEN
            IF (EVEX.b = 1)
                THEN
                    DEST[i+31:i] ←
            Convert_Double_Precision_Floating_Point_To_Integer(SRC[63:0])
                ELSE
                    DEST[i+31:i] ←
            Convert_Double_Precision_Floating_Point_To_Integer(SRC[k+63:k])
            FI;
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE                                ; zeroing-masking
                    DEST[i+31:i] ← 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL/2] ← 0
```

VCVTPD2DQ (VEX.256 encoded version)

```
DEST[31:0] ←Convert_Double_Precision_Floating_Point_To_Integer(SRC[63:0])
DEST[63:32] ←Convert_Double_Precision_Floating_Point_To_Integer(SRC[127:64])
DEST[95:64] ←Convert_Double_Precision_Floating_Point_To_Integer(SRC[191:128])
DEST[127:96] ←Convert_Double_Precision_Floating_Point_To_Integer(SRC[255:192)
DEST[MAXVL-1:128]←0
```

VCVTPD2DQ (VEX.128 encoded version)

```
DEST[31:0] ←Convert_Double_Precision_Floating_Point_To_Integer(SRC[63:0])
DEST[63:32] ←Convert_Double_Precision_Floating_Point_To_Integer(SRC[127:64])
DEST[MAXVL-1:64]←0
```

CVTPD2DQ (128-bit Legacy SSE version)

```
DEST[31:0] ←Convert_Double_Precision_Floating_Point_To_Integer(SRC[63:0])
DEST[63:32] ←Convert_Double_Precision_Floating_Point_To_Integer(SRC[127:64])
DEST[127:64] ←0
DEST[MAXVL-1:128] (unmodified)
```

Intel C/C++ Compiler Intrinsic Equivalent

VCVTPD2DQ __m256i _mm512_cvtpd_epi32(__m512d a);
VCVTPD2DQ __m256i _mm512_mask_cvtpd_epi32(__m256i s, __mmask8 k, __m512d a);
VCVTPD2DQ __m256i _mm512_maskz_cvtpd_epi32(__mmask8 k, __m512d a);
VCVTPD2DQ __m256i _mm512_cvt_roundpd_epi32(__m512d a, int r);
VCVTPD2DQ __m256i _mm512_mask_cvt_roundpd_epi32(__m256i s, __mmask8 k, __m512d a, int r);
VCVTPD2DQ __m256i _mm512_maskz_cvt_roundpd_epi32(__mmask8 k, __m512d a, int r);
VCVTPD2DQ __m128i _mm256_mask_cvtpd_epi32(__m128i s, __mmask8 k, __m256d a);
VCVTPD2DQ __m128i _mm256_maskz_cvtpd_epi32(__mmask8 k, __m256d a);
VCVTPD2DQ __m128i _mm_mask_cvtpd_epi32(__m128i s, __mmask8 k, __m128d a);
VCVTPD2DQ __m128i _mm_maskz_cvtpd_epi32(__mmask8 k, __m128d a);
VCVTPD2DQ __m128i _mm256_cvtpd_epi32 (__m256d src)
CVTPD2DQ __m128i _mm_cvtpd_epi32 (__m128d src)

SIMD Floating-Point Exceptions

Invalid, Precision

Other Exceptions

See Exceptions Type 2; additionally

EVEX-encoded instructions, see Exceptions Type E2.

#UD	If VEX.vvvv != 1111B or EVEX.vvvv != 1111B.

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

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

Instruction Operand Encoding

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

Description

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

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

When a conversion is inexact, the value returned is rounded according to the rounding control bits in the MXCSR register. If a converted result is larger than the maximum signed doubleword integer, the floating-point invalid exception is raised, and if this exception is masked, the indefinite integer value (80000000H) is returned.

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

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

Operation

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

Intel C/C++ Compiler Intrinsic Equivalent

CVTPD1PI: __m64 _mm_cvtpd_pi32(__m128d a)

SIMD Floating-Point Exceptions

Invalid, Precision.

Other Exceptions

See Table 22-4, "Exception Conditions for Legacy SIMD/MMX Instructions with FP Exception and 16-Byte Alignment," in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3B*.

CVTPD2PS—Convert Packed Double-Precision Floating-Point Values to Packed Single-Precision Floating-Point Values

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 5A /r CVTPD2PS xmm1, xmm2/m128	A	V/V	SSE2	Convert two packed double-precision floating-point values in xmm2/mem to two single-precision floating-point values in xmm1.
VEX.128.66.0F.WIG 5A /r VCVTPD2PS xmm1, xmm2/m128	A	V/V	AVX	Convert two packed double-precision floating-point values in xmm2/mem to two single-precision floating-point values in xmm1.
VEX.256.66.0F.WIG 5A /r VCVTPD2PS xmm1, ymm2/m256	A	V/V	AVX	Convert four packed double-precision floating-point values in ymm2/mem to four single-precision floating-point values in xmm1.
EVEX.128.66.0F.W1 5A /r VCVTPD2PS xmm1 {k1}{z}, xmm2/m128/m64bcst	B	V/V	AVX512VL AVX512F	Convert two packed double-precision floating-point values in xmm2/m128/m64bcst to two single-precision floating-point values in xmm1with writemask k1.
EVEX.256.66.0F.W1 5A /r VCVTPD2PS xmm1 {k1}{z}, ymm2/m256/m64bcst	B	V/V	AVX512VL AVX512F	Convert four packed double-precision floating-point values in ymm2/m256/m64bcst to four single-precision floating-point values in xmm1with writemask k1.
EVEX.512.66.0F.W1 5A /r VCVTPD2PS ymm1 {k1}{z}, zmm2/m512/m64bcst{er}	B	V/V	AVX512F	Convert eight packed double-precision floating-point values in zmm2/m512/m64bcst to eight single-precision floating-point values in ymm1with writemask k1.

Instruction Operand Encoding

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

Description

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

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

EVEX encoded versions: The source operand is a ZMM/YMM/XMM register, a 512/256/128-bit memory location, or a 512/256/128-bit vector broadcasted from a 64-bit memory location. The destination operand is a YMM/XMM/XMM (low 64-bits) register conditionally updated with writemask k1. The upper bits (MAXVL-1:256/128/64) of the corresponding destination are zeroed.

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

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

128-bit Legacy SSE version: The source operand is an XMM register or 128- bit memory location. The destination operand is an XMM register. Bits[127:64] of the destination XMM register are zeroed. However, the upper Bits (MAXVL-1:128) of the corresponding ZMM register destination are unmodified.

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

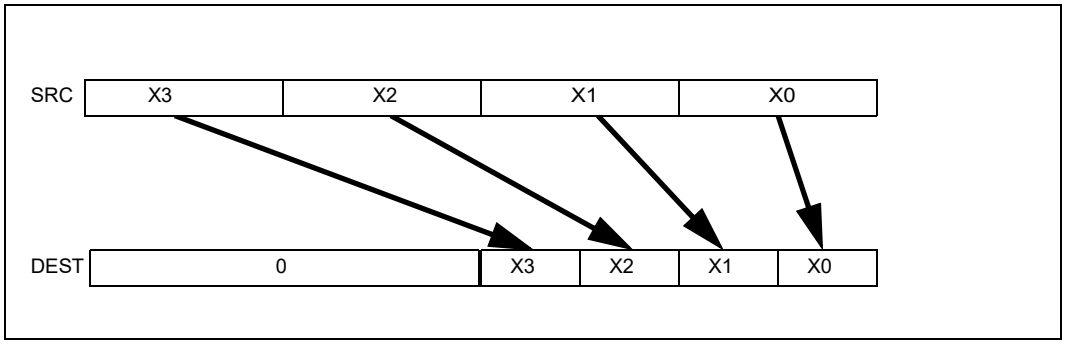


Figure 3-13. VCVTPD2PS (VEX.256 encoded version)

Operation

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

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
IF (VL = 512) AND (EVEX.b = 1)
    THEN
        SET_RM(EVEX.RC);
    ELSE
        SET_RM(MXCSR.RM);
FI;

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

VCVTPD2PS (EVEX encoded version) when src operand is a memory source
(KL, VL) = (2, 128), (4, 256), (8, 512)

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

VCVTPD2PS (VEX.256 encoded version)

```
DEST[31:0] ← Convert_Double_Precision_To_Single_Precision_Floating_Point(SRC[63:0])
DEST[63:32] ← Convert_Double_Precision_To_Single_Precision_Floating_Point(SRC[127:64])
DEST[95:64] ← Convert_Double_Precision_To_Single_Precision_Floating_Point(SRC[191:128])
DEST[127:96] ← Convert_Double_Precision_To_Single_Precision_Floating_Point(SRC[255:192)
DEST[MAXVL-1:128] ← 0
```

VCVTPD2PS (VEX.128 encoded version)

```
DEST[31:0] ← Convert_Double_Precision_To_Single_Precision_Floating_Point(SRC[63:0])
DEST[63:32] ← Convert_Double_Precision_To_Single_Precision_Floating_Point(SRC[127:64])
DEST[MAXVL-1:64] ← 0
```

CVTPD2PS (128-bit Legacy SSE version)

```
DEST[31:0] ← Convert_Double_Precision_To_Single_Precision_Floating_Point(SRC[63:0])
DEST[63:32] ← Convert_Double_Precision_To_Single_Precision_Floating_Point(SRC[127:64])
DEST[127:64] ← 0
DEST[MAXVL-1:128] (unmodified)
```

Intel C/C++ Compiler Intrinsic Equivalent

VCVTPD2PS __m256 _mm512_cvtpd_ps(__m512d a);
VCVTPD2PS __m256 _mm512_mask_cvtpd_ps(__m256 s, __mmask8 k, __m512d a);
VCVTPD2PS __m256 _mm512_maskz_cvtpd_ps(__mmask8 k, __m512d a);
VCVTPD2PS __m256 _mm512_cvt_roundpd_ps(__m512d a, int r);
VCVTPD2PS __m256 _mm512_mask_cvt_roundpd_ps(__m256 s, __mmask8 k, __m512d a, int r);
VCVTPD2PS __m256 _mm512_maskz_cvt_roundpd_ps(__mmask8 k, __m512d a, int r);
VCVTPD2PS __m128 _mm256_mask_cvtpd_ps(__m128 s, __mmask8 k, __m256d a);
VCVTPD2PS __m128 _mm256_maskz_cvtpd_ps(__mmask8 k, __m256d a);
VCVTPD2PS __m128 _mm_mask_cvtpd_ps(__m128 s, __mmask8 k, __m128d a);
VCVTPD2PS __m128 _mm_maskz_cvtpd_ps(__mmask8 k, __m128d a);
VCVTPD2PS __m128 _mm256_cvtpd_ps (__m256d a)
CVTPD2PS __m128 _mm_cvtpd_ps (__m128d a)

SIMD Floating-Point Exceptions

Invalid, Precision, Underflow, Overflow, Denormal

Other Exceptions

VEX-encoded instructions, see Exceptions Type 2;

EVEX-encoded instructions, see Exceptions Type E2.

#UD	If VEX.vvvv != 1111B or EVEX.vvvv != 1111B.

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

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

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

Instruction Operand Encoding

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

Description

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

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

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

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

Operation

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

Intel C/C++ Compiler Intrinsic Equivalent

CVTPI2PD: __m128d _mm_cvtpi32_pd(__m64 a)

SIMD Floating-Point Exceptions

None

Other Exceptions

See Table 22-6, "Exception Conditions for Legacy SIMD/MMX Instructions with XMM and without FP Exception," in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3B*.

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

Opcode/ Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
NP 0F 2A /r CVTPI2PS *xmm, mm/m64*	RM	Valid	Valid	Convert two signed doubleword integers from *mm/m64* to two single-precision floating-point values in *xmm*.

Instruction Operand Encoding

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

Description

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

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

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

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

Operation

```
DEST[31:0] ← Convert_Integer_To_Single_Precision_Floating_Point(SRC[31:0]);
DEST[63:32] ← Convert_Integer_To_Single_Precision_Floating_Point(SRC[63:32]);
(* High quadword of destination unchanged *)
```

Intel C/C++ Compiler Intrinsic Equivalent

CVTPI2PS: __m128 _mm_cvtpi32_ps(__m128 a, __m64 b)

SIMD Floating-Point Exceptions

Precision

Other Exceptions

See Table 22-5, "Exception Conditions for Legacy SIMD/MMX Instructions with XMM and FP Exception," in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3B*.

CVTPS2DQ—Convert Packed Single-Precision Floating-Point Values to Packed Signed Doubleword Integer Values

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 5B /r CVTPS2DQ xmm1, xmm2/m128	A	V/V	SSE2	Convert four packed single-precision floating-point values from xmm2/mem to four packed signed doubleword values in xmm1.
VEX.128.66.0F.WIG 5B /r VCVTPS2DQ xmm1, xmm2/m128	A	V/V	AVX	Convert four packed single-precision floating-point values from xmm2/mem to four packed signed doubleword values in xmm1.
VEX.256.66.0F.WIG 5B /r VCVTPS2DQ ymm1, ymm2/m256	A	V/V	AVX	Convert eight packed single-precision floating-point values from ymm2/mem to eight packed signed doubleword values in ymm1.
EVEX.128.66.0F.W0 5B /r VCVTPS2DQ xmm1 {k1}{z}, xmm2/m128/m32bcst	B	V/V	AVX512VL AVX512F	Convert four packed single precision floating-point values from xmm2/m128/m32bcst to four packed signed doubleword values in xmm1 subject to writemask k1.
EVEX.256.66.0F.W0 5B /r VCVTPS2DQ ymm1 {k1}{z}, ymm2/m256/m32bcst	B	V/V	AVX512VL AVX512F	Convert eight packed single precision floating-point values from ymm2/m256/m32bcst to eight packed signed doubleword values in ymm1 subject to writemask k1.
EVEX.512.66.0F.W0 5B /r VCVTPS2DQ zmm1 {k1}{z}, zmm2/m512/m32bcst{er}	B	V/V	AVX512F	Convert sixteen packed single-precision floating-point values from zmm2/m512/m32bcst to sixteen packed signed doubleword values in zmm1 subject to writemask k1.

Instruction Operand Encoding

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

Description

Converts four, eight or sixteen packed single-precision floating-point values in the source operand to four, eight or sixteen signed doubleword integers in the destination operand.

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

EVEX encoded versions: The source operand is a ZMM register, a 512-bit memory location or a 512-bit vector broadcasted from a 32-bit memory location. The destination operand is a ZMM register conditionally updated with writemask k1.

VEX.256 encoded version: The source operand is a YMM register or 256- bit memory location. The destination operand is a YMM register. The upper bits (MAXVL-1:256) of the corresponding ZMM register destination are zeroed.

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

128-bit Legacy SSE version: The source operand is an XMM register or 128- bit memory location. The destination operand is an XMM register. The upper bits (MAXVL-1:128) of the corresponding ZMM register destination are unmodified.

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

Operation

VCVTPS2DQ (encoded versions) when src operand is a register

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
IF (VL = 512) AND (EVEX.b = 1)
    THEN
        SET_RM(EVEX.RC);
    ELSE
        SET_RM(MXCSR.RM);
FI;

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

VCVTPS2DQ (EVEX encoded versions) when src operand is a memory source

```
(KL, VL) = (4, 128), (8, 256), (16, 512)

FOR j ← 0 TO 15
    i ← j * 32
    IF k1[j] OR *no writemask*
        THEN
            IF (EVEX.b = 1)
                THEN
                    DEST[i+31:i] ←
            Convert_Single_Precision_Floating_Point_To_Integer(SRC[31:0])
                ELSE
                    DEST[i+31:i] ←
            Convert_Single_Precision_Floating_Point_To_Integer(SRC[i+31:i])
            FI;
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE                            ; zeroing-masking
                    DEST[i+31:i] ← 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0
```

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

VCVTPS2DQ (VEX.128 encoded version)
DEST[31:0] ←Convert_Single_Precision_Floating_Point_To_Integer(SRC[31:0])
DEST[63:32] ←Convert_Single_Precision_Floating_Point_To_Integer(SRC[63:32])
DEST[95:64] ←Convert_Single_Precision_Floating_Point_To_Integer(SRC[95:64])
DEST[127:96] ←Convert_Single_Precision_Floating_Point_To_Integer(SRC[127:96])
DEST[MAXVL-1:128] ←0

CVTPS2DQ (128-bit Legacy SSE version)
DEST[31:0] ←Convert_Single_Precision_Floating_Point_To_Integer(SRC[31:0])
DEST[63:32] ←Convert_Single_Precision_Floating_Point_To_Integer(SRC[63:32])
DEST[95:64] ←Convert_Single_Precision_Floating_Point_To_Integer(SRC[95:64])
DEST[127:96] ←Convert_Single_Precision_Floating_Point_To_Integer(SRC[127:96])
DEST[MAXVL-1:128] (unmodified)

Intel C/C++ Compiler Intrinsic Equivalent

VCVTPS2DQ __m512i _mm512_cvtps_epi32(__m512 a);
VCVTPS2DQ __m512i _mm512_mask_cvtps_epi32(__m512i s, __mmask16 k, __m512 a);
VCVTPS2DQ __m512i _mm512_maskz_cvtps_epi32(__mmask16 k, __m512 a);
VCVTPS2DQ __m512i _mm512_cvt_roundps_epi32(__m512 a, int r);
VCVTPS2DQ __m512i _mm512_mask_cvt_roundps_epi32(__m512i s, __mmask16 k, __m512 a, int r);
VCVTPS2DQ __m512i _mm512_maskz_cvt_roundps_epi32(__mmask16 k, __m512 a, int r);
VCVTPS2DQ __m256i _mm256_mask_cvtps_epi32(__m256i s, __mmask8 k, __m256 a);
VCVTPS2DQ __m256i _mm256_maskz_cvtps_epi32(__mmask8 k, __m256 a);
VCVTPS2DQ __m128i _mm_mask_cvtps_epi32(__m128i s, __mmask8 k, __m128 a);
VCVTPS2DQ __m128i _mm_maskz_cvtps_epi32(__mmask8 k, __m128 a);
VCVTPS2DQ __ m256i _mm256_cvtps_epi32 (__m256 a)
CVTPS2DQ __m128i _mm_cvtps_epi32 (__m128 a)

SIMD Floating-Point Exceptions

Invalid, Precision

Other Exceptions

VEX-encoded instructions, see Exceptions Type 2;

EVEX-encoded instructions, see Exceptions Type E2.

#UD	If VEX.vvvv != 1111B or EVEX.vvvv != 1111B.

CVTPS2PD—Convert Packed Single-Precision Floating-Point Values to Packed Double-Precision Floating-Point Values

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F 5A /r CVTPS2PD xmm1, xmm2/m64	A	V/V	SSE2	Convert two packed single-precision floating-point values in xmm2/m64 to two packed double-precision floating-point values in xmm1.
VEX.128.0F.WIG 5A /r VCVTPS2PD xmm1, xmm2/m64	A	V/V	AVX	Convert two packed single-precision floating-point values in xmm2/m64 to two packed double-precision floating-point values in xmm1.
VEX.256.0F.WIG 5A /r VCVTPS2PD ymm1, xmm2/m128	A	V/V	AVX	Convert four packed single-precision floating-point values in xmm2/m128 to four packed double-precision floating-point values in ymm1.
EVEX.128.0F.W0 5A /r VCVTPS2PD xmm1 {k1}{z}, xmm2/m64/m32bcst	B	V/V	AVX512VL AVX512F	Convert two packed single-precision floating-point values in xmm2/m64/m32bcst to packed double-precision floating-point values in xmm1 with writemask k1.
EVEX.256.0F.W0 5A /r VCVTPS2PD ymm1 {k1}{z}, xmm2/m128/m32bcst	B	V/V	AVX512VL	Convert four packed single-precision floating-point values in xmm2/m128/m32bcst to packed double-precision floating-point values in ymm1 with writemask k1.
EVEX.512.0F.W0 5A /r VCVTPS2PD zmm1 {k1}{z}, ymm2/m256/m32bcst{sae}	B	V/V	AVX512F	Convert eight packed single-precision floating-point values in ymm2/m256/b32bcst to eight packed double-precision floating-point values in zmm1 with writemask k1.

Instruction Operand Encoding

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

Description

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

EVEX encoded versions: The source operand is a YMM/XMM/XMM (low 64-bits) register, a 256/128/64-bit memory location or a 256/128/64-bit vector broadcasted from a 32-bit memory location. The destination operand is a ZMM/YMM/XMM register conditionally updated with writemask k1.

VEX.256 encoded version: The source operand is an XMM register or 128- bit memory location. The destination operand is a YMM register. Bits (MAXVL-1:256) of the corresponding destination ZMM register are zeroed.

VEX.128 encoded version: The source operand is an XMM register or 64- bit memory location. The destination operand is a XMM register. The upper Bits (MAXVL-1:128) of the corresponding ZMM register destination are zeroed.

128-bit Legacy SSE version: The source operand is an XMM register or 64- bit memory location. The destination operand is an XMM register. The upper Bits (MAXVL-1:128) of the corresponding ZMM register destination are unmodified.

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

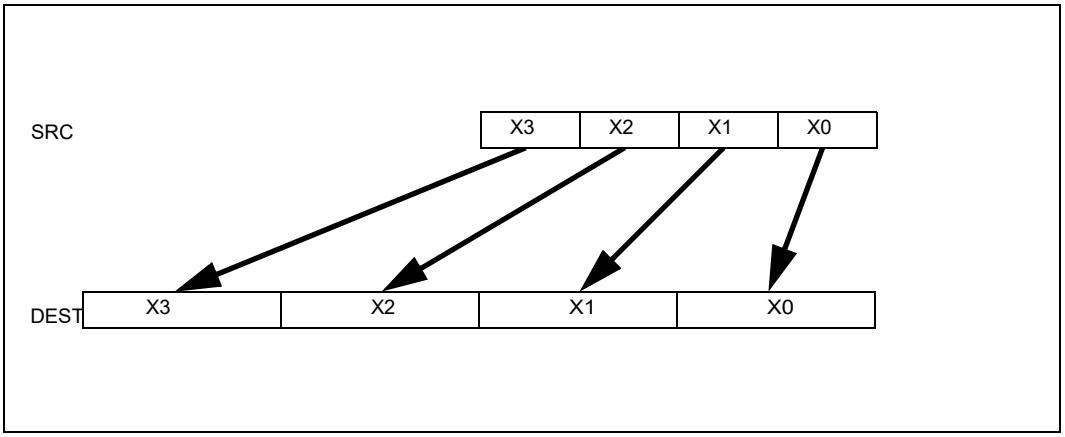


Figure 3-14. CVTPS2PD (VEX.256 encoded version)

Operation

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

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j ← 0 TO KL-1
    i ← j * 64
    k ← j * 32
    IF k1[j] OR *no writemask*
        THEN DEST[i+63:i] ←
            Convert_Single_Precision_To_Double_Precision_Floating_Point(SRC[k+31:k])
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE                            ; zeroing-masking
                    DEST[i+63:i] ← 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0
```

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

```
(KL, VL) = (2, 128), (4, 256), (8, 512)

FOR j ← 0 TO KL-1
    i ← j * 64
    k ← j * 32
    IF k1[j] OR *no writemask*
        THEN
            IF (EVEX.b = 1)
                THEN
                    DEST[i+63:i] ←
            Convert_Single_Precision_To_Double_Precision_Floating_Point(SRC[31:0])
                ELSE
                    DEST[i+63:i] ←
            Convert_Single_Precision_To_Double_Precision_Floating_Point(SRC[k+31:k])
            FI;
        ELSE
```

```
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE                                ; zeroing-masking
                    DEST[i+63:i] ← 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0
```

VCVTPS2PD (VEX.256 encoded version)

```
DEST[63:0] ← Convert_Single_Precision_To_Double_Precision_Floating_Point(SRC[31:0])
DEST[127:64] ← Convert_Single_Precision_To_Double_Precision_Floating_Point(SRC[63:32])
DEST[191:128] ← Convert_Single_Precision_To_Double_Precision_Floating_Point(SRC[95:64])
DEST[255:192] ← Convert_Single_Precision_To_Double_Precision_Floating_Point(SRC[127:96)
DEST[MAXVL-1:256] ← 0
```

VCVTPS2PD (VEX.128 encoded version)

```
DEST[63:0] ← Convert_Single_Precision_To_Double_Precision_Floating_Point(SRC[31:0])
DEST[127:64] ← Convert_Single_Precision_To_Double_Precision_Floating_Point(SRC[63:32])
DEST[MAXVL-1:128] ← 0
```

CVTPS2PD (128-bit Legacy SSE version)

```
DEST[63:0] ← Convert_Single_Precision_To_Double_Precision_Floating_Point(SRC[31:0])
DEST[127:64] ← Convert_Single_Precision_To_Double_Precision_Floating_Point(SRC[63:32])
DEST[MAXVL-1:128] (unmodified)
```

Intel C/C++ Compiler Intrinsic Equivalent

VCVTPS2PD __m512d _mm512_cvtps_pd(__m256 a);
VCVTPS2PD __m512d _mm512_mask_cvtps_pd(__m512d s, __mmask8 k, __m256 a);
VCVTPS2PD __m512d _mm512_maskz_cvtps_pd(__mmask8 k, __m256 a);
VCVTPS2PD __m512d _mm512_cvt_roundps_pd(__m256 a, int sae);
VCVTPS2PD __m512d _mm512_mask_cvt_roundps_pd(__m512d s, __mmask8 k, __m256 a, int sae);
VCVTPS2PD __m512d _mm512_maskz_cvt_roundps_pd(__mmask8 k, __m256 a, int sae);
VCVTPS2PD __m256d _mm256_mask_cvtps_pd(__m256d s, __mmask8 k, __m128 a);
VCVTPS2PD __m256d _mm256_maskz_cvtps_pd(__mmask8 k, __m128a);
VCVTPS2PD __m128d _mm_mask_cvtps_pd(__m128d s, __mmask8 k, __m128 a);
VCVTPS2PD __m128d _mm_maskz_cvtps_pd(__mmask8 k, __m128 a);
VCVTPS2PD __m256d _mm256_cvtps_pd (__m128 a)
CVTPS2PD __m128d _mm_cvtps_pd (__m128 a)

SIMD Floating-Point Exceptions

Invalid, Denormal

Other Exceptions

VEX-encoded instructions, see Exceptions Type 3;

EVEX-encoded instructions, see Exceptions Type E3.

#UD	If VEX.vvvv != 1111B or EVEX.vvvv != 1111B.

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

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

Instruction Operand Encoding

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

Description

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

The source operand can be an XMM register or a 128-bit memory location. The destination operand is an MMX technology register. When the source operand is an XMM register, the two single-precision floating-point values are contained in the low quadword of the register. When a conversion is inexact, the value returned is rounded according to the rounding control bits in the MXCSR register. If a converted result is larger than the maximum signed doubleword integer, the floating-point invalid exception is raised, and if this exception is masked, the indefinite integer value (80000000H) is returned.

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

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

Operation

```
DEST[31:0] ← Convert_Single_Precision_Floating_Point_To_Integer(SRC[31:0]);
DEST[63:32] ← Convert_Single_Precision_Floating_Point_To_Integer(SRC[63:32]);
```

Intel C/C++ Compiler Intrinsic Equivalent

CVTPS2PI: __m64 _mm_cvtps_pi32(__m128 a)

SIMD Floating-Point Exceptions

Invalid, Precision

Other Exceptions

See Table 22-5, "Exception Conditions for Legacy SIMD/MMX Instructions with XMM and FP Exception," in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3B*.

CVTSD2SI—Convert Scalar Double-Precision Floating-Point Value to Doubleword Integer

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
F2 0F 2D /r CVTSD2SI r32, xmm1/m64	A	V/V	SSE2	Convert one double-precision floating-point value from xmm1/m64 to one signed doubleword integer r32.
F2 REX.W 0F 2D /r CVTSD2SI r64, xmm1/m64	A	V/N.E.	SSE2	Convert one double-precision floating-point value from xmm1/m64 to one signed quadword integer sign-extended into r64.
VEX.LIG.F2.0F.W0 2D /r [1] VCVTSD2SI r32, xmm1/m64	A	V/V	AVX	Convert one double-precision floating-point value from xmm1/m64 to one signed doubleword integer r32.
VEX.LIG.F2.0F.W1 2D /r [1] VCVTSD2SI r64, xmm1/m64	A	V/N.E.[2]	AVX	Convert one double-precision floating-point value from xmm1/m64 to one signed quadword integer sign-extended into r64.
EVEX.LIG.F2.0F.W0 2D /r VCVTSD2SI r32, xmm1/m64{er}	B	V/V	AVX512F	Convert one double-precision floating-point value from xmm1/m64 to one signed doubleword integer r32.
EVEX.LIG.F2.0F.W1 2D /r VCVTSD2SI r64, xmm1/m64{er}	B	V/N.E.[2]	AVX512F	Convert one double-precision floating-point value from xmm1/m64 to one signed quadword integer sign-extended into r64.

NOTES:

1. Software should ensure VCVTSD2SI is encoded with VEX.L=0. Encoding VCVTSD2SI with VEX.L=1 may encounter unpredictable behavior across different processor generations.
2. VEX.W1/EVEX.W1 in non-64 bit is ignored; the instructions behaves as if the W0 version is used.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	NA	ModRM:reg (w)	ModRM:r/m (r)	NA	NA
B	Tuple1 Fixed	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

Description

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

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

If a converted result exceeds the range limits of signed doubleword integer (in non-64-bit modes or 64-bit mode with REX.W/VEX.W/EVEX.W=0), the floating-point invalid exception is raised, and if this exception is masked, the indefinite integer value (80000000H) is returned.

If a converted result exceeds the range limits of signed quadword integer (in 64-bit mode and REX.W/VEX.W/EVEX.W = 1), the floating-point invalid exception is raised, and if this exception is masked, the indefinite integer value (80000000_00000000H) is returned.

Legacy SSE instruction: Use of the REX.W prefix promotes the instruction to produce 64-bit data in 64-bit mode. See the summary chart at the beginning of this section for encoding data and limits.

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

Software should ensure VCVTSD2SI is encoded with VEX.L=0. Encoding VCVTSD2SI with VEX.L=1 may encounter unpredictable behavior across different processor generations.

Operation

VCVTSD2SI (EVEX encoded version)

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

(V)CVTSD2SI

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

Intel C/C++ Compiler Intrinsic Equivalent

VCVTSD2SI int _mm_cvtsd_i32(__m128d);
VCVTSD2SI int _mm_cvt_roundsd_i32(__m128d, int r);
VCVTSD2SI __int64 _mm_cvtsd_i64(__m128d);
VCVTSD2SI __int64 _mm_cvt_roundsd_i64(__m128d, int r);
CVTSD2SI __int64 _mm_cvtsd_si64(__m128d);
CVTSD2SI int _mm_cvtsd_si32(__m128d a)

SIMD Floating-Point Exceptions

Invalid, Precision

Other Exceptions

VEX-encoded instructions, see Exceptions Type 3;

EVEX-encoded instructions, see Exceptions Type E3NF.

#UD	If VEX.vvvv != 1111B or EVEX.vvvv != 1111B.

CVTSD2SS—Convert Scalar Double-Precision Floating-Point Value to Scalar Single-Precision Floating-Point Value

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
F2 0F 5A /r CVTSD2SS xmm1, xmm2/m64	A	V/V	SSE2	Convert one double-precision floating-point value in xmm2/m64 to one single-precision floating-point value in xmm1.
VEX.NDS.LIG.F2.0F.WIG 5A /r VCVTSD2SS xmm1,xmm2, xmm3/m64	B	V/V	AVX	Convert one double-precision floating-point value in xmm3/m64 to one single-precision floating-point value and merge with high bits in xmm2.
EVEX.NDS.LIG.F2.0F.W1 5A /r VCVTSD2SS xmm1 {k1}{z}, xmm2, xmm3/m64{er}	C	V/V	AVX512F	Convert one double-precision floating-point value in xmm3/m64 to one single-precision floating-point value and merge with high bits in xmm2 under writemask k1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	NA	ModRM:reg (w)	ModRM:r/m (r)	NA	NA
B	NA	ModRM:reg (w)	VEX.vvvv	ModRM:r/m (r)	NA
C	Tuple1 Scalar	ModRM:reg (w)	EVEX.vvvv	ModRM:r/m (r)	NA

Description

Converts a double-precision floating-point value in the "convert-from" source operand (the second operand in SSE2 version, otherwise the third operand) to a single-precision floating-point value in the destination operand.

When the "convert-from" operand is an XMM register, the double-precision floating-point value is contained in the low quadword of the register. The result is stored in the low doubleword of the destination operand. When the conversion is inexact, the value returned is rounded according to the rounding control bits in the MXCSR register.

128-bit Legacy SSE version: The "convert-from" source operand (the second operand) is an XMM register or memory location. Bits (MAXVL-1:32) of the corresponding destination register remain unchanged. The destination operand is an XMM register.

VEX.128 and EVEX encoded versions: The "convert-from" source operand (the third operand) can be an XMM register or a 64-bit memory location. The first source and destination operands are XMM registers. Bits (127:32) of the XMM register destination are copied from the corresponding bits in the first source operand. Bits (MAXVL-1:128) of the destination register are zeroed.

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

Software should ensure VCVTSD2SS is encoded with VEX.L=0. Encoding VCVTSD2SS with VEX.L=1 may encounter unpredictable behavior across different processor generations.

Operation

VCVTSD2SS (EVEX encoded version)

```
IF (SRC2 *is register*) AND (EVEX.b = 1)
    THEN
        SET_RM(EVEX.RC);
    ELSE
        SET_RM(MXCSR.RM);
FI;
IF k1[0] or *no writemask*
    THEN    DEST[31:0] ← Convert_Double_Precision_To_Single_Precision_Floating_Point(SRC2[63:0]);
    ELSE
        IF *merging-masking*                    ; merging-masking
            THEN *DEST[31:0] remains unchanged*
            ELSE                                ; zeroing-masking
                THEN DEST[31:0] ← 0
        FI;
FI;
DEST[127:32] ← SRC1[127:32]
DEST[MAXVL-1:128] ← 0
```

VCVTSD2SS (VEX.128 encoded version)

```
DEST[31:0] ←Convert_Double_Precision_To_Single_Precision_Floating_Point(SRC2[63:0]);
DEST[127:32] ←SRC1[127:32]
DEST[MAXVL-1:128] ←0
```

CVTSD2SS (128-bit Legacy SSE version)

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

Intel C/C++ Compiler Intrinsic Equivalent

VCVTSD2SS __m128 _mm_mask_cvtsd_ss(__m128 s, __mmask8 k, __m128 a, __m128d b);
VCVTSD2SS __m128 _mm_maskz_cvtsd_ss(__mmask8 k, __m128 a,__m128d b);
VCVTSD2SS __m128 _mm_cvt_roundsd_ss(__m128 a, __m128d b, int r);
VCVTSD2SS __m128 _mm_mask_cvt_roundsd_ss(__m128 s, __mmask8 k, __m128 a, __m128d b, int r);
VCVTSD2SS __m128 _mm_maskz_cvt_roundsd_ss(__mmask8 k, __m128 a,__m128d b, int r);
CVTSD2SS __m128_mm_cvtsd_ss(__m128 a, __m128d b)

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal

Other Exceptions

VEX-encoded instructions, see Exceptions Type 3.

EVEX-encoded instructions, see Exceptions Type E3.

CVTSI2SD—Convert Doubleword Integer to Scalar Double-Precision Floating-Point Value

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
F2 0F 2A /r CVTSI2SD xmm1, r32/m32	A	V/V	SSE2	Convert one signed doubleword integer from r32/m32 to one double-precision floating-point value in xmm1.
F2 REX.W 0F 2A /r CVTSI2SD xmm1, r/m64	A	V/N.E.	SSE2	Convert one signed quadword integer from r/m64 to one double-precision floating-point value in xmm1.
VEX.NDS.LIG.F2.0F.W0 2A /r VCVTSI2SD xmm1, xmm2, r/m32	B	V/V	AVX	Convert one signed doubleword integer from r/m32 to one double-precision floating-point value in xmm1.
VEX.NDS.LIG.F2.0F.W1 2A /r VCVTSI2SD xmm1, xmm2, r/m64	B	V/N.E.[1]	AVX	Convert one signed quadword integer from r/m64 to one double-precision floating-point value in xmm1.
EVEX.NDS.LIG.F2.0F.W0 2A /r VCVTSI2SD xmm1, xmm2, r/m32	C	V/V	AVX512F	Convert one signed doubleword integer from r/m32 to one double-precision floating-point value in xmm1.
EVEX.NDS.LIG.F2.0F.W1 2A /r VCVTSI2SD xmm1, xmm2, r/m64{er}	C	V/N.E.[1]	AVX512F	Convert one signed quadword integer from r/m64 to one double-precision floating-point value in xmm1.

NOTES:

1. VEX.W1/EVEX.W1 in non-64 bit is ignored; the instructions behaves as if the W0 version is used.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	NA	ModRM:reg (w)	ModRM:r/m (r)	NA	NA
B	NA	ModRM:reg (w)	VEX.vvvv	ModRM:r/m (r)	NA
C	Tuple1 Scalar	ModRM:reg (w)	EVEX.vvvv	ModRM:r/m (r)	NA

Description

Converts a signed doubleword integer (or signed quadword integer if operand size is 64 bits) in the "convert-from" source operand to a double-precision floating-point value in the destination operand. The result is stored in the low quadword of the destination operand, and the high quadword left unchanged. When conversion is inexact, the value returned is rounded according to the rounding control bits in the MXCSR register.

The second source operand can be a general-purpose register or a 32/64-bit memory location. The first source and destination operands are XMM registers.

128-bit Legacy SSE version: Use of the REX.W prefix promotes the instruction to 64-bit operands. The "convert-from" source operand (the second operand) is a general-purpose register or memory location. The destination is an XMM register Bits (MAXVL-1:64) of the corresponding destination register remain unchanged.

VEX.128 and EVEX encoded versions: The "convert-from" source operand (the third operand) can be a general-purpose register or a memory location. The first source and destination operands are XMM registers. Bits (127:64) of the XMM register destination are copied from the corresponding bits in the first source operand. Bits (MAXVL-1:128) of the destination register are zeroed.

EVEX.W0 version: attempt to encode this instruction with EVEX embedded rounding is ignored.

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

Software should ensure VCVTSI2SD is encoded with VEX.L=0. Encoding VCVTSI2SD with VEX.L=1 may encounter unpredictable behavior across different processor generations.

Operation

VCVTSI2SD (EVEX encoded version)

```
IF (SRC2 *is register*) AND (EVEX.b = 1)
    THEN
        SET_RM(EVEX.RC);
    ELSE
        SET_RM(MXCSR.RM);
FI;
IF 64-Bit Mode And OperandSize = 64
THEN
    DEST[63:0] ← Convert_Integer_To_Double_Precision_Floating_Point(SRC2[63:0]);
ELSE
    DEST[63:0] ← Convert_Integer_To_Double_Precision_Floating_Point(SRC2[31:0]);
FI;
DEST[127:64] ← SRC1[127:64]
DEST[MAXVL-1:128] ← 0
```

VCVTSI2SD (VEX.128 encoded version)

```
IF 64-Bit Mode And OperandSize = 64
THEN
    DEST[63:0] ←Convert_Integer_To_Double_Precision_Floating_Point(SRC2[63:0]);
ELSE
    DEST[63:0] ←Convert_Integer_To_Double_Precision_Floating_Point(SRC2[31:0]);
FI;
DEST[127:64] ←SRC1[127:64]
DEST[MAXVL-1:128] ←0
```

CVTSI2SD

```
IF 64-Bit Mode And OperandSize = 64
THEN
    DEST[63:0] ←Convert_Integer_To_Double_Precision_Floating_Point(SRC[63:0]);
ELSE
    DEST[63:0] ←Convert_Integer_To_Double_Precision_Floating_Point(SRC[31:0]);
FI;
DEST[MAXVL-1:64] (Unmodified)
```

Intel C/C++ Compiler Intrinsic Equivalent

VCVTSI2SD __m128d _mm_cvti32_sd(__m128d s, int a);
VCVTSI2SD __m128d _mm_cvti64_sd(__m128d s, __int64 a);
VCVTSI2SD __m128d _mm_cvt_roundi64_sd(__m128d s, __int64 a, int r);
CVTSI2SD __m128d _mm_cvtsi64_sd(__m128d s, __int64 a);
CVTSI2SD __m128d_mm_cvtsi32_sd(__m128d a, int b)

SIMD Floating-Point Exceptions

Precision

Other Exceptions

VEX-encoded instructions, see Exceptions Type 3 if W1, else Type 5.

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

CVTSI2SS—Convert Doubleword Integer to Scalar Single-Precision Floating-Point Value

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
F3 0F 2A /r CVTSI2SS xmm1, r/m32	A	V/V	SSE	Convert one signed doubleword integer from r/m32 to one single-precision floating-point value in xmm1.
F3 REX.W 0F 2A /r CVTSI2SS xmm1, r/m64	A	V/N.E.	SSE	Convert one signed quadword integer from r/m64 to one single-precision floating-point value in xmm1.
VEX.NDS.LIG.F3.0F.W0 2A /r VCVTSI2SS xmm1, xmm2, r/m32	B	V/V	AVX	Convert one signed doubleword integer from r/m32 to one single-precision floating-point value in xmm1.
VEX.NDS.LIG.F3.0F.W1 2A /r VCVTSI2SS xmm1, xmm2, r/m64	B	V/N.E.[1]	AVX	Convert one signed quadword integer from r/m64 to one single-precision floating-point value in xmm1.
EVEX.NDS.LIG.F3.0F.W0 2A /r VCVTSI2SS xmm1, xmm2, r/m32{er}	C	V/V	AVX512F	Convert one signed doubleword integer from r/m32 to one single-precision floating-point value in xmm1.
EVEX.NDS.LIG.F3.0F.W1 2A /r VCVTSI2SS xmm1, xmm2, r/m64{er}	C	V/N.E.[1]	AVX512F	Convert one signed quadword integer from r/m64 to one single-precision floating-point value in xmm1.

NOTES:

1. VEX.W1/EVEX.W1 in non-64 bit is ignored; the instructions behaves as if the W0 version is used.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	NA	ModRM:reg (w)	ModRM:r/m (r)	NA	NA
B	NA	ModRM:reg (w)	VEX.vvvv	ModRM:r/m (r)	NA
C	Tuple1 Scalar	ModRM:reg (w)	EVEX.vvvv	ModRM:r/m (r)	NA

Description

Converts a signed doubleword integer (or signed quadword integer if operand size is 64 bits) in the "convert-from" source operand to a single-precision floating-point value in the destination operand (first operand). The "convert-from" source operand can be a general-purpose register or a memory location. The destination operand is an XMM register. The result is stored in the low doubleword of the destination operand, and the upper three doublewords are left unchanged. When a conversion is inexact, the value returned is rounded according to the rounding control bits in the MXCSR register or the embedded rounding control bits.

128-bit Legacy SSE version: In 64-bit mode, Use of the REX.W prefix promotes the instruction to use 64-bit input value. The "convert-from" source operand (the second operand) is a general-purpose register or memory location. Bits (MAXVL-1:32) of the corresponding destination register remain unchanged.

VEX.128 and EVEX encoded versions: The "convert-from" source operand (the third operand) can be a general-purpose register or a memory location. The first source and destination operands are XMM registers. Bits (127:32) of the XMM register destination are copied from corresponding bits in the first source operand. Bits (MAXVL-1:128) of the destination register are zeroed.

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

Software should ensure VCVTSI2SS is encoded with VEX.L=0. Encoding VCVTSI2SS with VEX.L=1 may encounter unpredictable behavior across different processor generations.

Operation

VCVTSI2SS (EVEX encoded version)

```
IF (SRC2 *is register*) AND (EVEX.b = 1)
    THEN
        SET_RM(EVEX.RC);
    ELSE
        SET_RM(MXCSR.RM);
FI;
IF 64-Bit Mode And OperandSize = 64
THEN
    DEST[31:0] ← Convert_Integer_To_Single_Precision_Floating_Point(SRC[63:0]);
ELSE
    DEST[31:0] ← Convert_Integer_To_Single_Precision_Floating_Point(SRC[31:0]);
FI;
DEST[127:32] ← SRC1[127:32]
DEST[MAXVL-1:128] ← 0
```

VCVTSI2SS (VEX.128 encoded version)

```
IF 64-Bit Mode And OperandSize = 64
THEN
    DEST[31:0] ←Convert_Integer_To_Single_Precision_Floating_Point(SRC[63:0]);
ELSE
    DEST[31:0] ←Convert_Integer_To_Single_Precision_Floating_Point(SRC[31:0]);
FI;
DEST[127:32] ←SRC1[127:32]
DEST[MAXVL-1:128] ←0
```

CVTSI2SS (128-bit Legacy SSE version)

```
IF 64-Bit Mode And OperandSize = 64
THEN
    DEST[31:0] ←Convert_Integer_To_Single_Precision_Floating_Point(SRC[63:0]);
ELSE
    DEST[31:0] ←Convert_Integer_To_Single_Precision_Floating_Point(SRC[31:0]);
FI;
DEST[MAXVL-1:32] (Unmodified)
```

Intel C/C++ Compiler Intrinsic Equivalent

VCVTSI2SS __m128 _mm_cvti32_ss(__m128 s, int a);
VCVTSI2SS __m128 _mm_cvt_roundi32_ss(__m128 s, int a, int r);
VCVTSI2SS __m128 _mm_cvti64_ss(__m128 s, __int64 a);
VCVTSI2SS __m128 _mm_cvt_roundi64_ss(__m128 s, __int64 a, int r);
CVTSI2SS __m128 _mm_cvtsi64_ss(__m128 s, __int64 a);
CVTSI2SS __m128 _mm_cvtsi32_ss(__m128 a, int b);

SIMD Floating-Point Exceptions

Precision

Other Exceptions

VEX-encoded instructions, see Exceptions Type 3.
EVEX-encoded instructions, see Exceptions Type E3NF.

CVTSS2SD—Convert Scalar Single-Precision Floating-Point Value to Scalar Double-Precision Floating-Point Value

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
F3 0F 5A /r CVTSS2SD xmm1, xmm2/m32	A	V/V	SSE2	Convert one single-precision floating-point value in xmm2/m32 to one double-precision floating-point value in xmm1.
VEX.NDS.LIG.F3.0F.WIG 5A /r VCVTSS2SD xmm1, xmm2, xmm3/m32	B	V/V	AVX	Convert one single-precision floating-point value in xmm3/m32 to one double-precision floating-point value and merge with high bits of xmm2.
EVEX.NDS.LIG.F3.0F.W0 5A /r VCVTSS2SD xmm1 {k1}{z}, xmm2, xmm3/m32{sae}	C	V/V	AVX512F	Convert one single-precision floating-point value in xmm3/m32 to one double-precision floating-point value and merge with high bits of xmm2 under writemask k1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	NA	ModRM:reg (w)	ModRM:r/m (r)	NA	NA
B	NA	ModRM:reg (w)	VEX.vvvv	ModRM:r/m (r)	NA
C	Tuple1 Scalar	ModRM:reg (w)	EVEX.vvvv	ModRM:r/m (r)	NA

Description

Converts a single-precision floating-point value in the "convert-from" source operand to a double-precision floating-point value in the destination operand. When the "convert-from" source operand is an XMM register, the single-precision floating-point value is contained in the low doubleword of the register. The result is stored in the low quadword of the destination operand.

128-bit Legacy SSE version: The "convert-from" source operand (the second operand) is an XMM register or memory location. Bits (MAXVL-1:64) of the corresponding destination register remain unchanged. The destination operand is an XMM register.

VEX.128 and EVEX encoded versions: The "convert-from" source operand (the third operand) can be an XMM register or a 32-bit memory location. The first source and destination operands are XMM registers. Bits (127:64) of the XMM register destination are copied from the corresponding bits in the first source operand. Bits (MAXVL-1:128) of the destination register are zeroed.

Software should ensure VCVTSS2SD is encoded with VEX.L=0. Encoding VCVTSS2SD with VEX.L=1 may encounter unpredictable behavior across different processor generations.

Operation

VCVTSS2SD (EVEX encoded version)

```
IF k1[0] or *no writemask*
    THEN     DEST[63:0] ← Convert_Single_Precision_To_Double_Precision_Floating_Point(SRC2[31:0]);
    ELSE
        IF *merging-masking*                    ; merging-masking
            THEN *DEST[63:0] remains unchanged*
            ELSE                                ; zeroing-masking
                THEN DEST[63:0] = 0
        FI;
FI;
DEST[127:64] ← SRC1[127:64]
DEST[MAXVL-1:128] ← 0
```

VCVTSS2SD (VEX.128 encoded version)
DEST[63:0] ←Convert_Single_Precision_To_Double_Precision_Floating_Point(SRC2[31:0])
DEST[127:64] ←SRC1[127:64]
DEST[MAXVL-1:128] ←0

CVTSS2SD (128-bit Legacy SSE version)
DEST[63:0] ←Convert_Single_Precision_To_Double_Precision_Floating_Point(SRC[31:0]);
DEST[MAXVL-1:64] (Unmodified)

Intel C/C++ Compiler Intrinsic Equivalent

VCVTSS2SD __m128d _mm_cvt_roundss_sd(__m128d a, __m128 b, int r);
VCVTSS2SD __m128d _mm_mask_cvt_roundss_sd(__m128d s, __mmask8 m, __m128d a,__m128 b, int r);
VCVTSS2SD __m128d _mm_maskz_cvt_roundss_sd(__mmask8 k, __m128d a, __m128 a, int r);
VCVTSS2SD __m128d _mm_mask_cvtss_sd(__m128d s, __mmask8 m, __m128d a,__m128 b);
VCVTSS2SD __m128d _mm_maskz_cvtss_sd(__mmask8 m, __m128d a,__m128 b);
CVTSS2SD __m128d_mm_cvtss_sd(__m128d a, __m128 a);

SIMD Floating-Point Exceptions

Invalid, Denormal

Other Exceptions

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

CVTSS2SI—Convert Scalar Single-Precision Floating-Point Value to Doubleword Integer

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
F3 0F 2D /r CVTSS2SI r32, xmm1/m32	A	V/V	SSE	Convert one single-precision floating-point value from xmm1/m32 to one signed doubleword integer in r32.
F3 REX.W 0F 2D /r CVTSS2SI r64, xmm1/m32	A	V/N.E.	SSE	Convert one single-precision floating-point value from xmm1/m32 to one signed quadword integer in r64.
VEX.LIG.F3.0F.W0 2D /r [1] VCVTSS2SI r32, xmm1/m32	A	V/V	AVX	Convert one single-precision floating-point value from xmm1/m32 to one signed doubleword integer in r32.
VEX.LIG.F3.0F.W1 2D /r [1] VCVTSS2SI r64, xmm1/m32	A	V/N.E.[2]	AVX	Convert one single-precision floating-point value from xmm1/m32 to one signed quadword integer in r64.
EVEX.LIG.F3.0F.W0 2D /r VCVTSS2SI r32, xmm1/m32{er}	B	V/V	AVX512F	Convert one single-precision floating-point value from xmm1/m32 to one signed doubleword integer in r32.
EVEX.LIG.F3.0F.W1 2D /r VCVTSS2SI r64, xmm1/m32{er}	B	V/N.E.[2]	AVX512F	Convert one single-precision floating-point value from xmm1/m32 to one signed quadword integer in r64.

NOTES:

1. Software should ensure VCVTSS2SI is encoded with VEX.L=0. Encoding VCVTSS2SI with VEX.L=1 may encounter unpredictable behavior across different processor generations.
2. VEX.W1/EVEX.W1 in non-64 bit is ignored; the instructions behaves as if the W0 version is used.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	NA	ModRM:reg (w)	ModRM:r/m (r)	NA	NA
B	Tuple1 Fixed	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

Description

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

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

Legacy SSE instructions: In 64-bit mode, Use of the REX.W prefix promotes the instruction to produce 64-bit data. See the summary chart at the beginning of this section for encoding data and limits.

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

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

Software should ensure VCVTSS2SI is encoded with VEX.L=0. Encoding VCVTSS2SI with VEX.L=1 may encounter unpredictable behavior across different processor generations.

Operation

VCVTSS2SI (EVEX encoded version)

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

(V)CVTSS2SI (Legacy and VEX.128 encoded version)

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

Intel C/C++ Compiler Intrinsic Equivalent

VCVTSS2SI int _mm_cvtss_i32(__m128 a);
VCVTSS2SI int _mm_cvt_roundss_i32(__m128 a, int r);
VCVTSS2SI __int64 _mm_cvtss_i64(__m128 a);
VCVTSS2SI __int64 _mm_cvt_roundss_i64(__m128 a, int r);

SIMD Floating-Point Exceptions

Invalid, Precision

Other Exceptions

VEX-encoded instructions, see Exceptions Type 3; additionally

#UD	If VEX.vvvv != 1111B.

EVEX-encoded instructions, see Exceptions Type E3NF.

CVTTPD2DQ—Convert with Truncation Packed Double-Precision Floating-Point Values to Packed Doubleword Integers

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F E6 /r CVTTPD2DQ xmm1, xmm2/m128	A	V/V	SSE2	Convert two packed double-precision floating-point values in xmm2/mem to two signed doubleword integers in xmm1 using truncation.
VEX.128.66.0F.WIG E6 /r VCVTTPD2DQ xmm1, xmm2/m128	A	V/V	AVX	Convert two packed double-precision floating-point values in xmm2/mem to two signed doubleword integers in xmm1 using truncation.
VEX.256.66.0F.WIG E6 /r VCVTTPD2DQ xmm1, ymm2/m256	A	V/V	AVX	Convert four packed double-precision floating-point values in ymm2/mem to four signed doubleword integers in xmm1 using truncation.
EVEX.128.66.0F.W1 E6 /r VCVTTPD2DQ xmm1 {k1}{z}, xmm2/m128/m64bcst	B	V/V	AVX512VL AVX512F	Convert two packed double-precision floating-point values in xmm2/m128/m64bcst to two signed doubleword integers in xmm1 using truncation subject to writemask k1.
EVEX.256.66.0F.W1 E6 /r VCVTTPD2DQ xmm1 {k1}{z}, ymm2/m256/m64bcst	B	V/V	AVX512VL AVX512F	Convert four packed double-precision floating-point values in ymm2/m256/m64bcst to four signed doubleword integers in xmm1 using truncation subject to writemask k1.
EVEX.512.66.0F.W1 E6 /r VCVTTPD2DQ ymm1 {k1}{z}, zmm2/m512/m64bcst{sae}	B	V/V	AVX512F	Convert eight packed double-precision floating-point values in zmm2/m512/m64bcst to eight signed doubleword integers in ymm1 using truncation subject to writemask k1.

Instruction Operand Encoding

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

Description

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

When a conversion is inexact, a truncated (round toward zero) value is returned. If a converted result is larger than the maximum signed doubleword integer, the floating-point invalid exception is raised, and if this exception is masked, the indefinite integer value (80000000H) is returned.

EVEX encoded versions: The source operand is a ZMM/YMM/XMM register, a 512/256/128-bit memory location, or a 512/256/128-bit vector broadcasted from a 64-bit memory location. The destination operand is a YMM/XMM/XMM (low 64 bits) register conditionally updated with writemask k1. The upper bits (MAXVL-1:256) of the corresponding destination are zeroed.

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

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

128-bit Legacy SSE version: The source operand is an XMM register or 128- bit memory location. The destination operand is an XMM register. The upper bits (MAXVL-1:128) of the corresponding ZMM register destination are unmodified.

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

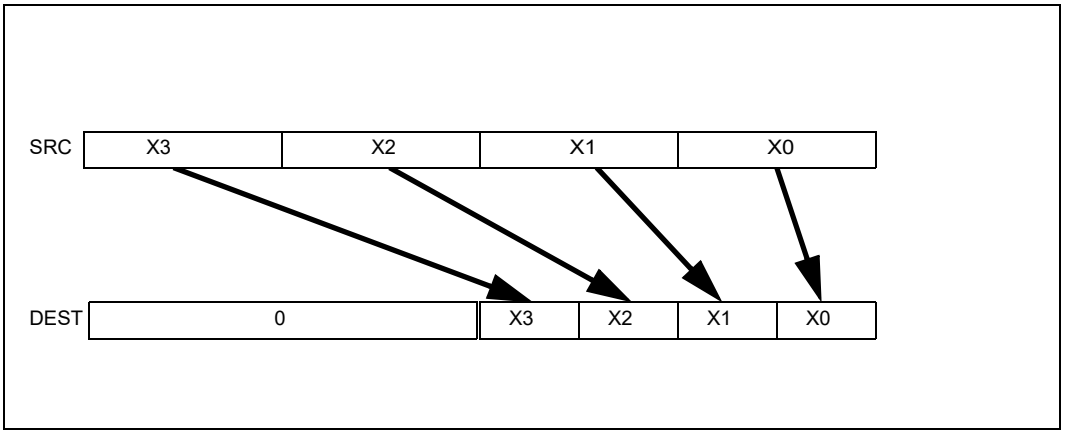


Figure 3-15. VCVTTPD2DQ (VEX.256 encoded version)

Operation

VCVTTPD2DQ (EVEX encoded versions) when src operand is a register
(KL, VL) = (2, 128), (4, 256), (8, 512)

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

VCVTTPD2DQ (EVEX encoded versions) when src operand is a memory source
(KL, VL) = (2, 128), (4, 256), (8, 512)

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

VCVTTPD2DQ (VEX.256 encoded version)
DEST[31:0] ←Convert_Double_Precision_Floating_Point_To_Integer_Truncate(SRC[63:0])
DEST[63:32] ←Convert_Double_Precision_Floating_Point_To_Integer_Truncate(SRC[127:64])
DEST[95:64] ←Convert_Double_Precision_Floating_Point_To_Integer_Truncate(SRC[191:128])
DEST[127:96] ←Convert_Double_Precision_Floating_Point_To_Integer_Truncate(SRC[255:192)
DEST[MAXVL-1:128]←0

VCVTTPD2DQ (VEX.128 encoded version)
DEST[31:0] ←Convert_Double_Precision_Floating_Point_To_Integer_Truncate(SRC[63:0])
DEST[63:32] ←Convert_Double_Precision_Floating_Point_To_Integer_Truncate(SRC[127:64])
DEST[MAXVL-1:64]←0

CVTTPD2DQ (128-bit Legacy SSE version)
DEST[31:0] ←Convert_Double_Precision_Floating_Point_To_Integer_Truncate(SRC[63:0])
DEST[63:32] ←Convert_Double_Precision_Floating_Point_To_Integer_Truncate(SRC[127:64])
DEST[127:64] ←0
DEST[MAXVL-1:128] (unmodified)

Intel C/C++ Compiler Intrinsic Equivalent

VCVTTPD2DQ __m256i _mm512_cvttpd_epi32(__m512d a);
VCVTTPD2DQ __m256i _mm512_mask_cvttpd_epi32(__m256i s, __mmask8 k, __m512d a);
VCVTTPD2DQ __m256i _mm512_maskz_cvttpd_epi32(__mmask8 k, __m512d a);
VCVTTPD2DQ __m256i _mm512_cvtt_roundpd_epi32(__m512d a, int sae);
VCVTTPD2DQ __m256i _mm512_mask_cvtt_roundpd_epi32(__m256i s, __mmask8 k, __m512d a, int sae);
VCVTTPD2DQ __m256i _mm512_maskz_cvtt_roundpd_epi32(__mmask8 k, __m512d a, int sae);
VCVTTPD2DQ __m128i _mm256_mask_cvttpd_epi32(__m128i s, __mmask8 k, __m256d a);
VCVTTPD2DQ __m128i _mm256_maskz_cvttpd_epi32(__mmask8 k, __m256d a);
VCVTTPD2DQ __m128i _mm_mask_cvttpd_epi32(__m128i s, __mmask8 k, __m128d a);
VCVTTPD2DQ __m128i _mm_maskz_cvttpd_epi32(__mmask8 k, __m128d a);
VCVTTPD2DQ __m128i _mm256_cvttpd_epi32 (__m256d src);
CVTTPD2DQ __m128i _mm_cvttpd_epi32 (__m128d src);

SIMD Floating-Point Exceptions

Invalid, Precision

Other Exceptions

VEX-encoded instructions, see Exceptions Type 2;
EVEX-encoded instructions, see Exceptions Type E2.

#UD	If VEX.vvvv != 1111B or EVEX.vvvv != 1111B.

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

Opcode/ Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
66 0F 2C /r CVTTPD2PI *mm, xmm/m128*	RM	Valid	Valid	Convert two packer double-precision floating-point values from *xmm/m128* to two packed signed doubleword integers in *mm* using truncation.

Instruction Operand Encoding

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

Description

Converts two packed double-precision floating-point values in the source operand (second operand) to two packed signed doubleword integers in the destination operand (first operand). The source operand can be an XMM register or a 128-bit memory location. The destination operand is an MMX technology register.

When a conversion is inexact, a truncated (round toward zero) result is returned. If a converted result is larger than the maximum signed doubleword integer, the floating-point invalid exception is raised, and if this exception is masked, the indefinite integer value (80000000H) is returned.

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

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

Operation

```
DEST[31:0] ← Convert_Double_Precision_Floating_Point_To_Integer32_Truncate(SRC[63:0]);
DEST[63:32] ← Convert_Double_Precision_Floating_Point_To_Integer32_
              Truncate(SRC[127:64]);
```

Intel C/C++ Compiler Intrinsic Equivalent

CVTTPD1PI: __m64 _mm_cvttpd_pi32(__m128d a)

SIMD Floating-Point Exceptions

Invalid, Precision

Other Mode Exceptions

See Table 22-4, "Exception Conditions for Legacy SIMD/MMX Instructions with FP Exception and 16-Byte Alignment," in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3B*.

CVTTPS2DQ—Convert with Truncation Packed Single-Precision Floating-Point Values to Packed Signed Doubleword Integer Values

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
F3 0F 5B /r CVTTPS2DQ xmm1, xmm2/m128	A	V/V	SSE2	Convert four packed single-precision floating-point values from xmm2/mem to four packed signed doubleword values in xmm1 using truncation.
VEX.128.F3.0F.WIG 5B /r VCVTTPS2DQ xmm1, xmm2/m128	A	V/V	AVX	Convert four packed single-precision floating-point values from xmm2/mem to four packed signed doubleword values in xmm1 using truncation.
VEX.256.F3.0F.WIG 5B /r VCVTTPS2DQ ymm1, ymm2/m256	A	V/V	AVX	Convert eight packed single-precision floating-point values from ymm2/mem to eight packed signed doubleword values in ymm1 using truncation.
EVEX.128.F3.0F.W0 5B /r VCVTTPS2DQ xmm1 {k1}{z}, xmm2/m128/m32bcst	B	V/V	AVX512VL AVX512F	Convert four packed single precision floating-point values from xmm2/m128/m32bcst to four packed signed doubleword values in xmm1 using truncation subject to writemask k1.
EVEX.256.F3.0F.W0 5B /r VCVTTPS2DQ ymm1 {k1}{z}, ymm2/m256/m32bcst	B	V/V	AVX512VL AVX512F	Convert eight packed single precision floating-point values from ymm2/m256/m32bcst to eight packed signed doubleword values in ymm1 using truncation subject to writemask k1.
EVEX.512.F3.0F.W0 5B /r VCVTTPS2DQ zmm1 {k1}{z}, zmm2/m512/m32bcst {sae}	B	V/V	AVX512F	Convert sixteen packed single-precision floating-point values from zmm2/m512/m32bcst to sixteen packed signed doubleword values in zmm1 using truncation subject to writemask k1.

Instruction Operand Encoding

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

Description

Converts four, eight or sixteen packed single-precision floating-point values in the source operand to four, eight or sixteen signed doubleword integers in the destination operand.

When a conversion is inexact, a truncated (round toward zero) value is returned. If a converted result is larger than the maximum signed doubleword integer, the floating-point invalid exception is raised, and if this exception is masked, the indefinite integer value (80000000H) is returned.

EVEX encoded versions: The source operand is a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 32-bit memory location. The destination operand is a ZMM/YMM/XMM register conditionally updated with writemask k1.

VEX.256 encoded version: The source operand is a YMM register or 256- bit memory location. The destination operand is a YMM register. The upper bits (MAXVL-1:256) of the corresponding ZMM register destination are zeroed.

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

128-bit Legacy SSE version: The source operand is an XMM register or 128- bit memory location. The destination operand is an XMM register. The upper bits (MAXVL-1:128) of the corresponding ZMM register destination are unmodified.

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

Operation

VCVTTPS2DQ (EVEX encoded versions) when src operand is a register
(KL, VL) = (4, 128), (8, 256), (16, 512)

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

VCVTTPS2DQ (EVEX encoded versions) when src operand is a memory source
(KL, VL) = (4, 128), (8, 256), (16, 512)

```
FOR j ← 0 TO 15
    i ← j * 32
    IF k1[j] OR *no writemask*
        THEN
            IF (EVEX.b = 1)
                THEN
                    DEST[i+31:i] ←
            Convert_Single_Precision_Floating_Point_To_Integer_Truncate(SRC[31:0])
                ELSE
                    DEST[i+31:i] ←
            Convert_Single_Precision_Floating_Point_To_Integer_Truncate(SRC[i+31:i])
            FI;
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE                            ; zeroing-masking
                    DEST[i+31:i] ← 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0
```

VCVTTPS2DQ (VEX.256 encoded version)

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

VCVTTPS2DQ (VEX.128 encoded version)

```
DEST[31:0] ←Convert_Single_Precision_Floating_Point_To_Integer_Truncate(SRC[31:0])
DEST[63:32] ←Convert_Single_Precision_Floating_Point_To_Integer_Truncate(SRC[63:32])
DEST[95:64] ←Convert_Single_Precision_Floating_Point_To_Integer_Truncate(SRC[95:64])
DEST[127:96] ←Convert_Single_Precision_Floating_Point_To_Integer_Truncate(SRC[127:96])
DEST[MAXVL-1:128] ←0
```

CVTTPS2DQ (128-bit Legacy SSE version)

```
DEST[31:0] ←Convert_Single_Precision_Floating_Point_To_Integer_Truncate(SRC[31:0])
DEST[63:32] ←Convert_Single_Precision_Floating_Point_To_Integer_Truncate(SRC[63:32])
DEST[95:64] ←Convert_Single_Precision_Floating_Point_To_Integer_Truncate(SRC[95:64])
DEST[127:96] ←Convert_Single_Precision_Floating_Point_To_Integer_Truncate(SRC[127:96])
DEST[MAXVL-1:128] (unmodified)
```

Intel C/C++ Compiler Intrinsic Equivalent

VCVTTPS2DQ __m512i _mm512_cvttps_epi32(__m512 a);
VCVTTPS2DQ __m512i _mm512_mask_cvttps_epi32(__m512i s, __mmask16 k, __m512 a);
VCVTTPS2DQ __m512i _mm512_maskz_cvttps_epi32(__mmask16 k, __m512 a);
VCVTTPS2DQ __m512i _mm512_cvtt_roundps_epi32(__m512 a, int sae);
VCVTTPS2DQ __m512i _mm512_mask_cvtt_roundps_epi32(__m512i s, __mmask16 k, __m512 a, int sae);
VCVTTPS2DQ __m512i _mm512_maskz_cvtt_roundps_epi32(__mmask16 k, __m512 a, int sae);
VCVTTPS2DQ __m256i _mm256_mask_cvttps_epi32(__m256i s, __mmask8 k, __m256 a);
VCVTTPS2DQ __m256i _mm256_maskz_cvttps_epi32(__mmask8 k, __m256 a);
VCVTTPS2DQ __m128i _mm_mask_cvttps_epi32(__m128i s, __mmask8 k, __m128 a);
VCVTTPS2DQ __m128i _mm_maskz_cvttps_epi32(__mmask8 k, __m128 a);
VCVTTPS2DQ __m256i _mm256_cvttps_epi32 (__m256 a)
CVTTPS2DQ __m128i _mm_cvttps_epi32 (__m128 a)

SIMD Floating-Point Exceptions

Invalid, Precision

Other Exceptions

VEX-encoded instructions, see Exceptions Type 2; additionally
EVEX-encoded instructions, see Exceptions Type E2.

#UD	If VEX.vvvv != 1111B or EVEX.vvvv != 1111B.

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

Opcode/ Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
NP 0F 2C /r CVTTPS2PI *mm, xmm/m64*	RM	Valid	Valid	Convert two single-precision floating-point values from *xmm/m64* to two signed doubleword signed integers in *mm* using truncation.

Instruction Operand Encoding

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

Description

Converts two packed single-precision floating-point values in the source operand (second operand) to two packed signed doubleword integers in the destination operand (first operand). The source operand can be an XMM register or a 64-bit memory location. The destination operand is an MMX technology register. When the source operand is an XMM register, the two single-precision floating-point values are contained in the low quadword of the register.

When a conversion is inexact, a truncated (round toward zero) result is returned. If a converted result is larger than the maximum signed doubleword integer, the floating-point invalid exception is raised, and if this exception is masked, the indefinite integer value (80000000H) is returned.

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

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

Operation

```
DEST[31:0] ← Convert_Single_Precision_Floating_Point_To_Integer_Truncate(SRC[31:0]);
DEST[63:32] ← Convert_Single_Precision_Floating_Point_To_Integer_Truncate(SRC[63:32]);
```

Intel C/C++ Compiler Intrinsic Equivalent

CVTTPS2PI: __m64 _mm_cvttps_pi32(__m128 a)

SIMD Floating-Point Exceptions

Invalid, Precision

Other Exceptions

See Table 22-5, "Exception Conditions for Legacy SIMD/MMX Instructions with XMM and FP Exception," in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3B*.

CVTTSD2SI—Convert with Truncation Scalar Double-Precision Floating-Point Value to Signed Integer

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
F2 0F 2C /r CVTTSD2SI r32, xmm1/m64	A	V/V	SSE2	Convert one double-precision floating-point value from xmm1/m64 to one signed doubleword integer in r32 using truncation.
F2 REX.W 0F 2C /r CVTTSD2SI r64, xmm1/m64	A	V/N.E.	SSE2	Convert one double-precision floating-point value from xmm1/m64 to one signed quadword integer in r64 using truncation.
VEX.LIG.F2.0F.W0 2C /r [1] VCVTTSD2SI r32, xmm1/m64	A	V/V	AVX	Convert one double-precision floating-point value from xmm1/m64 to one signed doubleword integer in r32 using truncation.
VEX.LIG.F2.0F.W1 2C /r [1] VCVTTSD2SI r64, xmm1/m64	B	V/N.E.[2]	AVX	Convert one double-precision floating-point value from xmm1/m64 to one signed quadword integer in r64 using truncation.
EVEX.LIG.F2.0F.W0 2C /r VCVTTSD2SI r32, xmm1/m64{sae}	B	V/V	AVX512F	Convert one double-precision floating-point value from xmm1/m64 to one signed doubleword integer in r32 using truncation.
EVEX.LIG.F2.0F.W1 2C /r VCVTTSD2SI r64, xmm1/m64{sae}	B	V/N.E.[2]	AVX512F	Convert one double-precision floating-point value from xmm1/m64 to one signed quadword integer in r64 using truncation.

NOTES:

1. Software should ensure VCVTTSD2SI is encoded with VEX.L=0. Encoding VCVTTSD2SI with VEX.L=1 may encounter unpredictable behavior across different processor generations.
2. For this specific instruction, VEX.W/EVEX.W in non-64 bit is ignored; the instructions behaves as if the W0 version is used.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	NA	ModRM:reg (w)	ModRM:r/m (r)	NA	NA
B	Tuple1 Fixed	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

Description

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

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

If a converted result exceeds the range limits of signed doubleword integer (in non-64-bit modes or 64-bit mode with REX.W/VEX.W/EVEX.W=0), the floating-point invalid exception is raised, and if this exception is masked, the indefinite integer value (80000000H) is returned.

If a converted result exceeds the range limits of signed quadword integer (in 64-bit mode and REX.W/VEX.W/EVEX.W = 1), the floating-point invalid exception is raised, and if this exception is masked, the indefinite integer value (80000000_00000000H) is returned.

Legacy SSE instructions: In 64-bit mode, Use of the REX.W prefix promotes the instruction to 64-bit operation. See the summary chart at the beginning of this section for encoding data and limits.

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

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

Software should ensure VCVTTSD2SI is encoded with VEX.L=0. Encoding VCVTTSD2SI with VEX.L=1 may encounter unpredictable behavior across different processor generations.

Operation

(V)CVTTSD2SI (All versions)

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

Intel C/C++ Compiler Intrinsic Equivalent

VCVTTSD2SI int _mm_cvttsd_i32(__m128d a);
VCVTTSD2SI int _mm_cvtt_roundsd_i32(__m128d a, int sae);
VCVTTSD2SI __int64 _mm_cvttsd_i64(__m128d a);
VCVTTSD2SI __int64 _mm_cvtt_roundsd_i64(__m128d a, int sae);
CVTTSD2SI int _mm_cvttsd_si32(__m128d a);
CVTTSD2SI __int64 _mm_cvttsd_si64(__m128d a);

SIMD Floating-Point Exceptions

Invalid, Precision

Other Exceptions

VEX-encoded instructions, see Exceptions Type 3; additionally

#UD	If VEX.vvvv != 1111B.

EVEX-encoded instructions, see Exceptions Type E3NF.

CVTTSS2SI—Convert with Truncation Scalar Single-Precision Floating-Point Value to Integer

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
F3 0F 2C /r CVTTSS2SI r32, xmm1/m32	A	V/V	SSE	Convert one single-precision floating-point value from xmm1/m32 to one signed doubleword integer in r32 using truncation.
F3 REX.W 0F 2C /r CVTTSS2SI r64, xmm1/m32	A	V/N.E.	SSE	Convert one single-precision floating-point value from xmm1/m32 to one signed quadword integer in r64 using truncation.
VEX.LIG.F3.0F.W0 2C /r [1] VCVTTSS2SI r32, xmm1/m32	A	V/V	AVX	Convert one single-precision floating-point value from xmm1/m32 to one signed doubleword integer in r32 using truncation.
VEX.LIG.F3.0F.W1 2C /r [1] VCVTTSS2SI r64, xmm1/m32	A	V/N.E.[2]	AVX	Convert one single-precision floating-point value from xmm1/m32 to one signed quadword integer in r64 using truncation.
EVEX.LIG.F3.0F.W0 2C /r VCVTTSS2SI r32, xmm1/m32{sae}	B	V/V	AVX512F	Convert one single-precision floating-point value from xmm1/m32 to one signed doubleword integer in r32 using truncation.
EVEX.LIG.F3.0F.W1 2C /r VCVTTSS2SI r64, xmm1/m32{sae}	B	V/N.E.[2]	AVX512F	Convert one single-precision floating-point value from xmm1/m32 to one signed quadword integer in r64 using truncation.

NOTES:

1. Software should ensure VCVTTSS2SI is encoded with VEX.L=0. Encoding VCVTTSS2SI with VEX.L=1 may encounter unpredictable behavior across different processor generations.
2. For this specific instruction, VEX.W/EVEX.W in non-64 bit is ignored; the instructions behaves as if the W0 version is used.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	NA	ModRM:reg (w)	ModRM:r/m (r)	NA	NA
B	Tuple1 Fixed	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

Description

Converts a single-precision floating-point value in the source operand (the second operand) to a signed doubleword integer (or signed quadword integer if operand size is 64 bits) in the destination operand (the first operand). The source operand can be an XMM register or a 32-bit memory location. The destination operand is a general purpose register. When the source operand is an XMM register, the single-precision floating-point value is contained in the low doubleword of the register.

When a conversion is inexact, a truncated (round toward zero) result is returned. If a converted result is larger than the maximum signed doubleword integer, the floating-point invalid exception is raised. If this exception is masked, the indefinite integer value (80000000H or 80000000_00000000H if operand size is 64 bits) is returned.

Legacy SSE instructions: In 64-bit mode, Use of the REX.W prefix promotes the instruction to 64-bit operation. See the summary chart at the beginning of this section for encoding data and limits.

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

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

Software should ensure VCVTTSS2SI is encoded with VEX.L=0. Encoding VCVTTSS2SI with VEX.L=1 may encounter unpredictable behavior across different processor generations.

Operation

(V)CVTTSS2SI (All versions)

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

Intel C/C++ Compiler Intrinsic Equivalent

VCVTTSS2SI int _mm_cvttss_i32(__m128 a);
VCVTTSS2SI int _mm_cvtt_roundss_i32(__m128 a, int sae);
VCVTTSS2SI __int64 _mm_cvttss_i64(__m128 a);
VCVTTSS2SI __int64 _mm_cvtt_roundss_i64(__m128 a, int sae);
CVTTSS2SI int _mm_cvttss_si32(__m128 a);
CVTTSS2SI __int64 _mm_cvttss_si64(__m128 a);

SIMD Floating-Point Exceptions

Invalid, Precision

Other Exceptions

See Exceptions Type 3; additionally

#UD	If VEX.vvvv != 1111B.

EVEX-encoded instructions, see Exceptions Type E3NF.

CWD/CDQ/CQO—Convert Word to Doubleword/Convert Doubleword to Quadword

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
99	CWD	ZO	Valid	Valid	DX:AX ← sign-extend of AX.
99	CDQ	ZO	Valid	Valid	EDX:EAX ← sign-extend of EAX.
REX.W + 99	CQO	ZO	Valid	N.E.	RDX:RAX← sign-extend of RAX.

Instruction Operand Encoding

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

Description

Doubles the size of the operand in register AX, EAX, or RAX (depending on the operand size) by means of sign extension and stores the result in registers DX:AX, EDX:EAX, or RDX:RAX, respectively. The CWD instruction copies the sign (bit 15) of the value in the AX register into every bit position in the DX register. The CDQ instruction copies the sign (bit 31) of the value in the EAX register into every bit position in the EDX register. The CQO instruction (available in 64-bit mode only) copies the sign (bit 63) of the value in the RAX register into every bit position in the RDX register.

The CWD instruction can be used to produce a doubleword dividend from a word before word division. The CDQ instruction can be used to produce a quadword dividend from a doubleword before doubleword division. The CQO instruction can be used to produce a double quadword dividend from a quadword before a quadword division.

The CWD and CDQ mnemonics reference the same opcode. The CWD instruction is intended for use when the operand-size attribute is 16 and the CDQ instruction for when the operand-size attribute is 32. Some assemblers may force the operand size to 16 when CWD is used and to 32 when CDQ is used. Others may treat these mnemonics as synonyms (CWD/CDQ) and use the current setting of the operand-size attribute to determine the size of values to be converted, regardless of the mnemonic used.

In 64-bit mode, use of the REX.W prefix promotes operation to 64 bits. The CQO mnemonics reference the same opcode as CWD/CDQ. See the summary chart at the beginning of this section for encoding data and limits.

Operation

```
IF OperandSize = 16 (* CWD instruction *)
    THEN
        DX ← SignExtend(AX);
    ELSE IF OperandSize = 32 (* CDQ instruction *)
        EDX ← SignExtend(EAX); FI;
    ELSE IF 64-Bit Mode and OperandSize = 64 (* CQO instruction*)
        RDX ← SignExtend(RAX); FI;
FI;
```

Flags Affected

None

Exceptions (All Operating Modes)

#UD	If the LOCK prefix is used.

DAA—Decimal Adjust AL after Addition

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
27	DAA	ZO	Invalid	Valid	Decimal adjust AL after addition.

Instruction Operand Encoding

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

Description

Adjusts the sum of two packed BCD values to create a packed BCD result. The AL register is the implied source and destination operand. The DAA instruction is only useful when it follows an ADD instruction that adds (binary addition) two 2-digit, packed BCD values and stores a byte result in the AL register. The DAA instruction then adjusts the contents of the AL register to contain the correct 2-digit, packed BCD result. If a decimal carry is detected, the CF and AF flags are set accordingly.

This instruction executes as described above in compatibility mode and legacy mode. It is not valid in 64-bit mode.

Operation

```
IF 64-Bit Mode
    THEN
        #UD;
    ELSE
        old_AL ← AL;
        old_CF ← CF;
        CF ← 0;
        IF (((AL AND 0FH) > 9) or AF = 1)
            THEN
                AL ← AL + 6;
                CF ← old_CF or (Carry from AL ← AL + 6);
                AF ← 1;
            ELSE
                AF ← 0;
        FI;
        IF ((old_AL > 99H) or (old_CF = 1))
            THEN
                AL ← AL + 60H;
                CF ← 1;
            ELSE
                CF ← 0;
        FI;
FI;
```

Example

```
ADD     AL, BL    Before: AL=79H BL=35H EFLAGS(OSZAPC)=XXXXXX
                  After: AL=AEH BL=35H EFLAGS(OSZAPC)=110000
DAA               Before: AL=AEH BL=35H EFLAGS(OSZAPC)=110000
                  After: AL=14H BL=35H EFLAGS(OSZAPC)=X00111
DAA               Before: AL=2EH BL=35H EFLAGS(OSZAPC)=110000
                  After: AL=34H BL=35H EFLAGS(OSZAPC)=X00101
```

Flags Affected

The CF and AF flags are set if the adjustment of the value results in a decimal carry in either digit of the result (see the "Operation" section above). The SF, ZF, and PF flags are set according to the result. The OF flag is undefined.

Protected Mode Exceptions

#UD	If the LOCK prefix is used.

Real-Address Mode Exceptions

#UD	If the LOCK prefix is used.

Virtual-8086 Mode Exceptions

#UD	If the LOCK prefix is used.

Compatibility Mode Exceptions

#UD	If the LOCK prefix is used.

64-Bit Mode Exceptions

#UD	If in 64-bit mode.

DAS—Decimal Adjust AL after Subtraction

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
2F	DAS	ZO	Invalid	Valid	Decimal adjust AL after subtraction.

Instruction Operand Encoding

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

Description

Adjusts the result of the subtraction of two packed BCD values to create a packed BCD result. The AL register is the implied source and destination operand. The DAS instruction is only useful when it follows a SUB instruction that subtracts (binary subtraction) one 2-digit, packed BCD value from another and stores a byte result in the AL register. The DAS instruction then adjusts the contents of the AL register to contain the correct 2-digit, packed BCD result. If a decimal borrow is detected, the CF and AF flags are set accordingly.

This instruction executes as described above in compatibility mode and legacy mode. It is not valid in 64-bit mode.

Operation

```
IF 64-Bit Mode
    THEN
        #UD;
    ELSE
        old_AL ← AL;
        old_CF ← CF;
        CF ← 0;
        IF (((AL AND 0FH) > 9) or AF = 1)
            THEN
                AL ← AL - 6;
                CF ← old_CF or (Borrow from AL ← AL – 6);
                AF ← 1;
            ELSE
                AF ← 0;
        FI;
        IF ((old_AL > 99H) or (old_CF = 1))
            THEN
                AL ← AL – 60H;
                CF ← 1;
        FI;
FI;
```

Example

```
SUB     AL, BL    Before: AL = 35H, BL = 47H, EFLAGS(OSZAPC) = XXXXXX
                  After: AL = EEH, BL = 47H, EFLAGS(OSZAPC) = 010111
DAA               Before: AL = EEH, BL = 47H, EFLAGS(OSZAPC) = 010111
                  After: AL = 88H, BL = 47H, EFLAGS(OSZAPC) = X10111
```

Flags Affected

The CF and AF flags are set if the adjustment of the value results in a decimal borrow in either digit of the result (see the "Operation" section above). The SF, ZF, and PF flags are set according to the result. The OF flag is undefined.

Protected Mode Exceptions

#UD	If the LOCK prefix is used.

Real-Address Mode Exceptions

#UD	If the LOCK prefix is used.

Virtual-8086 Mode Exceptions

#UD	If the LOCK prefix is used.

Compatibility Mode Exceptions

#UD	If the LOCK prefix is used.

64-Bit Mode Exceptions

#UD	If in 64-bit mode.

DEC—Decrement by 1

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
FE /1	DEC *r/m8*	M	Valid	Valid	Decrement *r/m8* by 1.
REX + FE /1	DEC *r/m8**	M	Valid	N.E.	Decrement *r/m8* by 1.
FF /1	DEC *r/m16*	M	Valid	Valid	Decrement *r/m16* by 1.
FF /1	DEC *r/m32*	M	Valid	Valid	Decrement *r/m32* by 1.
REX.W + FF /1	DEC *r/m64*	M	Valid	N.E.	Decrement *r/m64* by 1.
48+rw	DEC *r16*	O	N.E.	Valid	Decrement *r16* by 1.
48+rd	DEC *r32*	O	N.E.	Valid	Decrement *r32* by 1.

NOTES:

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

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
M	ModRM:r/m (r, w)	NA	NA	NA
O	opcode + rd (r, w)	NA	NA	NA

Description

Subtracts 1 from the destination operand, while preserving the state of the CF flag. The destination operand can be a register or a memory location. This instruction allows a loop counter to be updated without disturbing the CF flag. (To perform a decrement operation that updates the CF flag, use a SUB instruction with an immediate operand of 1.)

This instruction can be used with a LOCK prefix to allow the instruction to be executed atomically.

In 64-bit mode, DEC r16 and DEC r32 are not encodable (because opcodes 48H through 4FH are REX prefixes). Otherwise, the instruction's 64-bit mode default operation size is 32 bits. Use of the REX.R prefix permits access to additional registers (R8-R15). Use of the REX.W prefix promotes operation to 64 bits.

See the summary chart at the beginning of this section for encoding data and limits.

Operation

DEST ← DEST – 1;

Flags Affected

The CF flag is not affected. The OF, SF, ZF, AF, and PF flags are set according to the result.

Protected Mode Exceptions

#GP(0)	If the destination operand is located in a non-writable segment.
	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
	If the DS, ES, FS, or GS register contains a NULL segment selector.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If the LOCK prefix is used but the destination is not a memory operand.

Real-Address Mode Exceptions

#GP	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS	If a memory operand effective address is outside the SS segment limit.
#UD	If the LOCK prefix is used but the destination is not a memory operand.

Virtual-8086 Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.
#UD	If the LOCK prefix is used but the destination is not a memory operand.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If the LOCK prefix is used but the destination is not a memory operand.

DIV—Unsigned Divide

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
F6 /6	DIV *r/m8*	M	Valid	Valid	Unsigned divide AX by *r/m8*, with result stored in AL ← Quotient, AH ← Remainder.
REX + F6 /6	DIV *r/m8**	M	Valid	N.E.	Unsigned divide AX by *r/m8*, with result stored in AL ← Quotient, AH ← Remainder.
F7 /6	DIV *r/m16*	M	Valid	Valid	Unsigned divide DX:AX by *r/m16*, with result stored in AX ← Quotient, DX ← Remainder.
F7 /6	DIV *r/m32*	M	Valid	Valid	Unsigned divide EDX:EAX by *r/m32*, with result stored in EAX ← Quotient, EDX ← Remainder.
REX.W + F7 /6	DIV *r/m64*	M	Valid	N.E.	Unsigned divide RDX:RAX by *r/m64*, with result stored in RAX ← Quotient, RDX ← Remainder.

NOTES:

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

Instruction Operand Encoding

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

Description

Divides unsigned the value in the AX, DX:AX, EDX:EAX, or RDX:RAX registers (dividend) by the source operand (divisor) and stores the result in the AX (AH:AL), DX:AX, EDX:EAX, or RDX:RAX registers. The source operand can be a general-purpose register or a memory location. The action of this instruction depends on the operand size (dividend/divisor). Division using 64-bit operand is available only in 64-bit mode.

Non-integral results are truncated (chopped) towards 0. The remainder is always less than the divisor in magnitude. Overflow is indicated with the #DE (divide error) exception rather than with the CF flag.

In 64-bit mode, the instruction's default operation size is 32 bits. Use of the REX.R prefix permits access to additional registers (R8-R15). Use of the REX.W prefix promotes operation to 64 bits. In 64-bit mode when REX.W is applied, the instruction divides the unsigned value in RDX:RAX by the source operand and stores the quotient in RAX, the remainder in RDX.

See the summary chart at the beginning of this section for encoding data and limits. See Table 3-15.

Table 3-15. DIV Action

Operand Size	Dividend	Divisor	Quotient	Remainder	Maximum Quotient
Word/byte	AX	r/m8	AL	AH	255
Doubleword/word	DX:AX	r/m16	AX	DX	65,535
Quadword/doubleword	EDX:EAX	r/m32	EAX	EDX	$2^{32} - 1$
Doublequadword/ quadword	RDX:RAX	r/m64	RAX	RDX	$2^{64} - 1$

Operation

```
IF SRC = 0
    THEN #DE; FI; (* Divide Error *)
IF OperandSize = 8 (* Word/Byte Operation *)
    THEN
        temp ← AX / SRC;
        IF temp > FFH
            THEN #DE; (* Divide error *)
            ELSE
                AL ← temp;
                AH ← AX MOD SRC;
        FI;
    ELSE IF OperandSize = 16 (* Doubleword/word operation *)
        THEN
            temp ← DX:AX / SRC;
            IF temp > FFFFH
                THEN #DE; (* Divide error *)
            ELSE
                AX ← temp;
                DX ← DX:AX MOD SRC;
            FI;
        FI;
    ELSE IF Operandsize = 32 (* Quadword/doubleword operation *)
        THEN
            temp ← EDX:EAX / SRC;
            IF temp > FFFFFFFFH
                THEN #DE; (* Divide error *)
            ELSE
                EAX ← temp;
                EDX ← EDX:EAX MOD SRC;
            FI;
        FI;
    ELSE IF 64-Bit Mode and Operandsize = 64 (* Doublequadword/quadword operation *)
        THEN
            temp ← RDX:RAX / SRC;
            IF temp > FFFFFFFFFFFFFFFFH
                THEN #DE; (* Divide error *)
            ELSE
                RAX ← temp;
                RDX ← RDX:RAX MOD SRC;
            FI;
        FI;
FI;
```

Flags Affected

The CF, OF, SF, ZF, AF, and PF flags are undefined.

Protected Mode Exceptions

#DE	If the source operand (divisor) is 0
	If the quotient is too large for the designated register.
#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
	If the DS, ES, FS, or GS register contains a NULL segment selector.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If the LOCK prefix is used.

Real-Address Mode Exceptions

#DE	If the source operand (divisor) is 0.
	If the quotient is too large for the designated register.
#GP	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
	If the DS, ES, FS, or GS register contains a NULL segment selector.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#UD	If the LOCK prefix is used.

Virtual-8086 Mode Exceptions

#DE	If the source operand (divisor) is 0.
	If the quotient is too large for the designated register.
#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.
#UD	If the LOCK prefix is used.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form.
#DE	If the source operand (divisor) is 0
	If the quotient is too large for the designated register.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If the LOCK prefix is used.

DIVPD—Divide Packed Double-Precision Floating-Point Values

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 5E /r DIVPD xmm1, xmm2/m128	A	V/V	SSE2	Divide packed double-precision floating-point values in xmm1 by packed double-precision floating-point values in xmm2/mem.
VEX.NDS.128.66.0F.WIG 5E /r VDIVPD xmm1, xmm2, xmm3/m128	B	V/V	AVX	Divide packed double-precision floating-point values in xmm2 by packed double-precision floating-point values in xmm3/mem.
VEX.NDS.256.66.0F.WIG 5E /r VDIVPD ymm1, ymm2, ymm3/m256	B	V/V	AVX	Divide packed double-precision floating-point values in ymm2 by packed double-precision floating-point values in ymm3/mem.
EVEX.NDS.128.66.0F.W1 5E /r VDIVPD xmm1 {k1}{z}, xmm2, xmm3/m128/m64bcst	C	V/V	AVX512VL AVX512F	Divide packed double-precision floating-point values in xmm2 by packed double-precision floating-point values in xmm3/m128/m64bcst and write results to xmm1 subject to writemask k1.
EVEX.NDS.256.66.0F.W1 5E /r VDIVPD ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst	C	V/V	AVX512VL AVX512F	Divide packed double-precision floating-point values in ymm2 by packed double-precision floating-point values in ymm3/m256/m64bcst and write results to ymm1 subject to writemask k1.
EVEX.NDS.512.66.0F.W1 5E /r VDIVPD zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst{er}	C	V/V	AVX512F	Divide packed double-precision floating-point values in zmm2 by packed double-precision FP values in zmm3/m512/m64bcst and write results to zmm1 subject to writemask k1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	NA	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
B	NA	ModRM:reg (w)	VEX.vvvv	ModRM:r/m (r)	NA
C	Full Vector	ModRM:reg (w)	EVEX.vvvv	ModRM:r/m (r)	NA

Description

Performs a SIMD divide of the double-precision floating-point values in the first source operand by the floating-point values in the second source operand (the third operand). Results are written to the destination operand (the first operand).

EVEX encoded versions: The first source operand (the second operand) is a ZMM/YMM/XMM register. The second source operand can be a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 64-bit memory location. The destination operand is a ZMM/YMM/XMM register conditionally updated with writemask k1.

VEX.256 encoded version: The first source operand (the second operand) is a YMM register. The second source operand can be a YMM register or a 256-bit memory location. The destination operand is a YMM register. The upper bits (MAXVL-1:256) of the corresponding destination are zeroed.

VEX.128 encoded version: The first source operand (the second operand) is a XMM register. The second source operand can be a XMM register or a 128-bit memory location. The destination operand is a XMM register. The upper bits (MAXVL-1:128) of the corresponding destination are zeroed.

128-bit Legacy SSE version: The second source operand (the second operand) can be an XMM register or an 128-bit memory location. The destination is the same as the first source operand. The upper bits (MAXVL-1:128) of the corresponding destination are unmodified.

Operation

VDIVPD (EVEX encoded versions)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
IF (VL = 512) AND (EVEX.b = 1) AND SRC2 *is a register*
    THEN
        SET_RM(EVEX.RC);       ; refer to Table 2-4 in the Intel® Architecture Instruction Set Extensions Programming Reference
    ELSE
        SET_RM(MXCSR.RM);
FI;
FOR j ← 0 TO KL-1
    i ← j * 64
    IF k1[j] OR *no writemask*
        THEN
            IF (EVEX.b = 1) AND (SRC2 *is memory*)
                THEN
                    DEST[i+63:i] ← SRC1[i+63:i] / SRC2[63:0]
                ELSE
                    DEST[i+63:i] ← SRC1[i+63:i] / SRC2[i+63:i]
            FI;
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE                                ; zeroing-masking
                    DEST[i+63:i] ← 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0
```

VDIVPD (VEX.256 encoded version)

```
DEST[63:0] ←SRC1[63:0] / SRC2[63:0]
DEST[127:64] ←SRC1[127:64] / SRC2[127:64]
DEST[191:128] ←SRC1[191:128] / SRC2[191:128]
DEST[255:192] ←SRC1[255:192] / SRC2[255:192]
DEST[MAXVL-1:256] ←0;
```

VDIVPD (VEX.128 encoded version)

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

DIVPD (128-bit Legacy SSE version)

```
DEST[63:0] ←SRC1[63:0] / SRC2[63:0]
DEST[127:64] ←SRC1[127:64] / SRC2[127:64]
DEST[MAXVL-1:128] (Unmodified)
```

Intel C/C++ Compiler Intrinsic Equivalent

VDIVPD __m512d _mm512_div_pd(__m512d a, __m512d b);
VDIVPD __m512d _mm512_mask_div_pd(__m512d s, __mmask8 k, __m512d a, __m512d b);
VDIVPD __m512d _mm512_maskz_div_pd(__mmask8 k, __m512d a, __m512d b);
VDIVPD __m256d _mm256_mask_div_pd(__m256d s, __mmask8 k, __m256d a, __m256d b);
VDIVPD __m256d _mm256_maskz_div_pd(__mmask8 k, __m256d a, __m256d b);
VDIVPD __m128d _mm_mask_div_pd(__m128d s, __mmask8 k, __m128d a, __m128d b);
VDIVPD __m128d _mm_maskz_div_pd(__mmask8 k, __m128d a, __m128d b);
VDIVPD __m512d _mm512_div_round_pd(__m512d a, __m512d b, int);
VDIVPD __m512d _mm512_mask_div_round_pd(__m512d s, __mmask8 k, __m512d a, __m512d b, int);
VDIVPD __m512d _mm512_maskz_div_round_pd(__mmask8 k, __m512d a, __m512d b, int);
VDIVPD __m256d _mm256_div_pd (__m256d a, __m256d b);
DIVPD __m128d _mm_div_pd (__m128d a, __m128d b);

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Divide-by-Zero, Precision, Denormal

Other Exceptions

VEX-encoded instructions, see Exceptions Type 2.

EVEX-encoded instructions, see Exceptions Type E2.

DIVPS—Divide Packed Single-Precision Floating-Point Values

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F 5E /r DIVPS xmm1, xmm2/m128	A	V/V	SSE	Divide packed single-precision floating-point values in xmm1 by packed single-precision floating-point values in xmm2/mem.
VEX.NDS.128.0F.WIG 5E /r VDIVPS xmm1, xmm2, xmm3/m128	B	V/V	AVX	Divide packed single-precision floating-point values in xmm2 by packed single-precision floating-point values in xmm3/mem.
VEX.NDS.256.0F.WIG 5E /r VDIVPS ymm1, ymm2, ymm3/m256	B	V/V	AVX	Divide packed single-precision floating-point values in ymm2 by packed single-precision floating-point values in ymm3/mem.
EVEX.NDS.128.0F.W0 5E /r VDIVPS xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst	C	V/V	AVX512VL AVX512F	Divide packed single-precision floating-point values in xmm2 by packed single-precision floating-point values in xmm3/m128/m32bcst and write results to xmm1 subject to writemask k1.
EVEX.NDS.256.0F.W0 5E /r VDIVPS ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst	C	V/V	AVX512VL AVX512F	Divide packed single-precision floating-point values in ymm2 by packed single-precision floating-point values in ymm3/m256/m32bcst and write results to ymm1 subject to writemask k1.
EVEX.NDS.512.0F.W0 5E /r VDIVPS zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst{er}	C	V/V	AVX512F	Divide packed single-precision floating-point values in zmm2 by packed single-precision floating-point values in zmm3/m512/m32bcst and write results to zmm1 subject to writemask k1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	NA	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
B	NA	ModRM:reg (w)	VEX.vvvv	ModRM:r/m (r)	NA
C	Full Vector	ModRM:reg (w)	EVEX.vvvv	ModRM:r/m (r)	NA

Description

Performs a SIMD divide of the four, eight or sixteen packed single-precision floating-point values in the first source operand (the second operand) by the four, eight or sixteen packed single-precision floating-point values in the second source operand (the third operand). Results are written to the destination operand (the first operand).

EVEX encoded versions: The first source operand (the second operand) is a ZMM/YMM/XMM register. The second source operand can be a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 32-bit memory location. The destination operand is a ZMM/YMM/XMM register conditionally updated with writemask k1.

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

VEX.128 encoded version: The first source operand is a XMM register. The second source operand can be a XMM register or a 128-bit memory location. The destination operand is a XMM register. The upper bits (MAXVL-1:128) of the corresponding ZMM register destination are zeroed.

128-bit Legacy SSE version: The second source can be an XMM register or an 128-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (MAXVL-1:128) of the corresponding ZMM register destination are unmodified.

Operation

VDIVPS (EVEX encoded versions)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
IF (VL = 512) AND (EVEX.b = 1) AND SRC2 *is a register*
    THEN
        SET_RM(EVEX.RC);
    ELSE
        SET_RM(MXCSR.RM);
FI;
FOR j ← 0 TO KL-1
    i ← j * 32
    IF k1[j] OR *no writemask*
        THEN
            IF (EVEX.b = 1) AND (SRC2 *is memory*)
                THEN
                    DEST[i+31:i] ← SRC1[i+31:i] / SRC2[31:0]
                ELSE
                    DEST[i+31:i] ← SRC1[i+31:i] / SRC2[i+31:i]
            FI;
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE                                ; zeroing-masking
                    DEST[i+31:i] ← 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0
```

VDIVPS (VEX.256 encoded version)

```
DEST[31:0] ←SRC1[31:0] / SRC2[31:0]
DEST[63:32] ←SRC1[63:32] / SRC2[63:32]
DEST[95:64] ←SRC1[95:64] / SRC2[95:64]
DEST[127:96] ←SRC1[127:96] / SRC2[127:96]
DEST[159:128] ←SRC1[159:128] / SRC2[159:128]
DEST[191:160]←SRC1[191:160] / SRC2[191:160]
DEST[223:192] ←SRC1[223:192] / SRC2[223:192]
DEST[255:224] ←SRC1[255:224] / SRC2[255:224].
DEST[MAXVL-1:256] ←0;
```

VDIVPS (VEX.128 encoded version)

```
DEST[31:0] ←SRC1[31:0] / SRC2[31:0]
DEST[63:32] ←SRC1[63:32] / SRC2[63:32]
DEST[95:64] ←SRC1[95:64] / SRC2[95:64]
DEST[127:96] ←SRC1[127:96] / SRC2[127:96]
DEST[MAXVL-1:128] ←0
```

DIVPS (128-bit Legacy SSE version)
DEST[31:0] ←SRC1[31:0] / SRC2[31:0]
DEST[63:32] ←SRC1[63:32] / SRC2[63:32]
DEST[95:64] ←SRC1[95:64] / SRC2[95:64]
DEST[127:96] ←SRC1[127:96] / SRC2[127:96]
DEST[MAXVL-1:128] (Unmodified)

Intel C/C++ Compiler Intrinsic Equivalent

VDIVPS __m512 _mm512_div_ps(__m512 a, __m512 b);
VDIVPS __m512 _mm512_mask_div_ps(__m512 s, __mmask16 k, __m512 a, __m512 b);
VDIVPS __m512 _mm512_maskz_div_ps(__mmask16 k, __m512 a, __m512 b);
VDIVPD __m256d _mm256_mask_div_pd(__m256d s, __mmask8 k, __m256d a, __m256d b);
VDIVPD __m256d _mm256_maskz_div_pd(__mmask8 k, __m256d a, __m256d b);
VDIVPD __m128d _mm_mask_div_pd(__m128d s, __mmask8 k, __m128d a, __m128d b);
VDIVPD __m128d _mm_maskz_div_pd(__mmask8 k, __m128d a, __m128d b);
VDIVPS __m512 _mm512_div_round_ps(__m512 a, __m512 b, int);
VDIVPS __m512 _mm512_mask_div_round_ps(__m512 s, __mmask16 k, __m512 a, __m512 b, int);
VDIVPS __m512 _mm512_maskz_div_round_ps(__mmask16 k, __m512 a, __m512 b, int);
VDIVPS __m256 _mm256_div_ps (__m256 a, __m256 b);
DIVPS __m128 _mm_div_ps (__m128 a, __m128 b);

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Divide-by-Zero, Precision, Denormal

Other Exceptions

VEX-encoded instructions, see Exceptions Type 2.

EVEX-encoded instructions, see Exceptions Type E2.

DIVSD—Divide Scalar Double-Precision Floating-Point Value

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
F2 0F 5E /r DIVSD xmm1, xmm2/m64	A	V/V	SSE2	Divide low double-precision floating-point value in xmm1 by low double-precision floating-point value in xmm2/m64.
VEX.NDS.LIG.F2.0F.WIG 5E /r VDIVSD xmm1, xmm2, xmm3/m64	B	V/V	AVX	Divide low double-precision floating-point value in xmm2 by low double-precision floating-point value in xmm3/m64.
EVEX.NDS.LIG.F2.0F.W1 5E /r VDIVSD xmm1 {k1}{z}, xmm2, xmm3/m64{er}	C	V/V	AVX512F	Divide low double-precision floating-point value in xmm2 by low double-precision floating-point value in xmm3/m64.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	NA	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
B	NA	ModRM:reg (w)	VEX.vvvv	ModRM:r/m (r)	NA
C	Tuple1 Scalar	ModRM:reg (w)	EVEX.vvvv	ModRM:r/m (r)	NA

Description

Divides the low double-precision floating-point value in the first source operand by the low double-precision floating-point value in the second source operand, and stores the double-precision floating-point result in the destination operand. The second source operand can be an XMM register or a 64-bit memory location. The first source and destination are XMM registers.

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

VEX.128 encoded version: The first source operand is an xmm register encoded by VEX.vvvv. The quadword at bits 127:64 of the destination operand is copied from the corresponding quadword of the first source operand. Bits (MAXVL-1:128) of the destination register are zeroed.

EVEX.128 encoded version: The first source operand is an xmm register encoded by EVEX.vvvv. The quadword element of the destination operand at bits 127:64 are copied from the first source operand. Bits (MAXVL-1:128) of the destination register are zeroed.

EVEX version: The low quadword element of the destination is updated according to the writemask.

Software should ensure VDIVSD is encoded with VEX.L=0. Encoding VDIVSD with VEX.L=1 may encounter unpredictable behavior across different processor generations.

Operation

VDIVSD (EVEX encoded version)

```
IF (EVEX.b = 1) AND SRC2 *is a register*
    THEN
        SET_RM(EVEX.RC);
    ELSE
        SET_RM(MXCSR.RM);
FI;
IF k1[0] or *no writemask*
    THEN    DEST[63:0] ← SRC1[63:0] / SRC2[63:0]
    ELSE
        IF *merging-masking*                    ; merging-masking
            THEN *DEST[63:0] remains unchanged*
            ELSE                                ; zeroing-masking
                THEN DEST[63:0] ← 0
        FI;
FI;
DEST[127:64] ← SRC1[127:64]
DEST[MAXVL-1:128] ← 0
```

VDIVSD (VEX.128 encoded version)

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

DIVSD (128-bit Legacy SSE version)

```
DEST[63:0] ←DEST[63:0] / SRC[63:0]
DEST[MAXVL-1:64] (Unmodified)
```

Intel C/C++ Compiler Intrinsic Equivalent

VDIVSD __m128d _mm_mask_div_sd(__m128d s, __mmask8 k, __m128d a, __m128d b);
VDIVSD __m128d _mm_maskz_div_sd(__mmask8 k, __m128d a, __m128d b);
VDIVSD __m128d _mm_div_round_sd(__m128d a, __m128d b, int);
VDIVSD __m128d _mm_mask_div_round_sd(__m128d s, __mmask8 k, __m128d a, __m128d b, int);
VDIVSD __m128d _mm_maskz_div_round_sd(__mmask8 k, __m128d a, __m128d b, int);
DIVSD __m128d _mm_div_sd (__m128d a, __m128d b);

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Divide-by-Zero, Precision, Denormal

Other Exceptions

VEX-encoded instructions, see Exceptions Type 3.

EVEX-encoded instructions, see Exceptions Type E3.

DIVSS—Divide Scalar Single-Precision Floating-Point Values

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
F3 0F 5E /r DIVSS xmm1, xmm2/m32	A	V/V	SSE	Divide low single-precision floating-point value in xmm1 by low single-precision floating-point value in xmm2/m32.
VEX.NDS.LIG.F3.0F.WIG 5E /r VDIVSS xmm1, xmm2, xmm3/m32	B	V/V	AVX	Divide low single-precision floating-point value in xmm2 by low single-precision floating-point value in xmm3/m32.
EVEX.NDS.LIG.F3.0F.W0 5E /r VDIVSS xmm1 {k1}{z}, xmm2, xmm3/m32{er}	C	V/V	AVX512F	Divide low single-precision floating-point value in xmm2 by low single-precision floating-point value in xmm3/m32.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	NA	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
B	NA	ModRM:reg (w)	VEX.vvvv	ModRM:r/m (r)	NA
C	Tuple1 Scalar	ModRM:reg (w)	EVEX.vvvv	ModRM:r/m (r)	NA

Description

Divides the low single-precision floating-point value in the first source operand by the low single-precision floating-point value in the second source operand, and stores the single-precision floating-point result in the destination operand. The second source operand can be an XMM register or a 32-bit memory location.

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

VEX.128 encoded version: The first source operand is an xmm register encoded by VEX.vvvv. The three high-order doublewords of the destination operand are copied from the first source operand. Bits (MAXVL-1:128) of the destination register are zeroed.

EVEX.128 encoded version: The first source operand is an xmm register encoded by EVEX.vvvv. The doubleword elements of the destination operand at bits 127:32 are copied from the first source operand. Bits (MAXVL-1:128) of the destination register are zeroed.

EVEX version: The low doubleword element of the destination is updated according to the writemask.

Software should ensure VDIVSS is encoded with VEX.L=0. Encoding VDIVSS with VEX.L=1 may encounter unpredictable behavior across different processor generations.

Operation

VDIVSS (EVEX encoded version)

```
IF (EVEX.b = 1) AND SRC2 *is a register*
    THEN
        SET_RM(EVEX.RC);
    ELSE
        SET_RM(MXCSR.RM);
FI;
IF k1[0] or *no writemask*
    THEN    DEST[31:0] ← SRC1[31:0] / SRC2[31:0]
    ELSE
        IF *merging-masking*                    ; merging-masking
            THEN *DEST[31:0] remains unchanged*
            ELSE                                ; zeroing-masking
                THEN DEST[31:0] ← 0
        FI;
FI;
DEST[127:32] ← SRC1[127:32]
DEST[MAXVL-1:128] ← 0
```

VDIVSS (VEX.128 encoded version)

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

DIVSS (128-bit Legacy SSE version)

```
DEST[31:0] ←DEST[31:0] / SRC[31:0]
DEST[MAXVL-1:32] (Unmodified)
```

Intel C/C++ Compiler Intrinsic Equivalent

VDIVSS __m128 _mm_mask_div_ss(__m128 s, __mmask8 k, __m128 a, __m128 b);
VDIVSS __m128 _mm_maskz_div_ss(__mmask8 k, __m128 a, __m128 b);
VDIVSS __m128 _mm_div_round_ss(__m128 a, __m128 b, int);
VDIVSS __m128 _mm_mask_div_round_ss(__m128 s, __mmask8 k, __m128 a, __m128 b, int);
VDIVSS __m128 _mm_maskz_div_round_ss(__mmask8 k, __m128 a, __m128 b, int);
DIVSS __m128 _mm_div_ss(__m128 a, __m128 b);

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Divide-by-Zero, Precision, Denormal

Other Exceptions

VEX-encoded instructions, see Exceptions Type 3.

EVEX-encoded instructions, see Exceptions Type E3.

DPPD — Dot Product of Packed Double Precision Floating-Point Values

Opcode/ Instruction	Op/ En	64/32-bit Mode	CPUID Feature Flag	Description
66 0F 3A 41 /r ib DPPD *xmm1, xmm2/m128, imm8*	RMI	V/V	SSE4_1	Selectively multiply packed DP floating-point values from *xmm1* with packed DP floating-point values from *xmm2*, add and selectively store the packed DP floating-point values to *xmm1*.
VEX.NDS.128.66.0F3A.WIG 41 /r ib VDPPD xmm1,xmm2, xmm3/m128, imm8	RVMI	V/V	AVX	Selectively multiply packed DP floating-point values from xmm2 with packed DP floating-point values from xmm3, add and selectively store the packed DP floating-point values to xmm1.

Instruction Operand Encoding

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

Description

Conditionally multiplies the packed double-precision floating-point values in the destination operand (first operand) with the packed double-precision floating-point values in the source (second operand) depending on a mask extracted from bits [5:4] of the immediate operand (third operand). If a condition mask bit is zero, the corresponding multiplication is replaced by a value of 0.0 in the manner described by Section 12.8.4 of *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*.

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

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

DPPD follows the NaN forwarding rules stated in the Software Developer's Manual, vol. 1, table 4.7. These rules do not cover horizontal prioritization of NaNs. Horizontal propagation of NaNs to the destination and the positioning of those NaNs in the destination is implementation dependent. NaNs on the input sources or computationally generated NaNs will have at least one NaN propagated to the destination.

128-bit Legacy SSE version: The second source can be an XMM register or an 128-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (MAXVL-1:128) of the corresponding YMM register destination are unmodified.

VEX.128 encoded version: the first source operand is an XMM register or 128-bit memory location. The destination operand is an XMM register. The upper bits (MAXVL-1:128) of the corresponding YMM register destination are zeroed.

If VDPPD is encoded with VEX.L= 1, an attempt to execute the instruction encoded with VEX.L= 1 will cause an #UD exception.

Operation

DP_primitive (SRC1, SRC2)

```
IF (imm8[4] = 1)
    THEN Temp1[63:0] ← DEST[63:0] * SRC[63:0]; // update SIMD exception flags
    ELSE Temp1[63:0] ← +0.0; FI;
IF (imm8[5] = 1)
    THEN Temp1[127:64] ← DEST[127:64] * SRC[127:64]; // update SIMD exception flags
    ELSE Temp1[127:64] ← +0.0; FI;
/* if unmasked exception reported, execute exception handler*/

Temp2[63:0] ← Temp1[63:0] + Temp1[127:64]; // update SIMD exception flags
/* if unmasked exception reported, execute exception handler*/

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

DPPD (128-bit Legacy SSE version)

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

VDPPD (VEX.128 encoded version)

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

Flags Affected

None

Intel C/C++ Compiler Intrinsic Equivalent

DPPD: __m128d _mm_dp_pd (__m128d a, __m128d b, const int mask);

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal

Exceptions are determined separately for each add and multiply operation. Unmasked exceptions will leave the destination untouched.

Other Exceptions

See Exceptions Type 2; additionally

#UD	If VEX.L= 1.

DPPS — Dot Product of Packed Single Precision Floating-Point Values

Opcode/ Instruction	Op/ En	64/32-bit Mode	CPUID Feature Flag	Description
66 0F 3A 40 /r ib DPPS *xmm1, xmm2/m128, imm8*	RMI	V/V	SSE4_1	Selectively multiply packed SP floating-point values from *xmm1* with packed SP floating-point values from *xmm2*, add and selectively store the packed SP floating-point values or zero values to *xmm1*.
VEX.NDS.128.66.0F3A.WIG 40 /r ib VDPPS xmm1,xmm2, xmm3/m128, imm8	RVMI	V/V	AVX	Multiply packed SP floating point values from xmm1 with packed SP floating point values from xmm2/mem selectively add and store to xmm1.
VEX.NDS.256.66.0F3A.WIG 40 /r ib VDPPS ymm1, ymm2, ymm3/m256, imm8	RVMI	V/V	AVX	Multiply packed single-precision floating-point values from ymm2 with packed SP floating point values from ymm3/mem, selectively add pairs of elements and store to ymm1.

Instruction Operand Encoding

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

Description

Conditionally multiplies the packed single precision floating-point values in the destination operand (first operand) with the packed single-precision floats in the source (second operand) depending on a mask extracted from the high 4 bits of the immediate byte (third operand). If a condition mask bit in Imm8[7:4] is zero, the corresponding multiplication is replaced by a value of 0.0 in the manner described by Section 12.8.4 of *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*.

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

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

DPPS follows the NaN forwarding rules stated in the Software Developer's Manual, vol. 1, table 4.7. These rules do not cover horizontal prioritization of NaNs. Horizontal propagation of NaNs to the destination and the positioning of those NaNs in the destination is implementation dependent. NaNs on the input sources or computationally generated NaNs will have at least one NaN propagated to the destination.

128-bit Legacy SSE version: The second source can be an XMM register or an 128-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (MAXVL-1:128) of the corresponding YMM register destination are unmodified.

VEX.128 encoded version: the first source operand is an XMM register or 128-bit memory location. The destination operand is an XMM register. The upper bits (MAXVL-1:128) of the corresponding YMM register destination are zeroed.

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

Operation

DP_primitive (SRC1, SRC2)

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

Temp2[31:0] ← Temp1[31:0] + Temp1[63:32]; // update SIMD exception flags
/* if unmasked exception reported, execute exception handler*/
Temp3[31:0] ← Temp1[95:64] + Temp1[127:96]; // update SIMD exception flags
/* if unmasked exception reported, execute exception handler*/
Temp4[31:0] ← Temp2[31:0] + Temp3[31:0]; // update SIMD exception flags
/* if unmasked exception reported, execute exception handler*/

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

DPPS (128-bit Legacy SSE version)

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

VDPPS (VEX.128 encoded version)

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

VDPPS (VEX.256 encoded version)

```
DEST[127:0]←DP_Primitive(SRC1[127:0], SRC2[127:0]);
DEST[255:128]←DP_Primitive(SRC1[255:128], SRC2[255:128]);
```

Flags Affected

None

Intel C/C++ Compiler Intrinsic Equivalent

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

VDPPS: __m256 _mm256_dp_ps (__m256 a, __m256 b, const int mask);

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal

Exceptions are determined separately for each add and multiply operation, in the order of their execution. Unmasked exceptions will leave the destination operands unchanged.

Other Exceptions

See Exceptions Type 2.

EMMS—Empty MMX Technology State

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
NP 0F 77	EMMS	ZO	Valid	Valid	Set the x87 FPU tag word to empty.

Instruction Operand Encoding

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

Description

Sets the values of all the tags in the x87 FPU tag word to empty (all 1s). This operation marks the x87 FPU data registers (which are aliased to the MMX technology registers) as available for use by x87 FPU floating-point instructions. (See Figure 8-7 in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*, for the format of the x87 FPU tag word.) All other MMX instructions (other than the EMMS instruction) set all the tags in x87 FPU tag word to valid (all 0s).

The EMMS instruction must be used to clear the MMX technology state at the end of all MMX technology procedures or subroutines and before calling other procedures or subroutines that may execute x87 floating-point instructions. If a floating-point instruction loads one of the registers in the x87 FPU data register stack before the x87 FPU tag word has been reset by the EMMS instruction, an x87 floating-point register stack overflow can occur that will result in an x87 floating-point exception or incorrect result.

EMMS operation is the same in non-64-bit modes and 64-bit mode.

Operation

```
x87FPUTagWord ← FFFFH;
```

Intel C/C++ Compiler Intrinsic Equivalent

void _mm_empty()

Flags Affected

None

Protected Mode Exceptions

#UD	If CR0.EM[bit 2] = 1.
#NM	If CR0.TS[bit 3] = 1.
#MF	If there is a pending FPU exception.
#UD	If the LOCK prefix is used.

Real-Address Mode Exceptions

Same exceptions as in protected mode.

Virtual-8086 Mode Exceptions

Same exceptions as in protected mode.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

Same exceptions as in protected mode.

ENTER—Make Stack Frame for Procedure Parameters

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
C8 *iw* 00	ENTER *imm16*, 0	II	Valid	Valid	Create a stack frame for a procedure.
C8 *iw* 01	ENTER *imm16*,1	II	Valid	Valid	Create a stack frame with a nested pointer for a procedure.
C8 *iw* ib	ENTER *imm16, imm8*	II	Valid	Valid	Create a stack frame with nested pointers for a procedure.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
II	iw	imm8	NA	NA

Description

Creates a stack frame (comprising of space for dynamic storage and 1-32 frame pointer storage) for a procedure. The first operand (imm16) specifies the size of the dynamic storage in the stack frame (that is, the number of bytes of dynamically allocated on the stack for the procedure). The second operand (imm8) gives the lexical nesting level (0 to 31) of the procedure. The nesting level (imm8 mod 32) and the OperandSize attribute determine the size in bytes of the storage space for frame pointers.

The nesting level determines the number of frame pointers that are copied into the "display area" of the new stack frame from the preceding frame. The default size of the frame pointer is the StackAddrSize attribute, but can be overridden using the 66H prefix. Thus, the OperandSize attribute determines the size of each frame pointer that will be copied into the stack frame and the data being transferred from SP/ESP/RSP register into the BP/EBP/RBP register.

The ENTER and companion LEAVE instructions are provided to support block structured languages. The ENTER instruction (when used) is typically the first instruction in a procedure and is used to set up a new stack frame for a procedure. The LEAVE instruction is then used at the end of the procedure (just before the RET instruction) to release the stack frame.

If the nesting level is 0, the processor pushes the frame pointer from the BP/EBP/RBP register onto the stack, copies the current stack pointer from the SP/ESP/RSP register into the BP/EBP/RBP register, and loads the SP/ESP/RSP register with the current stack-pointer value minus the value in the size operand. For nesting levels of 1 or greater, the processor pushes additional frame pointers on the stack before adjusting the stack pointer. These additional frame pointers provide the called procedure with access points to other nested frames on the stack. See "Procedure Calls for Block-Structured Languages" in Chapter 6 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*, for more information about the actions of the ENTER instruction.

The ENTER instruction causes a page fault whenever a write using the final value of the stack pointer (within the current stack segment) would do so.

In 64-bit mode, default operation size is 64 bits; 32-bit operation size cannot be encoded. Use of 66H prefix changes frame pointer operand size to 16 bits.

When the 66H prefix is used and causing the OperandSize attribute to be less than the StackAddrSize, software is responsible for the following:

- The companion LEAVE instruction must also use the 66H prefix,
- The value in the RBP/EBP register prior to executing "66H ENTER" must be within the same 16KByte region of the current stack pointer (RSP/ESP), such that the value of RBP/EBP after "66H ENTER" remains a valid address in the stack. This ensures "66H LEAVE" can restore 16-bits of data from the stack.

Operation

```
AllocSize ← imm16;
NestingLevel ← imm8 MOD 32;
IF (OperandSize = 64)
    THEN
        Push(RBP); (* RSP decrements by 8 *)
        FrameTemp ← RSP;
    ELSE IF OperandSize = 32
        THEN
            Push(EBP); (* (E)SP decrements by 4 *)
            FrameTemp ← ESP; FI;
    ELSE (* OperandSize = 16 *)
            Push(BP); (* RSP or (E)SP decrements by 2 *)
            FrameTemp ← SP;
FI;

IF NestingLevel = 0
    THEN GOTO CONTINUE;
FI;

IF (NestingLevel > 1)
    THEN FOR i ← 1 to (NestingLevel - 1)
        DO
            IF (OperandSize = 64)
                THEN
                    RBP ← RBP - 8;
                    Push([RBP]); (* Quadword push *)
                ELSE IF OperandSize = 32
                    THEN
                        IF StackSize = 32
                            EBP ← EBP - 4;
                            Push([EBP]); (* Doubleword push *)
                        ELSE (* StackSize = 16 *)
                            BP ← BP - 4;
                            Push([BP]); (* Doubleword push *)
                        FI;
                    FI;
                ELSE (* OperandSize = 16 *)
                    IF StackSize = 32
                        THEN
                            EBP ← EBP - 2;
                            Push([EBP]); (* Word push *)
                        ELSE (* StackSize = 16 *)
                            BP ← BP - 2;
                            Push([BP]); (* Word push *)
                    FI;
                FI;
    OD;
FI;

IF (OperandSize = 64) (* nestinglevel 1 *)
    THEN
        Push(FrameTemp); (* Quadword push and RSP decrements by 8 *)
    ELSE IF OperandSize = 32
```

```
        THEN
            Push(FrameTemp); FI; (* Doubleword push and (E)SP decrements by 4 *)
    ELSE (* OperandSize = 16 *)
            Push(FrameTemp); (* Word push and RSP|ESP|SP decrements by 2 *)
FI;

CONTINUE:
IF 64-Bit Mode (StackSize = 64)
    THEN
            RBP ← FrameTemp;
            RSP ← RSP – AllocSize;
    ELSE IF OperandSize = 32
        THEN
            EBP ← FrameTemp;
            ESP ← ESP – AllocSize; FI;
    ELSE (* OperandSize = 16 *)
            BP ← FrameTemp[15:1]; (* Bits 16 and above of applicable RBP/EBP are unmodified *)
            SP ← SP – AllocSize;
FI;

END;
```

Flags Affected

None.

Protected Mode Exceptions

#SS(0)	If the new value of the SP or ESP register is outside the stack segment limit.
#PF(fault-code)	If a page fault occurs or if a write using the final value of the stack pointer (within the current stack segment) would cause a page fault.
#UD	If the LOCK prefix is used.

Real-Address Mode Exceptions

#SS	If the new value of the SP or ESP register is outside the stack segment limit.
#UD	If the LOCK prefix is used.

Virtual-8086 Mode Exceptions

#SS(0)	If the new value of the SP or ESP register is outside the stack segment limit.
#PF(fault-code)	If a page fault occurs or if a write using the final value of the stack pointer (within the current stack segment) would cause a page fault.
#UD	If the LOCK prefix is used.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#SS(0)	If the stack address is in a non-canonical form.
#PF(fault-code)	If a page fault occurs or if a write using the final value of the stack pointer (within the current stack segment) would cause a page fault.
#UD	If the LOCK prefix is used.

EXTRACTPS—Extract Packed Floating-Point Values

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 3A 17 /r ib EXTRACTPS reg/m32, xmm1, imm8	A	V/V	SSE4_1	Extract one single-precision floating-point value from xmm1 at the offset specified by imm8 and store the result in reg or m32. Zero extend the results in 64-bit register if applicable.
VEX.128.66.0F3A.WIG 17 /r ib VEXTRACTPS reg/m32, xmm1, imm8	A	V/V	AVX	Extract one single-precision floating-point value from xmm1 at the offset specified by imm8 and store the result in reg or m32. Zero extend the results in 64-bit register if applicable.
EVEX.128.66.0F3A.WIG 17 /r ib VEXTRACTPS reg/m32, xmm1, imm8	B	V/V	AVX512F	Extract one single-precision floating-point value from xmm1 at the offset specified by imm8 and store the result in reg or m32. Zero extend the results in 64-bit register if applicable.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	NA	ModRM:r/m (w)	ModRM:reg (r)	Imm8	NA
B	Tuple1 Scalar	ModRM:r/m (w)	ModRM:reg (r)	Imm8	NA

Description

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

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

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

VEX.128 and EVEX encoded version: When VEX.W1 or EVEX.W1 form is used in 64-bit mode with a general purpose register (GPR) as a destination operand, the packed single quantity is zero extended to 64 bits.

VEX.vvvv/EVEX.vvvv is reserved and must be 1111b otherwise instructions will #UD.

128-bit Legacy SSE version: When a REX.W prefix is used in 64-bit mode with a general purpose register (GPR) as a destination operand, the packed single quantity is zero extended to 64 bits.

The source register is an XMM register. Imm8[1:0] determine the starting DWORD offset from which to extract the 32-bit floating-point value.

If VEXTRACTPS is encoded with VEX.L= 1, an attempt to execute the instruction encoded with VEX.L= 1 will cause an #UD exception.

Operation

VEXTRACTPS (EVEX and VEX.128 encoded version)

```
SRC_OFFSET ← IMM8[1:0]
IF (64-Bit Mode and DEST is register)
    DEST[31:0] ← (SRC[127:0] >> (SRC_OFFSET*32)) AND 0FFFFFFFFh
    DEST[63:32] ← 0
ELSE
    DEST[31:0] ← (SRC[127:0] >> (SRC_OFFSET*32)) AND 0FFFFFFFFh
FI
```

EXTRACTPS (128-bit Legacy SSE version)

```
SRC_OFFSET ←IMM8[1:0]
IF (64-Bit Mode and DEST is register)
    DEST[31:0] ←(SRC[127:0] >> (SRC_OFFSET*32)) AND 0FFFFFFFFh
    DEST[63:32] ←0
ELSE
    DEST[31:0] ←(SRC[127:0] >> (SRC_OFFSET*32)) AND 0FFFFFFFFh
FI
```

Intel C/C++ Compiler Intrinsic Equivalent

EXTRACTPS int _mm_extract_ps (__m128 a, const int nidx);

SIMD Floating-Point Exceptions

None

Other Exceptions

VEX-encoded instructions, see Exceptions Type 5; Additionally

EVEX-encoded instructions, see Exceptions Type E9NF.

#UD	IF VEX.L = 0.
#UD	If VEX.vvvv != 1111B or EVEX.vvvv != 1111B.

F2XM1—Compute 2^x-1

Opcode	Instruction	64-Bit Mode	Compat/ Leg Mode	Description
D9 F0	F2XM1	Valid	Valid	Replace ST(0) with $(2^{ST(0)} - 1)$.

Description

Computes the exponential value of 2 to the power of the source operand minus 1. The source operand is located in register ST(0) and the result is also stored in ST(0). The value of the source operand must lie in the range –1.0 to +1.0. If the source value is outside this range, the result is undefined.

The following table shows the results obtained when computing the exponential value of various classes of numbers, assuming that neither overflow nor underflow occurs.

Table 3-16. Results Obtained from F2XM1

ST(0) SRC	ST(0) DEST
– 1.0 to –0	– 0.5 to – 0
– 0	– 0
+ 0	+ 0
+ 0 to +1.0	+ 0 to 1.0

Values other than 2 can be exponentiated using the following formula:

$$x^y \leftarrow 2^{(y * \log_2 x)}$$

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

Operation

$ST(0) \leftarrow (2^{ST(0)} - 1)$;

FPU Flags Affected

C1	Set to 0 if stack underflow occurred.
	Set if result was rounded up; cleared otherwise.
C0, C2, C3	Undefined.

Floating-Point Exceptions

#IS	Stack underflow occurred.
#IA	Source operand is an SNaN value or unsupported format.
#D	Source is a denormal value.
#U	Result is too small for destination format.
#P	Value cannot be represented exactly in destination format.

Protected Mode Exceptions

#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#UD	If the LOCK prefix is used.

Real-Address Mode Exceptions

Same exceptions as in protected mode.

Virtual-8086 Mode Exceptions

Same exceptions as in protected mode.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

Same exceptions as in protected mode.

FABS—Absolute Value

Opcode	Instruction	64-Bit Mode	Compat/ Leg Mode	Description
D9 E1	FABS	Valid	Valid	Replace ST with its absolute value.

Description

Clears the sign bit of ST(0) to create the absolute value of the operand. The following table shows the results obtained when creating the absolute value of various classes of numbers.

Table 3-17. Results Obtained from FABS

ST(0) SRC	ST(0) DEST
−∞	+∞
−F	+F
−0	+0
+0	+0
+F	+F
+∞	+∞
NaN	NaN

NOTES:

F Means finite floating-point value.

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

Operation

ST(0) ← |ST(0)|;

FPU Flags Affected

C1	Set to 0.
C0, C2, C3	Undefined.

Floating-Point Exceptions

#IS	Stack underflow occurred.

Protected Mode Exceptions

#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#UD	If the LOCK prefix is used.

Real-Address Mode Exceptions

Same exceptions as in protected mode.

Virtual-8086 Mode Exceptions

Same exceptions as in protected mode.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

Same exceptions as in protected mode.

FADD/FADDP/FIADD—Add

Opcode	Instruction	64-Bit Mode	Compat/ Leg Mode	Description
D8 /0	FADD *m32fp*	Valid	Valid	Add *m32fp* to ST(0) and store result in ST(0).
DC /0	FADD *m64fp*	Valid	Valid	Add *m64fp* to ST(0) and store result in ST(0).
D8 C0+i	FADD ST(0), ST(i)	Valid	Valid	Add ST(0) to ST(i) and store result in ST(0).
DC C0+i	FADD ST(i), ST(0)	Valid	Valid	Add ST(i) to ST(0) and store result in ST(i).
DE C0+i	FADDP ST(i), ST(0)	Valid	Valid	Add ST(0) to ST(i), store result in ST(i), and pop the register stack.
DE C1	FADDP	Valid	Valid	Add ST(0) to ST(1), store result in ST(1), and pop the register stack.
DA /0	FIADD *m32int*	Valid	Valid	Add *m32int* to ST(0) and store result in ST(0).
DE /0	FIADD *m16int*	Valid	Valid	Add *m16int* to ST(0) and store result in ST(0).

Description

Adds the destination and source operands and stores the sum in the destination location. The destination operand is always an FPU register; the source operand can be a register or a memory location. Source operands in memory can be in single-precision or double-precision floating-point format or in word or doubleword integer format.

The no-operand version of the instruction adds the contents of the ST(0) register to the ST(1) register. The one-operand version adds the contents of a memory location (either a floating-point or an integer value) to the contents of the ST(0) register. The two-operand version, adds the contents of the ST(0) register to the ST(i) register or vice versa. The value in ST(0) can be doubled by coding:

```
FADD ST(0), ST(0);
```

The FADDP instructions perform the additional operation of popping the FPU register stack after storing the result. To pop the register stack, the processor marks the ST(0) register as empty and increments the stack pointer (TOP) by 1. (The no-operand version of the floating-point add instructions always results in the register stack being popped. In some assemblers, the mnemonic for this instruction is FADD rather than FADDP.)

The FIADD instructions convert an integer source operand to double extended-precision floating-point format before performing the addition.

The table on the following page shows the results obtained when adding various classes of numbers, assuming that neither overflow nor underflow occurs.

When the sum of two operands with opposite signs is 0, the result is +0, except for the round toward $-\infty$ mode, in which case the result is –0. When the source operand is an integer 0, it is treated as a +0.

When both operand are infinities of the same sign, the result is ∞ of the expected sign. If both operands are infinities of opposite signs, an invalid-operation exception is generated. See Table 3-18.

Table 3-18. FADD/FADDP/FIADD Results

		DEST						
		– ∞	– F	– 0	+ 0	+ F	+ ∞	NaN
SRC	– ∞	– ∞	– ∞	– ∞	– ∞	– ∞	*	NaN
	– F or – I	– ∞	– F	SRC	SRC	± F or ± 0	+ ∞	NaN
	–0	– ∞	DEST	– 0	± 0	DEST	+ ∞	NaN
	+ 0	– ∞	DEST	± 0	+ 0	DEST	+ ∞	NaN
	+ F or + I	– ∞	± F or ± 0	SRC	SRC	+ F	+ ∞	NaN
	+ ∞	*	+ ∞	+ ∞	+ ∞	+ ∞	+ ∞	NaN
	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN

NOTES:

F Means finite floating-point value.

I Means integer.

* Indicates floating-point invalid-arithmetic-operand (#IA) exception.

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

Operation

```
IF Instruction = FIADD
    THEN
        DEST ← DEST + ConvertToDoubleExtendedPrecisionFP(SRC);
    ELSE (* Source operand is floating-point value *)
        DEST ← DEST + SRC;
FI;

IF Instruction = FADDP
    THEN
        PopRegisterStack;
FI;
```

FPU Flags Affected

C1	Set to 0 if stack underflow occurred. Set if result was rounded up; cleared otherwise.
C0, C2, C3	Undefined.

Floating-Point Exceptions

#IS	Stack underflow occurred.
#IA	Operand is an SNaN value or unsupported format. Operands are infinities of unlike sign.
#D	Source operand is a denormal value.
#U	Result is too small for destination format.
#O	Result is too large for destination format.
#P	Value cannot be represented exactly in destination format.

Protected Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit. If the DS, ES, FS, or GS register contains a NULL segment selector.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If the LOCK prefix is used.

Real-Address Mode Exceptions

#GP	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS	If a memory operand effective address is outside the SS segment limit.
#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#UD	If the LOCK prefix is used.

Virtual-8086 Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.
#UD	If the LOCK prefix is used.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form.
#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#MF	If there is a pending x87 FPU exception.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If the LOCK prefix is used.

FBLD—Load Binary Coded Decimal

Opcode	Instruction	64-Bit Mode	Compat/ Leg Mode	Description
DF /4	FBLD *m80dec*	Valid	Valid	Convert BCD value to floating-point and push onto the FPU stack.

Description

Converts the BCD source operand into double extended-precision floating-point format and pushes the value onto the FPU stack. The source operand is loaded without rounding errors. The sign of the source operand is preserved, including that of –0.

The packed BCD digits are assumed to be in the range 0 through 9; the instruction does not check for invalid digits (AH through FH). Attempting to load an invalid encoding produces an undefined result.

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

Operation

```
TOP ← TOP – 1;
ST(0) ← ConvertToDoubleExtendedPrecisionFP(SRC);
```

FPU Flags Affected

C1	Set to 1 if stack overflow occurred; otherwise, set to 0.
C0, C2, C3	Undefined.

Floating-Point Exceptions

#IS	Stack overflow occurred.

Protected Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
	If the DS, ES, FS, or GS register contains a NULL segment selector.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If the LOCK prefix is used.

Real-Address Mode Exceptions

#GP	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS	If a memory operand effective address is outside the SS segment limit.
#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#UD	If the LOCK prefix is used.

Virtual-8086 Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.
#UD	If the LOCK prefix is used.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form.
#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#MF	If there is a pending x87 FPU exception.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If the LOCK prefix is used.

FBSTP—Store BCD Integer and Pop

Opcode	Instruction	64-Bit Mode	Compat/ Leg Mode	Description
DF /6	FBSTP m80bcd	Valid	Valid	Store ST(0) in m80bcd and pop ST(0).

Description

Converts the value in the ST(0) register to an 18-digit packed BCD integer, stores the result in the destination operand, and pops the register stack. If the source value is a non-integral value, it is rounded to an integer value, according to rounding mode specified by the RC field of the FPU control word. To pop the register stack, the processor marks the ST(0) register as empty and increments the stack pointer (TOP) by 1.

The destination operand specifies the address where the first byte destination value is to be stored. The BCD value (including its sign bit) requires 10 bytes of space in memory.

The following table shows the results obtained when storing various classes of numbers in packed BCD format.

Table 3-19. FBSTP Results

ST(0)	DEST
– ∞ or Value Too Large for DEST Format	*
F ≤ – 1	– D
–1 < F < -0	**
– 0	– 0
+ 0	+ 0
+ 0 < F < +1	**
F ≥ +1	+ D
+ ∞ or Value Too Large for DEST Format	*
NaN	*

NOTES:

F Means finite floating-point value.

D Means packed-BCD number.

* Indicates floating-point invalid-operation (#IA) exception.

** ±0 or ±1, depending on the rounding mode.

If the converted value is too large for the destination format, or if the source operand is an ∞, SNaN, QNAN, or is in an unsupported format, an invalid-arithmetic-operand condition is signaled. If the invalid-operation exception is not masked, an invalid-arithmetic-operand exception (#IA) is generated and no value is stored in the destination operand. If the invalid-operation exception is masked, the packed BCD indefinite value is stored in memory.

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

Operation

```
DEST ← BCD(ST(0));
PopRegisterStack;
```

FPU Flags Affected

C1	Set to 0 if stack underflow occurred.
	Set if result was rounded up; cleared otherwise.
C0, C2, C3	Undefined.

Floating-Point Exceptions

#IS	Stack underflow occurred.
#IA	Converted value that exceeds 18 BCD digits in length.
	Source operand is an SNaN, QNaN, ±∞, or in an unsupported format.
#P	Value cannot be represented exactly in destination format.

Protected Mode Exceptions

#GP(0)	If a segment register is being loaded with a segment selector that points to a non-writable segment.
	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
	If the DS, ES, FS, or GS register contains a NULL segment selector.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If the LOCK prefix is used.

Real-Address Mode Exceptions

#GP	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS	If a memory operand effective address is outside the SS segment limit.
#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#UD	If the LOCK prefix is used.

Virtual-8086 Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.
#UD	If the LOCK prefix is used.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form.
#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#MF	If there is a pending x87 FPU exception.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If the LOCK prefix is used.

FCHS—Change Sign

Opcode	Instruction	64-Bit Mode	Compat/ Leg Mode	Description
D9 E0	FCHS	Valid	Valid	Complements sign of ST(0).

Description

Complements the sign bit of ST(0). This operation changes a positive value into a negative value of equal magnitude or vice versa. The following table shows the results obtained when changing the sign of various classes of numbers.

Table 3-20. FCHS Results

ST(0) SRC	ST(0) DEST
– ∞	+ ∞
– F	+ F
– 0	+ 0
+ 0	– 0
+ F	– F
+ ∞	– ∞
NaN	NaN

NOTES:

* F means finite floating-point value.

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

Operation

```
SignBit(ST(0)) ← NOT (SignBit(ST(0)));
```

FPU Flags Affected

C1	Set to 0.
C0, C2, C3	Undefined.

Floating-Point Exceptions

#IS	Stack underflow occurred.

Protected Mode Exceptions

#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#UD	If the LOCK prefix is used.

Real-Address Mode Exceptions

Same exceptions as in protected mode.

Virtual-8086 Mode Exceptions

Same exceptions as in protected mode.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

Same exceptions as in protected mode.

FCLEX/FNCLEX—Clear Exceptions

Opcode*	Instruction	64-Bit Mode	Compat/ Leg Mode	Description
9B DB E2	FCLEX	Valid	Valid	Clear floating-point exception flags after checking for pending unmasked floating-point exceptions.
DB E2	FNCLEX*	Valid	Valid	Clear floating-point exception flags without checking for pending unmasked floating-point exceptions.

NOTES:

* See IA-32 Architecture Compatibility section below.

Description

Clears the floating-point exception flags (PE, UE, OE, ZE, DE, and IE), the exception summary status flag (ES), the stack fault flag (SF), and the busy flag (B) in the FPU status word. The FCLEX instruction checks for and handles any pending unmasked floating-point exceptions before clearing the exception flags; the FNCLEX instruction does not.

The assembler issues two instructions for the FCLEX instruction (an FWAIT instruction followed by an FNCLEX instruction), and the processor executes each of these instructions separately. If an exception is generated for either of these instructions, the save EIP points to the instruction that caused the exception.

IA-32 Architecture Compatibility

When operating a Pentium or Intel486 processor in MS-DOS* compatibility mode, it is possible (under unusual circumstances) for an FNCLEX instruction to be interrupted prior to being executed to handle a pending FPU exception. See the section titled "No-Wait FPU Instructions Can Get FPU Interrupt in Window" in Appendix D of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*, for a description of these circumstances. An FNCLEX instruction cannot be interrupted in this way on later Intel processors, except for the Intel Quark™ X1000 processor.

This instruction affects only the x87 FPU floating-point exception flags. It does not affect the SIMD floating-point exception flags in the MXCRS register.

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

Operation

```
FPUStatusWord[0:7] ← 0;
FPUStatusWord[15] ← 0;
```

FPU Flags Affected

The PE, UE, OE, ZE, DE, IE, ES, SF, and B flags in the FPU status word are cleared. The C0, C1, C2, and C3 flags are undefined.

Floating-Point Exceptions

None

Protected Mode Exceptions

#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#UD	If the LOCK prefix is used.

Real-Address Mode Exceptions

Same exceptions as in protected mode.

Virtual-8086 Mode Exceptions

Same exceptions as in protected mode.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

Same exceptions as in protected mode.

FCMOV*cc*—Floating-Point Conditional Move

Opcode*	Instruction	64-Bit Mode	Compat/ Leg Mode*	Description
DA C0+i	FCMOVB ST(0), ST(i)	Valid	Valid	Move if below (CF=1).
DA C8+i	FCMOVE ST(0), ST(i)	Valid	Valid	Move if equal (ZF=1).
DA D0+i	FCMOVBE ST(0), ST(i)	Valid	Valid	Move if below or equal (CF=1 or ZF=1).
DA D8+i	FCMOVU ST(0), ST(i)	Valid	Valid	Move if unordered (PF=1).
DB C0+i	FCMOVNB ST(0), ST(i)	Valid	Valid	Move if not below (CF=0).
DB C8+i	FCMOVNE ST(0), ST(i)	Valid	Valid	Move if not equal (ZF=0).
DB D0+i	FCMOVNBE ST(0), ST(i)	Valid	Valid	Move if not below or equal (CF=0 and ZF=0).
DB D8+i	FCMOVNU ST(0), ST(i)	Valid	Valid	Move if not unordered (PF=0).

NOTES:

* See IA-32 Architecture Compatibility section below.

Description

Tests the status flags in the EFLAGS register and moves the source operand (second operand) to the destination operand (first operand) if the given test condition is true. The condition for each mnemonic os given in the Description column above and in Chapter 8 in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*. The source operand is always in the ST(i) register and the destination operand is always ST(0).

The FCMOV*cc* instructions are useful for optimizing small IF constructions. They also help eliminate branching overhead for IF operations and the possibility of branch mispredictions by the processor.

A processor may not support the FCMOV*cc* instructions. Software can check if the FCMOV*cc* instructions are supported by checking the processor's feature information with the CPUID instruction (see "COMISS—Compare Scalar Ordered Single-Precision Floating-Point Values and Set EFLAGS" in this chapter). If both the CMOV and FPU feature bits are set, the FCMOV*cc* instructions are supported.

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

IA-32 Architecture Compatibility

The FCMOVcc instructions were introduced to the IA-32 Architecture in the P6 family processors and are not available in earlier IA-32 processors.

Operation

```
IF condition TRUE
    THEN ST(0) ← ST(i);
FI;
```

FPU Flags Affected

C1	Set to 0 if stack underflow occurred.
C0, C2, C3	Undefined.

Floating-Point Exceptions

#IS	Stack underflow occurred.

Integer Flags Affected

None.

Protected Mode Exceptions

#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#UD	If the LOCK prefix is used.

Real-Address Mode Exceptions

Same exceptions as in protected mode.

Virtual-8086 Mode Exceptions

Same exceptions as in protected mode.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

Same exceptions as in protected mode.

FCOM/FCOMP/FCOMPP—Compare Floating Point Values

Opcode	Instruction	64-Bit Mode	Compat/ Leg Mode	Description
D8 /2	FCOM *m32fp*	Valid	Valid	Compare ST(0) with *m32fp*.
DC /2	FCOM *m64fp*	Valid	Valid	Compare ST(0) with *m64fp*.
D8 D0+i	FCOM ST(i)	Valid	Valid	Compare ST(0) with ST(i).
D8 D1	FCOM	Valid	Valid	Compare ST(0) with ST(1).
D8 /3	FCOMP *m32fp*	Valid	Valid	Compare ST(0) with *m32fp* and pop register stack.
DC /3	FCOMP *m64fp*	Valid	Valid	Compare ST(0) with m64fp and pop register stack.
D8 D8+i	FCOMP ST(i)	Valid	Valid	Compare ST(0) with ST(i) and pop register stack.
D8 D9	FCOMP	Valid	Valid	Compare ST(0) with ST(1) and pop register stack.
DE D9	FCOMPP	Valid	Valid	Compare ST(0) with ST(1) and pop register stack twice.

Description

Compares the contents of register ST(0) and source value and sets condition code flags C0, C2, and C3 in the FPU status word according to the results (see the table below). The source operand can be a data register or a memory location. If no source operand is given, the value in ST(0) is compared with the value in ST(1). The sign of zero is ignored, so that −0.0 is equal to +0.0.

Table 3-21. FCOM/FCOMP/FCOMPP Results

Condition	C3	C2	C0
ST(0) > SRC	0	0	0
ST(0) < SRC	0	0	1
ST(0) = SRC	1	0	0
Unordered*	1	1	1

NOTES:

* Flags not set if unmasked invalid-arithmetic-operand (#IA) exception is generated.

This instruction checks the class of the numbers being compared (see "FXAM—Examine Floating-Point" in this chapter). If either operand is a NaN or is in an unsupported format, an invalid-arithmetic-operand exception (#IA) is raised and, if the exception is masked, the condition flags are set to "unordered." If the invalid-arithmetic-operand exception is unmasked, the condition code flags are not set.

The FCOMP instruction pops the register stack following the comparison operation and the FCOMPP instruction pops the register stack twice following the comparison operation. To pop the register stack, the processor marks the ST(0) register as empty and increments the stack pointer (TOP) by 1.

The FCOM instructions perform the same operation as the FUCOM instructions. The only difference is how they handle QNaN operands. The FCOM instructions raise an invalid-arithmetic-operand exception (#IA) when either or both of the operands is a NaN value or is in an unsupported format. The FUCOM instructions perform the same operation as the FCOM instructions, except that they do not generate an invalid-arithmetic-operand exception for QNaNs.

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

Operation

```
CASE (relation of operands) OF
    ST > SRC:       C3, C2, C0 ← 000;
    ST < SRC:       C3, C2, C0 ← 001;
    ST = SRC:       C3, C2, C0 ← 100;
ESAC;

IF ST(0) or SRC = NaN or unsupported format
    THEN
        #IA
        IF FPUControlWord.IM = 1
            THEN
                C3, C2, C0 ← 111;
        FI;
FI;

IF Instruction = FCOMP
    THEN
        PopRegisterStack;
FI;

IF Instruction = FCOMPP
    THEN
        PopRegisterStack;
        PopRegisterStack;
FI;
```

FPU Flags Affected

C1	Set to 0.
C0, C2, C3	See table on previous page.

Floating-Point Exceptions

#IS	Stack underflow occurred.
#IA	One or both operands are NaN values or have unsupported formats.
	Register is marked empty.
#D	One or both operands are denormal values.

Protected Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
	If the DS, ES, FS, or GS register contains a NULL segment selector.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If the LOCK prefix is used.

Real-Address Mode Exceptions

#GP	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS	If a memory operand effective address is outside the SS segment limit.
#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#UD	If the LOCK prefix is used.

Virtual-8086 Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.
#UD	If the LOCK prefix is used.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form.
#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#MF	If there is a pending x87 FPU exception.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If the LOCK prefix is used.

FCOMI/FCOMIP/FUCOMI/FUCOMIP—Compare Floating Point Values and Set EFLAGS

Opcode	Instruction	64-Bit Mode	Compat/ Leg Mode	Description
DB F0+i	FCOMI ST, ST(i)	Valid	Valid	Compare ST(0) with ST(i) and set status flags accordingly.
DF F0+i	FCOMIP ST, ST(i)	Valid	Valid	Compare ST(0) with ST(i), set status flags accordingly, and pop register stack.
DB E8+i	FUCOMI ST, ST(i)	Valid	Valid	Compare ST(0) with ST(i), check for ordered values, and set status flags accordingly.
DF E8+i	FUCOMIP ST, ST(i)	Valid	Valid	Compare ST(0) with ST(i), check for ordered values, set status flags accordingly, and pop register stack.

Description

Performs an unordered comparison of the contents of registers ST(0) and ST(i) and sets the status flags ZF, PF, and CF in the EFLAGS register according to the results (see the table below). The sign of zero is ignored for comparisons, so that −0.0 is equal to +0.0.

Table 3-22. FCOMI/FCOMIP/ FUCOMI/FUCOMIP Results

Comparison Results*	ZF	PF	CF
ST0 > ST(i)	0	0	0
ST0 < ST(i)	0	0	1
ST0 = ST(i)	1	0	0
Unordered**	1	1	1

NOTES:

* See the IA-32 Architecture Compatibility section below.

** Flags not set if unmasked invalid-arithmetic-operand (#IA) exception is generated.

An unordered comparison checks the class of the numbers being compared (see "FXAM—Examine Floating-Point" in this chapter). The FUCOMI/FUCOMIP instructions perform the same operations as the FCOMI/FCOMIP instructions. The only difference is that the FUCOMI/FUCOMIP instructions raise the invalid-arithmetic-operand exception (#IA) only when either or both operands are an SNaN or are in an unsupported format; QNaNs cause the condition code flags to be set to unordered, but do not cause an exception to be generated. The FCOMI/FCOMIP instructions raise an invalid-operation exception when either or both of the operands are a NaN value of any kind or are in an unsupported format.

If the operation results in an invalid-arithmetic-operand exception being raised, the status flags in the EFLAGS register are set only if the exception is masked.

The FCOMI/FCOMIP and FUCOMI/FUCOMIP instructions set the OF, SF and AF flags to zero in the EFLAGS register (regardless of whether an invalid-operation exception is detected).

The FCOMIP and FUCOMIP instructions also pop the register stack following the comparison operation. To pop the register stack, the processor marks the ST(0) register as empty and increments the stack pointer (TOP) by 1.

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

IA-32 Architecture Compatibility

The FCOMI/FCOMIP/FUCOMI/FUCOMIP instructions were introduced to the IA-32 Architecture in the P6 family processors and are not available in earlier IA-32 processors.

Operation

```
CASE (relation of operands) OF
    ST(0) > ST(i):      ZF, PF, CF ← 000;
    ST(0) < ST(i):      ZF, PF, CF ← 001;
    ST(0) = ST(i):      ZF, PF, CF ← 100;
ESAC;

IF Instruction is FCOMI or FCOMIP
    THEN
        IF ST(0) or ST(i) = NaN or unsupported format
            THEN
                #IA
                IF FPUControlWord.IM = 1
                    THEN
                        ZF, PF, CF ← 111;
                FI;
        FI;
FI;

IF Instruction is FUCOMI or FUCOMIP
    THEN
        IF ST(0) or ST(i) = QNaN, but not SNaN or unsupported format
            THEN
                ZF, PF, CF ← 111;
            ELSE (* ST(0) or ST(i) is SNaN or unsupported format *)
                #IA;
                IF FPUControlWord.IM = 1
                    THEN
                        ZF, PF, CF ← 111;
                FI;
        FI;
FI;

IF Instruction is FCOMIP or FUCOMIP
    THEN
        PopRegisterStack;
FI;
```

FPU Flags Affected

C1	Set to 0.
C0, C2, C3	Not affected.

Floating-Point Exceptions

#IS	Stack underflow occurred.
#IA	(FCOMI or FCOMIP instruction) One or both operands are NaN values or have unsupported formats. (FUCOMI or FUCOMIP instruction) One or both operands are SNaN values (but not QNaNs) or have undefined formats. Detection of a QNaN value does not raise an invalid-operand exception.

Protected Mode Exceptions

#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#MF	If there is a pending x87 FPU exception.
#UD	If the LOCK prefix is used.

Real-Address Mode Exceptions

Same exceptions as in protected mode.

Virtual-8086 Mode Exceptions

Same exceptions as in protected mode.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

Same exceptions as in protected mode.

FCOS— Cosine

Opcode	Instruction	64-Bit Mode	Compat/ Leg Mode	Description
D9 FF	FCOS	Valid	Valid	Replace ST(0) with its approximate cosine.

Description

Computes the approximate cosine of the source operand in register ST(0) and stores the result in ST(0). The source operand must be given in radians and must be within the range -2^{63} to $+2^{63}$. The following table shows the results obtained when taking the cosine of various classes of numbers.

Table 3-23. FCOS Results

ST(0) SRC	ST(0) DEST
−∞	*
−F	−1 to +1
−0	+1
+0	+1
+F	−1 to +1
+∞	*
NaN	NaN

NOTES:

F Means finite floating-point value.

* Indicates floating-point invalid-arithmetic-operand (#IA) exception.

If the source operand is outside the acceptable range, the C2 flag in the FPU status word is set, and the value in register ST(0) remains unchanged. The instruction does not raise an exception when the source operand is out of range. It is up to the program to check the C2 flag for out-of-range conditions. Source values outside the range -2^{63} to $+2^{63}$ can be reduced to the range of the instruction by subtracting an appropriate integer multiple of 2π. However, even within the range -2^{63} to $+2^{63}$, inaccurate results can occur because the finite approximation of π used internally for argument reduction is not sufficient in all cases. Therefore, for accurate results it is safe to apply FCOS only to arguments reduced accurately in software, to a value smaller in absolute value than $3\pi/8$. See the sections titled "Approximation of Pi" and "Transcendental Instruction Accuracy" in Chapter 8 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*, for a discussion of the proper value to use for π in performing such reductions.

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

Operation

```
IF |ST(0)| < 2^63
THEN
    C2 ← 0;
    ST(0) ← FCOS(ST(0)); // approximation of cosine
ELSE (* Source operand is out-of-range *)
    C2 ← 1;
FI;
```

FPU Flags Affected

C1	Set to 0 if stack underflow occurred. Set if result was rounded up; cleared otherwise. Undefined if C2 is 1.
C2	Set to 1 if outside range (-2^{63} < source operand < $+2^{63}$); otherwise, set to 0.
C0, C3	Undefined.

Floating-Point Exceptions

#IS	Stack underflow occurred.
#IA	Source operand is an SNaN value, ∞, or unsupported format.
#D	Source is a denormal value.
#P	Value cannot be represented exactly in destination format.

Protected Mode Exceptions

#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#MF	If there is a pending x87 FPU exception.
#UD	If the LOCK prefix is used.

Real-Address Mode Exceptions

Same exceptions as in protected mode.

Virtual-8086 Mode Exceptions

Same exceptions as in protected mode.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

Same exceptions as in protected mode.

FDECSTP—Decrement Stack-Top Pointer

Opcode	Instruction	64-Bit Mode	Compat/ Leg Mode	Description
D9 F6	FDECSTP	Valid	Valid	Decrement TOP field in FPU status word.

Description

Subtracts one from the TOP field of the FPU status word (decrements the top-of-stack pointer). If the TOP field contains a 0, it is set to 7. The effect of this instruction is to rotate the stack by one position. The contents of the FPU data registers and tag register are not affected.

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

Operation

```
IF TOP = 0
    THEN TOP ← 7;
    ELSE TOP ← TOP – 1;
FI;
```

FPU Flags Affected

The C1 flag is set to 0. The C0, C2, and C3 flags are undefined.

Floating-Point Exceptions

None.

Protected Mode Exceptions

#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#MF	If there is a pending x87 FPU exception.
#UD	If the LOCK prefix is used.

Real-Address Mode Exceptions

Same exceptions as in protected mode.

Virtual-8086 Mode Exceptions

Same exceptions as in protected mode.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

Same exceptions as in protected mode.

FDIV/FDIVP/FIDIV—Divide

Opcode	Instruction	64-Bit Mode	Compat/ Leg Mode	Description
D8 /6	FDIV *m32fp*	Valid	Valid	Divide ST(0) by *m32fp* and store result in ST(0).
DC /6	FDIV *m64fp*	Valid	Valid	Divide ST(0) by *m64fp* and store result in ST(0).
D8 F0+i	FDIV ST(0), ST(i)	Valid	Valid	Divide ST(0) by ST(i) and store result in ST(0).
DC F8+i	FDIV ST(i), ST(0)	Valid	Valid	Divide ST(i) by ST(0) and store result in ST(i).
DE F8+i	FDIVP ST(i), ST(0)	Valid	Valid	Divide ST(i) by ST(0), store result in ST(i), and pop the register stack.
DE F9	FDIVP	Valid	Valid	Divide ST(1) by ST(0), store result in ST(1), and pop the register stack.
DA /6	FIDIV *m32int*	Valid	Valid	Divide ST(0) by *m32int* and store result in ST(0).
DE /6	FIDIV *m16int*	Valid	Valid	Divide ST(0) by *m16int* and store result in ST(0).

Description

Divides the destination operand by the source operand and stores the result in the destination location. The destination operand (dividend) is always in an FPU register; the source operand (divisor) can be a register or a memory location. Source operands in memory can be in single-precision or double-precision floating-point format, word or doubleword integer format.

The no-operand version of the instruction divides the contents of the ST(1) register by the contents of the ST(0) register. The one-operand version divides the contents of the ST(0) register by the contents of a memory location (either a floating-point or an integer value). The two-operand version, divides the contents of the ST(0) register by the contents of the ST(i) register or vice versa.

The FDIVP instructions perform the additional operation of popping the FPU register stack after storing the result. To pop the register stack, the processor marks the ST(0) register as empty and increments the stack pointer (TOP) by 1. The no-operand version of the floating-point divide instructions always results in the register stack being popped. In some assemblers, the mnemonic for this instruction is FDIV rather than FDIVP.

The FIDIV instructions convert an integer source operand to double extended-precision floating-point format before performing the division. When the source operand is an integer 0, it is treated as a +0.

If an unmasked divide-by-zero exception (#Z) is generated, no result is stored; if the exception is masked, an ∞ of the appropriate sign is stored in the destination operand.

The following table shows the results obtained when dividing various classes of numbers, assuming that neither overflow nor underflow occurs.

Table 3-24. FDIV/FDIVP/FIDIV Results

		DEST						
		−∞	−F	−0	+0	+F	+∞	NaN
SRC	−∞	*	+0	+0	−0	−0	*	NaN
	−F	+∞	+F	+0	−0	−F	−∞	NaN
	−I	+∞	+F	+0	−0	−F	−∞	NaN
	−0	+∞	**	*	*	**	−∞	NaN
	+0	−∞	**	*	*	**	+∞	NaN
	+I	−∞	−F	−0	+0	+F	+∞	NaN
	+F	−∞	−F	−0	+0	+F	+∞	NaN
	+∞	*	−0	−0	+0	+0	*	NaN
	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN

NOTES:

F Means finite floating-point value.

I Means integer.

* Indicates floating-point invalid-arithmetic-operand (#IA) exception.

** Indicates floating-point zero-divide (#Z) exception.

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

Operation

```
IF SRC = 0
    THEN
        #Z;
    ELSE
        IF Instruction is FIDIV
            THEN
                DEST ← DEST / ConvertToDoubleExtendedPrecisionFP(SRC);
            ELSE (* Source operand is floating-point value *)
                DEST ← DEST / SRC;
        FI;
FI;

IF Instruction = FDIVP
    THEN
        PopRegisterStack;
FI;
```

FPU Flags Affected

C1	Set to 0 if stack underflow occurred. Set if result was rounded up; cleared otherwise.
C0, C2, C3	Undefined.

Floating-Point Exceptions

#IS	Stack underflow occurred.
#IA	Operand is an SNaN value or unsupported format. $\pm\infty$ / $\pm\infty$; ±0 / ±0
#D	Source is a denormal value.
#Z	DEST / ±0, where DEST is not equal to ±0.
#U	Result is too small for destination format.
#O	Result is too large for destination format.
#P	Value cannot be represented exactly in destination format.

Protected Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit. If the DS, ES, FS, or GS register contains a NULL segment selector.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If the LOCK prefix is used.

Real-Address Mode Exceptions

#GP	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS	If a memory operand effective address is outside the SS segment limit.
#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#UD	If the LOCK prefix is used.

Virtual-8086 Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.
#UD	If the LOCK prefix is used.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form.
#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#MF	If there is a pending x87 FPU exception.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If the LOCK prefix is used.

FDIVR/FDIVRP/FIDIVR—Reverse Divide

Opcode	Instruction	64-Bit Mode	Compat/ Leg Mode	Description
D8 /7	FDIVR *m32fp*	Valid	Valid	Divide *m32fp* by ST(0) and store result in ST(0).
DC /7	FDIVR *m64fp*	Valid	Valid	Divide *m64fp* by ST(0) and store result in ST(0).
D8 F8+i	FDIVR ST(0), ST(i)	Valid	Valid	Divide ST(i) by ST(0) and store result in ST(0).
DC F0+i	FDIVR ST(i), ST(0)	Valid	Valid	Divide ST(0) by ST(i) and store result in ST(i).
DE F0+i	FDIVRP ST(i), ST(0)	Valid	Valid	Divide ST(0) by ST(i), store result in ST(i), and pop the register stack.
DE F1	FDIVRP	Valid	Valid	Divide ST(0) by ST(1), store result in ST(1), and pop the register stack.
DA /7	FIDIVR *m32int*	Valid	Valid	Divide *m32int* by ST(0) and store result in ST(0).
DE /7	FIDIVR *m16int*	Valid	Valid	Divide *m16int* by ST(0) and store result in ST(0).

Description

Divides the source operand by the destination operand and stores the result in the destination location. The destination operand (divisor) is always in an FPU register; the source operand (dividend) can be a register or a memory location. Source operands in memory can be in single-precision or double-precision floating-point format, word or doubleword integer format.

These instructions perform the reverse operations of the FDIV, FDIVP, and FIDIV instructions. They are provided to support more efficient coding.

The no-operand version of the instruction divides the contents of the ST(0) register by the contents of the ST(1) register. The one-operand version divides the contents of a memory location (either a floating-point or an integer value) by the contents of the ST(0) register. The two-operand version, divides the contents of the ST(i) register by the contents of the ST(0) register or vice versa.

The FDIVRP instructions perform the additional operation of popping the FPU register stack after storing the result. To pop the register stack, the processor marks the ST(0) register as empty and increments the stack pointer (TOP) by 1. The no-operand version of the floating-point divide instructions always results in the register stack being popped. In some assemblers, the mnemonic for this instruction is FDIVR rather than FDIVRP.

The FIDIVR instructions convert an integer source operand to double extended-precision floating-point format before performing the division.

If an unmasked divide-by-zero exception (#Z) is generated, no result is stored; if the exception is masked, an ∞ of the appropriate sign is stored in the destination operand.

The following table shows the results obtained when dividing various classes of numbers, assuming that neither overflow nor underflow occurs.

Table 3-25. FDIVR/FDIVRP/FIDIVR Results

		DEST						
		− ∞	− F	− 0	+ 0	+ F	+ ∞	NaN
SRC	− ∞	*	+ ∞	+ ∞	− ∞	− ∞	*	NaN
	− F	+ 0	+ F	**	**	− F	− 0	NaN
	− I	+ 0	+ F	**	**	− F	− 0	NaN
	− 0	+ 0	+ 0	*	*	− 0	− 0	NaN
	+ 0	− 0	− 0	*	*	+ 0	+ 0	NaN
	+ I	− 0	− F	**	**	+ F	+ 0	NaN
	+ F	− 0	− F	**	**	+ F	+ 0	NaN
	+ ∞	*	− ∞	− ∞	+ ∞	+ ∞	*	NaN
	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN

NOTES:

F Means finite floating-point value.

I Means integer.

* Indicates floating-point invalid-arithmetic-operand (#IA) exception.

** Indicates floating-point zero-divide (#Z) exception.

When the source operand is an integer 0, it is treated as a +0. This instruction's operation is the same in non-64-bit modes and 64-bit mode.

Operation

```
IF DEST = 0
    THEN
        #Z;
    ELSE
        IF Instruction = FIDIVR
            THEN
                DEST ← ConvertToDoubleExtendedPrecisionFP(SRC) / DEST;
            ELSE (* Source operand is floating-point value *)
                DEST ← SRC / DEST;
        FI;
FI;

IF Instruction = FDIVRP
    THEN
        PopRegisterStack;
FI;
```

FPU Flags Affected

C1	Set to 0 if stack underflow occurred. Set if result was rounded up; cleared otherwise.
C0, C2, C3	Undefined.

Floating-Point Exceptions

#IS	Stack underflow occurred.
#IA	Operand is an SNaN value or unsupported format.
	±∞ / ±∞; ±0 / ±0
#D	Source is a denormal value.
#Z	SRC / ±0, where SRC is not equal to ±0.
#U	Result is too small for destination format.
#O	Result is too large for destination format.
#P	Value cannot be represented exactly in destination format.

Protected Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
	If the DS, ES, FS, or GS register contains a NULL segment selector.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If the LOCK prefix is used.

Real-Address Mode Exceptions

#GP	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS	If a memory operand effective address is outside the SS segment limit.
#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#UD	If the LOCK prefix is used.

Virtual-8086 Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.
#UD	If the LOCK prefix is used.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form.
#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#MF	If there is a pending x87 FPU exception.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If the LOCK prefix is used.

FFREE—Free Floating-Point Register

Opcode	Instruction	64-Bit Mode	Compat/ Leg Mode	Description
DD C0+i	FFREE ST(i)	Valid	Valid	Sets tag for ST(i) to empty.

Description

Sets the tag in the FPU tag register associated with register ST(i) to empty (11B). The contents of ST(i) and the FPU stack-top pointer (TOP) are not affected.

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

Operation

TAG(i) ← 11B;

FPU Flags Affected

C0, C1, C2, C3 undefined.

Floating-Point Exceptions

None

Protected Mode Exceptions

#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#MF	If there is a pending x87 FPU exception.
#UD	If the LOCK prefix is used.

Real-Address Mode Exceptions

Same exceptions as in protected mode.

Virtual-8086 Mode Exceptions

Same exceptions as in protected mode.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

Same exceptions as in protected mode.

FICOM/FICOMP—Compare Integer

Opcode	Instruction	64-Bit Mode	Compat/ Leg Mode	Description
DE /2	FICOM *m16int*	Valid	Valid	Compare ST(0) with *m16int.*
DA /2	FICOM *m32int*	Valid	Valid	Compare ST(0) with *m32int.*
DE /3	FICOMP *m16int*	Valid	Valid	Compare ST(0) with *m16int* and pop stack register.
DA /3	FICOMP *m32int*	Valid	Valid	Compare ST(0) with *m32int* and pop stack register.

Description

Compares the value in ST(0) with an integer source operand and sets the condition code flags C0, C2, and C3 in the FPU status word according to the results (see table below). The integer value is converted to double extended-precision floating-point format before the comparison is made.

Table 3-26. FICOM/FICOMP Results

Condition	C3	C2	C0
ST(0) > SRC	0	0	0
ST(0) < SRC	0	0	1
ST(0) = SRC	1	0	0
Unordered	1	1	1

These instructions perform an "unordered comparison." An unordered comparison also checks the class of the numbers being compared (see "FXAM—Examine Floating-Point" in this chapter). If either operand is a NaN or is in an undefined format, the condition flags are set to "unordered."

The sign of zero is ignored, so that −0.0 ← +0.0.

The FICOMP instructions pop the register stack following the comparison. To pop the register stack, the processor marks the ST(0) register empty and increments the stack pointer (TOP) by 1.

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

Operation

```
CASE (relation of operands) OF
    ST(0) > SRC:        C3, C2, C0 ← 000;
    ST(0) < SRC:        C3, C2, C0 ← 001;
    ST(0) = SRC:        C3, C2, C0 ← 100;
    Unordered:          C3, C2, C0 ← 111;
ESAC;

IF Instruction = FICOMP
    THEN
        PopRegisterStack;
FI;
```

FPU Flags Affected

C1	Set to 0.
C0, C2, C3	See table on previous page.

Floating-Point Exceptions

#IS	Stack underflow occurred.
#IA	One or both operands are NaN values or have unsupported formats.
#D	One or both operands are denormal values.

Protected Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit. If the DS, ES, FS, or GS register contains a NULL segment selector.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If the LOCK prefix is used.

Real-Address Mode Exceptions

#GP	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS	If a memory operand effective address is outside the SS segment limit.
#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#UD	If the LOCK prefix is used.

Virtual-8086 Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.
#UD	If the LOCK prefix is used.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form.
#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#MF	If there is a pending x87 FPU exception.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If the LOCK prefix is used.

FILD—Load Integer

Opcode	Instruction	64-Bit Mode	Compat/ Leg Mode	Description
DF /0	FILD *m16int*	Valid	Valid	Push *m16int* onto the FPU register stack.
DB /0	FILD *m32int*	Valid	Valid	Push *m32int* onto the FPU register stack.
DF /5	FILD *m64int*	Valid	Valid	Push *m64int* onto the FPU register stack.

Description

Converts the signed-integer source operand into double extended-precision floating-point format and pushes the value onto the FPU register stack. The source operand can be a word, doubleword, or quadword integer. It is loaded without rounding errors. The sign of the source operand is preserved.

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

Operation

```
TOP ← TOP – 1;
ST(0) ← ConvertToDoubleExtendedPrecisionFP(SRC);
```

FPU Flags Affected

C1	Set to 1 if stack overflow occurred; set to 0 otherwise.
C0, C2, C3	Undefined.

Floating-Point Exceptions

#IS	Stack overflow occurred.

Protected Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit. If the DS, ES, FS, or GS register contains a NULL segment selector.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If the LOCK prefix is used.

Real-Address Mode Exceptions

#GP	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS	If a memory operand effective address is outside the SS segment limit.
#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#UD	If the LOCK prefix is used.

Virtual-8086 Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.
#UD	If the LOCK prefix is used.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form.
#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#MF	If there is a pending x87 FPU exception.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If the LOCK prefix is used.

FINCSTP—Increment Stack-Top Pointer

Opcode	Instruction	64-Bit Mode	Compat/ Leg Mode	Description
D9 F7	FINCSTP	Valid	Valid	Increment the TOP field in the FPU status register.

Description

Adds one to the TOP field of the FPU status word (increments the top-of-stack pointer). If the TOP field contains a 7, it is set to 0. The effect of this instruction is to rotate the stack by one position. The contents of the FPU data registers and tag register are not affected. This operation is not equivalent to popping the stack, because the tag for the previous top-of-stack register is not marked empty.

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

Operation

```
IF TOP = 7
    THEN TOP ← 0;
    ELSE TOP ← TOP + 1;
FI;
```

FPU Flags Affected

The C1 flag is set to 0. The C0, C2, and C3 flags are undefined.

Floating-Point Exceptions

None

Protected Mode Exceptions

#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#MF	If there is a pending x87 FPU exception.
#UD	If the LOCK prefix is used.

Real-Address Mode Exceptions

Same exceptions as in protected mode.

Virtual-8086 Mode Exceptions

Same exceptions as in protected mode.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

Same exceptions as in protected mode.

FINIT/FNINIT—Initialize Floating-Point Unit

Opcode	Instruction	64-Bit Mode	Compat/ Leg Mode	Description
9B DB E3	FINIT	Valid	Valid	Initialize FPU after checking for pending unmasked floating-point exceptions.
DB E3	FNINIT*	Valid	Valid	Initialize FPU without checking for pending unmasked floating-point exceptions.

NOTES:

* See IA-32 Architecture Compatibility section below.

Description

Sets the FPU control, status, tag, instruction pointer, and data pointer registers to their default states. The FPU control word is set to 037FH (round to nearest, all exceptions masked, 64-bit precision). The status word is cleared (no exception flags set, TOP is set to 0). The data registers in the register stack are left unchanged, but they are all tagged as empty (11B). Both the instruction and data pointers are cleared.

The FINIT instruction checks for and handles any pending unmasked floating-point exceptions before performing the initialization; the FNINIT instruction does not.

The assembler issues two instructions for the FINIT instruction (an FWAIT instruction followed by an FNINIT instruction), and the processor executes each of these instructions in separately. If an exception is generated for either of these instructions, the save EIP points to the instruction that caused the exception.

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

IA-32 Architecture Compatibility

When operating a Pentium or Intel486 processor in MS-DOS compatibility mode, it is possible (under unusual circumstances) for an FNINIT instruction to be interrupted prior to being executed to handle a pending FPU exception. See the section titled "No-Wait FPU Instructions Can Get FPU Interrupt in Window" in Appendix D of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*, for a description of these circumstances. An FNINIT instruction cannot be interrupted in this way on later Intel processors, except for the Intel Quark™ X1000 processor.

In the Intel387 math coprocessor, the FINIT/FNINIT instruction does not clear the instruction and data pointers.

This instruction affects only the x87 FPU. It does not affect the XMM and MXCSR registers.

Operation

```
FPUControlWord ← 037FH;
FPUStatusWord ← 0;
FPUTagWord ← FFFFH;
FPUDataPointer ← 0;
FPUInstructionPointer ← 0;
FPULastInstructionOpcode ← 0;
```

FPU Flags Affected

C0, C1, C2, C3 set to 0.

Floating-Point Exceptions

None

Protected Mode Exceptions

#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#MF	If there is a pending x87 FPU exception.
#UD	If the LOCK prefix is used.

Real-Address Mode Exceptions

Same exceptions as in protected mode.

Virtual-8086 Mode Exceptions

Same exceptions as in protected mode.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

Same exceptions as in protected mode.

FIST/FISTP—Store Integer

Opcode	Instruction	64-Bit Mode	Compat/ Leg Mode	Description
DF /2	FIST *m16int*	Valid	Valid	Store ST(0) in *m16int.*
DB /2	FIST *m32int*	Valid	Valid	Store ST(0) in *m32int.*
DF /3	FISTP *m16int*	Valid	Valid	Store ST(0) in *m16int* and pop register stack.
DB /3	FISTP *m32int*	Valid	Valid	Store ST(0) in *m32int* and pop register stack.
DF /7	FISTP *m64int*	Valid	Valid	Store ST(0) in *m64int* and pop register stack.

Description

The FIST instruction converts the value in the ST(0) register to a signed integer and stores the result in the destination operand. Values can be stored in word or doubleword integer format. The destination operand specifies the address where the first byte of the destination value is to be stored.

The FISTP instruction performs the same operation as the FIST instruction and then pops the register stack. To pop the register stack, the processor marks the ST(0) register as empty and increments the stack pointer (TOP) by 1. The FISTP instruction also stores values in quadword integer format.

The following table shows the results obtained when storing various classes of numbers in integer format.

Table 3-27. FIST/FISTP Results

ST(0)	DEST
$-\infty$ or Value Too Large for DEST Format	*
$F \leq -1$	−I
$-1 < F < -0$	**
−0	0
+0	0
$+0 < F < +1$	**
$F \geq +1$	+I
$+\infty$ or Value Too Large for DEST Format	*
NaN	*

NOTES:

F Means finite floating-point value.

I Means integer.

* Indicates floating-point invalid-operation (#IA) exception.

** 0 or ±1, depending on the rounding mode.

If the source value is a non-integral value, it is rounded to an integer value, according to the rounding mode specified by the RC field of the FPU control word.

If the converted value is too large for the destination format, or if the source operand is an ∞, SNaN, QNAN, or is in an unsupported format, an invalid-arithmetic-operand condition is signaled. If the invalid-operation exception is not masked, an invalid-arithmetic-operand exception (#IA) is generated and no value is stored in the destination operand. If the invalid-operation exception is masked, the integer indefinite value is stored in memory.

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

Operation

```
DEST ← Integer(ST(0));

IF Instruction = FISTP
    THEN
        PopRegisterStack;
FI;
```

FPU Flags Affected

C1	Set to 0 if stack underflow occurred. Indicates rounding direction of if the inexact exception (#P) is generated: 0 ← not roundup; 1 ← roundup. Set to 0 otherwise.
C0, C2, C3	Undefined.

Floating-Point Exceptions

#IS	Stack underflow occurred.
#IA	Converted value is too large for the destination format. Source operand is an SNaN, QNaN, ±∞, or unsupported format.
#P	Value cannot be represented exactly in destination format.

Protected Mode Exceptions

#GP(0)	If the destination is located in a non-writable segment. If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit. If the DS, ES, FS, or GS register is used to access memory and it contains a NULL segment selector.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If the LOCK prefix is used.

Real-Address Mode Exceptions

#GP	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS	If a memory operand effective address is outside the SS segment limit.
#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#UD	If the LOCK prefix is used.

Virtual-8086 Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.
#UD	If the LOCK prefix is used.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form.
#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#MF	If there is a pending x87 FPU exception.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If the LOCK prefix is used.

FISTTP—Store Integer with Truncation

Opcode	Instruction	64-Bit Mode	Compat/ Leg Mode	Description
DF /1	FISTTP *m16int*	Valid	Valid	Store ST(0) in *m16int with truncation.*
DB /1	FISTTP *m32int*	Valid	Valid	Store ST(0) in *m32int with truncation.*
DD /1	FISTTP *m64int*	Valid	Valid	Store ST(0) in *m64int* with truncation.

Description

FISTTP converts the value in ST into a signed integer using truncation (chop) as rounding mode, transfers the result to the destination, and pop ST. FISTTP accepts word, short integer, and long integer destinations.

The following table shows the results obtained when storing various classes of numbers in integer format.

Table 3-28. FISTTP Results

ST(0)	DEST
$-\infty$ or Value Too Large for DEST Format	*
F ≤ -1	−I
$-1 < F < +1$	0
F Š +1	+I
$+\infty$ or Value Too Large for DEST Format	*
NaN	*

NOTES:

F Means finite floating-point value.

I Means integer.

* Indicates floating-point invalid-operation (#IA) exception.

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

Operation

```
DEST ← ST;
pop ST;
```

Flags Affected

C1 is cleared; C0, C2, C3 undefined.

Numeric Exceptions

Invalid, Stack Invalid (stack underflow), Precision.

Protected Mode Exceptions

#GP(0)	If the destination is in a nonwritable segment.
	For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.
#SS(0)	For an illegal address in the SS segment.
#PF(fault-code)	For a page fault.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#NM	If CR0.EM[bit 2] = 1.
	If CR0.TS[bit 3] = 1.
#UD	If CPUID.01H:ECX.SSE3[bit 0] = 0.
	If the LOCK prefix is used.

Real Address Mode Exceptions

GP(0)	If any part of the operand would lie outside of the effective address space from 0 to 0FFFFH.
#NM	If CR0.EM[bit 2] = 1.
	If CR0.TS[bit 3] = 1.
#UD	If CPUID.01H:ECX.SSE3[bit 0] = 0.
	If the LOCK prefix is used.

Virtual 8086 Mode Exceptions

GP(0)	If any part of the operand would lie outside of the effective address space from 0 to 0FFFFH.
#NM	If CR0.EM[bit 2] = 1.
	If CR0.TS[bit 3] = 1.
#UD	If CPUID.01H:ECX.SSE3[bit 0] = 0.
	If the LOCK prefix is used.
#PF(fault-code)	For a page fault.
#AC(0)	For unaligned memory reference if the current privilege is 3.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form.
#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#MF	If there is a pending x87 FPU exception.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
	If the LOCK prefix is used.

FLD—Load Floating Point Value

Opcode	Instruction	64-Bit Mode	Compat/ Leg Mode	Description
D9 /0	FLD *m32fp*	Valid	Valid	Push *m32fp* onto the FPU register stack.
DD /0	FLD *m64fp*	Valid	Valid	Push *m64fp* onto the FPU register stack.
DB /5	FLD *m80fp*	Valid	Valid	Push *m80fp* onto the FPU register stack.
D9 C0+i	FLD ST(i)	Valid	Valid	Push ST(i) onto the FPU register stack.

Description

Pushes the source operand onto the FPU register stack. The source operand can be in single-precision, double-precision, or double extended-precision floating-point format. If the source operand is in single-precision or double-precision floating-point format, it is automatically converted to the double extended-precision floating-point format before being pushed on the stack.

The FLD instruction can also push the value in a selected FPU register [ST(i)] onto the stack. Here, pushing register ST(0) duplicates the stack top.

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

Operation

```
IF SRC is ST(i)
    THEN
        temp ← ST(i);
FI;

TOP ← TOP − 1;

IF SRC is memory-operand
    THEN
        ST(0) ← ConvertToDoubleExtendedPrecisionFP(SRC);
    ELSE (* SRC is ST(i) *)
        ST(0) ← temp;
FI;
```

FPU Flags Affected

C1	Set to 1 if stack overflow occurred; otherwise, set to 0.
C0, C2, C3	Undefined.

Floating-Point Exceptions

#IS	Stack underflow or overflow occurred.
#IA	Source operand is an SNaN. Does not occur if the source operand is in double extended-precision floating-point format (FLD m80fp or FLD ST(i)).
#D	Source operand is a denormal value. Does not occur if the source operand is in double extended-precision floating-point format.

Protected Mode Exceptions

#GP(0)	If destination is located in a non-writable segment.
	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
	If the DS, ES, FS, or GS register is used to access memory and it contains a NULL segment selector.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If the LOCK prefix is used.

Real-Address Mode Exceptions

#GP	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS	If a memory operand effective address is outside the SS segment limit.
#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#UD	If the LOCK prefix is used.

Virtual-8086 Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.
#UD	If the LOCK prefix is used.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form.
#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#MF	If there is a pending x87 FPU exception.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If the LOCK prefix is used.

FLD1/FLDL2T/FLDL2E/FLDPI/FLDLG2/FLDLN2/FLDZ—Load Constant

Opcode*	Instruction	64-Bit Mode	Compat/ Leg Mode	Description
D9 E8	FLD1	Valid	Valid	Push +1.0 onto the FPU register stack.
D9 E9	FLDL2T	Valid	Valid	Push $\log_2 10$ onto the FPU register stack.
D9 EA	FLDL2E	Valid	Valid	Push $\log_2 e$ onto the FPU register stack.
D9 EB	FLDPI	Valid	Valid	Push π onto the FPU register stack.
D9 EC	FLDLG2	Valid	Valid	Push $\log_{10} 2$ onto the FPU register stack.
D9 ED	FLDLN2	Valid	Valid	Push $\log_e 2$ onto the FPU register stack.
D9 EE	FLDZ	Valid	Valid	Push +0.0 onto the FPU register stack.

NOTES:

* See IA-32 Architecture Compatibility section below.

Description

Push one of seven commonly used constants (in double extended-precision floating-point format) onto the FPU register stack. The constants that can be loaded with these instructions include +1.0, +0.0, $\log_2 10$, $\log_2 e$, π, $\log_{10} 2$, and $\log_e 2$. For each constant, an internal 66-bit constant is rounded (as specified by the RC field in the FPU control word) to double extended-precision floating-point format. The inexact-result exception (#P) is not generated as a result of the rounding, nor is the C1 flag set in the x87 FPU status word if the value is rounded up.

See the section titled "Approximation of Pi" in Chapter 8 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*, for a description of the π constant.

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

IA-32 Architecture Compatibility

When the RC field is set to round-to-nearest, the FPU produces the same constants that is produced by the Intel 8087 and Intel 287 math coprocessors.

Operation

```
TOP ← TOP – 1;
ST(0) ← CONSTANT;
```

FPU Flags Affected

C1	Set to 1 if stack overflow occurred; otherwise, set to 0.
C0, C2, C3	Undefined.

Floating-Point Exceptions

#IS	Stack overflow occurred.

Protected Mode Exceptions

#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#MF	If there is a pending x87 FPU exception.
#UD	If the LOCK prefix is used.

Real-Address Mode Exceptions

Same exceptions as in protected mode.

Virtual-8086 Mode Exceptions

Same exceptions as in protected mode.

Virtual-8086 Mode Exceptions

Same exceptions as in protected mode.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

Same exceptions as in protected mode.

FLDCW—Load x87 FPU Control Word

Opcode	Instruction	64-Bit Mode	Compat/ Leg Mode	Description
D9 /5	FLDCW m2byte	Valid	Valid	Load FPU control word from *m2byte*.

Description

Loads the 16-bit source operand into the FPU control word. The source operand is a memory location. This instruction is typically used to establish or change the FPU's mode of operation.

If one or more exception flags are set in the FPU status word prior to loading a new FPU control word and the new control word unmasks one or more of those exceptions, a floating-point exception will be generated upon execution of the next floating-point instruction (except for the no-wait floating-point instructions, see the section titled "Software Exception Handling" in Chapter 8 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*). To avoid raising exceptions when changing FPU operating modes, clear any pending exceptions (using the FCLEX or FNCLEX instruction) before loading the new control word.

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

Operation

```
FPUControlWord ← SRC;
```

FPU Flags Affected

C0, C1, C2, C3 undefined.

Floating-Point Exceptions

None; however, this operation might unmask a pending exception in the FPU status word. That exception is then generated upon execution of the next "waiting" floating-point instruction.

Protected Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit. If the DS, ES, FS, or GS register is used to access memory and it contains a NULL segment selector.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If the LOCK prefix is used.

Real-Address Mode Exceptions

#GP	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS	If a memory operand effective address is outside the SS segment limit.
#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#UD	If the LOCK prefix is used.

Virtual-8086 Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.
#UD	If the LOCK prefix is used.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form.
#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#MF	If there is a pending x87 FPU exception.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If the LOCK prefix is used.

FLDENV—Load x87 FPU Environment

Opcode	Instruction	64-Bit Mode	Compat/ Leg Mode	Description
D9 /4	FLDENV *m14/28byte*	Valid	Valid	Load FPU environment from *m14byte* or *m28byte*.

Description

Loads the complete x87 FPU operating environment from memory into the FPU registers. The source operand specifies the first byte of the operating-environment data in memory. This data is typically written to the specified memory location by a FSTENV or FNSTENV instruction.

The FPU operating environment consists of the FPU control word, status word, tag word, instruction pointer, data pointer, and last opcode. Figures 8-9 through 8-12 in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*, show the layout in memory of the loaded environment, depending on the operating mode of the processor (protected or real) and the current operand-size attribute (16-bit or 32-bit). In virtual-8086 mode, the real mode layouts are used.

The FLDENV instruction should be executed in the same operating mode as the corresponding FSTENV/FNSTENV instruction.

If one or more unmasked exception flags are set in the new FPU status word, a floating-point exception will be generated upon execution of the next floating-point instruction (except for the no-wait floating-point instructions, see the section titled "Software Exception Handling" in Chapter 8 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*). To avoid generating exceptions when loading a new environment, clear all the exception flags in the FPU status word that is being loaded.

If a page or limit fault occurs during the execution of this instruction, the state of the x87 FPU registers as seen by the fault handler may be different than the state being loaded from memory. In such situations, the fault handler should ignore the status of the x87 FPU registers, handle the fault, and return. The FLDENV instruction will then complete the loading of the x87 FPU registers with no resulting context inconsistency.

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

Operation

```
FPUControlWord ← SRC[FPUControlWord];
FPUStatusWord ← SRC[FPUStatusWord];
FPUTagWord ← SRC[FPUTagWord];
FPUDataPointer ← SRC[FPUDataPointer];
FPUInstructionPointer ← SRC[FPUInstructionPointer];
FPULastInstructionOpcode ← SRC[FPULastInstructionOpcode];
```

FPU Flags Affected

The C0, C1, C2, C3 flags are loaded.

Floating-Point Exceptions

None; however, if an unmasked exception is loaded in the status word, it is generated upon execution of the next "waiting" floating-point instruction.

Protected Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit. If the DS, ES, FS, or GS register is used to access memory and it contains a NULL segment selector.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If the LOCK prefix is used.

Real-Address Mode Exceptions

#GP	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS	If a memory operand effective address is outside the SS segment limit.
#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#UD	If the LOCK prefix is used.

Virtual-8086 Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.
#UD	If the LOCK prefix is used.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form.
#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#MF	If there is a pending x87 FPU exception.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If the LOCK prefix is used.

FMUL/FMULP/FIMUL—Multiply

Opcode	Instruction	64-Bit Mode	Compat/ Leg Mode	Description
D8 /1	FMUL *m32fp*	Valid	Valid	Multiply ST(0) by *m32fp* and store result in ST(0).
DC /1	FMUL *m64fp*	Valid	Valid	Multiply ST(0) by *m64fp* and store result in ST(0).
D8 C8+i	FMUL ST(0), ST(i)	Valid	Valid	Multiply ST(0) by ST(i) and store result in ST(0).
DC C8+i	FMUL ST(i), ST(0)	Valid	Valid	Multiply ST(i) by ST(0) and store result in ST(i).
DE C8+i	FMULP ST(i), ST(0)	Valid	Valid	Multiply ST(i) by ST(0), store result in ST(i), and pop the register stack.
DE C9	FMULP	Valid	Valid	Multiply ST(1) by ST(0), store result in ST(1), and pop the register stack.
DA /1	FIMUL *m32int*	Valid	Valid	Multiply ST(0) by *m32int* and store result in ST(0).
DE /1	FIMUL *m16int*	Valid	Valid	Multiply ST(0) by *m16int* and store result in ST(0).

Description

Multiplies the destination and source operands and stores the product in the destination location. The destination operand is always an FPU data register; the source operand can be an FPU data register or a memory location. Source operands in memory can be in single-precision or double-precision floating-point format or in word or doubleword integer format.

The no-operand version of the instruction multiplies the contents of the ST(1) register by the contents of the ST(0) register and stores the product in the ST(1) register. The one-operand version multiplies the contents of the ST(0) register by the contents of a memory location (either a floating point or an integer value) and stores the product in the ST(0) register. The two-operand version, multiplies the contents of the ST(0) register by the contents of the ST(i) register, or vice versa, with the result being stored in the register specified with the first operand (the destination operand).

The FMULP instructions perform the additional operation of popping the FPU register stack after storing the product. To pop the register stack, the processor marks the ST(0) register as empty and increments the stack pointer (TOP) by 1. The no-operand version of the floating-point multiply instructions always results in the register stack being popped. In some assemblers, the mnemonic for this instruction is FMUL rather than FMULP.

The FIMUL instructions convert an integer source operand to double extended-precision floating-point format before performing the multiplication.

The sign of the result is always the exclusive-OR of the source signs, even if one or more of the values being multiplied is 0 or ∞. When the source operand is an integer 0, it is treated as a +0.

The following table shows the results obtained when multiplying various classes of numbers, assuming that neither overflow nor underflow occurs.

Table 3-29. FMUL/FMULP/FIMUL Results

		DEST						
		−∞	−F	−0	+0	+F	+∞	NaN
SRC	−∞	+∞	+∞	*	*	−∞	−∞	NaN
	−F	+∞	+F	+0	−0	−F	−∞	NaN
	−I	+∞	+F	+0	−0	−F	−∞	NaN
	−0	*	+0	+0	−0	−0	*	NaN
	+0	*	−0	−0	+0	+0	*	NaN
	+I	−∞	−F	−0	+0	+F	+∞	NaN
	+F	−∞	−F	−0	+0	+F	+∞	NaN
	+∞	−∞	−∞	*	*	+∞	+∞	NaN
	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN

NOTES:

F Means finite floating-point value.

I Means Integer.

* Indicates invalid-arithmetic-operand (#IA) exception.

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

Operation

```
IF Instruction = FIMUL
    THEN
        DEST ← DEST * ConvertToDoubleExtendedPrecisionFP(SRC);
    ELSE (* Source operand is floating-point value *)
        DEST ← DEST * SRC;
FI;

IF Instruction = FMULP
    THEN
        PopRegisterStack;
FI;
```

FPU Flags Affected

C1	Set to 0 if stack underflow occurred. Set if result was rounded up; cleared otherwise.
C0, C2, C3	Undefined.

Floating-Point Exceptions

#IS	Stack underflow occurred.
#IA	Operand is an SNaN value or unsupported format. One operand is ±0 and the other is ±∞.
#D	Source operand is a denormal value.
#U	Result is too small for destination format.
#O	Result is too large for destination format.
#P	Value cannot be represented exactly in destination format.

Protected Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit. If the DS, ES, FS, or GS register is used to access memory and it contains a NULL segment selector.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If the LOCK prefix is used.

Real-Address Mode Exceptions

#GP	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS	If a memory operand effective address is outside the SS segment limit.
#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#UD	If the LOCK prefix is used.

Virtual-8086 Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.
#UD	If the LOCK prefix is used.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form.
#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#MF	If there is a pending x87 FPU exception.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If the LOCK prefix is used.

FNOP—No Operation

Opcode	Instruction	64-Bit Mode	Compat/ Leg Mode	Description
D9 D0	FNOP	Valid	Valid	No operation is performed.

Description

Performs no FPU operation. This instruction takes up space in the instruction stream but does not affect the FPU or machine context, except the EIP register and the FPU Instruction Pointer.

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

FPU Flags Affected

C0, C1, C2, C3 undefined.

Floating-Point Exceptions

None

Protected Mode Exceptions

#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#MF	If there is a pending x87 FPU exception.
#UD	If the LOCK prefix is used.

Real-Address Mode Exceptions

Same exceptions as in protected mode.

Virtual-8086 Mode Exceptions

Same exceptions as in protected mode.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

Same exceptions as in protected mode.

FPATAN—Partial Arctangent

Opcode*	Instruction	64-Bit Mode	Compat/ Leg Mode	Description
D9 F3	FPATAN	Valid	Valid	Replace ST(1) with arctan(ST(1)/ST(0)) and pop the register stack.

NOTES:

* See IA-32 Architecture Compatibility section below.

Description

Computes the arctangent of the source operand in register ST(1) divided by the source operand in register ST(0), stores the result in ST(1), and pops the FPU register stack. The result in register ST(0) has the same sign as the source operand ST(1) and a magnitude less than +π.

The FPATAN instruction returns the angle between the X axis and the line from the origin to the point (X,Y), where Y (the ordinate) is ST(1) and X (the abscissa) is ST(0). The angle depends on the sign of X and Y independently, not just on the sign of the ratio Y/X. This is because a point (–X,Y) is in the second quadrant, resulting in an angle between π/2 and π, while a point (X,–Y) is in the fourth quadrant, resulting in an angle between 0 and –π/2. A point (–X,–Y) is in the third quadrant, giving an angle between –π/2 and –π.

The following table shows the results obtained when computing the arctangent of various classes of numbers, assuming that underflow does not occur.

Table 3-30. FPATAN Results

		ST(0)						
		– ∞	– F	– 0	+ 0	+ F	+ ∞	NaN
ST(1)	– ∞	– 3π/4*	– π/2	– π/2	– π/2	– π/2	– π/4*	NaN
	– F	-p	–π to –π/2	–π/2	–π/2	–π/2 to –0	- 0	NaN
	– 0	-p	-p	-p*	– 0*	– 0	– 0	NaN
	+ 0	+p	+ p	+ π*	+ 0*	+ 0	+ 0	NaN
	+ F	+p	+π to +π/2	+ π/2	+π/2	+π/2 to +0	+ 0	NaN
	+ ∞	+3π/4*	+π/2	+π/2	+π/2	+ π/2	+ π/4*	NaN
	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN

NOTES:

F Means finite floating-point value.

* Table 8-10 in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*, specifies that the ratios 0/0 and ∞/∞ generate the floating-point invalid arithmetic-operation exception and, if this exception is masked, the floating-point QNaN indefinite value is returned. With the FPATAN instruction, the 0/0 or ∞/∞ value is actually not calculated using division. Instead, the arctangent of the two variables is derived from a standard mathematical formulation that is generalized to allow complex numbers as arguments. In this complex variable formulation, arctangent(0,0) etc. has well defined values. These values are needed to develop a library to compute transcendental functions with complex arguments, based on the FPU functions that only allow floating-point values as arguments.

There is no restriction on the range of source operands that FPATAN can accept.

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

IA-32 Architecture Compatibility

The source operands for this instruction are restricted for the 80287 math coprocessor to the following range:

$0 \le |ST(1)| < |ST(0)| < +\infty$

Operation

```
ST(1) ← arctan(ST(1) / ST(0));
PopRegisterStack;
```

FPU Flags Affected

C1	Set to 0 if stack underflow occurred. Set if result was rounded up; cleared otherwise.
C0, C2, C3	Undefined.

Floating-Point Exceptions

#IS	Stack underflow occurred.
#IA	Source operand is an SNaN value or unsupported format.
#D	Source operand is a denormal value.
#U	Result is too small for destination format.
#P	Value cannot be represented exactly in destination format.

Protected Mode Exceptions

#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#MF	If there is a pending x87 FPU exception.
#UD	If the LOCK prefix is used.

Real-Address Mode Exceptions

Same exceptions as in protected mode.

Virtual-8086 Mode Exceptions

Same exceptions as in protected mode.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

Same exceptions as in protected mode.

FPREM—Partial Remainder

Opcode	Instruction	64-Bit Mode	Compat/ Leg Mode	Description
D9 F8	FPREM	Valid	Valid	Replace ST(0) with the remainder obtained from dividing ST(0) by ST(1).

Description

Computes the remainder obtained from dividing the value in the ST(0) register (the dividend) by the value in the ST(1) register (the divisor or **modulus**), and stores the result in ST(0). The remainder represents the following value:

Remainder ← ST(0) – (Q ∗ ST(1))

Here, Q is an integer value that is obtained by truncating the floating-point number quotient of [ST(0) / ST(1)] toward zero. The sign of the remainder is the same as the sign of the dividend. The magnitude of the remainder is less than that of the modulus, unless a partial remainder was computed (as described below).

This instruction produces an exact result; the inexact-result exception does not occur and the rounding control has no effect. The following table shows the results obtained when computing the remainder of various classes of numbers, assuming that underflow does not occur.

Table 3-31. FPREM Results

		ST(1)						
		-∞	-F	-0	+0	+F	+∞	NaN
ST(0)	-∞	*	*	*	*	*	*	NaN
	-F	ST(0)	-F or -0	**	**	-F or -0	ST(0)	NaN
	-0	-0	-0	*	*	-0	-0	NaN
	+0	+0	+0	*	*	+0	+0	NaN
	+F	ST(0)	+F or +0	**	**	+F or +0	ST(0)	NaN
	+∞	*	*	*	*	*	*	NaN
	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN

NOTES:

F Means finite floating-point value.

* Indicates floating-point invalid-arithmetic-operand (#IA) exception.

** Indicates floating-point zero-divide (#Z) exception.

When the result is 0, its sign is the same as that of the dividend. When the modulus is ∞, the result is equal to the value in ST(0).

The FPREM instruction does not compute the remainder specified in IEEE Std 754. The IEEE specified remainder can be computed with the FPREM1 instruction. The FPREM instruction is provided for compatibility with the Intel 8087 and Intel287 math coprocessors.

The FPREM instruction gets its name "partial remainder" because of the way it computes the remainder. This instruction arrives at a remainder through iterative subtraction. It can, however, reduce the exponent of ST(0) by no more than 63 in one execution of the instruction. If the instruction succeeds in producing a remainder that is less than the modulus, the operation is complete and the C2 flag in the FPU status word is cleared. Otherwise, C2 is set, and the result in ST(0) is called the **partial remainder**. The exponent of the partial remainder will be less than the exponent of the original dividend by at least 32. Software can re-execute the instruction (using the partial remainder in ST(0) as the dividend) until C2 is cleared. (Note that while executing such a remainder-computation loop, a higher-priority interrupting routine that needs the FPU can force a context switch in-between the instructions in the loop.)

An important use of the FPREM instruction is to reduce the arguments of periodic functions. When reduction is complete, the instruction stores the three least-significant bits of the quotient in the C3, C1, and C0 flags of the FPU

status word. This information is important in argument reduction for the tangent function (using a modulus of π/4), because it locates the original angle in the correct one of eight sectors of the unit circle.

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

Operation

```
D ← exponent(ST(0)) – exponent(ST(1));

IF D < 64
    THEN
        Q ← Integer(TruncateTowardZero(ST(0) / ST(1)));
        ST(0) ← ST(0) – (ST(1) * Q);
        C2 ← 0;
        C0, C3, C1 ← LeastSignificantBits(Q); (* Q2, Q1, Q0 *)
    ELSE
        C2 ← 1;
        N ← An implementation-dependent number between 32 and 63;
        QQ ← Integer(TruncateTowardZero((ST(0) / ST(1)) / 2^(D – N)));
        ST(0) ← ST(0) – (ST(1) * QQ * 2^(D – N));
FI;
```

FPU Flags Affected

C0	Set to bit 2 (Q2) of the quotient.
C1	Set to 0 if stack underflow occurred; otherwise, set to least significant bit of quotient (Q0).
C2	Set to 0 if reduction complete; set to 1 if incomplete.
C3	Set to bit 1 (Q1) of the quotient.

Floating-Point Exceptions

#IS	Stack underflow occurred.
#IA	Source operand is an SNaN value, modulus is 0, dividend is ∞, or unsupported format.
#D	Source operand is a denormal value.
#U	Result is too small for destination format.

Protected Mode Exceptions

#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#MF	If there is a pending x87 FPU exception.
#UD	If the LOCK prefix is used.

Real-Address Mode Exceptions

Same exceptions as in protected mode.

Virtual-8086 Mode Exceptions

Same exceptions as in protected mode.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

Same exceptions as in protected mode.

FPREM1—Partial Remainder

Opcode	Instruction	64-Bit Mode	Compat/ Leg Mode	Description
D9 F5	FPREM1	Valid	Valid	Replace ST(0) with the IEEE remainder obtained from dividing ST(0) by ST(1).

Description

Computes the IEEE remainder obtained from dividing the value in the ST(0) register (the dividend) by the value in the ST(1) register (the divisor or **modulus**), and stores the result in ST(0). The remainder represents the following value:

Remainder ← ST(0) – (Q ∗ ST(1))

Here, Q is an integer value that is obtained by rounding the floating-point number quotient of [ST(0) / ST(1)] toward the nearest integer value. The magnitude of the remainder is less than or equal to half the magnitude of the modulus, unless a partial remainder was computed (as described below).

This instruction produces an exact result; the precision (inexact) exception does not occur and the rounding control has no effect. The following table shows the results obtained when computing the remainder of various classes of numbers, assuming that underflow does not occur.

Table 3-32. FPREM1 Results

		ST(1)						
		– ∞	– F	– 0	+ 0	+ F	+ ∞	NaN
ST(0)	– ∞	*	*	*	*	*	*	NaN
	– F	ST(0)	±F or –0	**	**	± F or – 0	ST(0)	NaN
	– 0	– 0	– 0	*	*	– 0	-0	NaN
	+ 0	+ 0	+ 0	*	*	+ 0	+0	NaN
	+ F	ST(0)	± F or + 0	**	**	± F or + 0	ST(0)	NaN
	+ ∞	*	*	*	*	*	*	NaN
	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN

NOTES:

F Means finite floating-point value.

* Indicates floating-point invalid-arithmetic-operand (#IA) exception.

** Indicates floating-point zero-divide (#Z) exception.

When the result is 0, its sign is the same as that of the dividend. When the modulus is ∞, the result is equal to the value in ST(0).

The FPREM1 instruction computes the remainder specified in IEEE Standard 754. This instruction operates differently from the FPREM instruction in the way that it rounds the quotient of ST(0) divided by ST(1) to an integer (see the "Operation" section below).

Like the FPREM instruction, FPREM1 computes the remainder through iterative subtraction, but can reduce the exponent of ST(0) by no more than 63 in one execution of the instruction. If the instruction succeeds in producing a remainder that is less than one half the modulus, the operation is complete and the C2 flag in the FPU status word is cleared. Otherwise, C2 is set, and the result in ST(0) is called the **partial remainder**. The exponent of the partial remainder will be less than the exponent of the original dividend by at least 32. Software can re-execute the instruction (using the partial remainder in ST(0) as the dividend) until C2 is cleared. (Note that while executing such a remainder-computation loop, a higher-priority interrupting routine that needs the FPU can force a context switch in-between the instructions in the loop.)

An important use of the FPREM1 instruction is to reduce the arguments of periodic functions. When reduction is complete, the instruction stores the three least-significant bits of the quotient in the C3, C1, and C0 flags of the FPU

status word. This information is important in argument reduction for the tangent function (using a modulus of π/4), because it locates the original angle in the correct one of eight sectors of the unit circle.

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

Operation

```
D ← exponent(ST(0)) – exponent(ST(1));

IF D < 64
    THEN
        Q ← Integer(RoundTowardNearestInteger(ST(0) / ST(1)));
        ST(0) ← ST(0) – (ST(1) * Q);
        C2 ← 0;
        C0, C3, C1 ← LeastSignificantBits(Q); (* Q2, Q1, Q0 *)
    ELSE
        C2 ← 1;
        N ← An implementation-dependent number between 32 and 63;
        QQ ← Integer(TruncateTowardZero((ST(0) / ST(1)) / 2^(D – N)));
        ST(0) ← ST(0) – (ST(1) * QQ * 2^(D – N));
FI;
```

FPU Flags Affected

C0	Set to bit 2 (Q2) of the quotient.
C1	Set to 0 if stack underflow occurred; otherwise, set to least significant bit of quotient (Q0).
C2	Set to 0 if reduction complete; set to 1 if incomplete.
C3	Set to bit 1 (Q1) of the quotient.

Floating-Point Exceptions

#IS	Stack underflow occurred.
#IA	Source operand is an SNaN value, modulus (divisor) is 0, dividend is ∞, or unsupported format.
#D	Source operand is a denormal value.
#U	Result is too small for destination format.

Protected Mode Exceptions

#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#MF	If there is a pending x87 FPU exception.
#UD	If the LOCK prefix is used.

Real-Address Mode Exceptions

Same exceptions as in protected mode.

Virtual-8086 Mode Exceptions

Same exceptions as in protected mode.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

Same exceptions as in protected mode.

FPTAN—Partial Tangent

Opcode	Instruction	64-Bit Mode	Compat/ Leg Mode	Description
D9 F2	FPTAN	Valid	Valid	Replace ST(0) with its approximate tangent and push 1 onto the FPU stack.

Description

Computes the approximate tangent of the source operand in register ST(0), stores the result in ST(0), and pushes a 1.0 onto the FPU register stack. The source operand must be given in radians and must be less than $\pm 2^{63}$. The following table shows the unmasked results obtained when computing the partial tangent of various classes of numbers, assuming that underflow does not occur.

Table 3-33. FPTAN Results

ST(0) SRC	ST(0) DEST
$-\infty$	*
−F	−F to +F
−0	-0
+0	+0
+F	−F to +F
$+\infty$	*
NaN	NaN

NOTES:

F Means finite floating-point value.

* Indicates floating-point invalid-arithmetic-operand (#IA) exception.

If the source operand is outside the acceptable range, the C2 flag in the FPU status word is set, and the value in register ST(0) remains unchanged. The instruction does not raise an exception when the source operand is out of range. It is up to the program to check the C2 flag for out-of-range conditions. Source values outside the range -2^{63} to $+2^{63}$ can be reduced to the range of the instruction by subtracting an appropriate integer multiple of 2π. However, even within the range -2^{63} to $+2^{63}$, inaccurate results can occur because the finite approximation of π used internally for argument reduction is not sufficient in all cases. Therefore, for accurate results it is safe to apply FPTAN only to arguments reduced accurately in software, to a value smaller in absolute value than $3\pi/8$. See the sections titled "Approximation of Pi" and "Transcendental Instruction Accuracy" in Chapter 8 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*, for a discussion of the proper value to use for π in performing such reductions.

The value 1.0 is pushed onto the register stack after the tangent has been computed to maintain compatibility with the Intel 8087 and Intel287 math coprocessors. This operation also simplifies the calculation of other trigonometric functions. For instance, the cotangent (which is the reciprocal of the tangent) can be computed by executing a FDIVR instruction after the FPTAN instruction.

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

Operation

```
IF ST(0) < 2^63
    THEN
        C2 ← 0;
        ST(0) ← fptan(ST(0)); // approximation of tan
        TOP ← TOP – 1;
        ST(0) ← 1.0;
    ELSE (* Source operand is out-of-range *)
        C2 ← 1;
FI;
```

FPU Flags Affected

C1	Set to 0 if stack underflow occurred; set to 1 if stack overflow occurred. Set if result was rounded up; cleared otherwise.
C2	Set to 1 if outside range (-2^{63} < source operand < $+2^{63}$); otherwise, set to 0.
C0, C3	Undefined.

Floating-Point Exceptions

#IS	Stack underflow or overflow occurred.
#IA	Source operand is an SNaN value, ∞, or unsupported format.
#D	Source operand is a denormal value.
#U	Result is too small for destination format.
#P	Value cannot be represented exactly in destination format.

Protected Mode Exceptions

#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#MF	If there is a pending x87 FPU exception.
#UD	If the LOCK prefix is used.

Real-Address Mode Exceptions

Same exceptions as in protected mode.

Virtual-8086 Mode Exceptions

Same exceptions as in protected mode.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

Same exceptions as in protected mode.

FRNDINT—Round to Integer

Opcode	Instruction	64-Bit Mode	Compat/ Leg Mode	Description
D9 FC	FRNDINT	Valid	Valid	Round ST(0) to an integer.

Description

Rounds the source value in the ST(0) register to the nearest integral value, depending on the current rounding mode (setting of the RC field of the FPU control word), and stores the result in ST(0).

If the source value is ∞, the value is not changed. If the source value is not an integral value, the floating-point inexact-result exception (#P) is generated.

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

Operation

```
ST(0) ← RoundToIntegralValue(ST(0));
```

FPU Flags Affected

C1	Set to 0 if stack underflow occurred.
	Set if result was rounded up; cleared otherwise.
C0, C2, C3	Undefined.

Floating-Point Exceptions

#IS	Stack underflow occurred.
#IA	Source operand is an SNaN value or unsupported format.
#D	Source operand is a denormal value.
#P	Source operand is not an integral value.

Protected Mode Exceptions

#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#MF	If there is a pending x87 FPU exception.
#UD	If the LOCK prefix is used.

Real-Address Mode Exceptions

Same exceptions as in protected mode.

Virtual-8086 Mode Exceptions

Same exceptions as in protected mode.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

Same exceptions as in protected mode.

FRSTOR—Restore x87 FPU State

Opcode	Instruction	64-Bit Mode	Compat/ Leg Mode	Description
DD /4	FRSTOR *m94/108byte*	Valid	Valid	Load FPU state from *m94byte* or *m108byte*.

Description

Loads the FPU state (operating environment and register stack) from the memory area specified with the source operand. This state data is typically written to the specified memory location by a previous FSAVE/FNSAVE instruction.

The FPU operating environment consists of the FPU control word, status word, tag word, instruction pointer, data pointer, and last opcode. Figures 8-9 through 8-12 in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*, show the layout in memory of the stored environment, depending on the operating mode of the processor (protected or real) and the current operand-size attribute (16-bit or 32-bit). In virtual-8086 mode, the real mode layouts are used. The contents of the FPU register stack are stored in the 80 bytes immediately following the operating environment image.

The FRSTOR instruction should be executed in the same operating mode as the corresponding FSAVE/FNSAVE instruction.

If one or more unmasked exception bits are set in the new FPU status word, a floating-point exception will be generated. To avoid raising exceptions when loading a new operating environment, clear all the exception flags in the FPU status word that is being loaded.

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

Operation

```
FPUControlWord ← SRC[FPUControlWord];
FPUStatusWord ← SRC[FPUStatusWord];
FPUTagWord ← SRC[FPUTagWord];
FPUDataPointer ← SRC[FPUDataPointer];
FPUInstructionPointer ← SRC[FPUInstructionPointer];
FPULastInstructionOpcode ← SRC[FPULastInstructionOpcode];

ST(0) ← SRC[ST(0)];
ST(1) ← SRC[ST(1)];
ST(2) ← SRC[ST(2)];
ST(3) ← SRC[ST(3)];
ST(4) ← SRC[ST(4)];
ST(5) ← SRC[ST(5)];
ST(6) ← SRC[ST(6)];
ST(7) ← SRC[ST(7)];
```

FPU Flags Affected

The C0, C1, C2, C3 flags are loaded.

Floating-Point Exceptions

None; however, this operation might unmask an existing exception that has been detected but not generated, because it was masked. Here, the exception is generated at the completion of the instruction.

Protected Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit. If the DS, ES, FS, or GS register is used to access memory and it contains a NULL segment selector.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If the LOCK prefix is used.

Real-Address Mode Exceptions

#GP	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS	If a memory operand effective address is outside the SS segment limit.
#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#UD	If the LOCK prefix is used.

Virtual-8086 Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.
#UD	If the LOCK prefix is used.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form.
#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If the LOCK prefix is used.

FSAVE/FNSAVE—Store x87 FPU State

Opcode	Instruction	64-Bit Mode	Compat/ Leg Mode	Description
9B DD /6	FSAVE *m94/108byte*	Valid	Valid	Store FPU state to *m94byte* or *m108byte* after checking for pending unmasked floating-point exceptions. Then re-initialize the FPU.
DD /6	FNSAVE* *m94/108byte*	Valid	Valid	Store FPU environment to *m94byte* or *m108byte* without checking for pending unmasked floating-point exceptions. Then re-initialize the FPU.

NOTES:

* See IA-32 Architecture Compatibility section below.

Description

Stores the current FPU state (operating environment and register stack) at the specified destination in memory, and then re-initializes the FPU. The FSAVE instruction checks for and handles pending unmasked floating-point exceptions before storing the FPU state; the FNSAVE instruction does not.

The FPU operating environment consists of the FPU control word, status word, tag word, instruction pointer, data pointer, and last opcode. Figures 8-9 through 8-12 in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*, show the layout in memory of the stored environment, depending on the operating mode of the processor (protected or real) and the current operand-size attribute (16-bit or 32-bit). In virtual-8086 mode, the real mode layouts are used. The contents of the FPU register stack are stored in the 80 bytes immediately follow the operating environment image.

The saved image reflects the state of the FPU after all floating-point instructions preceding the FSAVE/FNSAVE instruction in the instruction stream have been executed.

After the FPU state has been saved, the FPU is reset to the same default values it is set to with the FINIT/FNINIT instructions (see "FINIT/FNINIT—Initialize Floating-Point Unit" in this chapter).

The FSAVE/FNSAVE instructions are typically used when the operating system needs to perform a context switch, an exception handler needs to use the FPU, or an application program needs to pass a "clean" FPU to a procedure.

The assembler issues two instructions for the FSAVE instruction (an FWAIT instruction followed by an FNSAVE instruction), and the processor executes each of these instructions separately. If an exception is generated for either of these instructions, the save EIP points to the instruction that caused the exception.

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

IA-32 Architecture Compatibility

For Intel math coprocessors and FPUs prior to the Intel Pentium processor, an FWAIT instruction should be executed before attempting to read from the memory image stored with a prior FSAVE/FNSAVE instruction. This FWAIT instruction helps ensure that the storage operation has been completed.

When operating a Pentium or Intel486 processor in MS-DOS compatibility mode, it is possible (under unusual circumstances) for an FNSAVE instruction to be interrupted prior to being executed to handle a pending FPU exception. See the section titled "No-Wait FPU Instructions Can Get FPU Interrupt in Window" in Appendix D of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*, for a description of these circumstances. An FNSAVE instruction cannot be interrupted in this way on later Intel processors, except for the Intel Quark™ X1000 processor.

Operation

```
(* Save FPU State and Registers *)

DEST[FPUControlWord] ← FPUControlWord;
DEST[FPUStatusWord] ← FPUStatusWord;
DEST[FPUTagWord] ← FPUTagWord;
DEST[FPUDataPointer] ← FPUDataPointer;
DEST[FPUInstructionPointer] ← FPUInstructionPointer;
DEST[FPULastInstructionOpcode] ← FPULastInstructionOpcode;

DEST[ST(0)] ← ST(0);
DEST[ST(1)] ← ST(1);
DEST[ST(2)] ← ST(2);
DEST[ST(3)] ← ST(3);
DEST[ST(4)]← ST(4);
DEST[ST(5)] ← ST(5);
DEST[ST(6)] ← ST(6);
DEST[ST(7)] ← ST(7);

(* Initialize FPU *)

FPUControlWord ← 037FH;
FPUStatusWord ← 0;
FPUTagWord ← FFFFH;
FPUDataPointer ← 0;
FPUInstructionPointer ← 0;
FPULastInstructionOpcode ← 0;
```

FPU Flags Affected

The C0, C1, C2, and C3 flags are saved and then cleared.

Floating-Point Exceptions

None.

Protected Mode Exceptions

#GP(0)	If destination is located in a non-writable segment.
	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
	If the DS, ES, FS, or GS register is used to access memory and it contains a NULL segment selector.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If the LOCK prefix is used.

Real-Address Mode Exceptions

#GP	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS	If a memory operand effective address is outside the SS segment limit.
#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#UD	If the LOCK prefix is used.

Virtual-8086 Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.
#UD	If the LOCK prefix is used.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form.
#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#MF	If there is a pending x87 FPU exception.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

FSCALE—Scale

Opcode	Instruction	64-Bit Mode	Compat/ Leg Mode	Description
D9 FD	FSCALE	Valid	Valid	Scale ST(0) by ST(1).

Description

Truncates the value in the source operand (toward 0) to an integral value and adds that value to the exponent of the destination operand. The destination and source operands are floating-point values located in registers ST(0) and ST(1), respectively. This instruction provides rapid multiplication or division by integral powers of 2. The following table shows the results obtained when scaling various classes of numbers, assuming that neither overflow nor underflow occurs.

Table 3-34. FSCALE Results

	ST(1)							
		−∞	−F	−0	+0	+F	+∞	NaN
ST(0)	−∞	NaN	−∞	−∞	−∞	−∞	−∞	NaN
	−F	−0	−F	−F	−F	−F	−∞	NaN
	−0	−0	−0	−0	−0	−0	NaN	NaN
	+0	+0	+0	+0	+0	+0	NaN	NaN
	+F	+0	+F	+F	+F	+F	+∞	NaN
	+∞	NaN	+∞	+∞	+∞	+∞	+∞	NaN
	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN

NOTES:

F Means finite floating-point value.

In most cases, only the exponent is changed and the mantissa (significand) remains unchanged. However, when the value being scaled in ST(0) is a denormal value, the mantissa is also changed and the result may turn out to be a normalized number. Similarly, if overflow or underflow results from a scale operation, the resulting mantissa will differ from the source's mantissa.

The FSCALE instruction can also be used to reverse the action of the FXTRACT instruction, as shown in the following example:

```
FXTRACT;
FSCALE;
FSTP ST(1);
```

In this example, the FXTRACT instruction extracts the significand and exponent from the value in ST(0) and stores them in ST(0) and ST(1) respectively. The FSCALE then scales the significand in ST(0) by the exponent in ST(1), recreating the original value before the FXTRACT operation was performed. The FSTP ST(1) instruction overwrites the exponent (extracted by the FXTRACT instruction) with the recreated value, which returns the stack to its original state with only one register [ST(0)] occupied.

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

Operation

$ST(0) \leftarrow ST(0) * 2^{RoundTowardZero(ST(1))};$

FPU Flags Affected

C1	Set to 0 if stack underflow occurred.
	Set if result was rounded up; cleared otherwise.
C0, C2, C3	Undefined.

Floating-Point Exceptions

#IS	Stack underflow occurred.
#IA	Source operand is an SNaN value or unsupported format.
#D	Source operand is a denormal value.
#U	Result is too small for destination format.
#O	Result is too large for destination format.
#P	Value cannot be represented exactly in destination format.

Protected Mode Exceptions

#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#MF	If there is a pending x87 FPU exception.
#UD	If the LOCK prefix is used.

Real-Address Mode Exceptions

Same exceptions as in protected mode.

Virtual-8086 Mode Exceptions

Same exceptions as in protected mode.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

Same exceptions as in protected mode.

FSIN—Sine

Opcode	Instruction	64-Bit Mode	Compat/ Leg Mode	Description
D9 FE	FSIN	Valid	Valid	Replace ST(0) with the approximate of its sine.

Description

Computes an approximation of the sine of the source operand in register ST(0) and stores the result in ST(0). The source operand must be given in radians and must be within the range -2^{63} to $+2^{63}$. The following table shows the results obtained when taking the sine of various classes of numbers, assuming that underflow does not occur.

Table 3-35. FSIN Results

SRC (ST(0))	DEST (ST(0))
− ∞	*
− F	− 1 to + 1
− 0	−0
+ 0	+ 0
+ F	− 1 to +1
+ ∞	*
NaN	NaN

NOTES:

F Means finite floating-point value.

* Indicates floating-point invalid-arithmetic-operand (#IA) exception.

If the source operand is outside the acceptable range, the C2 flag in the FPU status word is set, and the value in register ST(0) remains unchanged. The instruction does not raise an exception when the source operand is out of range. It is up to the program to check the C2 flag for out-of-range conditions. Source values outside the range -2^{63} to $+2^{63}$ can be reduced to the range of the instruction by subtracting an appropriate integer multiple of 2π. However, even within the range -2^{63} to $+2^{63}$, inaccurate results can occur because the finite approximation of π used internally for argument reduction is not sufficient in all cases. Therefore, for accurate results it is safe to apply FSIN only to arguments reduced accurately in software, to a value smaller in absolute value than $3\pi/4$. See the sections titled "Approximation of Pi" and "Transcendental Instruction Accuracy" in Chapter 8 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*, for a discussion of the proper value to use for π in performing such reductions.

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

Operation

```
IF -2^63 < ST(0) < 2^63
    THEN
        C2 ← 0;
        ST(0) ← fsin(ST(0)); // approximation of the mathematical sin function
    ELSE (* Source operand out of range *)
        C2 ← 1;
FI;
```

FPU Flags Affected

C1	Set to 0 if stack underflow occurred. Set if result was rounded up; cleared otherwise.
C2	Set to 1 if outside range (-2^{63} < source operand < $+2^{63}$); otherwise, set to 0.
C0, C3	Undefined.

Floating-Point Exceptions

#IS	Stack underflow occurred.
#IA	Source operand is an SNaN value, ∞, or unsupported format.
#D	Source operand is a denormal value.
#P	Value cannot be represented exactly in destination format.

Protected Mode Exceptions

#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#MF	If there is a pending x87 FPU exception.
#UD	If the LOCK prefix is used.

Real-Address Mode Exceptions

Same exceptions as in protected mode.

Virtual-8086 Mode Exceptions

Same exceptions as in protected mode.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

Same exceptions as in protected mode.

FSINCOS—Sine and Cosine

Opcode	Instruction	64-Bit Mode	Compat/ Leg Mode	Description
D9 FB	FSINCOS	Valid	Valid	Compute the sine and cosine of ST(0); replace ST(0) with the approximate sine, and push the approximate cosine onto the register stack.

Description

Computes both the approximate sine and the cosine of the source operand in register ST(0), stores the sine in ST(0), and pushes the cosine onto the top of the FPU register stack. (This instruction is faster than executing the FSIN and FCOS instructions in succession.)

The source operand must be given in radians and must be within the range -2^{63} to $+2^{63}$. The following table shows the results obtained when taking the sine and cosine of various classes of numbers, assuming that underflow does not occur.

Table 3-36. FSINCOS Results

SRC	DEST	
ST(0)	ST(1) Cosine	ST(0) Sine
$-\infty$	*	*
−F	−1 to +1	−1 to +1
−0	+1	−0
+0	+1	+0
+F	−1 to +1	−1 to +1
$+\infty$	*	*
NaN	NaN	NaN

NOTES:

F Means finite floating-point value.

* Indicates floating-point invalid-arithmetic-operand (#IA) exception.

If the source operand is outside the acceptable range, the C2 flag in the FPU status word is set, and the value in register ST(0) remains unchanged. The instruction does not raise an exception when the source operand is out of range. It is up to the program to check the C2 flag for out-of-range conditions. Source values outside the range -2^{63} to $+2^{63}$ can be reduced to the range of the instruction by subtracting an appropriate integer multiple of 2π. However, even within the range -2^{63} to $+2^{63}$, inaccurate results can occur because the finite approximation of π used internally for argument reduction is not sufficient in all cases. Therefore, for accurate results it is safe to apply FSINCOS only to arguments reduced accurately in software, to a value smaller in absolute value than $3\pi/8$. See the sections titled "Approximation of Pi" and "Transcendental Instruction Accuracy" in Chapter 8 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*, for a discussion of the proper value to use for π in performing such reductions.

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

Operation

```
IF ST(0) < 2^63
    THEN
        C2 ← 0;
        TEMP ← fcos(ST(0)); // approximation of cosine
        ST(0) ← fsin(ST(0)); // approximation of sine
        TOP ← TOP – 1;
        ST(0) ← TEMP;
    ELSE (* Source operand out of range *)
        C2 ← 1;
FI;
```

FPU Flags Affected

C1	Set to 0 if stack underflow occurred; set to 1 of stack overflow occurs. Set if result was rounded up; cleared otherwise.
C2	Set to 1 if outside range (-2^{63} < source operand < $+2^{63}$); otherwise, set to 0.
C0, C3	Undefined.

Floating-Point Exceptions

#IS	Stack underflow or overflow occurred.
#IA	Source operand is an SNaN value, ∞, or unsupported format.
#D	Source operand is a denormal value.
#U	Result is too small for destination format.
#P	Value cannot be represented exactly in destination format.

Protected Mode Exceptions

#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#MF	If there is a pending x87 FPU exception.
#UD	If the LOCK prefix is used.

Real-Address Mode Exceptions

Same exceptions as in protected mode.

Virtual-8086 Mode Exceptions

Same exceptions as in protected mode.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

Same exceptions as in protected mode.

FSQRT—Square Root

Opcode	Instruction	64-Bit Mode	Compat/ Leg Mode	Description
D9 FA	FSQRT	Valid	Valid	Computes square root of ST(0) and stores the result in ST(0).

Description

Computes the square root of the source value in the ST(0) register and stores the result in ST(0).

The following table shows the results obtained when taking the square root of various classes of numbers, assuming that neither overflow nor underflow occurs.

Table 3-37. FSQRT Results

SRC (ST(0))	DEST (ST(0))
– ∞	*
– F	*
– 0	– 0
+ 0	+ 0
+ F	+ F
+ ∞	+ ∞
NaN	NaN

NOTES:

F Means finite floating-point value.

* Indicates floating-point invalid-arithmetic-operand (#IA) exception.

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

Operation

ST(0) ← SquareRoot(ST(0));

FPU Flags Affected

C1	Set to 0 if stack underflow occurred.
	Set if result was rounded up; cleared otherwise.
C0, C2, C3	Undefined.

Floating-Point Exceptions

#IS	Stack underflow occurred.
#IA	Source operand is an SNaN value or unsupported format.
	Source operand is a negative value (except for –0).
#D	Source operand is a denormal value.
#P	Value cannot be represented exactly in destination format.

Protected Mode Exceptions

#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#MF	If there is a pending x87 FPU exception.
#UD	If the LOCK prefix is used.

Real-Address Mode Exceptions

Same exceptions as in protected mode.

Virtual-8086 Mode Exceptions

Same exceptions as in protected mode.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

Same exceptions as in protected mode.

FST/FSTP—Store Floating Point Value

Opcode	Instruction	64-Bit Mode	Compat/ Leg Mode	Description
D9 /2	FST *m32fp*	Valid	Valid	Copy ST(0) to *m32fp*.
DD /2	FST *m64fp*	Valid	Valid	Copy ST(0) to *m64fp*.
DD D0+i	FST ST(i)	Valid	Valid	Copy ST(0) to ST(i).
D9 /3	FSTP *m32fp*	Valid	Valid	Copy ST(0) to *m32fp* and pop register stack.
DD /3	FSTP *m64fp*	Valid	Valid	Copy ST(0) to *m64fp* and pop register stack.
DB /7	FSTP *m80fp*	Valid	Valid	Copy ST(0) to *m80fp* and pop register stack.
DD D8+i	FSTP ST(i)	Valid	Valid	Copy ST(0) to ST(i) and pop register stack.

Description

The FST instruction copies the value in the ST(0) register to the destination operand, which can be a memory location or another register in the FPU register stack. When storing the value in memory, the value is converted to single-precision or double-precision floating-point format.

The FSTP instruction performs the same operation as the FST instruction and then pops the register stack. To pop the register stack, the processor marks the ST(0) register as empty and increments the stack pointer (TOP) by 1. The FSTP instruction can also store values in memory in double extended-precision floating-point format.

If the destination operand is a memory location, the operand specifies the address where the first byte of the destination value is to be stored. If the destination operand is a register, the operand specifies a register in the register stack relative to the top of the stack.

If the destination size is single-precision or double-precision, the significand of the value being stored is rounded to the width of the destination (according to the rounding mode specified by the RC field of the FPU control word), and the exponent is converted to the width and bias of the destination format. If the value being stored is too large for the destination format, a numeric overflow exception (#O) is generated and, if the exception is unmasked, no value is stored in the destination operand. If the value being stored is a denormal value, the denormal exception (#D) is not generated. This condition is simply signaled as a numeric underflow exception (#U) condition.

If the value being stored is ±0, ±∞, or a NaN, the least-significant bits of the significand and the exponent are truncated to fit the destination format. This operation preserves the value's identity as a 0, ∞, or NaN.

If the destination operand is a non-empty register, the invalid-operation exception is not generated.

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

Operation

```
DEST ← ST(0);

IF Instruction = FSTP
    THEN
        PopRegisterStack;
FI;
```

FPU Flags Affected

C1	Set to 0 if stack underflow occurred. Indicates rounding direction of if the floating-point inexact exception (#P) is generated: 0 ← not roundup; 1 ← roundup.
C0, C2, C3	Undefined.

Floating-Point Exceptions

#IS	Stack underflow occurred.
#IA	If destination result is an SNaN value or unsupported format, except when the destination format is in double extended-precision floating-point format.
#U	Result is too small for the destination format.
#O	Result is too large for the destination format.
#P	Value cannot be represented exactly in destination format.

Protected Mode Exceptions

#GP(0)	If the destination is located in a non-writable segment. If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit. If the DS, ES, FS, or GS register is used to access memory and it contains a NULL segment selector.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If the LOCK prefix is used.

Real-Address Mode Exceptions

#GP	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS	If a memory operand effective address is outside the SS segment limit.
#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#UD	If the LOCK prefix is used.

Virtual-8086 Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.
#UD	If the LOCK prefix is used.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form.
#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#MF	If there is a pending x87 FPU exception.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If the LOCK prefix is used.

FSTCW/FNSTCW—Store x87 FPU Control Word

Opcode	Instruction	64-Bit Mode	Compat/ Leg Mode	Description
9B D9 /7	FSTCW *m2byte*	Valid	Valid	Store FPU control word to *m2byte* after checking for pending unmasked floating-point exceptions.
D9 /7	FNSTCW* *m2byte*	Valid	Valid	Store FPU control word to *m2byte* without checking for pending unmasked floating-point exceptions.

NOTES:

* See IA-32 Architecture Compatibility section below.

Description

Stores the current value of the FPU control word at the specified destination in memory. The FSTCW instruction checks for and handles pending unmasked floating-point exceptions before storing the control word; the FNSTCW instruction does not.

The assembler issues two instructions for the FSTCW instruction (an FWAIT instruction followed by an FNSTCW instruction), and the processor executes each of these instructions in separately. If an exception is generated for either of these instructions, the save EIP points to the instruction that caused the exception.

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

IA-32 Architecture Compatibility

When operating a Pentium or Intel486 processor in MS-DOS compatibility mode, it is possible (under unusual circumstances) for an FNSTCW instruction to be interrupted prior to being executed to handle a pending FPU exception. See the section titled "No-Wait FPU Instructions Can Get FPU Interrupt in Window" in Appendix D of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*, for a description of these circumstances. An FNSTCW instruction cannot be interrupted in this way on later Intel processors, except for the Intel Quark™ X1000 processor.

Operation

```
DEST ← FPUControlWord;
```

FPU Flags Affected

The C0, C1, C2, and C3 flags are undefined.

Floating-Point Exceptions

None.

Protected Mode Exceptions

#GP(0)	If the destination is located in a non-writable segment.
	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
	If the DS, ES, FS, or GS register is used to access memory and it contains a NULL segment selector.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If the LOCK prefix is used.

Real-Address Mode Exceptions

#GP	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS	If a memory operand effective address is outside the SS segment limit.
#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#UD	If the LOCK prefix is used.

Virtual-8086 Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.
#UD	If the LOCK prefix is used.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form.
#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#MF	If there is a pending x87 FPU exception.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If the LOCK prefix is used.

FSTENV/FNSTENV—Store x87 FPU Environment

Opcode	Instruction	64-Bit Mode	Compat/ Leg Mode	Description
9B D9 /6	FSTENV *m14/28byte*	Valid	Valid	Store FPU environment to *m14byte* or *m28byte* after checking for pending unmasked floating-point exceptions. Then mask all floating-point exceptions.
D9 /6	FNSTENV* *m14/28byte*	Valid	Valid	Store FPU environment to *m14byte* or *m28byte* without checking for pending unmasked floating-point exceptions. Then mask all floating-point exceptions.

NOTES:

* See IA-32 Architecture Compatibility section below.

Description

Saves the current FPU operating environment at the memory location specified with the destination operand, and then masks all floating-point exceptions. The FPU operating environment consists of the FPU control word, status word, tag word, instruction pointer, data pointer, and last opcode. Figures 8-9 through 8-12 in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*, show the layout in memory of the stored environment, depending on the operating mode of the processor (protected or real) and the current operand-size attribute (16-bit or 32-bit). In virtual-8086 mode, the real mode layouts are used.

The FSTENV instruction checks for and handles any pending unmasked floating-point exceptions before storing the FPU environment; the FNSTENV instruction does not. The saved image reflects the state of the FPU after all floating-point instructions preceding the FSTENV/FNSTENV instruction in the instruction stream have been executed.

These instructions are often used by exception handlers because they provide access to the FPU instruction and data pointers. The environment is typically saved in the stack. Masking all exceptions after saving the environment prevents floating-point exceptions from interrupting the exception handler.

The assembler issues two instructions for the FSTENV instruction (an FWAIT instruction followed by an FNSTENV instruction), and the processor executes each of these instructions separately. If an exception is generated for either of these instructions, the save EIP points to the instruction that caused the exception.

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

IA-32 Architecture Compatibility

When operating a Pentium or Intel486 processor in MS-DOS compatibility mode, it is possible (under unusual circumstances) for an FNSTENV instruction to be interrupted prior to being executed to handle a pending FPU exception. See the section titled "No-Wait FPU Instructions Can Get FPU Interrupt in Window" in Appendix D of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*, for a description of these circumstances. An FNSTENV instruction cannot be interrupted in this way on later Intel processors, except for the Intel Quark™ X1000 processor.

Operation

```
DEST[FPUControlWord] ← FPUControlWord;
DEST[FPUStatusWord] ← FPUStatusWord;
DEST[FPUTagWord] ← FPUTagWord;
DEST[FPUDataPointer] ← FPUDataPointer;
DEST[FPUInstructionPointer] ← FPUInstructionPointer;
DEST[FPULastInstructionOpcode] ← FPULastInstructionOpcode;
```

FPU Flags Affected

The C0, C1, C2, and C3 are undefined.

Floating-Point Exceptions

None

Protected Mode Exceptions

#GP(0)	If the destination is located in a non-writable segment. If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit. If the DS, ES, FS, or GS register is used to access memory and it contains a NULL segment selector.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If the LOCK prefix is used.

Real-Address Mode Exceptions

#GP	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS	If a memory operand effective address is outside the SS segment limit.
#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#UD	If the LOCK prefix is used.

Virtual-8086 Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.
#UD	If the LOCK prefix is used.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form.
#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#MF	If there is a pending x87 FPU exception.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If the LOCK prefix is used.

FSTSW/FNSTSW—Store x87 FPU Status Word

Opcode	Instruction	64-Bit Mode	Compat/ Leg Mode	Description
9B DD /7	FSTSW *m2byte*	Valid	Valid	Store FPU status word at *m2byte* after checking for pending unmasked floating-point exceptions.
9B DF E0	FSTSW AX	Valid	Valid	Store FPU status word in AX register after checking for pending unmasked floating-point exceptions.
DD /7	FNSTSW* *m2byte*	Valid	Valid	Store FPU status word at *m2byte* without checking for pending unmasked floating-point exceptions.
DF E0	FNSTSW* AX	Valid	Valid	Store FPU status word in AX register without checking for pending unmasked floating-point exceptions.

NOTES:

* See IA-32 Architecture Compatibility section below.

Description

Stores the current value of the x87 FPU status word in the destination location. The destination operand can be either a two-byte memory location or the AX register. The FSTSW instruction checks for and handles pending unmasked floating-point exceptions before storing the status word; the FNSTSW instruction does not.

The FNSTSW AX form of the instruction is used primarily in conditional branching (for instance, after an FPU comparison instruction or an FPREM, FPREM1, or FXAM instruction), where the direction of the branch depends on the state of the FPU condition code flags. (See the section titled "Branching and Conditional Moves on FPU Condition Codes" in Chapter 8 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*.) This instruction can also be used to invoke exception handlers (by examining the exception flags) in environments that do not use interrupts. When the FNSTSW AX instruction is executed, the AX register is updated before the processor executes any further instructions. The status stored in the AX register is thus guaranteed to be from the completion of the prior FPU instruction.

The assembler issues two instructions for the FSTSW instruction (an FWAIT instruction followed by an FNSTSW instruction), and the processor executes each of these instructions separately. If an exception is generated for either of these instructions, the save EIP points to the instruction that caused the exception.

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

IA-32 Architecture Compatibility

When operating a Pentium or Intel486 processor in MS-DOS compatibility mode, it is possible (under unusual circumstances) for an FNSTSW instruction to be interrupted prior to being executed to handle a pending FPU exception. See the section titled "No-Wait FPU Instructions Can Get FPU Interrupt in Window" in Appendix D of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*, for a description of these circumstances. An FNSTSW instruction cannot be interrupted in this way on later Intel processors, except for the Intel Quark™ X1000 processor.

Operation

```
DEST ← FPUStatusWord;
```

FPU Flags Affected

The C0, C1, C2, and C3 are undefined.

Floating-Point Exceptions

None

Protected Mode Exceptions

#GP(0)	If the destination is located in a non-writable segment.
	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
	If the DS, ES, FS, or GS register is used to access memory and it contains a NULL segment selector.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If the LOCK prefix is used.

Real-Address Mode Exceptions

#GP	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS	If a memory operand effective address is outside the SS segment limit.
#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.

Virtual-8086 Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.
#UD	If the LOCK prefix is used.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form.
#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#MF	If there is a pending x87 FPU exception.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If the LOCK prefix is used.

FSUB/FSUBP/FISUB—Subtract

Opcode	Instruction	64-Bit Mode	Compat/ Leg Mode	Description
D8 /4	FSUB *m32fp*	Valid	Valid	Subtract *m32fp* from ST(0) and store result in ST(0).
DC /4	FSUB *m64fp*	Valid	Valid	Subtract *m64fp* from ST(0) and store result in ST(0).
D8 E0+i	FSUB ST(0), ST(i)	Valid	Valid	Subtract ST(i) from ST(0) and store result in ST(0).
DC E8+i	FSUB ST(i), ST(0)	Valid	Valid	Subtract ST(0) from ST(i) and store result in ST(i).
DE E8+i	FSUBP ST(i), ST(0)	Valid	Valid	Subtract ST(0) from ST(i), store result in ST(i), and pop register stack.
DE E9	FSUBP	Valid	Valid	Subtract ST(0) from ST(1), store result in ST(1), and pop register stack.
DA /4	FISUB *m32int*	Valid	Valid	Subtract *m32int* from ST(0) and store result in ST(0).
DE /4	FISUB *m16int*	Valid	Valid	Subtract *m16int* from ST(0) and store result in ST(0).

Description

Subtracts the source operand from the destination operand and stores the difference in the destination location. The destination operand is always an FPU data register; the source operand can be a register or a memory location. Source operands in memory can be in single-precision or double-precision floating-point format or in word or doubleword integer format.

The no-operand version of the instruction subtracts the contents of the ST(0) register from the ST(1) register and stores the result in ST(1). The one-operand version subtracts the contents of a memory location (either a floating-point or an integer value) from the contents of the ST(0) register and stores the result in ST(0). The two-operand version, subtracts the contents of the ST(0) register from the ST(i) register or vice versa.

The FSUBP instructions perform the additional operation of popping the FPU register stack following the subtraction. To pop the register stack, the processor marks the ST(0) register as empty and increments the stack pointer (TOP) by 1. The no-operand version of the floating-point subtract instructions always results in the register stack being popped. In some assemblers, the mnemonic for this instruction is FSUB rather than FSUBP.

The FISUB instructions convert an integer source operand to double extended-precision floating-point format before performing the subtraction.

Table 3-38 shows the results obtained when subtracting various classes of numbers from one another, assuming that neither overflow nor underflow occurs. Here, the SRC value is subtracted from the DEST value (DEST – SRC = result).

When the difference between two operands of like sign is 0, the result is +0, except for the round toward –∞ mode, in which case the result is –0. This instruction also guarantees that +0 – (–0) = +0, and that –0 – (+0) = –0. When the source operand is an integer 0, it is treated as a +0.

When one operand is ∞, the result is ∞ of the expected sign. If both operands are ∞ of the same sign, an invalid-operation exception is generated.

Table 3-38. FSUB/FSUBP/FISUB Results

		SRC						
		– ∞	– F or – I	– 0	+ 0	+ F or + I	+ ∞	NaN
DEST	– ∞	*	– ∞	– ∞	– ∞	– ∞	– ∞	NaN
	– F	+ ∞	±F or ±0	DEST	DEST	– F	– ∞	NaN
	– 0	+ ∞	–SRC	±0	– 0	– SRC	– ∞	NaN
	+ 0	+ ∞	–SRC	+ 0	±0	– SRC	– ∞	NaN
	+ F	+ ∞	+ F	DEST	DEST	±F or ±0	– ∞	NaN
	+ ∞	+ ∞	+ ∞	+ ∞	+ ∞	+ ∞	*	NaN
	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN

NOTES:

F Means finite floating-point value.

I Means integer.

* Indicates floating-point invalid-arithmetic-operand (#IA) exception.

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

Operation

```
IF Instruction = FISUB
    THEN
        DEST ← DEST – ConvertToDoubleExtendedPrecisionFP(SRC);
    ELSE (* Source operand is floating-point value *)
        DEST ← DEST – SRC;
FI;

IF Instruction = FSUBP
    THEN
        PopRegisterStack;
FI;
```

FPU Flags Affected

C1	Set to 0 if stack underflow occurred.
	Set if result was rounded up; cleared otherwise.
C0, C2, C3	Undefined.

Floating-Point Exceptions

#IS	Stack underflow occurred.
#IA	Operand is an SNaN value or unsupported format.
	Operands are infinities of like sign.
#D	Source operand is a denormal value.
#U	Result is too small for destination format.
#O	Result is too large for destination format.
#P	Value cannot be represented exactly in destination format.

Protected Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit. If the DS, ES, FS, or GS register is used to access memory and it contains a NULL segment selector.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If the LOCK prefix is used.

Real-Address Mode Exceptions

#GP	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS	If a memory operand effective address is outside the SS segment limit.
#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#UD	If the LOCK prefix is used.

Virtual-8086 Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.
#UD	If the LOCK prefix is used.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form.
#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#MF	If there is a pending x87 FPU exception.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If the LOCK prefix is used.

FSUBR/FSUBRP/FISUBR—Reverse Subtract

Opcode	Instruction	64-Bit Mode	Compat/ Leg Mode	Description
D8 /5	FSUBR *m32fp*	Valid	Valid	Subtract ST(0) from *m32fp* and store result in ST(0).
DC /5	FSUBR *m64fp*	Valid	Valid	Subtract ST(0) from *m64fp* and store result in ST(0).
D8 E8+i	FSUBR ST(0), ST(i)	Valid	Valid	Subtract ST(0) from ST(i) and store result in ST(0).
DC E0+i	FSUBR ST(i), ST(0)	Valid	Valid	Subtract ST(i) from ST(0) and store result in ST(i).
DE E0+i	FSUBRP ST(i), ST(0)	Valid	Valid	Subtract ST(i) from ST(0), store result in ST(i), and pop register stack.
DE E1	FSUBRP	Valid	Valid	Subtract ST(1) from ST(0), store result in ST(1), and pop register stack.
DA /5	FISUBR *m32int*	Valid	Valid	Subtract ST(0) from *m32int* and store result in ST(0).
DE /5	FISUBR *m16int*	Valid	Valid	Subtract ST(0) from *m16int* and store result in ST(0).

Description

Subtracts the destination operand from the source operand and stores the difference in the destination location. The destination operand is always an FPU register; the source operand can be a register or a memory location. Source operands in memory can be in single-precision or double-precision floating-point format or in word or doubleword integer format.

These instructions perform the reverse operations of the FSUB, FSUBP, and FISUB instructions. They are provided to support more efficient coding.

The no-operand version of the instruction subtracts the contents of the ST(1) register from the ST(0) register and stores the result in ST(1). The one-operand version subtracts the contents of the ST(0) register from the contents of a memory location (either a floating-point or an integer value) and stores the result in ST(0). The two-operand version, subtracts the contents of the ST(i) register from the ST(0) register or vice versa.

The FSUBRP instructions perform the additional operation of popping the FPU register stack following the subtraction. To pop the register stack, the processor marks the ST(0) register as empty and increments the stack pointer (TOP) by 1. The no-operand version of the floating-point reverse subtract instructions always results in the register stack being popped. In some assemblers, the mnemonic for this instruction is FSUBR rather than FSUBRP.

The FISUBR instructions convert an integer source operand to double extended-precision floating-point format before performing the subtraction.

The following table shows the results obtained when subtracting various classes of numbers from one another, assuming that neither overflow nor underflow occurs. Here, the DEST value is subtracted from the SRC value (SRC – DEST = result).

When the difference between two operands of like sign is 0, the result is +0, except for the round toward $-\infty$ mode, in which case the result is –0. This instruction also guarantees that +0 – (–0) = +0, and that –0 – (+0) = –0. When the source operand is an integer 0, it is treated as a +0.

When one operand is ∞, the result is ∞ of the expected sign. If both operands are ∞ of the same sign, an invalid-operation exception is generated.

Table 3-39. FSUBR/FSUBRP/FISUBR Results

		SRC						
		– ∞	–F or –I	–0	+0	+F or +I	+ ∞	NaN
DEST	– ∞	*	+ ∞	+ ∞	+ ∞	+ ∞	+ ∞	NaN
	– F	– ∞	±F or ±0	–DEST	–DEST	+ F	+ ∞	NaN
	– 0	– ∞	SRC	±0	+ 0	SRC	+ ∞	NaN
	+ 0	– ∞	SRC	– 0	±0	SRC	+ ∞	NaN
	+ F	– ∞	– F	–DEST	–DEST	±F or ±0	+ ∞	NaN
	+ ∞	– ∞	– ∞	– ∞	– ∞	– ∞	*	NaN
	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN

NOTES:

F Means finite floating-point value.

I Means integer.

* Indicates floating-point invalid-arithmetic-operand (#IA) exception.

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

Operation

```
IF Instruction = FISUBR
    THEN
        DEST ← ConvertToDoubleExtendedPrecisionFP(SRC) – DEST;
    ELSE (* Source operand is floating-point value *)
        DEST ← SRC – DEST; FI;
IF Instruction = FSUBRP
    THEN
        PopRegisterStack; FI;
```

FPU Flags Affected

C1	Set to 0 if stack underflow occurred.
	Set if result was rounded up; cleared otherwise.
C0, C2, C3	Undefined.

Floating-Point Exceptions

#IS	Stack underflow occurred.
#IA	Operand is an SNaN value or unsupported format.
	Operands are infinities of like sign.
#D	Source operand is a denormal value.
#U	Result is too small for destination format.
#O	Result is too large for destination format.
#P	Value cannot be represented exactly in destination format.

Protected Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
	If the DS, ES, FS, or GS register is used to access memory and it contains a NULL segment selector.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If the LOCK prefix is used.

Real-Address Mode Exceptions

#GP	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS	If a memory operand effective address is outside the SS segment limit.
#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#UD	If the LOCK prefix is used.

Virtual-8086 Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.
#UD	If the LOCK prefix is used.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form.
#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#MF	If there is a pending x87 FPU exception.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If the LOCK prefix is used.

FTST—TEST

Opcode	Instruction	64-Bit Mode	Compat/ Leg Mode	Description
D9 E4	FTST	Valid	Valid	Compare ST(0) with 0.0.

Description

Compares the value in the ST(0) register with 0.0 and sets the condition code flags C0, C2, and C3 in the FPU status word according to the results (see table below).

Table 3-40. FTST Results

Condition	C3	C2	C0
ST(0) > 0.0	0	0	0
ST(0) < 0.0	0	0	1
ST(0) = 0.0	1	0	0
Unordered	1	1	1

This instruction performs an "unordered comparison." An unordered comparison also checks the class of the numbers being compared (see "FXAM—Examine Floating-Point" in this chapter). If the value in register ST(0) is a NaN or is in an undefined format, the condition flags are set to "unordered" and the invalid operation exception is generated.

The sign of zero is ignored, so that (– 0.0 ← +0.0).

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

Operation

```
CASE (relation of operands) OF
    Not comparable:  C3, C2, C0 ← 111;
    ST(0) > 0.0:     C3, C2, C0 ← 000;
    ST(0) < 0.0:     C3, C2, C0 ← 001;
    ST(0) = 0.0:     C3, C2, C0 ← 100;
ESAC;
```

FPU Flags Affected

C1	Set to 0.
C0, C2, C3	See Table 3-40.

Floating-Point Exceptions

#IS	Stack underflow occurred.
#IA	The source operand is a NaN value or is in an unsupported format.
#D	The source operand is a denormal value.

Protected Mode Exceptions

#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#MF	If there is a pending x87 FPU exception.
#UD	If the LOCK prefix is used.

Real-Address Mode Exceptions

Same exceptions as in protected mode.

Virtual-8086 Mode Exceptions

Same exceptions as in protected mode.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

Same exceptions as in protected mode.

FUCOM/FUCOMP/FUCOMPP—Unordered Compare Floating Point Values

Opcode	Instruction	64-Bit Mode	Compat/ Leg Mode	Description
DD E0+i	FUCOM ST(i)	Valid	Valid	Compare ST(0) with ST(i).
DD E1	FUCOM	Valid	Valid	Compare ST(0) with ST(1).
DD E8+i	FUCOMP ST(i)	Valid	Valid	Compare ST(0) with ST(i) and pop register stack.
DD E9	FUCOMP	Valid	Valid	Compare ST(0) with ST(1) and pop register stack.
DA E9	FUCOMPP	Valid	Valid	Compare ST(0) with ST(1) and pop register stack twice.

Description

Performs an unordered comparison of the contents of register ST(0) and ST(i) and sets condition code flags C0, C2, and C3 in the FPU status word according to the results (see the table below). If no operand is specified, the contents of registers ST(0) and ST(1) are compared. The sign of zero is ignored, so that −0.0 is equal to +0.0.

Table 3-41. FUCOM/FUCOMP/FUCOMPP Results

Comparison Results*	C3	C2	C0
ST0 > ST(i)	0	0	0
ST0 < ST(i)	0	0	1
ST0 = ST(i)	1	0	0
Unordered	1	1	1

NOTES:

* Flags not set if unmasked invalid-arithmetic-operand (#IA) exception is generated.

An unordered comparison checks the class of the numbers being compared (see "FXAM—Examine Floating-Point" in this chapter). The FUCOM/FUCOMP/FUCOMPP instructions perform the same operations as the FCOM/FCOMP/FCOMPP instructions. The only difference is that the FUCOM/FUCOMP/FUCOMPP instructions raise the invalid-arithmetic-operand exception (#IA) only when either or both operands are an SNaN or are in an unsupported format; QNaNs cause the condition code flags to be set to unordered, but do not cause an exception to be generated. The FCOM/FCOMP/FCOMPP instructions raise an invalid-operation exception when either or both of the operands are a NaN value of any kind or are in an unsupported format.

As with the FCOM/FCOMP/FCOMPP instructions, if the operation results in an invalid-arithmetic-operand exception being raised, the condition code flags are set only if the exception is masked.

The FUCOMP instruction pops the register stack following the comparison operation and the FUCOMPP instruction pops the register stack twice following the comparison operation. To pop the register stack, the processor marks the ST(0) register as empty and increments the stack pointer (TOP) by 1.

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

Operation

```
CASE (relation of operands) OF
    ST > SRC:       C3, C2, C0 ← 000;
    ST < SRC:       C3, C2, C0 ← 001;
    ST = SRC:       C3, C2, C0 ← 100;
ESAC;

IF ST(0) or SRC = QNaN, but not SNaN or unsupported format
    THEN
        C3, C2, C0 ← 111;
    ELSE (* ST(0) or SRC is SNaN or unsupported format *)
         #IA;
        IF FPUControlWord.IM = 1
            THEN
                C3, C2, C0 ← 111;
        FI;
FI;

IF Instruction = FUCOMP
    THEN
        PopRegisterStack;
FI;

IF Instruction = FUCOMPP
    THEN
        PopRegisterStack;
FI;
```

FPU Flags Affected

C1	Set to 0 if stack underflow occurred.
C0, C2, C3	See Table 3-41.

Floating-Point Exceptions

#IS	Stack underflow occurred.
#IA	One or both operands are SNaN values or have unsupported formats. Detection of a QNaN value in and of itself does not raise an invalid-operand exception.
#D	One or both operands are denormal values.

Protected Mode Exceptions

#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#MF	If there is a pending x87 FPU exception.
#UD	If the LOCK prefix is used.

Real-Address Mode Exceptions

Same exceptions as in protected mode.

Virtual-8086 Mode Exceptions

Same exceptions as in protected mode.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

Same exceptions as in protected mode.

FXAM—Examine Floating-Point

Opcode	Instruction	64-Bit Mode	Compat/ Leg Mode	Description
D9 E5	FXAM	Valid	Valid	Classify value or number in ST(0).

Description

Examines the contents of the ST(0) register and sets the condition code flags C0, C2, and C3 in the FPU status word to indicate the class of value or number in the register (see the table below).

Table 3-42. FXAM Results

Class	C3	C2	C0
Unsupported	0	0	0
NaN	0	0	1
Normal finite number	0	1	0
Infinity	0	1	1
Zero	1	0	0
Empty	1	0	1
Denormal number	1	1	0

The C1 flag is set to the sign of the value in ST(0), regardless of whether the register is empty or full.

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

Operation

```
C1 ← sign bit of ST; (* 0 for positive, 1 for negative *)

CASE (class of value or number in ST(0)) OF
    Unsupported:C3, C2, C0 ← 000;
    NaN:        C3, C2, C0 ← 001;
    Normal:     C3, C2, C0 ← 010;
    Infinity:   C3, C2, C0 ← 011;
    Zero:       C3, C2, C0 ← 100;
    Empty:      C3, C2, C0 ← 101;
    Denormal:   C3, C2, C0 ← 110;
ESAC;
```

FPU Flags Affected

C1	Sign of value in ST(0).
C0, C2, C3	See Table 3-42.

Floating-Point Exceptions

None

Protected Mode Exceptions

#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#MF	If there is a pending x87 FPU exception.
#UD	If the LOCK prefix is used.

Real-Address Mode Exceptions

Same exceptions as in protected mode.

Virtual-8086 Mode Exceptions

Same exceptions as in protected mode.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

Same exceptions as in protected mode.

FXCH—Exchange Register Contents

Opcode	Instruction	64-Bit Mode	Compat/ Leg Mode	Description
D9 C8+i	FXCH ST(i)	Valid	Valid	Exchange the contents of ST(0) and ST(i).
D9 C9	FXCH	Valid	Valid	Exchange the contents of ST(0) and ST(1).

Description

Exchanges the contents of registers ST(0) and ST(i). If no source operand is specified, the contents of ST(0) and ST(1) are exchanged.

This instruction provides a simple means of moving values in the FPU register stack to the top of the stack [ST(0)], so that they can be operated on by those floating-point instructions that can only operate on values in ST(0). For example, the following instruction sequence takes the square root of the third register from the top of the register stack:

```
FXCH ST(3);
FSQRT;
FXCH ST(3);
```

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

Operation

```
IF (Number-of-operands) is 1
    THEN
        temp ← ST(0);
        ST(0) ← SRC;
        SRC ← temp;
    ELSE
        temp ← ST(0);
        ST(0) ← ST(1);
        ST(1) ← temp;
FI;
```

FPU Flags Affected

C1	Set to 0.
C0, C2, C3	Undefined.

Floating-Point Exceptions

#IS	Stack underflow occurred.

Protected Mode Exceptions

#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#MF	If there is a pending x87 FPU exception.
#UD	If the LOCK prefix is used.

Real-Address Mode Exceptions

Same exceptions as in protected mode.

Virtual-8086 Mode Exceptions

Same exceptions as in protected mode.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

Same exceptions as in protected mode.

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

Opcode/ Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
NP 0F AE /1 FXRSTOR *m512byte*	M	Valid	Valid	Restore the x87 FPU, MMX, XMM, and MXCSR register state from *m512byte*.
NP REX.W + 0F AE /1 FXRSTOR64 *m512byte*	M	Valid	N.E.	Restore the x87 FPU, MMX, XMM, and MXCSR register state from *m512byte*.

Instruction Operand Encoding

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

Description

Reloads the x87 FPU, MMX technology, XMM, and MXCSR registers from the 512-byte memory image specified in the source operand. This data should have been written to memory previously using the FXSAVE instruction, and in the same format as required by the operating modes. The first byte of the data should be located on a 16-byte boundary. There are three distinct layouts of the FXSAVE state map: one for legacy and compatibility mode, a second format for 64-bit mode FXSAVE/FXRSTOR with REX.W=0, and the third format is for 64-bit mode with FXSAVE64/FXRSTOR64. Table 3-43 shows the layout of the legacy/compatibility mode state information in memory and describes the fields in the memory image for the FXRSTOR and FXSAVE instructions. Table 3-46 shows the layout of the 64-bit mode state information when REX.W is set (FXSAVE64/FXRSTOR64). Table 3-47 shows the layout of the 64-bit mode state information when REX.W is clear (FXSAVE/FXRSTOR).

The state image referenced with an FXRSTOR instruction must have been saved using an FXSAVE instruction or be in the same format as required by Table 3-43, Table 3-46, or Table 3-47. Referencing a state image saved with an FSAVE, FNSAVE instruction or incompatible field layout will result in an incorrect state restoration.

The FXRSTOR instruction does not flush pending x87 FPU exceptions. To check and raise exceptions when loading x87 FPU state information with the FXRSTOR instruction, use an FWAIT instruction after the FXRSTOR instruction.

If the OSFXSR bit in control register CR4 is not set, the FXRSTOR instruction may not restore the states of the XMM and MXCSR registers. This behavior is implementation dependent.

If the MXCSR state contains an unmasked exception with a corresponding status flag also set, loading the register with the FXRSTOR instruction will not result in a SIMD floating-point error condition being generated. Only the next occurrence of this unmasked exception will result in the exception being generated.

Bits 16 through 32 of the MXCSR register are defined as reserved and should be set to 0. Attempting to write a 1 in any of these bits from the saved state image will result in a general protection exception (#GP) being generated.

Bytes 464:511 of an FXSAVE image are available for software use. FXRSTOR ignores the content of bytes 464:511 in an FXSAVE state image.

Operation

```
IF 64-Bit Mode
    THEN
        (x87 FPU, MMX, XMM15-XMM0, MXCSR)    Load(SRC);
    ELSE
         (x87 FPU, MMX, XMM7-XMM0, MXCSR) ← Load(SRC);
FI;
```

x87 FPU and SIMD Floating-Point Exceptions

None.

Protected Mode Exceptions

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

Real-Address Mode Exceptions

#GP	If a memory operand is not aligned on a 16-byte boundary, regardless of segment.
	If any part of the operand lies outside the effective address space from 0 to FFFFH.
	For an attempt to set reserved bits in MXCSR.
#NM	If CR0.TS[bit 3] = 1.
	If CR0.EM[bit 2] = 1.
#UD	If CPUID.01H:EDX.FXSR[bit 24] = 0.
	If the LOCK prefix is used.

Virtual-8086 Mode Exceptions

Same exceptions as in real address mode.

#PF(fault-code)	For a page fault.
#AC	For unaligned memory reference.
#UD	If the LOCK prefix is used.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

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

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

Opcode/ Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
NP 0F AE /0 FXSAVE *m512byte*	M	Valid	Valid	Save the x87 FPU, MMX, XMM, and MXCSR register state to *m512byte*.
NP REX.W + 0F AE /0 FXSAVE64 *m512byte*	M	Valid	N.E.	Save the x87 FPU, MMX, XMM, and MXCSR register state to *m512byte*.

Instruction Operand Encoding

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

Description

Saves the current state of the x87 FPU, MMX technology, XMM, and MXCSR registers to a 512-byte memory location specified in the destination operand. The content layout of the 512 byte region depends on whether the processor is operating in non-64-bit operating modes or 64-bit sub-mode of IA-32e mode.

Bytes 464:511 are available to software use. The processor does not write to bytes 464:511 of an FXSAVE area.

The operation of FXSAVE in non-64-bit modes is described first.

Non-64-Bit Mode Operation

Table 3-43 shows the layout of the state information in memory when the processor is operating in legacy modes.

Table 3-43. Non-64-bit-Mode Layout of FXSAVE and FXRSTOR Memory Region

15 14	13 12	11 10	9 8	7 6	5	4	3 2	1 0	
Rsvd	FCS	FIP[31:0]		FOP	Rsvd	FTW	FSW	FCW	0
MXCSR_MASK		MXCSR		Rsrvd	FDS		FDP[31:0]		16
Reserved		ST0/MM0							32
Reserved		ST1/MM1							48
Reserved		ST2/MM2							64
Reserved		ST3/MM3							80
Reserved		ST4/MM4							96
Reserved		ST5/MM5							112
Reserved		ST6/MM6							128
Reserved		ST7/MM7							144
XMM0									160
XMM1									176
XMM2									192
XMM3									208
XMM4									224
XMM5									240
XMM6									256
XMM7									272
Reserved									288

Table 3-43. Non-64-bit-Mode Layout of FXSAVE and FXRSTOR Memory Region (Contd.)

15 14	13 12	11 10	9 8	7 6	5	4	3 2	1 0	
Reserved									304
Reserved									320
Reserved									336
Reserved									352
Reserved									368
Reserved									384
Reserved									400
Reserved									416
Reserved									432
Reserved									448
Available									464
Available									480
Available									496

The destination operand contains the first byte of the memory image, and it must be aligned on a 16-byte boundary. A misaligned destination operand will result in a general-protection (#GP) exception being generated (or in some cases, an alignment check exception [#AC]).

The FXSAVE instruction is used when an operating system needs to perform a context switch or when an exception handler needs to save and examine the current state of the x87 FPU, MMX technology, and/or XMM and MXCSR registers.

The fields in Table 3-43 are defined in Table 3-44.

Table 3-44. Field Definitions

Field	Definition
FCW	x87 FPU Control Word (16 bits). See Figure 8-6 in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*, for the layout of the x87 FPU control word.
FSW	x87 FPU Status Word (16 bits). See Figure 8-4 in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*, for the layout of the x87 FPU status word.
Abridged FTW	x87 FPU Tag Word (8 bits). The tag information saved here is abridged, as described in the following paragraphs.
FOP	x87 FPU Opcode (16 bits). The lower 11 bits of this field contain the opcode, upper 5 bits are reserved. See Figure 8-8 in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*, for the layout of the x87 FPU opcode field.
FIP	x87 FPU Instruction Pointer Offset (64 bits). The contents of this field differ depending on the current addressing mode (32-bit, 16-bit, or 64-bit) of the processor when the FXSAVE instruction was executed: 32-bit mode — 32-bit IP offset. 16-bit mode — low 16 bits are IP offset; high 16 bits are reserved. 64-bit mode with REX.W — 64-bit IP offset. 64-bit mode without REX.W — 32-bit IP offset. See "x87 FPU Instruction and Operand (Data) Pointers" in Chapter 8 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*, for a description of the x87 FPU instruction pointer.

Table 3-44. Field Definitions (Contd.)

Field	Definition
FCS	x87 FPU Instruction Pointer Selector (16 bits). If CPUID.(EAX=07H,ECX=0H):EBX[bit 13] = 1, the processor deprecates FCS and FDS, and this field is saved as 0000H.
FDP	x87 FPU Instruction Operand (Data) Pointer Offset (64 bits). The contents of this field differ depending on the current addressing mode (32-bit, 16-bit, or 64-bit) of the processor when the FXSAVE instruction was executed: 32-bit mode — 32-bit DP offset. 16-bit mode — low 16 bits are DP offset; high 16 bits are reserved. 64-bit mode with REX.W — 64-bit DP offset. 64-bit mode without REX.W — 32-bit DP offset. See "x87 FPU Instruction and Operand (Data) Pointers" in Chapter 8 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*, for a description of the x87 FPU operand pointer.
FDS	x87 FPU Instruction Operand (Data) Pointer Selector (16 bits). If CPUID.(EAX=07H,ECX=0H):EBX[bit 13] = 1, the processor deprecates FCS and FDS, and this field is saved as 0000H.
MXCSR	MXCSR Register State (32 bits). See Figure 10-3 in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*, for the layout of the MXCSR register. If the OSFXSR bit in control register CR4 is not set, the FXSAVE instruction may not save this register. This behavior is implementation dependent.
MXCSR_ MASK	MXCSR_MASK (32 bits). This mask can be used to adjust values written to the MXCSR register, ensuring that reserved bits are set to 0. Set the mask bits and flags in MXCSR to the mode of operation desired for SSE and SSE2 SIMD floating-point instructions. See "Guidelines for Writing to the MXCSR Register" in Chapter 11 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*, for instructions for how to determine and use the MXCSR_MASK value.
ST0/MM0 through ST7/MM7	x87 FPU or MMX technology registers. These 80-bit fields contain the x87 FPU data registers or the MMX technology registers, depending on the state of the processor prior to the execution of the FXSAVE instruction. If the processor had been executing x87 FPU instruction prior to the FXSAVE instruction, the x87 FPU data registers are saved; if it had been executing MMX instructions (or SSE or SSE2 instructions that operated on the MMX technology registers), the MMX technology registers are saved. When the MMX technology registers are saved, the high 16 bits of the field are reserved.
XMM0 through XMM7	XMM registers (128 bits per field). If the OSFXSR bit in control register CR4 is not set, the FXSAVE instruction may not save these registers. This behavior is implementation dependent.

The FXSAVE instruction saves an abridged version of the x87 FPU tag word in the FTW field (unlike the FSAVE instruction, which saves the complete tag word). The tag information is saved in physical register order (R0 through R7), rather than in top-of-stack (TOS) order. With the FXSAVE instruction, however, only a single bit (1 for valid or 0 for empty) is saved for each tag. For example, assume that the tag word is currently set as follows:

R7	R6	R5	R4	R3	R2	R1	R0
11	xx	xx	xx	11	11	11	11

Here, 11B indicates empty stack elements and "xx" indicates valid (00B), zero (01B), or special (10B).

For this example, the FXSAVE instruction saves only the following 8 bits of information:

R7	R6	R5	R4	R3	R2	R1	R0
0	1	1	1	0	0	0	0

Here, a 1 is saved for any valid, zero, or special tag, and a 0 is saved for any empty tag.

The operation of the FXSAVE instruction differs from that of the FSAVE instruction, the as follows:

- FXSAVE instruction does not check for pending unmasked floating-point exceptions. (The FXSAVE operation in this regard is similar to the operation of the FNSAVE instruction).
- After the FXSAVE instruction has saved the state of the x87 FPU, MMX technology, XMM, and MXCSR registers, the processor retains the contents of the registers. Because of this behavior, the FXSAVE instruction cannot be

used by an application program to pass a "clean" x87 FPU state to a procedure, since it retains the current state. To clean the x87 FPU state, an application must explicitly execute an FINIT instruction after an FXSAVE instruction to reinitialize the x87 FPU state.

- The format of the memory image saved with the FXSAVE instruction is the same regardless of the current addressing mode (32-bit or 16-bit) and operating mode (protected, real address, or system management). This behavior differs from the FSAVE instructions, where the memory image format is different depending on the addressing mode and operating mode. Because of the different image formats, the memory image saved with the FXSAVE instruction cannot be restored correctly with the FRSTOR instruction, and likewise the state saved with the FSAVE instruction cannot be restored correctly with the FXRSTOR instruction.

The FSAVE format for FTW can be recreated from the FTW valid bits and the stored 80-bit FP data (assuming the stored data was not the contents of MMX technology registers) using Table 3-45.

Table 3-45. Recreating FSAVE Format

Exponent all 1's	Exponent all 0's	Fraction all 0's	J and M bits	FTW valid bit	x87 FTW	
0	0	0	0x	1	Special	10
0	0	0	1x	1	Valid	00
0	0	1	00	1	Special	10
0	0	1	10	1	Valid	00
0	1	0	0x	1	Special	10
0	1	0	1x	1	Special	10
0	1	1	00	1	Zero	01
0	1	1	10	1	Special	10
1	0	0	1x	1	Special	10
1	0	0	1x	1	Special	10
1	0	1	00	1	Special	10
1	0	1	10	1	Special	10
For all legal combinations above.				0	Empty	11

The J-bit is defined to be the 1-bit binary integer to the left of the decimal place in the significand. The M-bit is defined to be the most significant bit of the fractional portion of the significand (i.e., the bit immediately to the right of the decimal place).

When the M-bit is the most significant bit of the fractional portion of the significand, it must be 0 if the fraction is all 0's.

IA-32e Mode Operation

In compatibility sub-mode of IA-32e mode, legacy SSE registers, XMM0 through XMM7, are saved according to the legacy FXSAVE map. In 64-bit mode, all of the SSE registers, XMM0 through XMM15, are saved. Additionally, there are two different layouts of the FXSAVE map in 64-bit mode, corresponding to FXSAVE64 (which requires REX.W=1) and FXSAVE (REX.W=0). In the FXSAVE64 map (Table 3-46), the FPU IP and FPU DP pointers are 64-bit wide. In the FXSAVE map for 64-bit mode (Table 3-47), the FPU IP and FPU DP pointers are 32-bits.

Table 3-46. Layout of the 64-bit-mode FXSAVE64 Map (requires REX.W = 1)

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

Table 3-47. Layout of the 64-bit-mode FXSAVE Map (REX.W = 0)

15 14	13 12	11 10	9 8	7 6	5	4	3 2	1 0	
Reserved	FCS	FIP[31:0]		FOP	Reserved	FTW	FSW	FCW	0
MXCSR_MASK		MXCSR		Reserved	FDS		FDP[31:0]		16
Reserved		ST0/MM0							32
Reserved		ST1/MM1							48
Reserved		ST2/MM2							64
Reserved		ST3/MM3							80
Reserved		ST4/MM4							96
Reserved		ST5/MM5							112
Reserved		ST6/MM6							128
Reserved		ST7/MM7							144
XMM0									160
XMM1									176
XMM2									192
XMM3									208
XMM4									224
XMM5									240
XMM6									256
XMM7									272
XMM8									288
XMM9									304
XMM10									320
XMM11									336
XMM12									352
XMM13									368
XMM14									384
XMM15									400
Reserved									416
Reserved									432
Reserved									448
Available									464
Available									480
Available									496

Operation

```
IF 64-Bit Mode
    THEN
        IF REX.W = 1
            THEN
                DEST ← Save64BitPromotedFxsave(x87 FPU, MMX, XMM15-XMM0,
                MXCSR);
            ELSE
                DEST ← Save64BitDefaultFxsave(x87 FPU, MMX, XMM15-XMM0, MXCSR);
        FI;
    ELSE
        DEST ← SaveLegacyFxsave(x87 FPU, MMX, XMM7-XMM0, MXCSR);
FI;
```

Protected Mode Exceptions

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

Real-Address Mode Exceptions

#GP	If a memory operand is not aligned on a 16-byte boundary, regardless of segment.
	If any part of the operand lies outside the effective address space from 0 to FFFFH.
#NM	If CR0.TS[bit 3] = 1.
	If CR0.EM[bit 2] = 1.
#UD	If CPUID.01H:EDX.FXSR[bit 24] = 0.
	If the LOCK prefix is used.

Virtual-8086 Mode Exceptions

Same exceptions as in real address mode.

#PF(fault-code)	For a page fault.
#AC	For unaligned memory reference.
#UD	If the LOCK prefix is used.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

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

Implementation Note

The order in which the processor signals general-protection (#GP) and page-fault (#PF) exceptions when they both occur on an instruction boundary is given in Table 5-2 in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3B*. This order vary for FXSAVE for different processor implementations.

FXTRACT—Extract Exponent and Significand

Opcode/ Instruction	64-Bit Mode	Compat/ Leg Mode	Description
D9 F4 FXTRACT	Valid	Valid	Separate value in ST(0) into exponent and significand, store exponent in ST(0), and push the significand onto the register stack.

Description

Separates the source value in the ST(0) register into its exponent and significand, stores the exponent in ST(0), and pushes the significand onto the register stack. Following this operation, the new top-of-stack register ST(0) contains the value of the original significand expressed as a floating-point value. The sign and significand of this value are the same as those found in the source operand, and the exponent is 3FFFH (biased value for a true exponent of zero). The ST(1) register contains the value of the original operand's true (unbiased) exponent expressed as a floating-point value. (The operation performed by this instruction is a superset of the IEEE-recommended logb(x) function.)

This instruction and the F2XM1 instruction are useful for performing power and range scaling operations. The FXTRACT instruction is also useful for converting numbers in double extended-precision floating-point format to decimal representations (e.g., for printing or displaying).

If the floating-point zero-divide exception (#Z) is masked and the source operand is zero, an exponent value of $-\infty$ is stored in register ST(1) and 0 with the sign of the source operand is stored in register ST(0).

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

Operation

```
TEMP ← Significand(ST(0));
ST(0) ← Exponent(ST(0));
TOP← TOP – 1;
ST(0) ← TEMP;
```

FPU Flags Affected

C1	Set to 0 if stack underflow occurred; set to 1 if stack overflow occurred.
C0, C2, C3	Undefined.

Floating-Point Exceptions

#IS	Stack underflow or overflow occurred.
#IA	Source operand is an SNaN value or unsupported format.
#Z	ST(0) operand is ±0.
#D	Source operand is a denormal value.

Protected Mode Exceptions

#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#MF	If there is a pending x87 FPU exception.
#UD	If the LOCK prefix is used.

Real-Address Mode Exceptions

Same exceptions as in protected mode.

Virtual-8086 Mode Exceptions

Same exceptions as in protected mode.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

Same exceptions as in protected mode.

FYL2X—Compute y ∗ $\log_2$x

Opcode	Instruction	64-Bit Mode	Compat/ Leg Mode	Description
D9 F1	FYL2X	Valid	Valid	Replace ST(1) with (ST(1) ∗ $\log_2$ST(0)) and pop the register stack.

Description

Computes (ST(1) ∗ $\log_2$ (ST(0))), stores the result in resister ST(1), and pops the FPU register stack. The source operand in ST(0) must be a non-zero positive number.

The following table shows the results obtained when taking the log of various classes of numbers, assuming that neither overflow nor underflow occurs.

Table 3-48. FYL2X Results

		ST(0)							
		− ∞	− F	±0	+0<+F<+1	+ 1	+ F > + 1	+ ∞	NaN
ST(1)	− ∞	*	*	+ ∞	+ ∞	*	− ∞	− ∞	NaN
	− F	*	*	**	+ F	− 0	− F	− ∞	NaN
	− 0	*	*	*	+ 0	− 0	− 0	*	NaN
	+ 0	*	*	*	− 0	+ 0	+ 0	*	NaN
	+ F	*	*	**	− F	+ 0	+ F	+ ∞	NaN
	+ ∞	*	*	− ∞	− ∞	*	+ ∞	+ ∞	NaN
	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN

NOTES:

F Means finite floating-point value.

* Indicates floating-point invalid-operation (#IA) exception.

** Indicates floating-point zero-divide (#Z) exception.

If the divide-by-zero exception is masked and register ST(0) contains ±0, the instruction returns ∞ with a sign that is the opposite of the sign of the source operand in register ST(1).

The FYL2X instruction is designed with a built-in multiplication to optimize the calculation of logarithms with an arbitrary positive base (b):

$$\log_b x \leftarrow (\log_2 b)^{-1} * \log_2 x$$

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

Operation

ST(1) ← ST(1) ∗ $\log_2$ST(0);
PopRegisterStack;

FPU Flags Affected

C1	Set to 0 if stack underflow occurred.
	Set if result was rounded up; cleared otherwise.
C0, C2, C3	Undefined.

Floating-Point Exceptions

#IS	Stack underflow occurred.
#IA	Either operand is an SNaN or unsupported format.
	Source operand in register ST(0) is a negative finite value (not -0).
#Z	Source operand in register ST(0) is ±0.
#D	Source operand is a denormal value.
#U	Result is too small for destination format.
#O	Result is too large for destination format.
#P	Value cannot be represented exactly in destination format.

Protected Mode Exceptions

#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#MF	If there is a pending x87 FPU exception.
#UD	If the LOCK prefix is used.

Real-Address Mode Exceptions

Same exceptions as in protected mode.

Virtual-8086 Mode Exceptions

Same exceptions as in protected mode.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

Same exceptions as in protected mode.

FYL2XP1—Compute y ∗ log$_2$(x +1)

Opcode	Instruction	64-Bit Mode	Compat/ Leg Mode	Description
D9 F9	FYL2XP1	Valid	Valid	Replace ST(1) with ST(1) ∗ log$_2$(ST(0) + 1.0) and pop the register stack.

Description

Computes (ST(1) ∗ log$_2$(ST(0) + 1.0)), stores the result in register ST(1), and pops the FPU register stack. The source operand in ST(0) must be in the range:

$$-(1-\sqrt{2}/2))\text{to}(1-\sqrt{2}/2)$$

The source operand in ST(1) can range from –∞ to +∞. If the ST(0) operand is outside of its acceptable range, the result is undefined and software should not rely on an exception being generated. Under some circumstances exceptions may be generated when ST(0) is out of range, but this behavior is implementation specific and not guaranteed.

The following table shows the results obtained when taking the log epsilon of various classes of numbers, assuming that underflow does not occur.

Table 3-49. FYL2XP1 Results

		ST(0)				
		–(1 – ($\sqrt{2}/2$)) to –0	-0	+0	+0 to +(1 - ($\sqrt{2}/2$))	NaN
ST(1)	– ∞	+∞	*	*	– ∞	NaN
	– F	+F	+0	-0	– F	NaN
	– 0	+0	+0	-0	– 0	NaN
	+0	– 0	– 0	+0	+0	NaN
	+F	– F	– 0	+0	+F	NaN
	+∞	– ∞	*	*	+∞	NaN
	NaN	NaN	NaN	NaN	NaN	NaN

NOTES:

F Means finite floating-point value.

* Indicates floating-point invalid-operation (#IA) exception.

This instruction provides optimal accuracy for values of epsilon [the value in register ST(0)] that are close to 0. For small epsilon (ε) values, more significant digits can be retained by using the FYL2XP1 instruction than by using (ε+1) as an argument to the FYL2X instruction. The (ε+1) expression is commonly found in compound interest and annuity calculations. The result can be simply converted into a value in another logarithm base by including a scale factor in the ST(1) source operand. The following equation is used to calculate the scale factor for a particular logarithm base, where n is the logarithm base desired for the result of the FYL2XP1 instruction:

scale factor ← log$_n$ 2

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

Operation

ST(1) ← ST(1) ∗ log$_2$(ST(0) + 1.0);
PopRegisterStack;

FPU Flags Affected

C1	Set to 0 if stack underflow occurred. Set if result was rounded up; cleared otherwise.
C0, C2, C3	Undefined.

Floating-Point Exceptions

#IS	Stack underflow occurred.
#IA	Either operand is an SNaN value or unsupported format.
#D	Source operand is a denormal value.
#U	Result is too small for destination format.
#O	Result is too large for destination format.
#P	Value cannot be represented exactly in destination format.

Protected Mode Exceptions

#NM	CR0.EM[bit 2] or CR0.TS[bit 3] = 1.
#MF	If there is a pending x87 FPU exception.
#UD	If the LOCK prefix is used.

Real-Address Mode Exceptions

Same exceptions as in protected mode.

Virtual-8086 Mode Exceptions

Same exceptions as in protected mode.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

Same exceptions as in protected mode.

HADDPD—Packed Double-FP Horizontal Add

Opcode/ Instruction	Op/ En	64/32-bit Mode	CPUID Feature Flag	Description
66 0F 7C /r HADDPD *xmm1, xmm2/m128*	RM	V/V	SSE3	Horizontal add packed double-precision floating-point values from *xmm2/m128* to *xmm1*.
VEX.NDS.128.66.0F.WIG 7C /r VHADDPD xmm1,xmm2, xmm3/m128	RVM	V/V	AVX	Horizontal add packed double-precision floating-point values from xmm2 and xmm3/mem.
VEX.NDS.256.66.0F.WIG 7C /r VHADDPD ymm1, ymm2, ymm3/m256	RVM	V/V	AVX	Horizontal add packed double-precision floating-point values from ymm2 and ymm3/mem.

Instruction Operand Encoding

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

Description

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

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

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

See Figure 3-16 for HADDPD; see Figure 3-17 for VHADDPD.

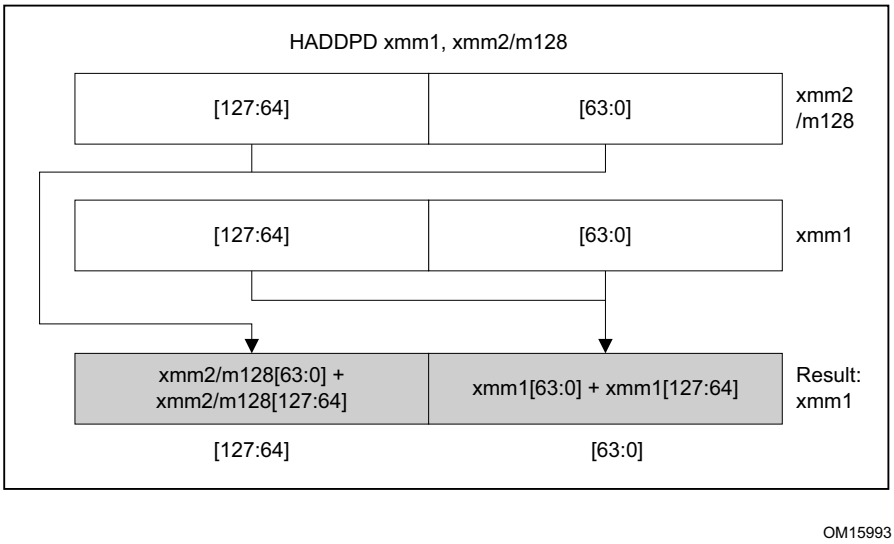


Figure 3-16. HADDPD—Packed Double-FP Horizontal Add

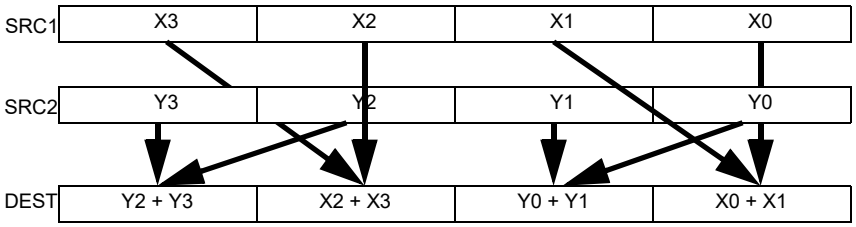


Figure 3-17. VHADDPD operation

128-bit Legacy SSE version: The second source can be an XMM register or an 128-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (MAXVL-1:128) of the corresponding YMM register destination are unmodified.

VEX.128 encoded version: the first source operand is an XMM register or 128-bit memory location. The destination operand is an XMM register. The upper bits (MAXVL-1:128) of the corresponding YMM register destination are zeroed.

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

Operation

HADDPD (128-bit Legacy SSE version)

```
DEST[63:0] ← SRC1[127:64] + SRC1[63:0]
DEST[127:64] ← SRC2[127:64] + SRC2[63:0]
DEST[MAXVL-1:128] (Unmodified)
```

VHADDPD (VEX.128 encoded version)

```
DEST[63:0] ← SRC1[127:64] + SRC1[63:0]
DEST[127:64] ← SRC2[127:64] + SRC2[63:0]
DEST[MAXVL-1:128] ← 0
```

VHADDPD (VEX.256 encoded version)

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

Intel C/C++ Compiler Intrinsic Equivalent

VHADDPD: __m256d _mm256_hadd_pd (__m256d a, __m256d b);

HADDPD: __m128d _mm_hadd_pd (__m128d a, __m128d b);

Exceptions

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

Numeric Exceptions

Overflow, Underflow, Invalid, Precision, Denormal

Other Exceptions

See Exceptions Type 2.

HADDPS—Packed Single-FP Horizontal Add

Opcode/ Instruction	Op/ En	64/32-bit Mode	CPUID Feature Flag	Description
F2 0F 7C /r HADDPS *xmm1, xmm2/m128*	RM	V/V	SSE3	Horizontal add packed single-precision floating-point values from *xmm2/m128* to *xmm1*.
VEX.NDS.128.F2.0F.WIG 7C /r VHADDPS xmm1, xmm2, xmm3/m128	RVM	V/V	AVX	Horizontal add packed single-precision floating-point values from xmm2 and xmm3/mem.
VEX.NDS.256.F2.0F.WIG 7C /r VHADDPS ymm1, ymm2, ymm3/m256	RVM	V/V	AVX	Horizontal add packed single-precision floating-point values from ymm2 and ymm3/mem.

Instruction Operand Encoding

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

Description

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

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

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

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

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

See Figure 3-18 for HADDPS; see Figure 3-19 for VHADDPS.

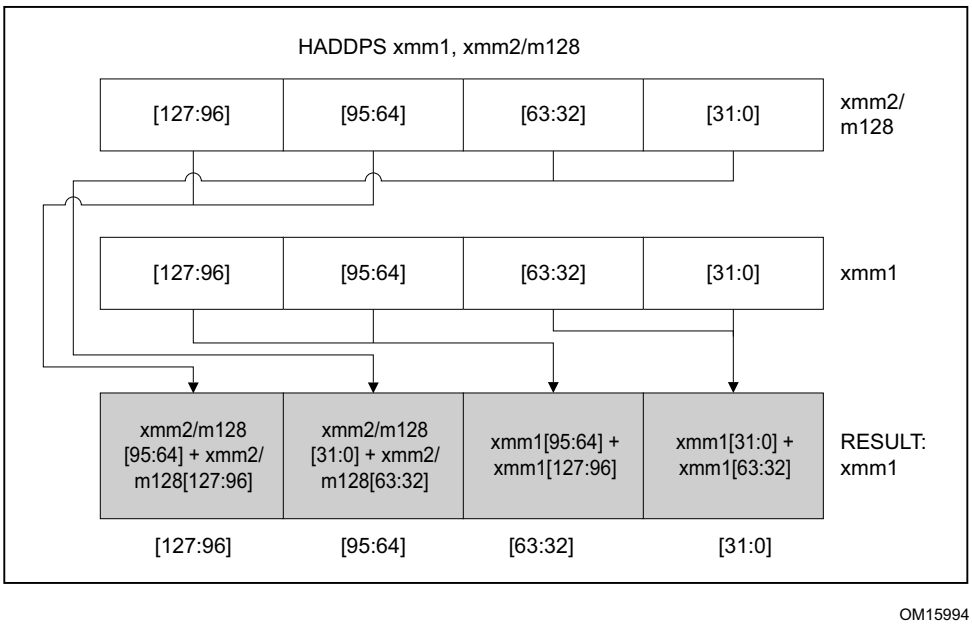


Figure 3-18. HADDPS—Packed Single-FP Horizontal Add

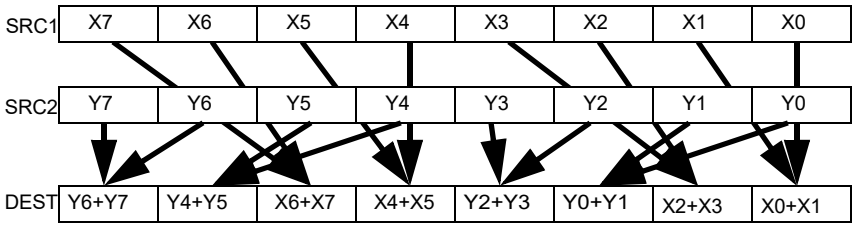


Figure 3-19. VHADDPS operation

128-bit Legacy SSE version: The second source can be an XMM register or an 128-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (MAXVL-1:128) of the corresponding YMM register destination are unmodified.

VEX.128 encoded version: the first source operand is an XMM register or 128-bit memory location. The destination operand is an XMM register. The upper bits (MAXVL-1:128) of the corresponding YMM register destination are zeroed.

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

Operation

HADDPS (128-bit Legacy SSE version)

```
DEST[31:0] ← SRC1[63:32] + SRC1[31:0]
DEST[63:32] ← SRC1[127:96] + SRC1[95:64]
DEST[95:64] ← SRC2[63:32] + SRC2[31:0]
DEST[127:96] ← SRC2[127:96] + SRC2[95:64]
DEST[MAXVL-1:128] (Unmodified)
```

VHADDPS (VEX.128 encoded version)

```
DEST[31:0] ← SRC1[63:32] + SRC1[31:0]
DEST[63:32] ← SRC1[127:96] + SRC1[95:64]
DEST[95:64] ← SRC2[63:32] + SRC2[31:0]
DEST[127:96] ← SRC2[127:96] + SRC2[95:64]
DEST[MAXVL-1:128] ← 0
```

VHADDPS (VEX.256 encoded version)

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

Intel C/C++ Compiler Intrinsic Equivalent

HADDPS: __m128 _mm_hadd_ps (__m128 a, __m128 b);

VHADDPS: __m256 _mm256_hadd_ps (__m256 a, __m256 b);

Exceptions

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

Numeric Exceptions

Overflow, Underflow, Invalid, Precision, Denormal

Other Exceptions

See Exceptions Type 2.

HLT—Halt

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
F4	HLT	ZO	Valid	Valid	Halt

Instruction Operand Encoding

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

Description

Stops instruction execution and places the processor in a HALT state. An enabled interrupt (including NMI and SMI), a debug exception, the BINIT# signal, the INIT# signal, or the RESET# signal will resume execution. If an interrupt (including NMI) is used to resume execution after a HLT instruction, the saved instruction pointer (CS:EIP) points to the instruction following the HLT instruction.

When a HLT instruction is executed on an Intel 64 or IA-32 processor supporting Intel Hyper-Threading Technology, only the logical processor that executes the instruction is halted. The other logical processors in the physical processor remain active, unless they are each individually halted by executing a HLT instruction.

The HLT instruction is a privileged instruction. When the processor is running in protected or virtual-8086 mode, the privilege level of a program or procedure must be 0 to execute the HLT instruction.

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

Operation

```
Enter Halt state;
```

Flags Affected

None

Protected Mode Exceptions

#GP(0)	If the current privilege level is not 0.
#UD	If the LOCK prefix is used.

Real-Address Mode Exceptions

None.

Virtual-8086 Mode Exceptions

Same exceptions as in protected mode.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

Same exceptions as in protected mode.

HSUBPD—Packed Double-FP Horizontal Subtract

Opcode/ Instruction	Op/ En	64/32-bit Mode	CPUID Feature Flag	Description
66 0F 7D /r HSUBPD *xmm1, xmm2/m128*	RM	V/V	SSE3	Horizontal subtract packed double-precision floating-point values from *xmm2/m128* to *xmm1*.
VEX.NDS.128.66.0F.WIG 7D /r VHSUBPD xmm1,xmm2, xmm3/m128	RVM	V/V	AVX	Horizontal subtract packed double-precision floating-point values from xmm2 and xmm3/mem.
VEX.NDS.256.66.0F.WIG 7D /r VHSUBPD ymm1, ymm2, ymm3/m256	RVM	V/V	AVX	Horizontal subtract packed double-precision floating-point values from ymm2 and ymm3/mem.

Instruction Operand Encoding

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

Description

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

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

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

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

See Figure 3-20 for HSUBPD; see Figure 3-21 for VHSUBPD.

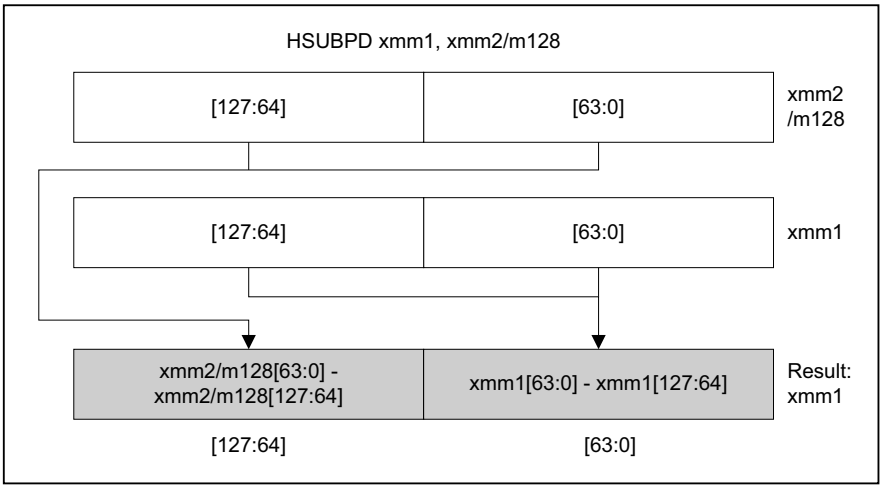


Figure 3-20. HSUBPD—Packed Double-FP Horizontal Subtract

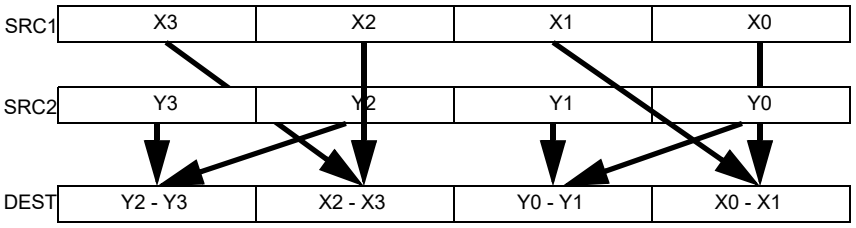


Figure 3-21. VHSUBPD operation

128-bit Legacy SSE version: The second source can be an XMM register or an 128-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (MAXVL-1:128) of the corresponding YMM register destination are unmodified.

VEX.128 encoded version: the first source operand is an XMM register or 128-bit memory location. The destination operand is an XMM register. The upper bits (MAXVL-1:128) of the corresponding YMM register destination are zeroed.

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

Operation

HSUBPD (128-bit Legacy SSE version)

```
DEST[63:0] ← SRC1[63:0] - SRC1[127:64]
DEST[127:64] ← SRC2[63:0] - SRC2[127:64]
DEST[MAXVL-1:128] (Unmodified)
```

VHSUBPD (VEX.128 encoded version)

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

VHSUBPD (VEX.256 encoded version)

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

Intel C/C++ Compiler Intrinsic Equivalent

HSUBPD: __m128d _mm_hsub_pd(__m128d a, __m128d b)

VHSUBPD: __m256d _mm256_hsub_pd (__m256d a, __m256d b);

Exceptions

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

Numeric Exceptions

Overflow, Underflow, Invalid, Precision, Denormal

Other Exceptions

See Exceptions Type 2.

HSUBPS—Packed Single-FP Horizontal Subtract

Opcode/ Instruction	Op/ En	64/32-bit Mode	CPUID Feature Flag	Description
F2 0F 7D /r HSUBPS *xmm1, xmm2/m128*	RM	V/V	SSE3	Horizontal subtract packed single-precision floating-point values from *xmm2/m128* to *xmm1*.
VEX.NDS.128.F2.0F.WIG 7D /r VHSUBPS xmm1, xmm2, xmm3/m128	RVM	V/V	AVX	Horizontal subtract packed single-precision floating-point values from xmm2 and xmm3/mem.
VEX.NDS.256.F2.0F.WIG 7D /r VHSUBPS ymm1, ymm2, ymm3/m256	RVM	V/V	AVX	Horizontal subtract packed single-precision floating-point values from ymm2 and ymm3/mem.

Instruction Operand Encoding

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

Description

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

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

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

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

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

See Figure 3-22 for HSUBPS; see Figure 3-23 for VHSUBPS.

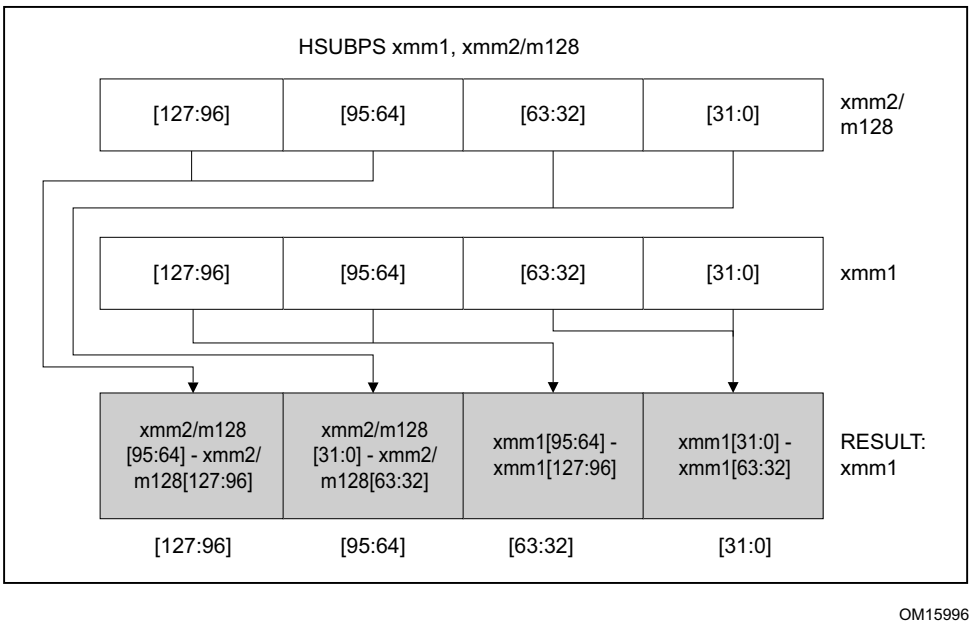


Figure 3-22. HSUBPS—Packed Single-FP Horizontal Subtract

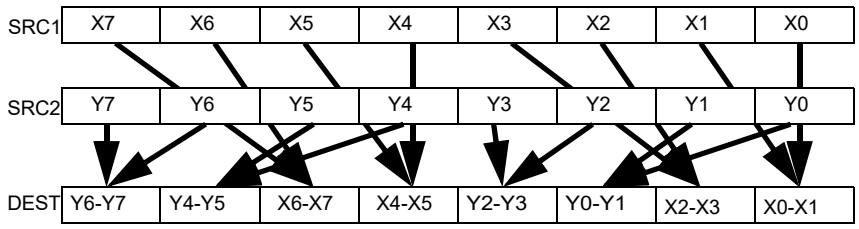


Figure 3-23. VHSUBPS operation

128-bit Legacy SSE version: The second source can be an XMM register or an 128-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (MAXVL-1:128) of the corresponding YMM register destination are unmodified.

VEX.128 encoded version: the first source operand is an XMM register or 128-bit memory location. The destination operand is an XMM register. The upper bits (MAXVL-1:128) of the corresponding YMM register destination are zeroed.

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

Operation

HSUBPS (128-bit Legacy SSE version)

```
DEST[31:0] ← SRC1[31:0] - SRC1[63:32]
DEST[63:32] ← SRC1[95:64] - SRC1[127:96]
DEST[95:64] ← SRC2[31:0] - SRC2[63:32]
DEST[127:96] ← SRC2[95:64] - SRC2[127:96]
DEST[MAXVL-1:128] (Unmodified)
```

VHSUBPS (VEX.128 encoded version)

```
DEST[31:0] ← SRC1[31:0] - SRC1[63:32]
DEST[63:32] ← SRC1[95:64] - SRC1[127:96]
DEST[95:64] ← SRC2[31:0] - SRC2[63:32]
DEST[127:96] ← SRC2[95:64] - SRC2[127:96]
DEST[MAXVL-1:128] ← 0
```

VHSUBPS (VEX.256 encoded version)

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

Intel C/C++ Compiler Intrinsic Equivalent

HSUBPS: __m128 _mm_hsub_ps(__m128 a, __m128 b);

VHSUBPS: __m256 _mm256_hsub_ps (__m256 a, __m256 b);

Exceptions

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

Numeric Exceptions

Overflow, Underflow, Invalid, Precision, Denormal

Other Exceptions

See Exceptions Type 2.

IDIV—Signed Divide

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
F6 /7	IDIV *r/m8*	M	Valid	Valid	Signed divide AX by *r/m*8, with result stored in: AL ← Quotient, AH ← Remainder.
REX + F6 /7	IDIV *r/m8**	M	Valid	N.E.	Signed divide AX by *r/m*8, with result stored in AL ← Quotient, AH ← Remainder.
F7 /7	IDIV *r/m16*	M	Valid	Valid	Signed divide DX:AX by *r/m16*, with result stored in AX ← Quotient, DX ← Remainder.
F7 /7	IDIV *r/m32*	M	Valid	Valid	Signed divide EDX:EAX by *r/m32*, with result stored in EAX ← Quotient, EDX ← Remainder.
REX.W + F7 /7	IDIV *r/m64*	M	Valid	N.E.	Signed divide RDX:RAX by *r/m64*, with result stored in RAX ← Quotient, RDX ← Remainder.

NOTES:

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

Instruction Operand Encoding

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

Description

Divides the (signed) value in the AX, DX:AX, or EDX:EAX (dividend) by the source operand (divisor) and stores the result in the AX (AH:AL), DX:AX, or EDX:EAX registers. The source operand can be a general-purpose register or a memory location. The action of this instruction depends on the operand size (dividend/divisor).

Non-integral results are truncated (chopped) towards 0. The remainder is always less than the divisor in magnitude. Overflow is indicated with the #DE (divide error) exception rather than with the CF flag.

In 64-bit mode, the instruction's default operation size is 32 bits. Use of the REX.R prefix permits access to additional registers (R8-R15). Use of the REX.W prefix promotes operation to 64 bits. In 64-bit mode when REX.W is applied, the instruction divides the signed value in RDX:RAX by the source operand. RAX contains a 64-bit quotient; RDX contains a 64-bit remainder.

See the summary chart at the beginning of this section for encoding data and limits. See Table 3-50.

Table 3-50. IDIV Results

Operand Size	Dividend	Divisor	Quotient	Remainder	Quotient Range
Word/byte	AX	r/m8	AL	AH	–128 to +127
Doubleword/word	DX:AX	r/m16	AX	DX	–32,768 to +32,767
Quadword/doubleword	EDX:EAX	r/m32	EAX	EDX	-2^{31} to $2^{31} - 1$
Doublequadword/ quadword	RDX:RAX	r/m64	RAX	RDX	-2^{63} to $2^{63} - 1$

Operation

```
IF SRC = 0
    THEN #DE; (* Divide error *)
FI;

IF OperandSize = 8 (* Word/byte operation *)
    THEN
        temp ← AX / SRC; (* Signed division *)
        IF (temp > 7FH) or (temp < 80H)
        (* If a positive result is greater than 7FH or a negative result is less than 80H *)
            THEN #DE; (* Divide error *)
            ELSE
                AL ← temp;
                AH ← AX SignedModulus SRC;
        FI;
    ELSE IF OperandSize = 16 (* Doubleword/word operation *)
        THEN
            temp ← DX:AX / SRC; (* Signed division *)
            IF (temp > 7FFFH) or (temp < 8000H)
            (* If a positive result is greater than 7FFFH
            or a negative result is less than 8000H *)
                THEN
                    #DE; (* Divide error *)
                ELSE
                    AX ← temp;
                    DX ← DX:AX SignedModulus SRC;
            FI;
        FI;
    ELSE IF OperandSize = 32 (* Quadword/doubleword operation *)
            temp ← EDX:EAX / SRC; (* Signed division *)
            IF (temp > 7FFFFFFFH) or (temp < 80000000H)
            (* If a positive result is greater than 7FFFFFFFH
            or a negative result is less than 80000000H *)
                THEN
                    #DE; (* Divide error *)
                ELSE
                    EAX ← temp;
                    EDX ← EDXE:AX SignedModulus SRC;
            FI;
        FI;
    ELSE IF OperandSize = 64 (* Doublequadword/quadword operation *)
            temp ← RDX:RAX / SRC; (* Signed division *)
            IF (temp > 7FFFFFFFFFFFFFFFH) or (temp < 8000000000000000H)
            (* If a positive result is greater than 7FFFFFFFFFFFFFFFH
            or a negative result is less than 8000000000000000H *)
                THEN
                    #DE; (* Divide error *)
                ELSE
                    RAX ← temp;
                    RDX ← RDE:RAX SignedModulus SRC;
            FI;
        FI;
FI;
```

Flags Affected

The CF, OF, SF, ZF, AF, and PF flags are undefined.

Protected Mode Exceptions

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

Real-Address Mode Exceptions

#DE	If the source operand (divisor) is 0.
	The signed result (quotient) is too large for the destination.
#GP	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS	If a memory operand effective address is outside the SS segment limit.
#UD	If the LOCK prefix is used.

Virtual-8086 Mode Exceptions

#DE	If the source operand (divisor) is 0.
	The signed result (quotient) is too large for the destination.
#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.
#UD	If the LOCK prefix is used.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form.
#DE	If the source operand (divisor) is 0
	If the quotient is too large for the designated register.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If the LOCK prefix is used.

IMUL—Signed Multiply

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
F6 /5	IMUL *r/m8**	M	Valid	Valid	AX← AL ∗ *r/m* byte.
F7 /5	IMUL *r/m16*	M	Valid	Valid	DX:AX ← AX ∗ *r/m* word.
F7 /5	IMUL *r/m32*	M	Valid	Valid	EDX:EAX ← EAX ∗ *r/m*32.
REX.W + F7 /5	IMUL *r/m64*	M	Valid	N.E.	RDX:RAX ← RAX ∗ *r/m*64.
0F AF /*r*	IMUL *r16, r/m16*	RM	Valid	Valid	word register ← word register ∗ *r/m*16.
0F AF /*r*	IMUL *r32, r/m32*	RM	Valid	Valid	doubleword register ← doubleword register ∗ *r/m32.*
REX.W + 0F AF /*r*	IMUL *r64, r/m64*	RM	Valid	N.E.	Quadword register ← Quadword register ∗ *r/m64.*
6B /*r ib*	IMUL *r16, r/m16, imm8*	RMI	Valid	Valid	word register ← *r/m16* ∗ sign-extended immediate byte.
6B /*r ib*	IMUL *r32, r/m32, imm8*	RMI	Valid	Valid	doubleword register ← *r/m32* ∗ sign-extended immediate byte.
REX.W + 6B /*r ib*	IMUL *r64, r/m64, imm8*	RMI	Valid	N.E.	Quadword register ← *r/m64* ∗ sign-extended immediate byte.
69 /*r iw*	IMUL *r16, r/m16, imm16*	RMI	Valid	Valid	word register ← *r/m16* ∗ immediate word.
69 /*r id*	IMUL *r32, r/m32, imm32*	RMI	Valid	Valid	doubleword register ← *r/m32* ∗ immediate doubleword.
REX.W + 69 /*r id*	IMUL *r64, r/m64, imm32*	RMI	Valid	N.E.	Quadword register ← *r/m64* ∗ immediate doubleword.

NOTES:

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

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
M	ModRM:r/m (r, w)	NA	NA	NA
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
RMI	ModRM:reg (r, w)	ModRM:r/m (r)	imm8/16/32	NA

Description

Performs a signed multiplication of two operands. This instruction has three forms, depending on the number of operands.

- **One-operand form** — This form is identical to that used by the MUL instruction. Here, the source operand (in a general-purpose register or memory location) is multiplied by the value in the AL, AX, EAX, or RAX register (depending on the operand size) and the product (twice the size of the input operand) is stored in the AX, DX:AX, EDX:EAX, or RDX:RAX registers, respectively.
- **Two-operand form** — With this form the destination operand (the first operand) is multiplied by the source operand (second operand). The destination operand is a general-purpose register and the source operand is an immediate value, a general-purpose register, or a memory location. The intermediate product (twice the size of the input operand) is truncated and stored in the destination operand location.
- **Three-operand form** — This form requires a destination operand (the first operand) and two source operands (the second and the third operands). Here, the first source operand (which can be a general-purpose register or a memory location) is multiplied by the second source operand (an immediate value). The intermediate product (twice the size of the first source operand) is truncated and stored in the destination operand (a general-purpose register).

When an immediate value is used as an operand, it is sign-extended to the length of the destination operand format.

The CF and OF flags are set when the signed integer value of the intermediate product differs from the sign extended operand-size-truncated product, otherwise the CF and OF flags are cleared.

The three forms of the IMUL instruction are similar in that the length of the product is calculated to twice the length of the operands. With the one-operand form, the product is stored exactly in the destination. With the two- and three- operand forms, however, the result is truncated to the length of the destination before it is stored in the destination register. Because of this truncation, the CF or OF flag should be tested to ensure that no significant bits are lost.

The two- and three-operand forms may also be used with unsigned operands because the lower half of the product is the same regardless if the operands are signed or unsigned. The CF and OF flags, however, cannot be used to determine if the upper half of the result is non-zero.

In 64-bit mode, the instruction's default operation size is 32 bits. Use of the REX.R prefix permits access to additional registers (R8-R15). Use of the REX.W prefix promotes operation to 64 bits. Use of REX.W modifies the three forms of the instruction as follows.

- **One-operand form** —The source operand (in a 64-bit general-purpose register or memory location) is multiplied by the value in the RAX register and the product is stored in the RDX:RAX registers.
- **Two-operand form** — The source operand is promoted to 64 bits if it is a register or a memory location. The destination operand is promoted to 64 bits.
- **Three-operand form** — The first source operand (either a register or a memory location) and destination operand are promoted to 64 bits. If the source operand is an immediate, it is sign extended to 64 bits.

Operation

```
IF (NumberOfOperands = 1)
    THEN IF (OperandSize = 8)
        THEN
            TMP_XP ← AL * SRC (* Signed multiplication; TMP_XP is a signed integer at twice the width of the SRC *);
            AX ← TMP_XP[15:0];
            IF SignExtend(TMP_XP[7:0]) = TMP_XP
                THEN CF ← 0; OF ← 0;
                ELSE CF ← 1; OF ← 1; FI;
        ELSE IF OperandSize = 16
            THEN
                TMP_XP ← AX * SRC (* Signed multiplication; TMP_XP is a signed integer at twice the width of the SRC *)
                DX:AX ← TMP_XP[31:0];
                IF SignExtend(TMP_XP[15:0]) = TMP_XP
                    THEN CF ← 0; OF ← 0;
                    ELSE CF ← 1; OF ← 1; FI;
            ELSE IF OperandSize = 32
                THEN
                    TMP_XP ← EAX * SRC (* Signed multiplication; TMP_XP is a signed integer at twice the width of the SRC*)
                    EDX:EAX ← TMP_XP[63:0];
                    IF SignExtend(TMP_XP[31:0]) = TMP_XP
                        THEN CF ← 0; OF ← 0;
                        ELSE CF ← 1; OF ← 1; FI;
                ELSE (* OperandSize = 64 *)
                    TMP_XP ← RAX * SRC (* Signed multiplication; TMP_XP is a signed integer at twice the width of the SRC *)
                    EDX:EAX ← TMP_XP[127:0];
                    IF SignExtend(TMP_XP[63:0]) = TMP_XP
                        THEN CF ← 0; OF ← 0;
                        ELSE CF ← 1; OF ← 1; FI;
                FI;
```

```
        FI;
    ELSE IF (NumberOfOperands = 2)
        THEN
            TMP_XP ← DEST * SRC (* Signed multiplication; TMP_XP is a signed integer at twice the width of the SRC *)
            DEST ← TruncateToOperandSize(TMP_XP);
            IF SignExtend(DEST) ≠ TMP_XP
                THEN CF ← 1; OF ← 1;
                ELSE CF ← 0; OF ← 0; FI;
        ELSE (* NumberOfOperands = 3 *)
            TMP_XP ← SRC1 * SRC2 (* Signed multiplication; TMP_XP is a signed integer at twice the width of the SRC1 *)
            DEST ← TruncateToOperandSize(TMP_XP);
            IF SignExtend(DEST) ≠ TMP_XP
                THEN CF ← 1; OF ← 1;
                ELSE CF ← 0; OF ← 0; FI;
    FI;
FI;
```

Flags Affected

For the one operand form of the instruction, the CF and OF flags are set when significant bits are carried into the upper half of the result and cleared when the result fits exactly in the lower half of the result. For the two- and three-operand forms of the instruction, the CF and OF flags are set when the result must be truncated to fit in the destination operand size and cleared when the result fits exactly in the destination operand size. The SF, ZF, AF, and PF flags are undefined.

Protected Mode Exceptions

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

Real-Address Mode Exceptions

#GP	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS	If a memory operand effective address is outside the SS segment limit.
#UD	If the LOCK prefix is used.

Virtual-8086 Mode Exceptions

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

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If the LOCK prefix is used.

IN—Input from Port

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
E4 *ib*	IN AL, i*mm8*	I	Valid	Valid	Input byte from *imm8* I/O port address into AL.
E5 *ib*	IN AX, i*mm8*	I	Valid	Valid	Input word from *imm8* I/O port address into AX.
E5 *ib*	IN EAX, i*mm8*	I	Valid	Valid	Input dword from *imm8* I/O port address into EAX.
EC	IN AL,DX	ZO	Valid	Valid	Input byte from I/O port in DX into AL.
ED	IN AX,DX	ZO	Valid	Valid	Input word from I/O port in DX into AX.
ED	IN EAX,DX	ZO	Valid	Valid	Input doubleword from I/O port in DX into EAX.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
I	imm8	NA	NA	NA
ZO	NA	NA	NA	NA

Description

Copies the value from the I/O port specified with the second operand (source operand) to the destination operand (first operand). The source operand can be a byte-immediate or the DX register; the destination operand can be register AL, AX, or EAX, depending on the size of the port being accessed (8, 16, or 32 bits, respectively). Using the DX register as a source operand allows I/O port addresses from 0 to 65,535 to be accessed; using a byte immediate allows I/O port addresses 0 to 255 to be accessed.

When accessing an 8-bit I/O port, the opcode determines the port size; when accessing a 16- and 32-bit I/O port, the operand-size attribute determines the port size. At the machine code level, I/O instructions are shorter when accessing 8-bit I/O ports. Here, the upper eight bits of the port address will be 0.

This instruction is only useful for accessing I/O ports located in the processor's I/O address space. See Chapter 18, "Input/Output," in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*, for more information on accessing I/O ports in the I/O address space.

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

Operation

```
IF ((PE = 1) and ((CPL > IOPL) or (VM = 1)))
    THEN (* Protected mode with CPL > IOPL or virtual-8086 mode *)
        IF (Any I/O Permission Bit for I/O port being accessed = 1)
            THEN (* I/O operation is not allowed *)
                #GP(0);
            ELSE ( * I/O operation is allowed *)
                DEST ← SRC; (* Read from selected I/O port *)
        FI;
    ELSE (Real Mode or Protected Mode with CPL ≤ IOPL *)
        DEST ← SRC; (* Read from selected I/O port *)
FI;
```

Flags Affected

None

Protected Mode Exceptions

#GP(0)	If the CPL is greater than (has less privilege) the I/O privilege level (IOPL) and any of the corresponding I/O permission bits in TSS for the I/O port being accessed is 1.
#UD	If the LOCK prefix is used.

Real-Address Mode Exceptions

#UD	If the LOCK prefix is used.

Virtual-8086 Mode Exceptions

#GP(0)	If any of the I/O permission bits in the TSS for the I/O port being accessed is 1.
#PF(fault-code)	If a page fault occurs.
#UD	If the LOCK prefix is used.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#GP(0)	If the CPL is greater than (has less privilege) the I/O privilege level (IOPL) and any of the corresponding I/O permission bits in TSS for the I/O port being accessed is 1.
#UD	If the LOCK prefix is used.

INC—Increment by 1

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
FE /0	INC *r/m8*	M	Valid	Valid	Increment *r/m* byte by 1.
REX + FE /0	INC *r/m8**	M	Valid	N.E.	Increment *r/m* byte by 1.
FF /0	INC *r/m16*	M	Valid	Valid	Increment *r/m* word by 1.
FF /0	INC *r/m32*	M	Valid	Valid	Increment *r/m* doubleword by 1.
REX.W + FF /0	INC *r/m64*	M	Valid	N.E.	Increment *r/m* quadword by 1.
40+ *rw***	INC *r16*	O	N.E.	Valid	Increment word register by 1.
40+ *rd*	INC *r32*	O	N.E.	Valid	Increment doubleword register by 1.

NOTES:

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

** 40H through 47H are REX prefixes in 64-bit mode.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
M	ModRM:r/m (r, w)	NA	NA	NA
O	opcode + rd (r, w)	NA	NA	NA

Description

Adds 1 to the destination operand, while preserving the state of the CF flag. The destination operand can be a register or a memory location. This instruction allows a loop counter to be updated without disturbing the CF flag. (Use a ADD instruction with an immediate operand of 1 to perform an increment operation that does updates the CF flag.)

This instruction can be used with a LOCK prefix to allow the instruction to be executed atomically.

In 64-bit mode, INC r16 and INC r32 are not encodable (because opcodes 40H through 47H are REX prefixes). Otherwise, the instruction's 64-bit mode default operation size is 32 bits. Use of the REX.R prefix permits access to additional registers (R8-R15). Use of the REX.W prefix promotes operation to 64 bits.

Operation

```
DEST ← DEST + 1;
```

AFlags Affected

The CF flag is not affected. The OF, SF, ZF, AF, and PF flags are set according to the result.

Protected Mode Exceptions

#GP(0)	If the destination operand is located in a non-writable segment.
	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
	If the DS, ES, FS, or GS register is used to access memory and it contains a NULLsegment selector.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If the LOCK prefix is used but the destination is not a memory operand.

Real-Address Mode Exceptions

#GP	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS	If a memory operand effective address is outside the SS segment limit.
#UD	If the LOCK prefix is used but the destination is not a memory operand.

Virtual-8086 Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.
#UD	If the LOCK prefix is used but the destination is not a memory operand.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If the LOCK prefix is used but the destination is not a memory operand.

INS/INSB/INSW/INSD—Input from Port to String

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
6C	INS *m8*, DX	ZO	Valid	Valid	Input byte from I/O port specified in DX into memory location specified in ES:(E)DI or RDI.*
6D	INS *m16*, DX	ZO	Valid	Valid	Input word from I/O port specified in DX into memory location specified in ES:(E)DI or RDI.[1]
6D	INS *m32*, DX	ZO	Valid	Valid	Input doubleword from I/O port specified in DX into memory location specified in ES:(E)DI or RDI.[1]
6C	INSB	ZO	Valid	Valid	Input byte from I/O port specified in DX into memory location specified with ES:(E)DI or RDI.[1]
6D	INSW	ZO	Valid	Valid	Input word from I/O port specified in DX into memory location specified in ES:(E)DI or RDI.[1]
6D	INSD	ZO	Valid	Valid	Input doubleword from I/O port specified in DX into memory location specified in ES:(E)DI or RDI.[1]

NOTES:

* In 64-bit mode, only 64-bit (RDI) and 32-bit (EDI) address sizes are supported. In non-64-bit mode, only 32-bit (EDI) and 16-bit (DI) address sizes are supported.

Instruction Operand Encoding

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

Description

Copies the data from the I/O port specified with the source operand (second operand) to the destination operand (first operand). The source operand is an I/O port address (from 0 to 65,535) that is read from the DX register. The destination operand is a memory location, the address of which is read from either the ES:DI, ES:EDI or the RDI registers (depending on the address-size attribute of the instruction, 16, 32 or 64, respectively). (The ES segment cannot be overridden with a segment override prefix.) The size of the I/O port being accessed (that is, the size of the source and destination operands) is determined by the opcode for an 8-bit I/O port or by the operand-size attribute of the instruction for a 16- or 32-bit I/O port.

At the assembly-code level, two forms of this instruction are allowed: the "explicit-operands" form and the "no-operands" form. The explicit-operands form (specified with the INS mnemonic) allows the source and destination operands to be specified explicitly. Here, the source operand must be "DX," and the destination operand should be a symbol that indicates the size of the I/O port and the destination address. This explicit-operands form is provided to allow documentation; however, note that the documentation provided by this form can be misleading. That is, the destination operand symbol must specify the correct **type** (size) of the operand (byte, word, or doubleword), but it does not have to specify the correct **location**. The location is always specified by the ES:(E)DI registers, which must be loaded correctly before the INS instruction is executed.

The no-operands form provides "short forms" of the byte, word, and doubleword versions of the INS instructions. Here also DX is assumed by the processor to be the source operand and ES:(E)DI is assumed to be the destination operand. The size of the I/O port is specified with the choice of mnemonic: INSB (byte), INSW (word), or INSD (doubleword).

After the byte, word, or doubleword is transfer from the I/O port to the memory location, the DI/EDI/RDI register is incremented or decremented automatically according to the setting of the DF flag in the EFLAGS register. (If the DF flag is 0, the (E)DI register is incremented; if the DF flag is 1, the (E)DI register is decremented.) The (E)DI register is incremented or decremented by 1 for byte operations, by 2 for word operations, or by 4 for doubleword operations.

The INS, INSB, INSW, and INSD instructions can be preceded by the REP prefix for block input of ECX bytes, words, or doublewords. See "REP/REPE/REPZ /REPNE/REPNZ—Repeat String Operation Prefix" in Chapter 4 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2B*, for a description of the REP prefix.

These instructions are only useful for accessing I/O ports located in the processor's I/O address space. See Chapter 18, "Input/Output," in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*, for more information on accessing I/O ports in the I/O address space.

In 64-bit mode, default address size is 64 bits, 32 bit address size is supported using the prefix 67H. The address of the memory destination is specified by RDI or EDI. 16-bit address size is not supported in 64-bit mode. The operand size is not promoted.

These instructions may read from the I/O port without writing to the memory location if an exception or VM exit occurs due to the write (e.g. #PF). If this would be problematic, for example because the I/O port read has side-effects, software should ensure the write to the memory location does not cause an exception or VM exit.

Operation

```
IF ((PE = 1) and ((CPL > IOPL) or (VM = 1)))
    THEN (* Protected mode with CPL > IOPL or virtual-8086 mode *)
        IF (Any I/O Permission Bit for I/O port being accessed = 1)
            THEN (* I/O operation is not allowed *)
                #GP(0);
            ELSE (* I/O operation is allowed *)
                DEST ← SRC; (* Read from I/O port *)
        FI;
    ELSE (Real Mode or Protected Mode with CPL IOPL *)
        DEST ← SRC; (* Read from I/O port *)
FI;
```

Non-64-bit Mode:

```
IF (Byte transfer)
    THEN IF DF = 0
        THEN (E)DI ← (E)DI + 1;
        ELSE (E)DI ← (E)DI – 1; FI;
    ELSE IF (Word transfer)
        THEN IF DF = 0
            THEN (E)DI ← (E)DI + 2;
            ELSE (E)DI ← (E)DI – 2; FI;
        ELSE (* Doubleword transfer *)
            THEN IF DF = 0
                THEN (E)DI ← (E)DI + 4;
                ELSE (E)DI ← (E)DI – 4; FI;
        FI;
FI;
```

FI64-bit Mode:

```
IF (Byte transfer)
    THEN IF DF = 0
        THEN (E|R)DI ← (E|R)DI + 1;
        ELSE (E|R)DI ← (E|R)DI – 1; FI;
    ELSE IF (Word transfer)
        THEN IF DF = 0
            THEN (E)DI ← (E)DI + 2;
            ELSE (E)DI ← (E)DI – 2; FI;
        ELSE (* Doubleword transfer *)
```

```
            THEN IF DF = 0
                THEN (E|R)DI ← (E|R)DI + 4;
                ELSE (E|R)DI ← (E|R)DI – 4; FI;
        FI;
FI;
```

Flags Affected

None

Protected Mode Exceptions

#GP(0)	If the CPL is greater than (has less privilege) the I/O privilege level (IOPL) and any of the corresponding I/O permission bits in TSS for the I/O port being accessed is 1.
	If the destination is located in a non-writable segment.
	If an illegal memory operand effective address in the ES segments is given.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If the LOCK prefix is used.

Real-Address Mode Exceptions

#GP	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS	If a memory operand effective address is outside the SS segment limit.
#UD	If the LOCK prefix is used.

Virtual-8086 Mode Exceptions

#GP(0)	If any of the I/O permission bits in the TSS for the I/O port being accessed is 1.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.
#UD	If the LOCK prefix is used.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the CPL is greater than (has less privilege) the I/O privilege level (IOPL) and any of the corresponding I/O permission bits in TSS for the I/O port being accessed is 1.
	If the memory address is in a non-canonical form.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If the LOCK prefix is used.

INSERTPS—Insert Scalar Single-Precision Floating-Point Value

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 3A 21 /r ib INSERTPS xmm1, xmm2/m32, imm8	A	V/V	SSE4_1	Insert a single-precision floating-point value selected by imm8 from xmm2/m32 into xmm1 at the specified destination element specified by imm8 and zero out destination elements in xmm1 as indicated in imm8.
VEX.NDS.128.66.0F3A.WIG 21 /r ib VINSERTPS xmm1, xmm2, xmm3/m32, imm8	B	V/V	AVX	Insert a single-precision floating-point value selected by imm8 from xmm3/m32 and merge with values in xmm2 at the specified destination element specified by imm8 and write out the result and zero out destination elements in xmm1 as indicated in imm8.
EVEX.NDS.128.66.0F3A.W0 21 /r ib VINSERTPS xmm1, xmm2, xmm3/m32, imm8	C	V/V	AVX512F	Insert a single-precision floating-point value selected by imm8 from xmm3/m32 and merge with values in xmm2 at the specified destination element specified by imm8 and write out the result and zero out destination elements in xmm1 as indicated in imm8.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	NA	ModRM:reg (r, w)	ModRM:r/m (r)	Imm8	NA
B	NA	ModRM:reg (w)	VEX.vvvv	ModRM:r/m (r)	Imm8
C	Tuple1 Scalar	ModRM:reg (w)	EVEX.vvvv	ModRM:r/m (r)	Imm8

Description

(register source form)

Copy a single-precision scalar floating-point element into a 128-bit vector register. The immediate operand has three fields, where the ZMask bits specify which elements of the destination will be set to zero, the Count_D bits specify which element of the destination will be overwritten with the scalar value, and for vector register sources the Count_S bits specify which element of the source will be copied. When the scalar source is a memory operand the Count_S bits are ignored.

(memory source form)

Load a floating-point element from a 32-bit memory location and destination operand it into the first source at the location indicated by the Count_D bits of the immediate operand. Store in the destination and zero out destination elements based on the ZMask bits of the immediate operand.

128-bit Legacy SSE version: The first source register is an XMM register. The second source operand is either an XMM register or a 32-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (MAXVL-1:128) of the corresponding register destination are unmodified.

VEX.128 and EVEX encoded version: The destination and first source register is an XMM register. The second source operand is either an XMM register or a 32-bit memory location. The upper bits (MAXVL-1:128) of the corresponding register destination are zeroed.

If VINSERTPS is encoded with VEX.L= 1, an attempt to execute the instruction encoded with VEX.L= 1 will cause an #UD exception.

Operation

VINSERTPS (VEX.128 and EVEX encoded version)

```
IF (SRC = REG) THEN COUNT_S ← imm8[7:6]
    ELSE COUNT_S ← 0
COUNT_D ← imm8[5:4]
ZMASK ← imm8[3:0]
CASE (COUNT_S) OF
    0: TMP ← SRC2[31:0]
    1: TMP ← SRC2[63:32]
    2: TMP ← SRC2[95:64]
    3: TMP ← SRC2[127:96]
ESAC;
CASE (COUNT_D) OF
    0: TMP2[31:0] ← TMP
        TMP2[127:32] ← SRC1[127:32]
    1: TMP2[63:32] ← TMP
        TMP2[31:0] ← SRC1[31:0]
        TMP2[127:64] ← SRC1[127:64]
    2: TMP2[95:64] ← TMP
        TMP2[63:0] ← SRC1[63:0]
        TMP2[127:96] ← SRC1[127:96]
    3: TMP2[127:96] ← TMP
        TMP2[95:0] ← SRC1[95:0]
ESAC;

IF (ZMASK[0] = 1) THEN DEST[31:0] ← 00000000H
    ELSE DEST[31:0] ← TMP2[31:0]
IF (ZMASK[1] = 1) THEN DEST[63:32] ← 00000000H
    ELSE DEST[63:32] ← TMP2[63:32]
IF (ZMASK[2] = 1) THEN DEST[95:64] ← 00000000H
    ELSE DEST[95:64] ← TMP2[95:64]
IF (ZMASK[3] = 1) THEN DEST[127:96] ← 00000000H
    ELSE DEST[127:96] ← TMP2[127:96]
DEST[MAXVL-1:128] ← 0
```

INSERTPS (128-bit Legacy SSE version)

```
IF (SRC = REG) THEN COUNT_S ←imm8[7:6]
    ELSE COUNT_S ←0
COUNT_D ←imm8[5:4]
ZMASK ←imm8[3:0]
CASE (COUNT_S) OF
    0: TMP ←SRC[31:0]
    1: TMP ←SRC[63:32]
    2: TMP ←SRC[95:64]
    3: TMP ←SRC[127:96]
ESAC;

CASE (COUNT_D) OF
    0: TMP2[31:0] ←TMP
        TMP2[127:32] ←DEST[127:32]
    1: TMP2[63:32] ←TMP
        TMP2[31:0] ←DEST[31:0]
        TMP2[127:64] ←DEST[127:64]
    2: TMP2[95:64] ←TMP
```

```
            TMP2[63:0] ←DEST[63:0]
            TMP2[127:96] ←DEST[127:96]
    3: TMP2[127:96] ←TMP
            TMP2[95:0] ←DEST[95:0]
ESAC;

IF (ZMASK[0] = 1) THEN DEST[31:0] ←00000000H
    ELSE DEST[31:0] ←TMP2[31:0]
IF (ZMASK[1] = 1) THEN DEST[63:32] ←00000000H
    ELSE DEST[63:32] ←TMP2[63:32]
IF (ZMASK[2] = 1) THEN DEST[95:64] ←00000000H
    ELSE DEST[95:64] ←TMP2[95:64]
IF (ZMASK[3] = 1) THEN DEST[127:96] ←00000000H
    ELSE DEST[127:96] ←TMP2[127:96]
DEST[MAXVL-1:128] (Unmodified)
```

Intel C/C++ Compiler Intrinsic Equivalent

VINSERTPS __m128 _mm_insert_ps(__m128 dst, __m128 src, const int nidx);
INSETRTPS __m128 _mm_insert_ps(__m128 dst, __m128 src, const int nidx);

SIMD Floating-Point Exceptions

None

Other Exceptions

Non-EVEX-encoded instruction, see Exceptions Type 5; additionally

#UD	If VEX.L = 0.

EVEX-encoded instruction, see Exceptions Type E9NF.

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

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
CC	INT 3	ZO	Valid	Valid	Interrupt 3—trap to debugger.
CD *ib*	INT *imm8*	I	Valid	Valid	Interrupt vector specified by immediate byte.
CE	INTO	ZO	Invalid	Valid	Interrupt 4—if overflow flag is 1.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
ZO	NA	NA	NA	NA
I	imm8	NA	NA	NA

Description

The INT *n* instruction generates a call to the interrupt or exception handler specified with the destination operand (see the section titled "Interrupts and Exceptions" in Chapter 6 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*). The destination operand specifies a vector from 0 to 255, encoded as an 8-bit unsigned intermediate value. Each vector provides an index to a gate descriptor in the IDT. The first 32 vectors are reserved by Intel for system use. Some of these vectors are used for internally generated exceptions.

The INT *n* instruction is the general mnemonic for executing a software-generated call to an interrupt handler. The INTO instruction is a special mnemonic for calling overflow exception (#OF), exception 4. The overflow interrupt checks the OF flag in the EFLAGS register and calls the overflow interrupt handler if the OF flag is set to 1. (The INTO instruction cannot be used in 64-bit mode.)

The INT 3 instruction generates a special one byte opcode (CC) that is intended for calling the debug exception handler. (This one byte form is valuable because it can be used to replace the first byte of any instruction with a breakpoint, including other one byte instructions, without over-writing other code).

An interrupt generated by INTO or INT3 (CC) differs from one generated by INT *n* in the following ways:

- The normal IOPL checks do not occur in virtual-8086 mode. The interrupt is taken (without fault) with any IOPL value.
- The interrupt redirection enabled by the virtual-8086 mode extensions (VME) does not occur. The interrupt is always handled by a protected-mode handler.

(These features do not pertain to CD03, the "normal" 2-byte opcode for INT 3. Intel and Microsoft assemblers will not generate the CD03 opcode from any mnemonic, but this opcode can be created by direct numeric code definition or by self-modifying code.)

The action of the INT *n* instruction (including the INTO and INT 3 instructions) is similar to that of a far call made with the CALL instruction. The primary difference is that with the INT *n* instruction, the EFLAGS register is pushed onto the stack before the return address. (The return address is a far address consisting of the current values of the CS and EIP registers.) Returns from interrupt procedures are handled with the IRET instruction, which pops the EFLAGS information and return address from the stack.

The vector specifies an interrupt descriptor in the interrupt descriptor table (IDT); that is, it provides index into the IDT. The selected interrupt descriptor in turn contains a pointer to an interrupt or exception handler procedure. In protected mode, the IDT contains an array of 8-byte descriptors, each of which is an interrupt gate, trap gate, or task gate. In real-address mode, the IDT is an array of 4-byte far pointers (2-byte code segment selector and a 2-byte instruction pointer), each of which point directly to a procedure in the selected segment. (Note that in real-address mode, the IDT is called the **interrupt vector table**, and its pointers are called interrupt vectors.)

The following decision table indicates which action in the lower portion of the table is taken given the conditions in the upper portion of the table. Each Y in the lower section of the decision table represents a procedure defined in the "Operation" section for this instruction (except #GP).

Table 3-51. Decision Table

PE	0	1	1	1	1	1	1	1
VM	–	–	–	–	–	0	1	1
IOPL	–	–	–	–	–	–	<3	=3
DPL/CPL RELATIONSHIP	–	DPL< CPL	–	DPL> CPL	DPL= CPL or C	DPL< CPL & NC	–	–
INTERRUPT TYPE	–	S/W	–	–	–	–	–	–
GATE TYPE	–	–	Task	Trap or Interrupt	Trap or Interrupt	Trap or Interrupt	Trap or Interrupt	Trap or Interrupt
REAL-ADDRESS-MODE	Y							
PROTECTED-MODE		Y	Y	Y	Y	Y	Y	Y
TRAP-OR-INTERRUPT-GATE				Y	Y	Y	Y	Y
INTER-PRIVILEGE-LEVEL-INTERRUPT						Y		
INTRA-PRIVILEGE-LEVEL-INTERRUPT					Y			
INTERRUPT-FROM-VIRTUAL-8086-MODE								Y
TASK-GATE			Y					
#GP		Y		Y			Y	

NOTES:

– Don't Care.

Y Yes, action taken.

Blank Action not taken.

When the processor is executing in virtual-8086 mode, the IOPL determines the action of the INT n instruction. If the IOPL is less than 3, the processor generates a #GP(selector) exception; if the IOPL is 3, the processor executes a protected mode interrupt to privilege level 0. The interrupt gate's DPL must be set to 3 and the target CPL of the interrupt handler procedure must be 0 to execute the protected mode interrupt to privilege level 0.

The interrupt descriptor table register (IDTR) specifies the base linear address and limit of the IDT. The initial base address value of the IDTR after the processor is powered up or reset is 0.

Operation

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

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

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

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

```
IF PE = 0
    THEN
        GOTO REAL-ADDRESS-MODE;
    ELSE (* PE = 1 *)
        IF (EFLAGS.VM = 1 AND CR4.VME = 0 AND IOPL < 3 AND INT n)
            THEN
                #GP(0); (* Bit 0 of error code is 0 because INT n *)
            ELSE
                IF (EFLAGS.VM = 1 AND CR4.VME = 1 AND INT n)
                    THEN
                        Consult bit n of the software interrupt redirection bit map in the TSS;
                        IF bit n is clear
                            THEN (* redirect interrupt to 8086 program interrupt handler *)
                                Push EFLAGS[15:0];      (* if IOPL < 3, save VIF in IF position and save IOPL position as 3 *)
                                Push CS;
                                Push IP;
                                IF IOPL = 3
                                    THEN IF ← 0; (* Clear interrupt flag *)
                                    ELSE VIF ← 0; (* Clear virtual interrupt flag *)
                                FI;
                                TF ← 0; (* Clear trap flag *)
                                load CS and EIP (lower 16 bits only) from entry n in interrupt vector table referenced from TSS;
                            ELSE
                                IF IOPL = 3
                                    THEN GOTO PROTECTED-MODE;
                                    ELSE #GP(0); (* Bit 0 of error code is 0 because INT n *)
                                FI;
                        FI;
                    ELSE (* Protected mode, IA-32e mode, or virtual-8086 mode interrupt *)
                        IF (IA32_EFER.LMA = 0)
                            THEN (* Protected mode, or virtual-8086 mode interrupt *)
                                GOTO PROTECTED-MODE;
                            ELSE (* IA-32e mode interrupt *)
                            GOTO IA-32e-MODE;
                        FI;
                FI;
        FI;
FI;
REAL-ADDRESS-MODE:
    IF ((vector_number « 2) + 3) is not within IDT limit
        THEN #GP; FI;
    IF stack not large enough for a 6-byte return information
        THEN #SS; FI;
    Push (EFLAGS[15:0]);
    IF ← 0; (* Clear interrupt flag *)
    TF ← 0; (* Clear trap flag *)
    AC ← 0; (* Clear AC flag *)
    Push(CS);
    Push(IP);
    (* No error codes are pushed in real-address mode*)
    CS ← IDT(Descriptor (vector_number « 2), selector));
    EIP ← IDT(Descriptor (vector_number « 2), offset)); (* 16 bit offset AND 0000FFFFH *)
END;
PROTECTED-MODE:
```

```
    IF ((vector_number « 3) + 7) is not within IDT limits
    or selected IDT descriptor is not an interrupt-, trap-, or task-gate type
        THEN #GP(error_code(vector_number,1,EXT)); FI;
        (* idt operand to error_code set because vector is used *)
    IF software interrupt (* Generated by INT n, INT3, or INTO *)
        THEN
            IF gate DPL < CPL (* PE = 1, DPL < CPL, software interrupt *)
                THEN #GP(error_code(vector_number,1,0)); FI;
                (* idt operand to error_code set because vector is used *)
                (* ext operand to error_code is 0 because INT n, INT3, or INTO*)
    FI;
    IF gate not present
        THEN #NP(error_code(vector_number,1,EXT)); FI;
        (* idt operand to error_code set because vector is used *)
    IF task gate (* Specified in the selected interrupt table descriptor *)
        THEN GOTO TASK-GATE;
        ELSE GOTO TRAP-OR-INTERRUPT-GATE; (* PE = 1, trap/interrupt gate *)
    FI;
END;
IA-32e-MODE:
    IF INTO and CS.L = 1 (64-bit mode)
        THEN #UD;
    FI;
    IF ((vector_number « 4) + 15) is not in IDT limits
    or selected IDT descriptor is not an interrupt-, or trap-gate type
        THEN #GP(error_code(vector_number,1,EXT));
        (* idt operand to error_code set because vector is used *)
    FI;
    IF software interrupt (* Generated by INT n, INT 3, or INTO *)
        THEN
            IF gate DPL < CPL (* PE = 1, DPL < CPL, software interrupt *)
                THEN #GP(error_code(vector_number,1,0));
                (* idt operand to error_code set because vector is used *)
                (* ext operand to error_code is 0 because INT n, INT3, or INTO*)
            FI;
    FI;
    IF gate not present
        THEN #NP(error_code(vector_number,1,EXT));
        (* idt operand to error_code set because vector is used *)
    FI;
    GOTO TRAP-OR-INTERRUPT-GATE; (* Trap/interrupt gate *)
END;
TASK-GATE: (* PE = 1, task gate *)
    Read TSS selector in task gate (IDT descriptor);
        IF local/global bit is set to local or index not within GDT limits
            THEN #GP(error_code(TSS selector,0,EXT)); FI;
            (* idt operand to error_code is 0 because selector is used *)
        Access TSS descriptor in GDT;
        IF TSS descriptor specifies that the TSS is busy (low-order 5 bits set to 00001)
            THEN #GP(TSS selector,0,EXT)); FI;
            (* idt operand to error_code is 0 because selector is used *)
        IF TSS not present
            THEN #NP(TSS selector,0,EXT)); FI;
            (* idt operand to error_code is 0 because selector is used *)
```

```
    SWITCH-TASKS (with nesting) to TSS;
    IF interrupt caused by fault with error code
        THEN
            IF stack limit does not allow push of error code
                THEN #SS(EXT); FI;
            Push(error code);
    FI;
    IF EIP not within code segment limit
        THEN #GP(EXT); FI;
END;
TRAP-OR-INTERRUPT-GATE:
    Read new code-segment selector for trap or interrupt gate (IDT descriptor);
    IF new code-segment selector is NULL
        THEN #GP(EXT); FI; (* Error code contains NULL selector *)
    IF new code-segment selector is not within its descriptor table limits
        THEN #GP(error_code(new code-segment selector,0,EXT)); FI;
        (* idt operand to error_code is 0 because selector is used *)
    Read descriptor referenced by new code-segment selector;
    IF descriptor does not indicate a code segment or new code-segment DPL > CPL
        THEN #GP(error_code(new code-segment selector,0,EXT)); FI;
        (* idt operand to error_code is 0 because selector is used *)
    IF new code-segment descriptor is not present,
        THEN #NP(error_code(new code-segment selector,0,EXT)); FI;
        (* idt operand to error_code is 0 because selector is used *)
    IF new code segment is non-conforming with DPL < CPL
        THEN
            IF VM = 0
                THEN
                    GOTO INTER-PRIVILEGE-LEVEL-INTERRUPT;
                    (* PE = 1, VM = 0, interrupt or trap gate, nonconforming code segment,
                    DPL < CPL *)
                ELSE (* VM = 1 *)
                    IF new code-segment DPL ≠ 0
                        THEN #GP(error_code(new code-segment selector,0,EXT));
                        (* idt operand to error_code is 0 because selector is used *)
                    GOTO INTERRUPT-FROM-VIRTUAL-8086-MODE; FI;
                    (* PE = 1, interrupt or trap gate, DPL < CPL, VM = 1 *)
            FI;
        ELSE (* PE = 1, interrupt or trap gate, DPL ≥ CPL *)
            IF VM = 1
                THEN #GP(error_code(new code-segment selector,0,EXT));
                (* idt operand to error_code is 0 because selector is used *)
            IF new code segment is conforming or new code-segment DPL = CPL
                THEN
                    GOTO INTRA-PRIVILEGE-LEVEL-INTERRUPT;
                ELSE (* PE = 1, interrupt or trap gate, nonconforming code segment, DPL > CPL *)
                    #GP(error_code(new code-segment selector,0,EXT));
                    (* idt operand to error_code is 0 because selector is used *)
            FI;
    FI;
END;
INTER-PRIVILEGE-LEVEL-INTERRUPT:
    (* PE = 1, interrupt or trap gate, non-conforming code segment, DPL < CPL *)
    IF (IA32_EFER.LMA = 0) (* Not IA-32e mode *)
```

```
        THEN
        (* Identify stack-segment selector for new privilege level in current TSS *)
            IF current TSS is 32-bit
                THEN
                    TSSstackAddress ← (new code-segment DPL « 3) + 4;
                    IF (TSSstackAddress + 5) > current TSS limit
                        THEN #TS(error_code(current TSS selector,0,EXT)); FI;
                        (* idt operand to error_code is 0 because selector is used *)
                    NewSS ← 2 bytes loaded from (TSS base + TSSstackAddress + 4);
                    NewESP ← 4 bytes loaded from (TSS base + TSSstackAddress);
                ELSE    (* current TSS is 16-bit *)
                    TSSstackAddress ← (new code-segment DPL « 2) + 2
                    IF (TSSstackAddress + 3) > current TSS limit
                        THEN #TS(error_code(current TSS selector,0,EXT)); FI;
                        (* idt operand to error_code is 0 because selector is used *)
                    NewSS ← 2 bytes loaded from (TSS base + TSSstackAddress + 2);
                    NewESP ← 2 bytes loaded from (TSS base + TSSstackAddress);
            FI;
            IF NewSS is NULL
                THEN #TS(EXT); FI;
            IF NewSS index is not within its descriptor-table limits
            or NewSS RPL ≠ new code-segment DPL
                THEN #TS(error_code(NewSS,0,EXT)); FI;
                (* idt operand to error_code is 0 because selector is used *)
            Read new stack-segment descriptor for NewSS in GDT or LDT;
            IF new stack-segment DPL ≠ new code-segment DPL
            or new stack-segment Type does not indicate writable data segment
                THEN #TS(error_code(NewSS,0,EXT)); FI;
                (* idt operand to error_code is 0 because selector is used *)
            IF NewSS is not present
                THEN #SS(error_code(NewSS,0,EXT)); FI;
                (* idt operand to error_code is 0 because selector is used *)
        ELSE (* IA-32e mode *)
            IF IDT-gate IST = 0
                THEN TSSstackAddress ← (new code-segment DPL « 3) + 4;
                ELSE TSSstackAddress ← (IDT gate IST « 3) + 28;
            FI;
            IF (TSSstackAddress + 7) > current TSS limit
                THEN #TS(error_code(current TSS selector,0,EXT); FI;
                (* idt operand to error_code is 0 because selector is used *)
            NewRSP ← 8 bytes loaded from (current TSS base + TSSstackAddress);
            NewSS ← new code-segment DPL; (* NULL selector with RPL = new CPL *)
    FI;
    IF IDT gate is 32-bit
            THEN
                IF new stack does not have room for 24 bytes (error code pushed)
                or 20 bytes (no error code pushed)
                    THEN #SS(error_code(NewSS,0,EXT)); FI;
                    (* idt operand to error_code is 0 because selector is used *)
            FI
        ELSE
            IF IDT gate is 16-bit
                THEN
                    IF new stack does not have room for 12 bytes (error code pushed)
```

```
                    or 10 bytes (no error code pushed);
                        THEN #SS(error_code(NewSS,0,EXT)); FI;
                        (* idt operand to error_code is 0 because selector is used *)
            ELSE (* 64-bit IDT gate*)
                IF StackAddress is non-canonical
                    THEN #SS(EXT); FI; (* Error code contains NULL selector *)
        FI;
FI;
IF (IA32_EFER.LMA = 0) (* Not IA-32e mode *)
        THEN
            IF instruction pointer from IDT gate is not within new code-segment limits
                THEN #GP(EXT); FI; (* Error code contains NULL selector *)
            ESP ← NewESP;
            SS ← NewSS; (* Segment descriptor information also loaded *)
        ELSE (* IA-32e mode *)
            IF instruction pointer from IDT gate contains a non-canonical address
                THEN #GP(EXT); FI; (* Error code contains NULL selector *)
            RSP ← NewRSP & FFFFFFFFFFFFFFF0H;
            SS ← NewSS;
FI;
IF IDT gate is 32-bit
        THEN
            CS:EIP ← Gate(CS:EIP); (* Segment descriptor information also loaded *)
        ELSE
            IF IDT gate 16-bit
                THEN
                    CS:IP ← Gate(CS:IP);
                    (* Segment descriptor information also loaded *)
                ELSE (* 64-bit IDT gate *)
                    CS:RIP ← Gate(CS:RIP);
                    (* Segment descriptor information also loaded *)
            FI;
FI;
IF IDT gate is 32-bit
            THEN
                Push(far pointer to old stack);
                (* Old SS and ESP, 3 words padded to 4 *)
                Push(EFLAGS);
                Push(far pointer to return instruction);
                (* Old CS and EIP, 3 words padded to 4 *)
                Push(ErrorCode); (* If needed, 4 bytes *)
            ELSE
                IF IDT gate 16-bit
                    THEN
                        Push(far pointer to old stack);
                        (* Old SS and SP, 2 words *)
                        Push(EFLAGS(15-0]);
                        Push(far pointer to return instruction);
                        (* Old CS and IP, 2 words *)
                        Push(ErrorCode); (* If needed, 2 bytes *)
                    ELSE (* 64-bit IDT gate *)
                        Push(far pointer to old stack);
                        (* Old SS and SP, each an 8-byte push *)
                        Push(RFLAGS); (* 8-byte push *)
```

```
                        Push(far pointer to return instruction);
                        (* Old CS and RIP, each an 8-byte push *)
                        Push(ErrorCode); (* If needed, 8-bytes *)
            FI;
    FI;
    CPL ← new code-segment DPL;
    CS(RPL) ← CPL;
    IF IDT gate is interrupt gate
        THEN IF ← 0 (* Interrupt flag set to 0, interrupts disabled *); FI;
    TF ← 0;
    VM ← 0;
    RF ← 0;
    NT ← 0;
END;
INTERRUPT-FROM-VIRTUAL-8086-MODE:
    (* Identify stack-segment selector for privilege level 0 in current TSS *)
    IF current TSS is 32-bit
        THEN
            IF TSS limit < 9
                THEN #TS(error_code(current TSS selector,0,EXT)); FI;
                (* idt operand to error_code is 0 because selector is used *)
            NewSS ← 2 bytes loaded from (current TSS base + 8);
            NewESP ← 4 bytes loaded from (current TSS base + 4);
        ELSE (* current TSS is 16-bit *)
            IF TSS limit < 5
                THEN #TS(error_code(current TSS selector,0,EXT)); FI;
                (* idt operand to error_code is 0 because selector is used *)
            NewSS ← 2 bytes loaded from (current TSS base + 4);
            NewESP ← 2 bytes loaded from (current TSS base + 2);
    FI;
    IF NewSS is NULL
        THEN #TS(EXT); FI; (* Error code contains NULL selector *)
    IF NewSS index is not within its descriptor table limits
    or NewSS RPL ≠ 0
        THEN #TS(error_code(NewSS,0,EXT)); FI;
        (* idt operand to error_code is 0 because selector is used *)
    Read new stack-segment descriptor for NewSS in GDT or LDT;
    IF new stack-segment DPL ≠ 0 or stack segment does not indicate writable data segment
        THEN #TS(error_code(NewSS,0,EXT)); FI;
        (* idt operand to error_code is 0 because selector is used *)
    IF new stack segment not present
        THEN #SS(error_code(NewSS,0,EXT)); FI;
        (* idt operand to error_code is 0 because selector is used *)
    IF IDT gate is 32-bit
        THEN
            IF new stack does not have room for 40 bytes (error code pushed)
            or 36 bytes (no error code pushed)
                THEN #SS(error_code(NewSS,0,EXT)); FI;
                (* idt operand to error_code is 0 because selector is used *)
        ELSE (* IDT gate is 16-bit)
            IF new stack does not have room for 20 bytes (error code pushed)
            or 18 bytes (no error code pushed)
                THEN #SS(error_code(NewSS,0,EXT)); FI;
                (* idt operand to error_code is 0 because selector is used *)
```

```
    FI;
    IF instruction pointer from IDT gate is not within new code-segment limits
        THEN #GP(EXT); FI; (* Error code contains NULL selector *)
    tempEFLAGS ← EFLAGS;
    VM ← 0;
    TF ← 0;
    RF ← 0;
    NT ← 0;
    IF service through interrupt gate
        THEN IF = 0; FI;
    TempSS ← SS;
    TempESP ← ESP;
    SS ← NewSS;
    ESP ← NewESP;
    (* Following pushes are 16 bits for 16-bit IDT gates and 32 bits for 32-bit IDT gates;
    Segment selector pushes in 32-bit mode are padded to two words *)
    Push(GS);
    Push(FS);
    Push(DS);
    Push(ES);
    Push(TempSS);
    Push(TempESP);
    Push(TempEFlags);
    Push(CS);
    Push(EIP);
    GS ← 0; (* Segment registers made NULL, invalid for use in protected mode *)
    FS ← 0;
    DS ← 0;
    ES ← 0;
    CS:IP ← Gate(CS); (* Segment descriptor information also loaded *)
    IF OperandSize = 32
        THEN
            EIP ← Gate(instruction pointer);
        ELSE (* OperandSize is 16 *)
            EIP ← Gate(instruction pointer) AND 0000FFFFH;
    FI;
    (* Start execution of new routine in Protected Mode *)
END;
INTRA-PRIVILEGE-LEVEL-INTERRUPT:
    (* PE = 1, DPL = CPL or conforming segment *)
    IF IA32_EFER.LMA = 1 (* IA-32e mode *)
        IF IDT-descriptor IST ≠ 0
            THEN
                TSSstackAddress ← (IDT-descriptor IST « 3) + 28;
                IF (TSSstackAddress + 7) > TSS limit
                    THEN #TS(error_code(current TSS selector,0,EXT)); FI;
                    (* idt operand to error_code is 0 because selector is used *)
                NewRSP ← 8 bytes loaded from (current TSS base + TSSstackAddress);
                ELSE NewRSP ← RSP;
                FI;
    FI;
    IF 32-bit gate (* implies IA32_EFER.LMA = 0 *)
        THEN
            IF current stack does not have room for 16 bytes (error code pushed)
```

```
            or 12 bytes (no error code pushed)
                THEN #SS(EXT); FI; (* Error code contains NULL selector *)
        ELSE IF 16-bit gate (* implies IA32_EFER.LMA = 0 *)
            IF current stack does not have room for 8 bytes (error code pushed)
            or 6 bytes (no error code pushed)
                THEN #SS(EXT); FI; (* Error code contains NULL selector *)
        ELSE (* IA32_EFER.LMA = 1, 64-bit gate*)
                IF NewRSP contains a non-canonical address
                    THEN #SS(EXT); (* Error code contains NULL selector *)
        FI;
    FI;
    IF (IA32_EFER.LMA = 0) (* Not IA-32e mode *)
        THEN
            IF instruction pointer from IDT gate is not within new code-segment limit
                THEN #GP(EXT); FI; (* Error code contains NULL selector *)
        ELSE
            IF instruction pointer from IDT gate contains a non-canonical address
                THEN #GP(EXT); FI; (* Error code contains NULL selector *)
            RSP ← NewRSP & FFFFFFFFFFFFFFF0H;
    FI;
    IF IDT gate is 32-bit (* implies IA32_EFER.LMA = 0 *)
        THEN
            Push (EFLAGS);
            Push (far pointer to return instruction); (* 3 words padded to 4 *)
            CS:EIP ← Gate(CS:EIP); (* Segment descriptor information also loaded *)
            Push (ErrorCode); (* If any *)
        ELSE
            IF IDT gate is 16-bit (* implies IA32_EFER.LMA = 0 *)
                THEN
                    Push (FLAGS);
                    Push (far pointer to return location); (* 2 words *)
                    CS:IP ← Gate(CS:IP);
                    (* Segment descriptor information also loaded *)
                    Push (ErrorCode); (* If any *)
                ELSE (* IA32_EFER.LMA = 1, 64-bit gate*)
                    Push(far pointer to old stack);
                    (* Old SS and SP, each an 8-byte push *)
                    Push(RFLAGS); (* 8-byte push *)
                    Push(far pointer to return instruction);
                    (* Old CS and RIP, each an 8-byte push *)
                    Push(ErrorCode); (* If needed, 8 bytes *)
                    CS:RIP ← GATE(CS:RIP);
                    (* Segment descriptor information also loaded *)
            FI;
    FI;
    CS(RPL) ← CPL;
    IF IDT gate is interrupt gate
        THEN IF ← 0; FI; (* Interrupt flag set to 0; interrupts disabled *)
    TF ← 0;
    NT ← 0;
    VM ← 0;
    RF ← 0;
END;
```

Flags Affected

The EFLAGS register is pushed onto the stack. The IF, TF, NT, AC, RF, and VM flags may be cleared, depending on the mode of operation of the processor when the INT instruction is executed (see the "Operation" section). If the interrupt uses a task gate, any flags may be set or cleared, controlled by the EFLAGS image in the new task's TSS.

Protected Mode Exceptions

#GP(error_code)	If the instruction pointer in the IDT or in the interrupt-, trap-, or task gate is beyond the code segment limits.
	If the segment selector in the interrupt-, trap-, or task gate is NULL.
	If an interrupt-, trap-, or task gate, code segment, or TSS segment selector index is outside its descriptor table limits.
	If the vector selects a descriptor outside the IDT limits.
	If an IDT descriptor is not an interrupt-, trap-, or task-descriptor.
	If an interrupt is generated by the INT *n*, INT 3, or INTO instruction and the DPL of an interrupt-, trap-, or task-descriptor is less than the CPL.
	If the segment selector in an interrupt- or trap-gate does not point to a segment descriptor for a code segment.
	If the segment selector for a TSS has its local/global bit set for local.
	If a TSS segment descriptor specifies that the TSS is busy or not available.
#SS(error_code)	If pushing the return address, flags, or error code onto the stack exceeds the bounds of the stack segment and no stack switch occurs.
	If the SS register is being loaded and the segment pointed to is marked not present.
	If pushing the return address, flags, error code, or stack segment pointer exceeds the bounds of the new stack segment when a stack switch occurs.
#NP(error_code)	If code segment, interrupt-, trap-, or task gate, or TSS is not present.
#TS(error_code)	If the RPL of the stack segment selector in the TSS is not equal to the DPL of the code segment being accessed by the interrupt or trap gate.
	If DPL of the stack segment descriptor pointed to by the stack segment selector in the TSS is not equal to the DPL of the code segment descriptor for the interrupt or trap gate.
	If the stack segment selector in the TSS is NULL.
	If the stack segment for the TSS is not a writable data segment.
	If segment-selector index for stack segment is outside descriptor table limits.
#PF(fault-code)	If a page fault occurs.
#UD	If the LOCK prefix is used.
#AC(EXT)	If alignment checking is enabled, the gate DPL is 3, and a stack push is unaligned.

Real-Address Mode Exceptions

#GP	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
	If the interrupt vector number is outside the IDT limits.
#SS	If stack limit violation on push.
	If pushing the return address, flags, or error code onto the stack exceeds the bounds of the stack segment.
#UD	If the LOCK prefix is used.

Virtual-8086 Mode Exceptions

#GP(error_code)	(For INT *n*, INTO, or BOUND instruction) If the IOPL is less than 3 or the DPL of the interrupt-, trap-, or task-gate descriptor is not equal to 3.
	If the instruction pointer in the IDT or in the interrupt-, trap-, or task gate is beyond the code segment limits.
	If the segment selector in the interrupt-, trap-, or task gate is NULL.
	If a interrupt-, trap-, or task gate, code segment, or TSS segment selector index is outside its descriptor table limits.
	If the vector selects a descriptor outside the IDT limits.
	If an IDT descriptor is not an interrupt-, trap-, or task-descriptor.
	If an interrupt is generated by the INT *n* instruction and the DPL of an interrupt-, trap-, or task-descriptor is less than the CPL.
	If the segment selector in an interrupt- or trap-gate does not point to a segment descriptor for a code segment.
	If the segment selector for a TSS has its local/global bit set for local.
#SS(error_code)	If the SS register is being loaded and the segment pointed to is marked not present.
	If pushing the return address, flags, error code, stack segment pointer, or data segments exceeds the bounds of the stack segment.
#NP(error_code)	If code segment, interrupt-, trap-, or task gate, or TSS is not present.
#TS(error_code)	If the RPL of the stack segment selector in the TSS is not equal to the DPL of the code segment being accessed by the interrupt or trap gate.
	If DPL of the stack segment descriptor for the TSS's stack segment is not equal to the DPL of the code segment descriptor for the interrupt or trap gate.
	If the stack segment selector in the TSS is NULL.
	If the stack segment for the TSS is not a writable data segment.
	If segment-selector index for stack segment is outside descriptor table limits.
#PF(fault-code)	If a page fault occurs.
#BP	If the INT 3 instruction is executed.
#OF	If the INTO instruction is executed and the OF flag is set.
#UD	If the LOCK prefix is used.
#AC(EXT)	If alignment checking is enabled, the gate DPL is 3, and a stack push is unaligned.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#GP(error_code)	If the instruction pointer in the 64-bit interrupt gate or 64-bit trap gate is non-canonical.
	If the segment selector in the 64-bit interrupt or trap gate is NULL.
	If the vector selects a descriptor outside the IDT limits.
	If the vector points to a gate which is in non-canonical space.
	If the vector points to a descriptor which is not a 64-bit interrupt gate or 64-bit trap gate.
	If the descriptor pointed to by the gate selector is outside the descriptor table limit.
	If the descriptor pointed to by the gate selector is in non-canonical space.
	If the descriptor pointed to by the gate selector is not a code segment.
	If the descriptor pointed to by the gate selector doesn't have the L-bit set, or has both the L-bit and D-bit set.
	If the descriptor pointed to by the gate selector has DPL > CPL.
#SS(error_code)	If a push of the old EFLAGS, CS selector, EIP, or error code is in non-canonical space with no stack switch.
	If a push of the old SS selector, ESP, EFLAGS, CS selector, EIP, or error code is in non-canonical space on a stack switch (either CPL change or no-CPL with IST).
#NP(error_code)	If the 64-bit interrupt-gate, 64-bit trap-gate, or code segment is not present.
#TS(error_code)	If an attempt to load RSP from the TSS causes an access to non-canonical space.
	If the RSP from the TSS is outside descriptor table limits.
#PF(fault-code)	If a page fault occurs.
#UD	If the LOCK prefix is used.
#AC(EXT)	If alignment checking is enabled, the gate DPL is 3, and a stack push is unaligned.

INVD—Invalidate Internal Caches

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
0F 08	INVD	ZO	Valid	Valid	Flush internal caches; initiate flushing of external caches.

NOTES:

* See the IA-32 Architecture Compatibility section below.

Instruction Operand Encoding

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

Description

Invalidates (flushes) the processor's internal caches and issues a special-function bus cycle that directs external caches to also flush themselves. Data held in internal caches is not written back to main memory.

After executing this instruction, the processor does not wait for the external caches to complete their flushing operation before proceeding with instruction execution. It is the responsibility of hardware to respond to the cache flush signal.

The INVD instruction is a privileged instruction. When the processor is running in protected mode, the CPL of a program or procedure must be 0 to execute this instruction.

The INVD instruction may be used when the cache is used as temporary memory and the cache contents need to be invalidated rather than written back to memory. When the cache is used as temporary memory, no external device should be actively writing data to main memory.

Use this instruction with care. Data cached internally and not written back to main memory will be lost. Note that any data from an external device to main memory (for example, via a PCIWrite) can be temporarily stored in the caches; these data can be lost when an INVD instruction is executed. Unless there is a specific requirement or benefit to flushing caches without writing back modified cache lines (for example, temporary memory, testing, or fault recovery where cache coherency with main memory is not a concern), software should instead use the WBINVD instruction.

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

IA-32 Architecture Compatibility

The INVD instruction is implementation dependent; it may be implemented differently on different families of Intel 64 or IA-32 processors. This instruction is not supported on IA-32 processors earlier than the Intel486 processor.

Operation

```
Flush(InternalCaches);
SignalFlush(ExternalCaches);
Continue (* Continue execution *)
```

Flags Affected

None

Protected Mode Exceptions

#GP(0)	If the current privilege level is not 0.
#UD	If the LOCK prefix is used.

Real-Address Mode Exceptions

#UD	If the LOCK prefix is used.

Virtual-8086 Mode Exceptions

#GP(0)	The INVD instruction cannot be executed in virtual-8086 mode.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

Same exceptions as in protected mode.

INVLPG—Invalidate TLB Entries

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
0F 01/7	INVLPG *m*	M	Valid	Valid	Invalidate TLB entries for page containing *m*.

NOTES:

* See the IA-32 Architecture Compatibility section below.

Instruction Operand Encoding

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

Description

Invalidates any translation lookaside buffer (TLB) entries specified with the source operand. The source operand is a memory address. The processor determines the page that contains that address and flushes all TLB entries for that page.[1]

The INVLPG instruction is a privileged instruction. When the processor is running in protected mode, the CPL must be 0 to execute this instruction.

The INVLPG instruction normally flushes TLB entries only for the specified page; however, in some cases, it may flush more entries, even the entire TLB. The instruction is guaranteed to invalidates only TLB entries associated with the current PCID. (If PCIDs are disabled — CR4.PCIDE = 0 — the current PCID is 000H.) The instruction also invalidates any global TLB entries for the specified page, regardless of PCID.

For more details on operations that flush the TLB, see "MOV—Move to/from Control Registers" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2B* and Section 4.10.4.1, "Operations that Invalidate TLBs and Paging-Structure Caches," in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*.

This instruction's operation is the same in all non-64-bit modes. It also operates the same in 64-bit mode, except if the memory address is in non-canonical form. In this case, INVLPG is the same as a NOP.

IA-32 Architecture Compatibility

The INVLPG instruction is implementation dependent, and its function may be implemented differently on different families of Intel 64 or IA-32 processors. This instruction is not supported on IA-32 processors earlier than the Intel486 processor.

Operation

```
Invalidate(RelevantTLBEntries);
Continue; (* Continue execution *)
```

Flags Affected

None.

Protected Mode Exceptions

#GP(0)	If the current privilege level is not 0.
#UD	Operand is a register.
	If the LOCK prefix is used.

1. If the paging structures map the linear address using a page larger than 4 KBytes and there are multiple TLB entries for that page (see Section 4.10.2.3, "Details of TLB Use," in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*), the instruction invalidates all of them.

Real-Address Mode Exceptions

#UD	Operand is a register. If the LOCK prefix is used.

Virtual-8086 Mode Exceptions

#GP(0)	The INVLPG instruction cannot be executed at the virtual-8086 mode.

64-Bit Mode Exceptions

#GP(0)	If the current privilege level is not 0.
#UD	Operand is a register. If the LOCK prefix is used.

INVPCID—Invalidate Process-Context Identifier

Opcode/Instruction	Op/ En	64/32-bit Mode	CPUID Feature Flag	Description
66 0F 38 82 /r INVPCID r32, m128	RM	NE/V	INVPCID	Invalidates entries in the TLBs and paging-structure caches based on invalidation type in r32 and descriptor in m128.
66 0F 38 82 /r INVPCID r64, m128	RM	V/NE	INVPCID	Invalidates entries in the TLBs and paging-structure caches based on invalidation type in r64 and descriptor in m128.

Instruction Operand Encoding

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

Description

Invalidates mappings in the translation lookaside buffers (TLBs) and paging-structure caches based on process-context identifier (PCID). (See Section 4.10, "Caching Translation Information," in *Intel 64 and IA-32 Architecture Software Developer's Manual, Volume 3A*.) Invalidation is based on the INVPCID type specified in the register operand and the INVPCID descriptor specified in the memory operand.

Outside 64-bit mode, the register operand is always 32 bits, regardless of the value of CS.D. In 64-bit mode the register operand has 64 bits.

There are four INVPCID types currently defined:

- Individual-address invalidation: If the INVPCID type is 0, the logical processor invalidates mappings—except global translations—for the linear address and PCID specified in the INVPCID descriptor.[1] In some cases, the instruction may invalidate global translations or mappings for other linear addresses (or other PCIDs) as well.
- Single-context invalidation: If the INVPCID type is 1, the logical processor invalidates all mappings—except global translations—associated with the PCID specified in the INVPCID descriptor. In some cases, the instruction may invalidate global translations or mappings for other PCIDs as well.
- All-context invalidation, including global translations: If the INVPCID type is 2, the logical processor invalidates all mappings—including global translations—associated with any PCID.
- All-context invalidation: If the INVPCID type is 3, the logical processor invalidates all mappings—except global translations—associated with any PCID. In some case, the instruction may invalidate global translations as well.

The INVPCID descriptor comprises 128 bits and consists of a PCID and a linear address as shown in Figure 3-24. For INVPCID type 0, the processor uses the full 64 bits of the linear address even outside 64-bit mode; the linear address is not used for other INVPCID types.

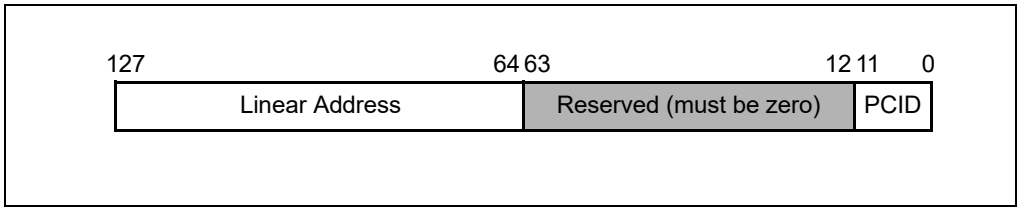


Figure 3-24. INVPCID Descriptor

1. If the paging structures map the linear address using a page larger than 4 KBytes and there are multiple TLB entries for that page (see Section 4.10.2.3, "Details of TLB Use," in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*), the instruction invalidates all of them.

If CR4.PCIDE = 0, a logical processor does not cache information for any PCID other than 000H. In this case, executions with INVPCID types 0 and 1 are allowed only if the PCID specified in the INVPCID descriptor is 000H; executions with INVPCID types 2 and 3 invalidate mappings only for PCID 000H. Note that CR4.PCIDE must be 0 outside 64-bit mode (see Chapter 4.10.1, "Process-Context Identifiers (PCIDs)," of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*).

Operation

```
INVPCID_TYPE ← value of register operand;        // must be in the range of 0–3
INVPCID_DESC ← value of memory operand;
CASE INVPCID_TYPE OF
    0:              // individual-address invalidation
        PCID ← INVPCID_DESC[11:0];
        L_ADDR ← INVPCID_DESC[127:64];
        Invalidate mappings for L_ADDR associated with PCID except global translations;
        BREAK;
    1:              // single PCID invalidation
        PCID ← INVPCID_DESC[11:0];
        Invalidate all mappings associated with PCID except global translations;
        BREAK;
    2:              // all PCID invalidation including global translations
        Invalidate all mappings for all PCIDs, including global translations;
        BREAK;
    3:              // all PCID invalidation retaining global translations
        Invalidate all mappings for all PCIDs except global translations;
        BREAK;
ESAC;
```

Intel C/C++ Compiler Intrinsic Equivalent

INVPCID: void _invpcid(unsigned __int32 type, void * descriptor);

SIMD Floating-Point Exceptions

None

Protected Mode Exceptions

#GP(0)	If the current privilege level is not 0.
	If the memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
	If the DS, ES, FS, or GS register contains an unusable segment.
	If the source operand is located in an execute-only code segment.
	If an invalid type is specified in the register operand, i.e., INVPCID_TYPE > 3.
	If bits 63:12 of INVPCID_DESC are not all zero.
	If INVPCID_TYPE is either 0 or 1 and INVPCID_DESC[11:0] is not zero.
	If INVPCID_TYPE is 0 and the linear address in INVPCID_DESC[127:64] is not canonical.
#PF(fault-code)	If a page fault occurs in accessing the memory operand.
#SS(0)	If the memory operand effective address is outside the SS segment limit.
	If the SS register contains an unusable segment.
#UD	If if CPUID.(EAX=07H, ECX=0H):EBX.INVPCID (bit 10) = 0.
	If the LOCK prefix is used.

Real-Address Mode Exceptions

#GP	If an invalid type is specified in the register operand, i.e., INVPCID_TYPE > 3.
	If bits 63:12 of INVPCID_DESC are not all zero.
	If INVPCID_TYPE is either 0 or 1 and INVPCID_DESC[11:0] is not zero.
	If INVPCID_TYPE is 0 and the linear address in INVPCID_DESC[127:64] is not canonical.
#UD	If CPUID.(EAX=07H, ECX=0H):EBX.INVPCID (bit 10) = 0.
	If the LOCK prefix is used.

Virtual-8086 Mode Exceptions

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

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#GP(0)	If the current privilege level is not 0.
	If the memory operand is in the CS, DS, ES, FS, or GS segments and the memory address is in a non-canonical form.
	If an invalid type is specified in the register operand, i.e., INVPCID_TYPE > 3.
	If bits 63:12 of INVPCID_DESC are not all zero.
	If CR4.PCIDE=0, INVPCID_TYPE is either 0 or 1, and INVPCID_DESC[11:0] is not zero.
	If INVPCID_TYPE is 0 and the linear address in INVPCID_DESC[127:64] is not canonical.
#PF(fault-code)	If a page fault occurs in accessing the memory operand.
#SS(0)	If the memory destination operand is in the SS segment and the memory address is in a non-canonical form.
#UD	If the LOCK prefix is used.
	If CPUID.(EAX=07H, ECX=0H):EBX.INVPCID (bit 10) = 0.

IRET/IRETD—Interrupt Return

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
CF	IRET	ZO	Valid	Valid	Interrupt return (16-bit operand size).
CF	IRETD	ZO	Valid	Valid	Interrupt return (32-bit operand size).
REX.W + CF	IRETQ	ZO	Valid	N.E.	Interrupt return (64-bit operand size).

Instruction Operand Encoding

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

Description

Returns program control from an exception or interrupt handler to a program or procedure that was interrupted by an exception, an external interrupt, or a software-generated interrupt. These instructions are also used to perform a return from a nested task. (A nested task is created when a CALL instruction is used to initiate a task switch or when an interrupt or exception causes a task switch to an interrupt or exception handler.) See the section titled "Task Linking" in Chapter 7 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*.

IRET and IRETD are mnemonics for the same opcode. The IRETD mnemonic (interrupt return double) is intended for use when returning from an interrupt when using the 32-bit operand size; however, most assemblers use the IRET mnemonic interchangeably for both operand sizes.

In Real-Address Mode, the IRET instruction preforms a far return to the interrupted program or procedure. During this operation, the processor pops the return instruction pointer, return code segment selector, and EFLAGS image from the stack to the EIP, CS, and EFLAGS registers, respectively, and then resumes execution of the interrupted program or procedure.

In Protected Mode, the action of the IRET instruction depends on the settings of the NT (nested task) and VM flags in the EFLAGS register and the VM flag in the EFLAGS image stored on the current stack. Depending on the setting of these flags, the processor performs the following types of interrupt returns:

- Return from virtual-8086 mode.
- Return to virtual-8086 mode.
- Intra-privilege level return.
- Inter-privilege level return.
- Return from nested task (task switch).

If the NT flag (EFLAGS register) is cleared, the IRET instruction performs a far return from the interrupt procedure, without a task switch. The code segment being returned to must be equally or less privileged than the interrupt handler routine (as indicated by the RPL field of the code segment selector popped from the stack).

As with a real-address mode interrupt return, the IRET instruction pops the return instruction pointer, return code segment selector, and EFLAGS image from the stack to the EIP, CS, and EFLAGS registers, respectively, and then resumes execution of the interrupted program or procedure. If the return is to another privilege level, the IRET instruction also pops the stack pointer and SS from the stack, before resuming program execution. If the return is to virtual-8086 mode, the processor also pops the data segment registers from the stack.

If the NT flag is set, the IRET instruction performs a task switch (return) from a nested task (a task called with a CALL instruction, an interrupt, or an exception) back to the calling or interrupted task. The updated state of the task executing the IRET instruction is saved in its TSS. If the task is re-entered later, the code that follows the IRET instruction is executed.

If the NT flag is set and the processor is in IA-32e mode, the IRET instruction causes a general protection exception.

If nonmaskable interrupts (NMIs) are blocked (see Section 6.7.1, "Handling Multiple NMIs" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*), execution of the IRET instruction unblocks NMIs.

This unblocking occurs even if the instruction causes a fault. In such a case, NMIs are unmasked before the exception handler is invoked.

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

See "Changes to Instruction Behavior in VMX Non-Root Operation" in Chapter 25 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3C*, for more information about the behavior of this instruction in VMX non-root operation.

Operation

```
IF PE = 0
    THEN GOTO REAL-ADDRESS-MODE;
ELSIF (IA32_EFER.LMA = 0)
    THEN
        IF (EFLAGS.VM = 1)
            THEN GOTO RETURN-FROM-VIRTUAL-8086-MODE;
            ELSE GOTO PROTECTED-MODE;
        FI;
    ELSE GOTO IA-32e-MODE;
FI;

REAL-ADDRESS-MODE;
    IF OperandSize = 32
        THEN
            EIP ← Pop();
            CS ← Pop(); (* 32-bit pop, high-order 16 bits discarded *)
            tempEFLAGS ← Pop();
            EFLAGS ← (tempEFLAGS AND 257FD5H) OR (EFLAGS AND 1A0000H);
        ELSE (* OperandSize = 16 *)
            EIP ← Pop(); (* 16-bit pop; clear upper 16 bits *)
            CS ← Pop(); (* 16-bit pop *)
            EFLAGS[15:0] ← Pop();
    FI;
    END;

RETURN-FROM-VIRTUAL-8086-MODE:
(* Processor is in virtual-8086 mode when IRET is executed and stays in virtual-8086 mode *)
    IF IOPL = 3 (* Virtual mode: PE = 1, VM = 1, IOPL = 3 *)
        THEN IF OperandSize = 32
            THEN
                EIP ← Pop();
                CS ← Pop(); (* 32-bit pop, high-order 16 bits discarded *)
                EFLAGS ← Pop();
                (* VM, IOPL,VIP and VIF EFLAG bits not modified by pop *)
                IF EIP not within CS limit
                    THEN #GP(0); FI;
            ELSE (* OperandSize = 16 *)
                EIP ← Pop(); (* 16-bit pop; clear upper 16 bits *)
                CS ← Pop(); (* 16-bit pop *)
                EFLAGS[15:0] ← Pop(); (* IOPL in EFLAGS not modified by pop *)
                IF EIP not within CS limit
                    THEN #GP(0); FI;
            FI;
        ELSE
```

```
            #GP(0); (* Trap to virtual-8086 monitor: PE = 1, VM = 1, IOPL < 3 *)
    FI;
END;

PROTECTED-MODE:
    IF NT = 1
        THEN GOTO TASK-RETURN; (* PE = 1, VM = 0, NT = 1 *)
    FI;
    IF OperandSize = 32
        THEN
            EIP ← Pop();
            CS ← Pop(); (* 32-bit pop, high-order 16 bits discarded *)
            tempEFLAGS ← Pop();
        ELSE (* OperandSize = 16 *)
            EIP ← Pop(); (* 16-bit pop; clear upper bits *)
            CS ← Pop(); (* 16-bit pop *)
            tempEFLAGS ← Pop(); (* 16-bit pop; clear upper bits *)
    FI;
    IF tempEFLAGS(VM) = 1 and CPL = 0
        THEN GOTO RETURN-TO-VIRTUAL-8086-MODE;
        ELSE GOTO PROTECTED-MODE-RETURN;
    FI;

TASK-RETURN: (* PE = 1, VM = 0, NT = 1 *)
    SWITCH-TASKS (without nesting) to TSS specified in link field of current TSS;
    Mark the task just abandoned as NOT BUSY;
    IF EIP is not within CS limit
        THEN #GP(0); FI;
END;

RETURN-TO-VIRTUAL-8086-MODE:
    (* Interrupted procedure was in virtual-8086 mode: PE = 1, CPL=0, VM = 1 in flag image *)
    IF EIP not within CS limit
        THEN #GP(0); FI;
    EFLAGS ← tempEFLAGS;
    ESP ← Pop();
    SS ← Pop(); (* Pop 2 words; throw away high-order word *)
    ES ← Pop(); (* Pop 2 words; throw away high-order word *)
    DS ← Pop(); (* Pop 2 words; throw away high-order word *)
    FS ← Pop(); (* Pop 2 words; throw away high-order word *)
    GS ← Pop(); (* Pop 2 words; throw away high-order word *)
    CPL ← 3;
    (* Resume execution in Virtual-8086 mode *)
END;

PROTECTED-MODE-RETURN: (* PE = 1 *)
    IF CS(RPL) > CPL
        THEN GOTO RETURN-TO-OUTER-PRIVILEGE-LEVEL;
        ELSE GOTO RETURN-TO-SAME-PRIVILEGE-LEVEL; FI;
END;

RETURN-TO-OUTER-PRIVILEGE-LEVEL:
    IF OperandSize = 32
        THEN
```

```
                ESP ← Pop();
                SS ← Pop(); (* 32-bit pop, high-order 16 bits discarded *)
        ELSE IF OperandSize = 16
            THEN
                ESP ← Pop(); (* 16-bit pop; clear upper bits *)
                SS ← Pop(); (* 16-bit pop *)
            ELSE (* OperandSize = 64 *)
                RSP ← Pop();
                SS ← Pop(); (* 64-bit pop, high-order 48 bits discarded *)
        FI;
        IF new mode ≠ 64-Bit Mode
            THEN
                IF EIP is not within CS limit
                    THEN #GP(0); FI;
            ELSE (* new mode = 64-bit mode *)
                IF RIP is non-canonical
                        THEN #GP(0); FI;
        FI;
        EFLAGS (CF, PF, AF, ZF, SF, TF, DF, OF, NT) ← tempEFLAGS;
        IF OperandSize = 32 or or OperandSize = 64
            THEN EFLAGS(RF, AC, ID) ← tempEFLAGS; FI;
        IF CPL ≤ IOPL
            THEN EFLAGS(IF) ← tempEFLAGS; FI;
        IF CPL = 0
            THEN
                EFLAGS(IOPL) ← tempEFLAGS;
                IF OperandSize = 32 or OperandSize = 64
                    THEN EFLAGS(VIF, VIP) ← tempEFLAGS; FI;
        FI;
        CPL ← CS(RPL);
        FOR each SegReg in (ES, FS, GS, and DS)
            DO
                tempDesc ← descriptor cache for SegReg (* hidden part of segment register *)
                IF (SegmentSelector == NULL) OR (tempDesc(DPL) < CPL AND tempDesc(Type) is (data or non-conforming code)))
                    THEN (* Segment register invalid *)
                        SegmentSelector ← 0; (*Segment selector becomes null*)
                FI;
            OD;
END;

RETURN-TO-SAME-PRIVILEGE-LEVEL: (* PE = 1, RPL = CPL *)
        IF new mode ≠ 64-Bit Mode
            THEN
                IF EIP is not within CS limit
                    THEN #GP(0); FI;
            ELSE (* new mode = 64-bit mode *)
                IF RIP is non-canonical
                        THEN #GP(0); FI;
        FI;
        EFLAGS (CF, PF, AF, ZF, SF, TF, DF, OF, NT) ← tempEFLAGS;
        IF OperandSize = 32 or OperandSize = 64
            THEN EFLAGS(RF, AC, ID) ← tempEFLAGS; FI;
        IF CPL ≤ IOPL
            THEN EFLAGS(IF) ← tempEFLAGS; FI;
```

```
    IF CPL = 0
        THEN
            EFLAGS(IOPL) ← tempEFLAGS;
            IF OperandSize = 32 or OperandSize = 64
                THEN EFLAGS(VIF, VIP) ← tempEFLAGS; FI;
    FI;
END;

IA-32e-MODE:
    IF NT = 1
        THEN #GP(0);
    ELSE IF OperandSize = 32
        THEN
            EIP ← Pop();
            CS ← Pop();
            tempEFLAGS ← Pop();
        ELSE IF OperandSize = 16
            THEN
                EIP ← Pop(); (* 16-bit pop; clear upper bits *)
                CS ← Pop(); (* 16-bit pop *)
                tempEFLAGS ← Pop(); (* 16-bit pop; clear upper bits *)
            FI;
        ELSE (* OperandSize = 64 *)
            THEN
                    RIP ← Pop();
                    CS ← Pop(); (* 64-bit pop, high-order 48 bits discarded *)
                    tempRFLAGS ← Pop();
    FI;
    IF CS.RPL > CPL
        THEN GOTO RETURN-TO-OUTER-PRIVILEGE-LEVEL;
        ELSE
            IF instruction began in 64-Bit Mode
                THEN
                    IF OperandSize = 32
                        THEN
                            ESP ← Pop();
                            SS ← Pop(); (* 32-bit pop, high-order 16 bits discarded *)
                    ELSE IF OperandSize = 16
                        THEN
                            ESP ← Pop(); (* 16-bit pop; clear upper bits *)
                            SS ← Pop(); (* 16-bit pop *)
                        ELSE (* OperandSize = 64 *)
                            RSP ← Pop();
                            SS ← Pop(); (* 64-bit pop, high-order 48 bits discarded *)
                    FI;
            FI;
            GOTO RETURN-TO-SAME-PRIVILEGE-LEVEL; FI;
END;
```

Flags Affected

All the flags and fields in the EFLAGS register are potentially modified, depending on the mode of operation of the processor. If performing a return from a nested task to a previous task, the EFLAGS register will be modified according to the EFLAGS image stored in the previous task's TSS.

Protected Mode Exceptions

#GP(0)	If the return code or stack segment selector is NULL.
	If the return instruction pointer is not within the return code segment limit.
#GP(selector)	If a segment selector index is outside its descriptor table limits.
	If the return code segment selector RPL is less than the CPL.
	If the DPL of a conforming-code segment is greater than the return code segment selector RPL.
	If the DPL for a nonconforming-code segment is not equal to the RPL of the code segment selector.
	If the stack segment descriptor DPL is not equal to the RPL of the return code segment selector.
	If the stack segment is not a writable data segment.
	If the stack segment selector RPL is not equal to the RPL of the return code segment selector.
	If the segment descriptor for a code segment does not indicate it is a code segment.
	If the segment selector for a TSS has its local/global bit set for local.
	If a TSS segment descriptor specifies that the TSS is not busy.
	If a TSS segment descriptor specifies that the TSS is not available.
#SS(0)	If the top bytes of stack are not within stack limits.
#NP(selector)	If the return code or stack segment is not present.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If an unaligned memory reference occurs when the CPL is 3 and alignment checking is enabled.
#UD	If the LOCK prefix is used.

Real-Address Mode Exceptions

#GP	If the return instruction pointer is not within the return code segment limit.
#SS	If the top bytes of stack are not within stack limits.

Virtual-8086 Mode Exceptions

#GP(0)	If the return instruction pointer is not within the return code segment limit.
	IF IOPL not equal to 3.
#PF(fault-code)	If a page fault occurs.
#SS(0)	If the top bytes of stack are not within stack limits.
#AC(0)	If an unaligned memory reference occurs and alignment checking is enabled.
#UD	If the LOCK prefix is used.

Compatibility Mode Exceptions

#GP(0)	If EFLAGS.NT[bit 14] = 1.

Other exceptions same as in Protected Mode.

64-Bit Mode Exceptions

#GP(0)	If EFLAGS.NT[bit 14] = 1.
	If the return code segment selector is NULL.
	If the stack segment selector is NULL going back to compatibility mode.
	If the stack segment selector is NULL going back to CPL3 64-bit mode.
	If a NULL stack segment selector RPL is not equal to CPL going back to non-CPL3 64-bit mode.
	If the return instruction pointer is not within the return code segment limit.
	If the return instruction pointer is non-canonical.
#GP(Selector)	If a segment selector index is outside its descriptor table limits.
	If a segment descriptor memory address is non-canonical.
	If the segment descriptor for a code segment does not indicate it is a code segment.
	If the proposed new code segment descriptor has both the D-bit and L-bit set.
	If the DPL for a nonconforming-code segment is not equal to the RPL of the code segment selector.
	If CPL is greater than the RPL of the code segment selector.
	If the DPL of a conforming-code segment is greater than the return code segment selector RPL.
	If the stack segment is not a writable data segment.
	If the stack segment descriptor DPL is not equal to the RPL of the return code segment selector.
	If the stack segment selector RPL is not equal to the RPL of the return code segment selector.
#SS(0)	If an attempt to pop a value off the stack violates the SS limit.
	If an attempt to pop a value off the stack causes a non-canonical address to be referenced.
#NP(selector)	If the return code or stack segment is not present.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If an unaligned memory reference occurs when the CPL is 3 and alignment checking is enabled.
#UD	If the LOCK prefix is used.

J*cc*—Jump if Condition Is Met

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
77 *cb*	JA *rel8*	D	Valid	Valid	Jump short if above (CF=0 and ZF=0).
73 *cb*	JAE *rel8*	D	Valid	Valid	Jump short if above or equal (CF=0).
72 *cb*	JB *rel8*	D	Valid	Valid	Jump short if below (CF=1).
76 *cb*	JBE *rel8*	D	Valid	Valid	Jump short if below or equal (CF=1 or ZF=1).
72 *cb*	JC *rel8*	D	Valid	Valid	Jump short if carry (CF=1).
E3 *cb*	JCXZ *rel8*	D	N.E.	Valid	Jump short if CX register is 0.
E3 *cb*	JECXZ *rel8*	D	Valid	Valid	Jump short if ECX register is 0.
E3 *cb*	JRCXZ *rel8*	D	Valid	N.E.	Jump short if RCX register is 0.
74 *cb*	JE *rel8*	D	Valid	Valid	Jump short if equal (ZF=1).
7F *cb*	JG *rel8*	D	Valid	Valid	Jump short if greater (ZF=0 and SF=OF).
7D *cb*	JGE *rel8*	D	Valid	Valid	Jump short if greater or equal (SF=OF).
7C *cb*	JL *rel8*	D	Valid	Valid	Jump short if less (SF≠ OF).
7E *cb*	JLE *rel8*	D	Valid	Valid	Jump short if less or equal (ZF=1 or SF≠ OF).
76 *cb*	JNA *rel8*	D	Valid	Valid	Jump short if not above (CF=1 or ZF=1).
72 *cb*	JNAE *rel8*	D	Valid	Valid	Jump short if not above or equal (CF=1).
73 *cb*	JNB *rel8*	D	Valid	Valid	Jump short if not below (CF=0).
77 *cb*	JNBE *rel8*	D	Valid	Valid	Jump short if not below or equal (CF=0 and ZF=0).
73 *cb*	JNC *rel8*	D	Valid	Valid	Jump short if not carry (CF=0).
75 *cb*	JNE *rel8*	D	Valid	Valid	Jump short if not equal (ZF=0).
7E *cb*	JNG *rel8*	D	Valid	Valid	Jump short if not greater (ZF=1 or SF≠ OF).
7C *cb*	JNGE *rel8*	D	Valid	Valid	Jump short if not greater or equal (SF≠ OF).
7D *cb*	JNL *rel8*	D	Valid	Valid	Jump short if not less (SF=OF).
7F *cb*	JNLE *rel8*	D	Valid	Valid	Jump short if not less or equal (ZF=0 and SF=OF).
71 *cb*	JNO *rel8*	D	Valid	Valid	Jump short if not overflow (OF=0).
7B *cb*	JNP *rel8*	D	Valid	Valid	Jump short if not parity (PF=0).
79 *cb*	JNS *rel8*	D	Valid	Valid	Jump short if not sign (SF=0).
75 *cb*	JNZ *rel8*	D	Valid	Valid	Jump short if not zero (ZF=0).
70 *cb*	JO *rel8*	D	Valid	Valid	Jump short if overflow (OF=1).
7A *cb*	JP *rel8*	D	Valid	Valid	Jump short if parity (PF=1).
7A *cb*	JPE *rel8*	D	Valid	Valid	Jump short if parity even (PF=1).
7B *cb*	JPO *rel8*	D	Valid	Valid	Jump short if parity odd (PF=0).
78 *cb*	JS *rel8*	D	Valid	Valid	Jump short if sign (SF=1).
74 *cb*	JZ *rel8*	D	Valid	Valid	Jump short if zero (ZF = 1).
0F 87 *cw*	JA *rel16*	D	N.S.	Valid	Jump near if above (CF=0 and ZF=0). Not supported in 64-bit mode.
0F 87 *cd*	JA *rel32*	D	Valid	Valid	Jump near if above (CF=0 and ZF=0).
0F 83 *cw*	JAE *rel16*	D	N.S.	Valid	Jump near if above or equal (CF=0). Not supported in 64-bit mode.

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
0F 83 *cd*	JAE *rel32*	D	Valid	Valid	Jump near if above or equal (CF=0).
0F 82 *cw*	JB *rel16*	D	N.S.	Valid	Jump near if below (CF=1). Not supported in 64-bit mode.
0F 82 *cd*	JB *rel32*	D	Valid	Valid	Jump near if below (CF=1).
0F 86 *cw*	JBE *rel16*	D	N.S.	Valid	Jump near if below or equal (CF=1 or ZF=1). Not supported in 64-bit mode.
0F 86 *cd*	JBE *rel32*	D	Valid	Valid	Jump near if below or equal (CF=1 or ZF=1).
0F 82 *cw*	JC *rel16*	D	N.S.	Valid	Jump near if carry (CF=1). Not supported in 64-bit mode.
0F 82 *cd*	JC *rel32*	D	Valid	Valid	Jump near if carry (CF=1).
0F 84 *cw*	JE *rel16*	D	N.S.	Valid	Jump near if equal (ZF=1). Not supported in 64-bit mode.
0F 84 *cd*	JE *rel32*	D	Valid	Valid	Jump near if equal (ZF=1).
0F 84 *cw*	JZ *rel16*	D	N.S.	Valid	Jump near if 0 (ZF=1). Not supported in 64-bit mode.
0F 84 *cd*	JZ *rel32*	D	Valid	Valid	Jump near if 0 (ZF=1).
0F 8F *cw*	JG *rel16*	D	N.S.	Valid	Jump near if greater (ZF=0 and SF=OF). Not supported in 64-bit mode.
0F 8F *cd*	JG *rel32*	D	Valid	Valid	Jump near if greater (ZF=0 and SF=OF).
0F 8D *cw*	JGE *rel16*	D	N.S.	Valid	Jump near if greater or equal (SF=OF). Not supported in 64-bit mode.
0F 8D *cd*	JGE *rel32*	D	Valid	Valid	Jump near if greater or equal (SF=OF).
0F 8C *cw*	JL *rel16*	D	N.S.	Valid	Jump near if less (SF≠ OF). Not supported in 64-bit mode.
0F 8C *cd*	JL *rel32*	D	Valid	Valid	Jump near if less (SF≠ OF).
0F 8E *cw*	JLE *rel16*	D	N.S.	Valid	Jump near if less or equal (ZF=1 or SF≠ OF). Not supported in 64-bit mode.
0F 8E *cd*	JLE *rel32*	D	Valid	Valid	Jump near if less or equal (ZF=1 or SF≠ OF).
0F 86 *cw*	JNA *rel16*	D	N.S.	Valid	Jump near if not above (CF=1 or ZF=1). Not supported in 64-bit mode.
0F 86 *cd*	JNA *rel32*	D	Valid	Valid	Jump near if not above (CF=1 or ZF=1).
0F 82 *cw*	JNAE *rel16*	D	N.S.	Valid	Jump near if not above or equal (CF=1). Not supported in 64-bit mode.
0F 82 *cd*	JNAE *rel32*	D	Valid	Valid	Jump near if not above or equal (CF=1).
0F 83 *cw*	JNB *rel16*	D	N.S.	Valid	Jump near if not below (CF=0). Not supported in 64-bit mode.
0F 83 *cd*	JNB *rel32*	D	Valid	Valid	Jump near if not below (CF=0).
0F 87 *cw*	JNBE *rel16*	D	N.S.	Valid	Jump near if not below or equal (CF=0 and ZF=0). Not supported in 64-bit mode.
0F 87 *cd*	JNBE *rel32*	D	Valid	Valid	Jump near if not below or equal (CF=0 and ZF=0).
0F 83 *cw*	JNC *rel16*	D	N.S.	Valid	Jump near if not carry (CF=0). Not supported in 64-bit mode.
0F 83 *cd*	JNC *rel32*	D	Valid	Valid	Jump near if not carry (CF=0).

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
0F 85 *cw*	JNE *rel16*	D	N.S.	Valid	Jump near if not equal (ZF=0). Not supported in 64-bit mode.
0F 85 *cd*	JNE *rel32*	D	Valid	Valid	Jump near if not equal (ZF=0).
0F 8E *cw*	JNG *rel16*	D	N.S.	Valid	Jump near if not greater (ZF=1 or SF≠ OF). Not supported in 64-bit mode.
0F 8E *cd*	JNG *rel32*	D	Valid	Valid	Jump near if not greater (ZF=1 or SF≠ OF).
0F 8C *cw*	JNGE *rel16*	D	N.S.	Valid	Jump near if not greater or equal (SF≠ OF). Not supported in 64-bit mode.
0F 8C *cd*	JNGE *rel32*	D	Valid	Valid	Jump near if not greater or equal (SF≠ OF).
0F 8D *cw*	JNL *rel16*	D	N.S.	Valid	Jump near if not less (SF=OF). Not supported in 64-bit mode.
0F 8D *cd*	JNL *rel32*	D	Valid	Valid	Jump near if not less (SF=OF).
0F 8F *cw*	JNLE *rel16*	D	N.S.	Valid	Jump near if not less or equal (ZF=0 and SF=OF). Not supported in 64-bit mode.
0F 8F *cd*	JNLE *rel32*	D	Valid	Valid	Jump near if not less or equal (ZF=0 and SF=OF).
0F 81 *cw*	JNO *rel16*	D	N.S.	Valid	Jump near if not overflow (OF=0). Not supported in 64-bit mode.
0F 81 *cd*	JNO *rel32*	D	Valid	Valid	Jump near if not overflow (OF=0).
0F 8B *cw*	JNP *rel16*	D	N.S.	Valid	Jump near if not parity (PF=0). Not supported in 64-bit mode.
0F 8B *cd*	JNP *rel32*	D	Valid	Valid	Jump near if not parity (PF=0).
0F 89 *cw*	JNS *rel16*	D	N.S.	Valid	Jump near if not sign (SF=0). Not supported in 64-bit mode.
0F 89 *cd*	JNS *rel32*	D	Valid	Valid	Jump near if not sign (SF=0).
0F 85 *cw*	JNZ *rel16*	D	N.S.	Valid	Jump near if not zero (ZF=0). Not supported in 64-bit mode.
0F 85 *cd*	JNZ *rel32*	D	Valid	Valid	Jump near if not zero (ZF=0).
0F 80 *cw*	JO *rel16*	D	N.S.	Valid	Jump near if overflow (OF=1). Not supported in 64-bit mode.
0F 80 *cd*	JO *rel32*	D	Valid	Valid	Jump near if overflow (OF=1).
0F 8A *cw*	JP *rel16*	D	N.S.	Valid	Jump near if parity (PF=1). Not supported in 64-bit mode.
0F 8A *cd*	JP *rel32*	D	Valid	Valid	Jump near if parity (PF=1).
0F 8A *cw*	JPE *rel16*	D	N.S.	Valid	Jump near if parity even (PF=1). Not supported in 64-bit mode.
0F 8A *cd*	JPE *rel32*	D	Valid	Valid	Jump near if parity even (PF=1).
0F 8B *cw*	JPO *rel16*	D	N.S.	Valid	Jump near if parity odd (PF=0). Not supported in 64-bit mode.
0F 8B *cd*	JPO *rel32*	D	Valid	Valid	Jump near if parity odd (PF=0).
0F 88 *cw*	JS *rel16*	D	N.S.	Valid	Jump near if sign (SF=1). Not supported in 64-bit mode.

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

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
D	Offset	NA	NA	NA

Description

Checks the state of one or more of the status flags in the EFLAGS register (CF, OF, PF, SF, and ZF) and, if the flags are in the specified state (condition), performs a jump to the target instruction specified by the destination operand. A condition code (*cc*) is associated with each instruction to indicate the condition being tested for. If the condition is not satisfied, the jump is not performed and execution continues with the instruction following the J*cc* instruction.

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

The conditions for each J*cc* mnemonic are given in the "Description" column of the table on the preceding page. The terms "less" and "greater" are used for comparisons of signed integers and the terms "above" and "below" are used for unsigned integers.

Because a particular state of the status flags can sometimes be interpreted in two ways, two mnemonics are defined for some opcodes. For example, the JA (jump if above) instruction and the JNBE (jump if not below or equal) instruction are alternate mnemonics for the opcode 77H.

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

```
JZ FARLABEL;
```

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

```
JNZ BEYOND;
JMP FARLABEL;
BEYOND:
```

The JRCXZ, JECXZ and JCXZ instructions differ from other J*cc* instructions because they do not check status flags. Instead, they check RCX, ECX or CX for 0. The register checked is determined by the address-size attribute. These instructions are useful when used at the beginning of a loop that terminates with a conditional loop instruction (such as LOOPNE). They can be used to prevent an instruction sequence from entering a loop when RCX, ECX or CX is 0. This would cause the loop to execute 2^{64}, 2^{32} or 64K times (not zero times).

All conditional jumps are converted to code fetches of one or two cache lines, regardless of jump address or cacheability.

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

Operation

```
IF condition
    THEN
        tempEIP ← EIP + SignExtend(DEST);
        IF OperandSize = 16
            THEN tempEIP ← tempEIP AND 0000FFFFH;
        FI;
    IF tempEIP is not within code segment limit
        THEN #GP(0);
        ELSE EIP ← tempEIP
    FI;
FI;
```

Flags Affected

None

Protected Mode Exceptions

#GP(0)	If the offset being jumped to is beyond the limits of the CS segment.
#UD	If the LOCK prefix is used.

Real-Address Mode Exceptions

#GP	If the offset being jumped to is beyond the limits of the CS segment or is outside of the effective address space from 0 to FFFFH. This condition can occur if a 32-bit address size override prefix is used.
#UD	If the LOCK prefix is used.

Virtual-8086 Mode Exceptions

Same exceptions as in real address mode.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

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

JMP—Jump

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
EB *cb*	JMP *rel8*	D	Valid	Valid	Jump short, RIP = RIP + 8-bit displacement sign extended to 64-bits
E9 *cw*	JMP *rel16*	D	N.S.	Valid	Jump near, relative, displacement relative to next instruction. Not supported in 64-bit mode.
E9 *cd*	JMP *rel32*	D	Valid	Valid	Jump near, relative, RIP = RIP + 32-bit displacement sign extended to 64-bits
FF /4	JMP *r/m16*	M	N.S.	Valid	Jump near, absolute indirect, address = zero-extended *r/m16.* Not supported in 64-bit mode.
FF /4	JMP *r/m32*	M	N.S.	Valid	Jump near, absolute indirect, address given in *r/m32.* Not supported in 64-bit mode.
FF /4	JMP *r/m64*	M	Valid	N.E.	Jump near, absolute indirect, RIP = 64-Bit offset from register or memory
EA *cd*	JMP *ptr16:16*	D	Inv.	Valid	Jump far, absolute, address given in operand
EA *cp*	JMP *ptr16:32*	D	Inv.	Valid	Jump far, absolute, address given in operand
FF /5	JMP *m16:16*	D	Valid	Valid	Jump far, absolute indirect, address given in *m16:16*
FF /5	JMP *m16:32*	D	Valid	Valid	Jump far, absolute indirect, address given in *m16:32.*
REX.W + FF /5	JMP *m16:64*	D	Valid	N.E.	Jump far, absolute indirect, address given in *m16:64.*

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
D	Offset	NA	NA	NA
M	ModRM:r/m (r)	NA	NA	NA

Description

Transfers program control to a different point in the instruction stream without recording return information. The destination (target) operand specifies the address of the instruction being jumped to. This operand can be an immediate value, a general-purpose register, or a memory location.

This instruction can be used to execute four different types of jumps:

- Near jump—A jump to an instruction within the current code segment (the segment currently pointed to by the CS register), sometimes referred to as an intrasegment jump.
- Short jump—A near jump where the jump range is limited to –128 to +127 from the current EIP value.
- Far jump—A jump to an instruction located in a different segment than the current code segment but at the same privilege level, sometimes referred to as an intersegment jump.
- Task switch—A jump to an instruction located in a different task.

A task switch can only be executed in protected mode (see Chapter 7, in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*, for information on performing task switches with the JMP instruction).

Near and Short Jumps. When executing a near jump, the processor jumps to the address (within the current code segment) that is specified with the target operand. The target operand specifies either an absolute offset (that is an offset from the base of the code segment) or a relative offset (a signed displacement relative to the current

value of the instruction pointer in the EIP register). A near jump to a relative offset of 8-bits (*rel8*) is referred to as a short jump. The CS register is not changed on near and short jumps.

An absolute offset is specified indirectly in a general-purpose register or a memory location (*r/m16* or *r/m32*). The operand-size attribute determines the size of the target operand (16 or 32 bits). Absolute offsets are loaded directly into the EIP register. If the operand-size attribute is 16, the upper two bytes of the EIP register are cleared, resulting in a maximum instruction pointer size of 16 bits.

A relative offset (*rel8, rel16*, or *rel32*) is generally specified as a label in assembly code, but at the machine code level, it is encoded as a signed 8-, 16-, or 32-bit immediate value. This value is added to the value in the EIP register. (Here, the EIP register contains the address of the instruction following the JMP instruction). When using relative offsets, the opcode (for short vs. near jumps) and the operand-size attribute (for near relative jumps) determines the size of the target operand (8, 16, or 32 bits).

Far Jumps in Real-Address or Virtual-8086 Mode. When executing a far jump in real-address or virtual-8086 mode, the processor jumps to the code segment and offset specified with the target operand. Here the target operand specifies an absolute far address either directly with a pointer (*ptr16:16* or *ptr16:32*) or indirectly with a memory location (*m16:16* or *m16:32*). With the pointer method, the segment and address of the called procedure is encoded in the instruction, using a 4-byte (16-bit operand size) or 6-byte (32-bit operand size) far address immediate. With the indirect method, the target operand specifies a memory location that contains a 4-byte (16-bit operand size) or 6-byte (32-bit operand size) far address. The far address is loaded directly into the CS and EIP registers. If the operand-size attribute is 16, the upper two bytes of the EIP register are cleared.

Far Jumps in Protected Mode. When the processor is operating in protected mode, the JMP instruction can be used to perform the following three types of far jumps:

- A far jump to a conforming or non-conforming code segment.
- A far jump through a call gate.
- A task switch.

(The JMP instruction cannot be used to perform inter-privilege-level far jumps.)

In protected mode, the processor always uses the segment selector part of the far address to access the corresponding descriptor in the GDT or LDT. The descriptor type (code segment, call gate, task gate, or TSS) and access rights determine the type of jump to be performed.

If the selected descriptor is for a code segment, a far jump to a code segment at the same privilege level is performed. (If the selected code segment is at a different privilege level and the code segment is non-conforming, a general-protection exception is generated.) A far jump to the same privilege level in protected mode is very similar to one carried out in real-address or virtual-8086 mode. The target operand specifies an absolute far address either directly with a pointer (*ptr16:16* or *ptr16:32*) or indirectly with a memory location (*m16:16* or *m16:32*). The operand-size attribute determines the size of the offset (16 or 32 bits) in the far address. The new code segment selector and its descriptor are loaded into CS register, and the offset from the instruction is loaded into the EIP register. Note that a call gate (described in the next paragraph) can also be used to perform far call to a code segment at the same privilege level. Using this mechanism provides an extra level of indirection and is the preferred method of making jumps between 16-bit and 32-bit code segments.

When executing a far jump through a call gate, the segment selector specified by the target operand identifies the call gate. (The offset part of the target operand is ignored.) The processor then jumps to the code segment specified in the call gate descriptor and begins executing the instruction at the offset specified in the call gate. No stack switch occurs. Here again, the target operand can specify the far address of the call gate either directly with a pointer (*ptr16:16* or *ptr16:32*) or indirectly with a memory location (*m16:16* or *m16:32*).

Executing a task switch with the JMP instruction is somewhat similar to executing a jump through a call gate. Here the target operand specifies the segment selector of the task gate for the task being switched to (and the offset part of the target operand is ignored). The task gate in turn points to the TSS for the task, which contains the segment selectors for the task's code and stack segments. The TSS also contains the EIP value for the next instruction that was to be executed before the task was suspended. This instruction pointer value is loaded into the EIP register so that the task begins executing again at this next instruction.

The JMP instruction can also specify the segment selector of the TSS directly, which eliminates the indirection of the task gate. See Chapter 7 in *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*, for detailed information on the mechanics of a task switch.

Note that when you execute at task switch with a JMP instruction, the nested task flag (NT) is not set in the EFLAGS register and the new TSS's previous task link field is not loaded with the old task's TSS selector. A return to the previous task can thus not be carried out by executing the IRET instruction. Switching tasks with the JMP instruction differs in this regard from the CALL instruction which does set the NT flag and save the previous task link information, allowing a return to the calling task with an IRET instruction.

In 64-Bit Mode — The instruction's operation size is fixed at 64 bits. If a selector points to a gate, then RIP equals the 64-bit displacement taken from gate; else RIP equals the zero-extended offset from the far pointer referenced in the instruction.

See the summary chart at the beginning of this section for encoding data and limits.

Operation

```
IF near jump
    IF 64-bit Mode
        THEN
            IF near relative jump
             THEN
                tempRIP ← RIP + DEST; (* RIP is instruction following JMP instruction*)
             ELSE (* Near absolute jump *)
                tempRIP ← DEST;
            FI;
        ELSE
            IF near relative jump
             THEN
                tempEIP ← EIP + DEST; (* EIP is instruction following JMP instruction*)
             ELSE (* Near absolute jump *)
                tempEIP ← DEST;
            FI;
    FI;
    IF (IA32_EFER.LMA = 0 or target mode = Compatibility mode)
    and tempEIP outside code segment limit
        THEN #GP(0); FI
    IF 64-bit mode and tempRIP is not canonical
        THEN #GP(0);
    FI;
    IF OperandSize = 32
        THEN
            EIP ← tempEIP;
        ELSE
            IF OperandSize = 16
                THEN (* OperandSize = 16 *)
                    EIP ← tempEIP AND 0000FFFFH;
                ELSE (* OperandSize = 64)
                    RIP ← tempRIP;
            FI;
     FI;
FI;
IF far jump and (PE = 0 or (PE = 1 AND VM = 1)) (* Real-address or virtual-8086 mode *)
     THEN
        tempEIP ← DEST(Offset); (* DEST is ptr16:32 or [m16:32] *)
        IF tempEIP is beyond code segment limit
            THEN #GP(0); FI;
        CS ← DEST(segment selector); (* DEST is ptr16:32 or [m16:32] *)
        IF OperandSize = 32
```

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

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

CALL-GATE:
    IF call gate DPL < CPL
    or call gate DPL < call gate segment-selector RPL
            THEN #GP(call gate selector); FI;
    IF call gate not present
        THEN #NP(call gate selector); FI;
    IF call gate code-segment selector is NULL
        THEN #GP(0); FI;
    IF call gate code-segment selector index outside descriptor table limits
        THEN #GP(code segment selector); FI;
    Read code segment descriptor;
    IF code-segment segment descriptor does not indicate a code segment
    or code-segment segment descriptor is conforming and DPL > CPL
    or code-segment segment descriptor is non-conforming and DPL ≠ CPL
            THEN #GP(code segment selector); FI;
    IF IA32_EFER.LMA = 1 and (code-segment descriptor is not a 64-bit code segment
    or code-segment segment descriptor has both L-Bit and D-bit set)
            THEN #GP(code segment selector); FI;
    IF code segment is not present
        THEN #NP(code-segment selector); FI;
     tempEIP ← DEST(Offset);
     IF GateSize = 16
        THEN tempEIP ← tempEIP AND 0000FFFFH; FI;
    IF (IA32_EFER.LMA = 0 OR target mode = Compatibility mode) AND tempEIP
    outside code segment limit
        THEN #GP(0); FI
    CS ← DEST[SegmentSelector); (* Segment descriptor information also loaded *)
    CS(RPL) ← CPL;
    EIP ← tempEIP;
END;
TASK-GATE:
    IF task gate DPL < CPL
    or task gate DPL < task gate segment-selector RPL
        THEN #GP(task gate selector); FI;
    IF task gate not present
```

```
        THEN #NP(gate selector); FI;
    Read the TSS segment selector in the task-gate descriptor;
    IF TSS segment selector local/global bit is set to local
    or index not within GDT limits
    or descriptor is not a TSS segment
    or TSS descriptor specifies that the TSS is busy
        THEN #GP(TSS selector); FI;
     IF TSS not present
        THEN #NP(TSS selector); FI;
     SWITCH-TASKS to TSS;
     IF EIP not within code segment limit
        THEN #GP(0); FI;
END;
TASK-STATE-SEGMENT:
    IF TSS DPL < CPL
    or TSS DPL < TSS segment-selector RPL
    or TSS descriptor indicates TSS not available
        THEN #GP(TSS selector); FI;
    IF TSS is not present
        THEN #NP(TSS selector); FI;
    SWITCH-TASKS to TSS;
    IF EIP not within code segment limit
        THEN #GP(0); FI;
END;
```

Flags Affected

All flags are affected if a task switch occurs; no flags are affected if a task switch does not occur.

Protected Mode Exceptions

#GP(0)	If offset in target operand, call gate, or TSS is beyond the code segment limits.
	If the segment selector in the destination operand, call gate, task gate, or TSS is NULL.
	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
	If the DS, ES, FS, or GS register is used to access memory and it contains a NULL segment selector.
#GP(selector)	If the segment selector index is outside descriptor table limits.
	If the segment descriptor pointed to by the segment selector in the destination operand is not for a conforming-code segment, nonconforming-code segment, call gate, task gate, or task state segment.
	If the DPL for a nonconforming-code segment is not equal to the CPL
	(When not using a call gate.) If the RPL for the segment's segment selector is greater than the CPL.
	If the DPL for a conforming-code segment is greater than the CPL.
	If the DPL from a call-gate, task-gate, or TSS segment descriptor is less than the CPL or than the RPL of the call-gate, task-gate, or TSS's segment selector.
	If the segment descriptor for selector in a call gate does not indicate it is a code segment.
	If the segment descriptor for the segment selector in a task gate does not indicate an available TSS.
	If the segment selector for a TSS has its local/global bit set for local.
	If a TSS segment descriptor specifies that the TSS is busy or not available.
#SS(0)	If a memory operand effective address is outside the SS segment limit.

#NP (selector)	If the code segment being accessed is not present.
	If call gate, task gate, or TSS not present.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3. (Only occurs when fetching target from memory.)
#UD	If the LOCK prefix is used.

Real-Address Mode Exceptions

#GP	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS	If a memory operand effective address is outside the SS segment limit.
#UD	If the LOCK prefix is used.

Virtual-8086 Mode Exceptions

#GP(0)	If the target operand is beyond the code segment limits.
	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made. (Only occurs when fetching target from memory.)
#UD	If the LOCK prefix is used.

Compatibility Mode Exceptions

Same as 64-bit mode exceptions.

64-Bit Mode Exceptions

#GP(0)	If a memory address is non-canonical.
	If target offset in destination operand is non-canonical.
	If target offset in destination operand is beyond the new code segment limit.
	If the segment selector in the destination operand is NULL.
	If the code segment selector in the 64-bit gate is NULL.
#GP(selector)	If the code segment or 64-bit call gate is outside descriptor table limits.
	If the code segment or 64-bit call gate overlaps non-canonical space.
	If the segment descriptor from a 64-bit call gate is in non-canonical space.
	If the segment descriptor pointed to by the segment selector in the destination operand is not for a conforming-code segment, nonconforming-code segment, 64-bit call gate.
	If the segment descriptor pointed to by the segment selector in the destination operand is a code segment, and has both the D-bit and the L-bit set.
	If the DPL for a nonconforming-code segment is not equal to the CPL, or the RPL for the segment's segment selector is greater than the CPL.
	If the DPL for a conforming-code segment is greater than the CPL.
	If the DPL from a 64-bit call-gate is less than the CPL or than the RPL of the 64-bit call-gate.
	If the upper type field of a 64-bit call gate is not 0x0.
	If the segment selector from a 64-bit call gate is beyond the descriptor table limits.
	If the code segment descriptor pointed to by the selector in the 64-bit gate doesn't have the L-bit set and the D-bit clear.
	If the segment descriptor for a segment selector from the 64-bit call gate does not indicate it is a code segment.
	If the code segment is non-conforming and CPL ≠ DPL.

	If the code segment is confirming and CPL < DPL.
#NP(selector)	If a code segment or 64-bit call gate is not present.
#UD	(64-bit mode only) If a far jump is direct to an absolute address in memory.
	If the LOCK prefix is used.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

KADDW/KADDB/KADDQ/KADDD—ADD Two Masks

Opcode/ Instruction	Op/En	64/32 bit Mode Support	CPUID Feature Flag	Description
VEX.L1.0F.W0 4A /r KADDW k1, k2, k3	RVR	V/V	AVX512DQ	Add 16 bits masks in k2 and k3 and place result in k1.
VEX.L1.66.0F.W0 4A /r KADDB k1, k2, k3	RVR	V/V	AVX512DQ	Add 8 bits masks in k2 and k3 and place result in k1.
VEX.L1.0F.W1 4A /r KADDQ k1, k2, k3	RVR	V/V	AVX512BW	Add 64 bits masks in k2 and k3 and place result in k1.
VEX.L1.66.0F.W1 4A /r KADDD k1, k2, k3	RVR	V/V	AVX512BW	Add 32 bits masks in k2 and k3 and place result in k1.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3
RVR	ModRM:reg (w)	VEX.1vvv (r)	ModRM:r/m (r, ModRM:[7:6] must be 11b)

Description

Adds the vector mask k2 and the vector mask k3, and writes the result into vector mask k1.

Operation

KADDW

DEST[15:0] ← SRC1[15:0] + SRC2[15:0]
DEST[MAX_KL-1:16] ← 0

KADDB

DEST[7:0] ← SRC1[7:0] + SRC2[7:0]
DEST[MAX_KL-1:8] ← 0

KADDQ

DEST[63:0] ← SRC1[63:0] + SRC2[63:0]
DEST[MAX_KL-1:64] ← 0

KADDD

DEST[31:0] ← SRC1[31:0] + SRC2[31:0]
DEST[MAX_KL-1:32] ← 0

Intel C/C++ Compiler Intrinsic Equivalent

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type K20.

KANDW/KANDB/KANDQ/KANDD—Bitwise Logical AND Masks

Opcode/ Instruction	Op/En	64/32 bit Mode Support	CPUID Feature Flag	Description
VEX.NDS.L1.0F.W0 41 /r KANDW k1, k2, k3	RVR	V/V	AVX512F	Bitwise AND 16 bits masks k2 and k3 and place result in k1.
VEX.L1.66.0F.W0 41 /r KANDB k1, k2, k3	RVR	V/V	AVX512DQ	Bitwise AND 8 bits masks k2 and k3 and place result in k1.
VEX.L1.0F.W1 41 /r KANDQ k1, k2, k3	RVR	V/V	AVX512BW	Bitwise AND 64 bits masks k2 and k3 and place result in k1.
VEX.L1.66.0F.W1 41 /r KANDD k1, k2, k3	RVR	V/V	AVX512BW	Bitwise AND 32 bits masks k2 and k3 and place result in k1.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3
RVR	ModRM:reg (w)	VEX.1vvv (r)	ModRM:r/m (r, ModRM:[7:6] must be 11b)

Description

Performs a bitwise AND between the vector mask k2 and the vector mask k3, and writes the result into vector mask k1.

Operation

KANDW

```
DEST[15:0] ← SRC1[15:0] BITWISE AND SRC2[15:0]
DEST[MAX_KL-1:16] ← 0
```

KANDB

```
DEST[7:0] ← SRC1[7:0] BITWISE AND SRC2[7:0]
DEST[MAX_KL-1:8] ← 0
```

KANDQ

```
DEST[63:0] ← SRC1[63:0] BITWISE AND SRC2[63:0]
DEST[MAX_KL-1:64] ← 0
```

KANDD

```
DEST[31:0] ← SRC1[31:0] BITWISE AND SRC2[31:0]
DEST[MAX_KL-1:32] ← 0
```

Intel C/C++ Compiler Intrinsic Equivalent

KANDW __mmask16 _mm512_kand(__mmask16 a, __mmask16 b);

Flags Affected

None

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type K20.

KANDNW/KANDNB/KANDNQ/KANDND—Bitwise Logical AND NOT Masks

Opcode/ Instruction	Op/En	64/32 bit Mode Support	CPUID Feature Flag	Description
VEX.NDS.L1.0F.W0 42 /r KANDNW k1, k2, k3	RVR	V/V	AVX512F	Bitwise AND NOT 16 bits masks k2 and k3 and place result in k1.
VEX.L1.66.0F.W0 42 /r KANDNB k1, k2, k3	RVR	V/V	AVX512DQ	Bitwise AND NOT 8 bits masks k1 and k2 and place result in k1.
VEX.L1.0F.W1 42 /r KANDNQ k1, k2, k3	RVR	V/V	AVX512BW	Bitwise AND NOT 64 bits masks k2 and k3 and place result in k1.
VEX.L1.66.0F.W1 42 /r KANDND k1, k2, k3	RVR	V/V	AVX512BW	Bitwise AND NOT 32 bits masks k2 and k3 and place result in k1.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3
RVR	ModRM:reg (w)	VEX.1vvv (r)	ModRM:r/m (r, ModRM:[7:6] must be 11b)

Description

Performs a bitwise AND NOT between the vector mask k2 and the vector mask k3, and writes the result into vector mask k1.

Operation

KANDNW

```
DEST[15:0] ← (BITWISE NOT SRC1[15:0]) BITWISE AND SRC2[15:0]
DEST[MAX_KL-1:16] ← 0
```

KANDNB

```
DEST[7:0] ← (BITWISE NOT SRC1[7:0]) BITWISE AND SRC2[7:0]
DEST[MAX_KL-1:8] ← 0
```

KANDNQ

```
DEST[63:0] ← (BITWISE NOT SRC1[63:0]) BITWISE AND SRC2[63:0]
DEST[MAX_KL-1:64] ← 0
```

KANDND

```
DEST[31:0] ← (BITWISE NOT SRC1[31:0]) BITWISE AND SRC2[31:0]
DEST[MAX_KL-1:32] ← 0
```

Intel C/C++ Compiler Intrinsic Equivalent

KANDNW __mmask16 _mm512_kandn(__mmask16 a, __mmask16 b);

Flags Affected

None

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type K20.

KMOVW/KMOVB/KMOVQ/KMOVD—Move from and to Mask Registers

Opcode/ Instruction	Op/En	64/32 bit Mode Support	CPUID Feature Flag	Description
VEX.L0.0F.W0 90 /r KMOVW k1, k2/m16	RM	V/V	AVX512F	Move 16 bits mask from k2/m16 and store the result in k1.
VEX.L0.66.0F.W0 90 /r KMOVB k1, k2/m8	RM	V/V	AVX512DQ	Move 8 bits mask from k2/m8 and store the result in k1.
VEX.L0.0F.W1 90 /r KMOVQ k1, k2/m64	RM	V/V	AVX512BW	Move 64 bits mask from k2/m64 and store the result in k1.
VEX.L0.66.0F.W1 90 /r KMOVD k1, k2/m32	RM	V/V	AVX512BW	Move 32 bits mask from k2/m32 and store the result in k1.
VEX.L0.0F.W0 91 /r KMOVW m16, k1	MR	V/V	AVX512F	Move 16 bits mask from k1 and store the result in m16.
VEX.L0.66.0F.W0 91 /r KMOVB m8, k1	MR	V/V	AVX512DQ	Move 8 bits mask from k1 and store the result in m8.
VEX.L0.0F.W1 91 /r KMOVQ m64, k1	MR	V/V	AVX512BW	Move 64 bits mask from k1 and store the result in m64.
VEX.L0.66.0F.W1 91 /r KMOVD m32, k1	MR	V/V	AVX512BW	Move 32 bits mask from k1 and store the result in m32.
VEX.L0.0F.W0 92 /r KMOVW k1, r32	RR	V/V	AVX512F	Move 16 bits mask from r32 to k1.
VEX.L0.66.0F.W0 92 /r KMOVB k1, r32	RR	V/V	AVX512DQ	Move 8 bits mask from r32 to k1.
VEX.L0.F2.0F.W1 92 /r KMOVQ k1, r64	RR	V/I	AVX512BW	Move 64 bits mask from r64 to k1.
VEX.L0.F2.0F.W0 92 /r KMOVD k1, r32	RR	V/V	AVX512BW	Move 32 bits mask from r32 to k1.
VEX.L0.0F.W0 93 /r KMOVW r32, k1	RR	V/V	AVX512F	Move 16 bits mask from k1 to r32.
VEX.L0.66.0F.W0 93 /r KMOVB r32, k1	RR	V/V	AVX512DQ	Move 8 bits mask from k1 to r32.
VEX.L0.F2.0F.W1 93 /r KMOVQ r64, k1	RR	V/I	AVX512BW	Move 64 bits mask from k1 to r64.
VEX.L0.F2.0F.W0 93 /r KMOVD r32, k1	RR	V/V	AVX512BW	Move 32 bits mask from k1 to r32.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2
RM	ModRM:reg (w)	ModRM:r/m (r)
MR	ModRM:r/m (w, ModRM:[7:6] must not be 11b)	ModRM:reg (r)
RR	ModRM:reg (w)	ModRM:r/m (r, ModRM:[7:6] must be 11b)

Description

Copies values from the source operand (second operand) to the destination operand (first operand). The source and destination operands can be mask registers, memory location or general purpose. The instruction cannot be used to transfer data between general purpose registers and or memory locations.

When moving to a mask register, the result is zero extended to MAX_KL size (i.e., 64 bits currently). When moving to a general-purpose register (GPR), the result is zero-extended to the size of the destination. In 32-bit mode, the default GPR destination's size is 32 bits. In 64-bit mode, the default GPR destination's size is 64 bits. Note that REX.W cannot be used to modify the size of the general-purpose destination.

Operation

KMOVW

```
IF *destination is a memory location*
    DEST[15:0] ← SRC[15:0]
IF *destination is a mask register or a GPR *
    DEST ← ZeroExtension(SRC[15:0])
```

KMOVB

```
IF *destination is a memory location*
    DEST[7:0] ← SRC[7:0]
IF *destination is a mask register or a GPR *
    DEST ← ZeroExtension(SRC[7:0])
```

KMOVQ

```
IF *destination is a memory location or a GPR*
    DEST[63:0] ← SRC[63:0]
IF *destination is a mask register*
    DEST ← ZeroExtension(SRC[63:0])
```

KMOVD

```
IF *destination is a memory location*
    DEST[31:0] ← SRC[31:0]
IF *destination is a mask register or a GPR *
    DEST ← ZeroExtension(SRC[31:0])
```

Intel C/C++ Compiler Intrinsic Equivalent

KMOVW __mmask16 _mm512_kmov(__mmask16 a);

Flags Affected

None

SIMD Floating-Point Exceptions

None

Other Exceptions

Instructions with RR operand encoding See Exceptions Type K20.

Instructions with RM or MR operand encoding See Exceptions Type K21.

KNOTW/KNOTB/KNOTQ/KNOTD—NOT Mask Register

Opcode/ Instruction	Op/En	64/32 bit Mode Support	CPUID Feature Flag	Description
VEX.L0.0F.W0 44 /r KNOTW k1, k2	RR	V/V	AVX512F	Bitwise NOT of 16 bits mask k2.
VEX.L0.66.0F.W0 44 /r KNOTB k1, k2	RR	V/V	AVX512DQ	Bitwise NOT of 8 bits mask k2.
VEX.L0.0F.W1 44 /r KNOTQ k1, k2	RR	V/V	AVX512BW	Bitwise NOT of 64 bits mask k2.
VEX.L0.66.0F.W1 44 /r KNOTD k1, k2	RR	V/V	AVX512BW	Bitwise NOT of 32 bits mask k2.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2
RR	ModRM:reg (w)	ModRM:r/m (r, ModRM:[7:6] must be 11b)

Description

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

Operation

KNOTW

```
DEST[15:0] ← BITWISE NOT SRC[15:0]
DEST[MAX_KL-1:16] ← 0
```

KNOTB

```
DEST[7:0] ← BITWISE NOT SRC[7:0]
DEST[MAX_KL-1:8] ← 0
```

KNOTQ

```
DEST[63:0] ← BITWISE NOT SRC[63:0]
DEST[MAX_KL-1:64] ← 0
```

KNOTD

```
DEST[31:0] ← BITWISE NOT SRC[31:0]
DEST[MAX_KL-1:32] ← 0
```

Intel C/C++ Compiler Intrinsic Equivalent

KNOTW __mmask16 _mm512_knot(__mmask16 a);

Flags Affected

None

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type K20.

KORW/KORB/KORQ/KORD—Bitwise Logical OR Masks

Opcode/ Instruction	Op/En	64/32 bit Mode Support	CPUID Feature Flag	Description
VEX.NDS.L1.0F.W0 45 /r KORW k1, k2, k3	RVR	V/V	AVX512F	Bitwise OR 16 bits masks k2 and k3 and place result in k1.
VEX.L1.66.0F.W0 45 /r KORB k1, k2, k3	RVR	V/V	AVX512DQ	Bitwise OR 8 bits masks k2 and k3 and place result in k1.
VEX.L1.0F.W1 45 /r KORQ k1, k2, k3	RVR	V/V	AVX512BW	Bitwise OR 64 bits masks k2 and k3 and place result in k1.
VEX.L1.66.0F.W1 45 /r KORD k1, k2, k3	RVR	V/V	AVX512BW	Bitwise OR 32 bits masks k2 and k3 and place result in k1.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3
RVR	ModRM:reg (w)	VEX.1vvv (r)	ModRM:r/m (r, ModRM:[7:6] must be 11b)

Description

Performs a bitwise OR between the vector mask k2 and the vector mask k3, and writes the result into vector mask k1 (three-operand form).

Operation

KORW

```
DEST[15:0] ← SRC1[15:0] BITWISE OR SRC2[15:0]
DEST[MAX_KL-1:16] ← 0
```

KORB

```
DEST[7:0] ← SRC1[7:0] BITWISE OR SRC2[7:0]
DEST[MAX_KL-1:8] ← 0
```

KORQ

```
DEST[63:0] ← SRC1[63:0] BITWISE OR SRC2[63:0]
DEST[MAX_KL-1:64] ← 0
```

KORD

```
DEST[31:0] ← SRC1[31:0] BITWISE OR SRC2[31:0]
DEST[MAX_KL-1:32] ← 0
```

Intel C/C++ Compiler Intrinsic Equivalent

KORW __mmask16 _mm512_kor(__mmask16 a, __mmask16 b);

Flags Affected

None

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type K20.

KORTESTW/KORTESTB/KORTESTQ/KORTESTD—OR Masks And Set Flags

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
VEX.L0.0F.W0 98 /r KORTESTW k1, k2	RR	V/V	AVX512F	Bitwise OR 16 bits masks k1 and k2 and update ZF and CF accordingly.
VEX.L0.66.0F.W0 98 /r KORTESTB k1, k2	RR	V/V	AVX512DQ	Bitwise OR 8 bits masks k1 and k2 and update ZF and CF accordingly.
VEX.L0.0F.W1 98 /r KORTESTQ k1, k2	RR	V/V	AVX512BW	Bitwise OR 64 bits masks k1 and k2 and update ZF and CF accordingly.
VEX.L0.66.0F.W1 98 /r KORTESTD k1, k2	RR	V/V	AVX512BW	Bitwise OR 32 bits masks k1 and k2 and update ZF and CF accordingly.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2
RR	ModRM:reg (w)	ModRM:r/m (r, ModRM:[7:6] must be 11b)

Description

Performs a bitwise OR between the vector mask register k2, and the vector mask register k1, and sets CF and ZF based on the operation result.

ZF flag is set if both sources are 0x0. CF is set if, after the OR operation is done, the operation result is all 1's.

Operation

KORTESTW

```
TMP[15:0] ← DEST[15:0] BITWISE OR SRC[15:0]
IF(TMP[15:0]=0)
    THEN ZF ← 1
    ELSE ZF ← 0
FI;
IF(TMP[15:0]=FFFFh)
    THEN CF ← 1
    ELSE CF ← 0
FI;
```

KORTESTB

```
TMP[7:0] ← DEST[7:0] BITWISE OR SRC[7:0]
IF(TMP[7:0]=0)
    THEN ZF ← 1
    ELSE ZF ← 0
FI;
IF(TMP[7:0]==FFh)
    THEN CF ← 1
    ELSE CF ← 0

FI;
```

KORTESTQ

```
TMP[63:0] ← DEST[63:0] BITWISE OR SRC[63:0]
IF(TMP[63:0]=0)
    THEN ZF ← 1
    ELSE ZF ← 0
FI;
IF(TMP[63:0]==FFFFFFFF_FFFFFFFFh)
    THEN CF ← 1
    ELSE CF ← 0
FI;
```

KORTESTD

```
TMP[31:0] ← DEST[31:0] BITWISE OR SRC[31:0]
IF(TMP[31:0]=0)
    THEN ZF ← 1
    ELSE ZF ← 0
FI;
IF(TMP[31:0]=FFFFFFFFh)
    THEN CF ← 1
    ELSE CF ← 0
FI;
```

Intel C/C++ Compiler Intrinsic Equivalent

KORTESTW __mmask16 _mm512_kortest[cz](__mmask16 a, __mmask16 b);

Flags Affected

The ZF flag is set if the result of OR-ing both sources is all 0s.
The CF flag is set if the result of OR-ing both sources is all 1s.
The OF, SF, AF, and PF flags are set to 0.

Other Exceptions

See Exceptions Type K20.

KSHIFTLW/KSHIFTLB/KSHIFTLQ/KSHIFTLD—Shift Left Mask Registers

Opcode/ Instruction	Op/En	64/32 bit Mode Support	CPUID Feature Flag	Description
VEX.L0.66.0F3A.W1 32 /r KSHIFTLW k1, k2, imm8	RRI	V/V	AVX512F	Shift left 16 bits in k2 by immediate and write result in k1.
VEX.L0.66.0F3A.W0 32 /r KSHIFTLB k1, k2, imm8	RRI	V/V	AVX512DQ	Shift left 8 bits in k2 by immediate and write result in k1.
VEX.L0.66.0F3A.W1 33 /r KSHIFTLQ k1, k2, imm8	RRI	V/V	AVX512BW	Shift left 64 bits in k2 by immediate and write result in k1.
VEX.L0.66.0F3A.W0 33 /r KSHIFTLD k1, k2, imm8	RRI	V/V	AVX512BW	Shift left 32 bits in k2 by immediate and write result in k1.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3
RRI	ModRM:reg (w)	ModRM:r/m (r, ModRM:[7:6] must be 11b)	Imm8

Description

Shifts 8/16/32/64 bits in the second operand (source operand) left by the count specified in immediate byte and place the least significant 8/16/32/64 bits of the result in the destination operand. The higher bits of the destination are zero-extended. The destination is set to zero if the count value is greater than 7 (for byte shift), 15 (for word shift), 31 (for doubleword shift) or 63 (for quadword shift).

Operation

KSHIFTLW

```
COUNT ← imm8[7:0]
DEST[MAX_KL-1:0] ← 0
IF COUNT <=15
    THEN DEST[15:0] ← SRC1[15:0] << COUNT;
FI;
```

KSHIFTLB

```
COUNT ← imm8[7:0]
DEST[MAX_KL-1:0] ← 0
IF COUNT <=7
        THEN    DEST[7:0] ← SRC1[7:0] << COUNT;
FI;
```

KSHIFTLQ

```
COUNT ← imm8[7:0]
DEST[MAX_KL-1:0] ← 0
IF COUNT <=63
        THEN    DEST[63:0] ← SRC1[63:0] << COUNT;
FI;
```

KSHIFTLD

```
COUNT ← imm8[7:0]
DEST[MAX_KL-1:0] ← 0
IF COUNT <=31
        THEN    DEST[31:0] ← SRC1[31:0] << COUNT;
FI;
```

Intel C/C++ Compiler Intrinsic Equivalent

Compiler auto generates KSHIFTLW when needed.

Flags Affected

None

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type K20.

KSHIFTRW/KSHIFTRB/KSHIFTRQ/KSHIFTRD—Shift Right Mask Registers

Opcode/ Instruction	Op/En	64/32 bit Mode Support	CPUID Feature Flag	Description
VEX.L0.66.0F3A.W1 30 /r KSHIFTRW k1, k2, imm8	RRI	V/V	AVX512F	Shift right 16 bits in k2 by immediate and write result in k1.
VEX.L0.66.0F3A.W0 30 /r KSHIFTRB k1, k2, imm8	RRI	V/V	AVX512DQ	Shift right 8 bits in k2 by immediate and write result in k1.
VEX.L0.66.0F3A.W1 31 /r KSHIFTRQ k1, k2, imm8	RRI	V/V	AVX512BW	Shift right 64 bits in k2 by immediate and write result in k1.
VEX.L0.66.0F3A.W0 31 /r KSHIFTRD k1, k2, imm8	RRI	V/V	AVX512BW	Shift right 32 bits in k2 by immediate and write result in k1.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3
RRI	ModRM:reg (w)	ModRM:r/m (r, ModRM:[7:6] must be 11b)	Imm8

Description

Shifts 8/16/32/64 bits in the second operand (source operand) right by the count specified in immediate and place the least significant 8/16/32/64 bits of the result in the destination operand. The higher bits of the destination are zero-extended. The destination is set to zero if the count value is greater than 7 (for byte shift), 15 (for word shift), 31 (for doubleword shift) or 63 (for quadword shift).

Operation

KSHIFTRW

```
COUNT ← imm8[7:0]
DEST[MAX_KL-1:0] ← 0
IF COUNT <=15
    THEN DEST[15:0] ← SRC1[15:0] >> COUNT;
FI;
```

KSHIFTRB

```
COUNT ← imm8[7:0]
DEST[MAX_KL-1:0] ← 0
IF COUNT <=7
        THEN    DEST[7:0] ← SRC1[7:0] >> COUNT;
FI;
```

KSHIFTRQ

```
COUNT ← imm8[7:0]
DEST[MAX_KL-1:0] ← 0
IF COUNT <=63
        THEN    DEST[63:0] ← SRC1[63:0] >> COUNT;
FI;
```

KSHIFTRD

```
COUNT ← imm8[7:0]
DEST[MAX_KL-1:0] ← 0
IF COUNT <=31
        THEN    DEST[31:0] ← SRC1[31:0] >> COUNT;
FI;
```

Intel C/C++ Compiler Intrinsic Equivalent

Compiler auto generates KSHIFTRW when needed.

Flags Affected

None

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type K20.

KTESTW/KTESTB/KTESTQ/KTESTD—Packed Bit Test Masks and Set Flags

Opcode/ Instruction	Op En	64/32 bit Mode Support	CPUID Feature Flag	Description
VEX.L0.0F.W0 99 /r KTESTW k1, k2	RR	V/V	AVX512DQ	Set ZF and CF depending on sign bit AND and ANDN of 16 bits mask register sources.
VEX.L0.66.0F.W0 99 /r KTESTB k1, k2	RR	V/V	AVX512DQ	Set ZF and CF depending on sign bit AND and ANDN of 8 bits mask register sources.
VEX.L0.0F.W1 99 /r KTESTQ k1, k2	RR	V/V	AVX512BW	Set ZF and CF depending on sign bit AND and ANDN of 64 bits mask register sources.
VEX.L0.66.0F.W1 99 /r KTESTD k1, k2	RR	V/V	AVX512BW	Set ZF and CF depending on sign bit AND and ANDN of 32 bits mask register sources.

Instruction Operand Encoding

Op/En	Operand 1	Operand2
RR	ModRM:reg (r)	ModRM:r/m (r, ModRM:[7:6] must be 11b)

Description

Performs a bitwise comparison of the bits of the first source operand and corresponding bits in the second source operand. If the AND operation produces all zeros, the ZF is set else the ZF is clear. If the bitwise AND operation of the inverted first source operand with the second source operand produces all zeros the CF is set else the CF is clear. Only the EFLAGS register is updated.

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

Operation

KTESTW

```
TEMP[15:0] ← SRC2[15:0] AND SRC1[15:0]
IF (TEMP[15:0] = = 0)
    THEN ZF ←1;
    ELSE ZF ← 0;
FI;
TEMP[15:0] ← SRC2[15:0] AND NOT SRC1[15:0]
IF (TEMP[15:0] = = 0)
    THEN CF ←1;
    ELSE CF ← 0;
FI;
AF ← OF ← PF ← SF ← 0;
```

KTESTB

```
TEMP[7:0] ← SRC2[7:0] AND SRC1[7:0]
IF (TEMP[7:0] = = 0)
    THEN ZF ←1;
    ELSE ZF ← 0;
FI;
TEMP[7:0] ← SRC2[7:0] AND NOT SRC1[7:0]
IF (TEMP[7:0] = = 0)
    THEN CF ←1;
    ELSE CF ← 0;
FI;
AF ← OF ← PF ← SF ← 0;
```

KTESTQ

```
TEMP[63:0] ← SRC2[63:0] AND SRC1[63:0]
IF (TEMP[63:0] = = 0)
    THEN ZF ←1;
    ELSE ZF ← 0;
FI;
TEMP[63:0] ← SRC2[63:0] AND NOT SRC1[63:0]
IF (TEMP[63:0] = = 0)
    THEN CF ←1;
    ELSE CF ← 0;
FI;
AF ← OF ← PF ← SF ← 0;
```

KTESTD

```
TEMP[31:0] ← SRC2[31:0] AND SRC1[31:0]
IF (TEMP[31:0] = = 0)
    THEN ZF ←1;
    ELSE ZF ← 0;
FI;
TEMP[31:0] ← SRC2[31:0] AND NOT SRC1[31:0]
IF (TEMP[31:0] = = 0)
    THEN CF ←1;
    ELSE CF ← 0;
FI;
AF ← OF ← PF ← SF ← 0;
```

Intel C/C++ Compiler Intrinsic Equivalent

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type K20.

KUNPCKBW/KUNPCKWD/KUNPCKDQ—Unpack for Mask Registers

Opcode/ Instruction	Op/En	64/32 bit Mode Support	CPUID Feature Flag	Description
VEX.NDS.L1.66.0F.W0 4B /r KUNPCKBW k1, k2, k3	RVR	V/V	AVX512F	Unpack and interleave 8 bits masks in k2 and k3 and write word result in k1.
VEX.NDS.L1.0F.W0 4B /r KUNPCKWD k1, k2, k3	RVR	V/V	AVX512BW	Unpack and interleave 16 bits in k2 and k3 and write double-word result in k1.
VEX.NDS.L1.0F.W1 4B /r KUNPCKDQ k1, k2, k3	RVR	V/V	AVX512BW	Unpack and interleave 32 bits masks in k2 and k3 and write quadword result in k1.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3
RVR	ModRM:reg (w)	VEX.1vvv (r)	ModRM:r/m (r, ModRM:[7:6] must be 11b)

Description

Unpacks the lower 8/16/32 bits of the second and third operands (source operands) into the low part of the first operand (destination operand), starting from the low bytes. The result is zero-extended in the destination.

Operation

KUNPCKBW

DEST[7:0] ← SRC2[7:0]
DEST[15:8] ← SRC1[7:0]
DEST[MAX_KL-1:16] ← 0

KUNPCKWD

DEST[15:0] ← SRC2[15:0]
DEST[31:16] ← SRC1[15:0]
DEST[MAX_KL-1:32] ← 0

KUNPCKDQ

DEST[31:0] ← SRC2[31:0]
DEST[63:32] ← SRC1[31:0]
DEST[MAX_KL-1:64] ← 0

Intel C/C++ Compiler Intrinsic Equivalent

KUNPCKBW __mmask16 _mm512_kunpackb(__mmask16 a, __mmask16 b);
KUNPCKDQ __mmask64 _mm512_kunpackd(__mmask64 a, __mmask64 b);
KUNPCKWD __mmask32 _mm512_kunpackw(__mmask32 a, __mmask32 b);

Flags Affected

None

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type K20.

KXNORW/KXNORB/KXNORQ/KXNORD—Bitwise Logical XNOR Masks

Opcode/ Instruction	Op/En	64/32 bit Mode Support	CPUID Feature Flag	Description
VEX.NDS.L1.0F.W0 46 /r KXNORW k1, k2, k3	RVR	V/V	AVX512F	Bitwise XNOR 16 bits masks k2 and k3 and place result in k1.
VEX.L1.66.0F.W0 46 /r KXNORB k1, k2, k3	RVR	V/V	AVX512DQ	Bitwise XNOR 8 bits masks k2 and k3 and place result in k1.
VEX.L1.0F.W1 46 /r KXNORQ k1, k2, k3	RVR	V/V	AVX512BW	Bitwise XNOR 64 bits masks k2 and k3 and place result in k1.
VEX.L1.66.0F.W1 46 /r KXNORD k1, k2, k3	RVR	V/V	AVX512BW	Bitwise XNOR 32 bits masks k2 and k3 and place result in k1.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3
RVR	ModRM:reg (w)	VEX.1vvv (r)	ModRM:r/m (r, ModRM:[7:6] must be 11b)

Description

Performs a bitwise XNOR between the vector mask k2 and the vector mask k3, and writes the result into vector mask k1 (three-operand form).

Operation

KXNORW

```
DEST[15:0] ← NOT (SRC1[15:0] BITWISE XOR SRC2[15:0])
DEST[MAX_KL-1:16] ← 0
```

KXNORB

```
DEST[7:0] ← NOT (SRC1[7:0] BITWISE XOR SRC2[7:0])
DEST[MAX_KL-1:8] ← 0
```

KXNORQ

```
DEST[63:0] ← NOT (SRC1[63:0] BITWISE XOR SRC2[63:0])
DEST[MAX_KL-1:64] ← 0
```

KXNORD

```
DEST[31:0] ← NOT (SRC1[31:0] BITWISE XOR SRC2[31:0])
DEST[MAX_KL-1:32] ← 0
```

Intel C/C++ Compiler Intrinsic Equivalent

KXNORW __mmask16 _mm512_kxnor(__mmask16 a, __mmask16 b);

Flags Affected

None

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type K20.

KXORW/KXORB/KXORQ/KXORD—Bitwise Logical XOR Masks

Opcode/ Instruction	Op/En	64/32 bit Mode Support	CPUID Feature Flag	Description
VEX.NDS.L1.0F.W0 47 /r KXORW k1, k2, k3	RVR	V/V	AVX512F	Bitwise XOR 16 bits masks k2 and k3 and place result in k1.
VEX.L1.66.0F.W0 47 /r KXORB k1, k2, k3	RVR	V/V	AVX512DQ	Bitwise XOR 8 bits masks k2 and k3 and place result in k1.
VEX.L1.0F.W1 47 /r KXORQ k1, k2, k3	RVR	V/V	AVX512BW	Bitwise XOR 64 bits masks k2 and k3 and place result in k1.
VEX.L1.66.0F.W1 47 /r KXORD k1, k2, k3	RVR	V/V	AVX512BW	Bitwise XOR 32 bits masks k2 and k3 and place result in k1.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3
RVR	ModRM:reg (w)	VEX.1vvv (r)	ModRM:r/m (r, ModRM:[7:6] must be 11b)

Description

Performs a bitwise XOR between the vector mask k2 and the vector mask k3, and writes the result into vector mask k1 (three-operand form).

Operation

KXORW

```
DEST[15:0] ← SRC1[15:0] BITWISE XOR SRC2[15:0]
DEST[MAX_KL-1:16] ← 0
```

KXORB

```
DEST[7:0] ← SRC1[7:0] BITWISE XOR SRC2[7:0]
DEST[MAX_KL-1:8] ← 0
```

KXORQ

```
DEST[63:0] ← SRC1[63:0] BITWISE XOR SRC2[63:0]
DEST[MAX_KL-1:64] ← 0
```

KXORD

```
DEST[31:0] ← SRC1[31:0] BITWISE XOR SRC2[31:0]
DEST[MAX_KL-1:32] ← 0
```

Intel C/C++ Compiler Intrinsic Equivalent

KXORW __mmask16 _mm512_kxor(__mmask16 a, __mmask16 b);

Flags Affected

None

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type K20.

LAHF—Load Status Flags into AH Register

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
9F	LAHF	ZO	Invalid*	Valid	Load: AH ← EFLAGS(SF:ZF:0:AF:0:PF:1:CF).

NOTES:
*Valid in specific steppings. See Description section.

Instruction Operand Encoding

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

Description

This instruction executes as described above in compatibility mode and legacy mode. It is valid in 64-bit mode only if CPUID.80000001H:ECX.LAHF-SAHF[bit 0] = 1.

Operation

```
IF 64-Bit Mode
    THEN
        IF CPUID.80000001H:ECX.LAHF-SAHF[bit 0] = 1;
            THEN AH ← RFLAGS(SF:ZF:0:AF:0:PF:1:CF);
            ELSE #UD;
        FI;
    ELSE
        AH ← EFLAGS(SF:ZF:0:AF:0:PF:1:CF);
FI;
```

Flags Affected

None. The state of the flags in the EFLAGS register is not affected.

Protected Mode Exceptions

#UD	If the LOCK prefix is used.

Real-Address Mode Exceptions

Same exceptions as in protected mode.

Virtual-8086 Mode Exceptions

Same exceptions as in protected mode.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#UD	If CPUID.80000001H:ECX.LAHF-SAHF[bit 0] = 0.
	If the LOCK prefix is used.

LAR—Load Access Rights Byte

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
0F 02 /r	LAR *r16, r16/m16*	RM	Valid	Valid	*r16* ← access rights referenced by *r16/m16*
0F 02 /r	LAR *reg, r32/m16*[1]	RM	Valid	Valid	*reg* ← access rights referenced by *r32/m16*

NOTES:

1. For all loads (regardless of source or destination sizing) only bits 16-0 are used. Other bits are ignored.

Instruction Operand Encoding

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

Description

Loads the access rights from the segment descriptor specified by the second operand (source operand) into the first operand (destination operand) and sets the ZF flag in the flag register. The source operand (which can be a register or a memory location) contains the segment selector for the segment descriptor being accessed. If the source operand is a memory address, only 16 bits of data are accessed. The destination operand is a general-purpose register.

The processor performs access checks as part of the loading process. Once loaded in the destination register, software can perform additional checks on the access rights information.

The access rights for a segment descriptor include fields located in the second doubleword (bytes 4–7) of the segment descriptor. The following fields are loaded by the LAR instruction:

- Bits 7:0 are returned as 0
- Bits 11:8 return the segment type.
- Bit 12 returns the S flag.
- Bits 14:13 return the DPL.
- Bit 15 returns the P flag.
- The following fields are returned only if the operand size is greater than 16 bits:
  - Bits 19:16 are undefined.
  - Bit 20 returns the software-available bit in the descriptor.
  - Bit 21 returns the L flag.
  - Bit 22 returns the D/B flag.
  - Bit 23 returns the G flag.
  - Bits 31:24 are returned as 0.

This instruction performs the following checks before it loads the access rights in the destination register:

- Checks that the segment selector is not NULL.
- Checks that the segment selector points to a descriptor that is within the limits of the GDT or LDT being accessed
- Checks that the descriptor type is valid for this instruction. All code and data segment descriptors are valid for (can be accessed with) the LAR instruction. The valid system segment and gate descriptor types are given in Table 3-52.
- If the segment is not a conforming code segment, it checks that the specified segment descriptor is visible at the CPL (that is, if the CPL and the RPL of the segment selector are less than or equal to the DPL of the segment selector).

If the segment descriptor cannot be accessed or is an invalid type for the instruction, the ZF flag is cleared and no access rights are loaded in the destination operand.

The LAR instruction can only be executed in protected mode and IA-32e mode.

Table 3-52. Segment and Gate Types

Type	Protected Mode		IA-32e Mode	
	Name	Valid	Name	Valid
0	Reserved	No	Reserved	No
1	Available 16-bit TSS	Yes	Reserved	No
2	LDT	Yes	LDT	Yes
3	Busy 16-bit TSS	Yes	Reserved	No
4	16-bit call gate	Yes	Reserved	No
5	16-bit/32-bit task gate	Yes	Reserved	No
6	16-bit interrupt gate	No	Reserved	No
7	16-bit trap gate	No	Reserved	No
8	Reserved	No	Reserved	No
9	Available 32-bit TSS	Yes	Available 64-bit TSS	Yes
A	Reserved	No	Reserved	No
B	Busy 32-bit TSS	Yes	Busy 64-bit TSS	Yes
C	32-bit call gate	Yes	64-bit call gate	Yes
D	Reserved	No	Reserved	No
E	32-bit interrupt gate	No	64-bit interrupt gate	No
F	32-bit trap gate	No	64-bit trap gate	No

Operation

```
IF Offset(SRC) > descriptor table limit
    THEN
        ZF ← 0;
    ELSE
        SegmentDescriptor ← descriptor referenced by SRC;
        IF SegmentDescriptor(Type) ≠ conforming code segment
        and (CPL > DPL) or (RPL > DPL)
        or SegmentDescriptor(Type) is not valid for instruction
            THEN
                ZF ← 0;
            ELSE
                DEST ← access rights from SegmentDescriptor as given in Description section;
                ZF ← 1;
        FI;
FI;
```

Flags Affected

The ZF flag is set to 1 if the access rights are loaded successfully; otherwise, it is cleared to 0.

Protected Mode Exceptions

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

Real-Address Mode Exceptions

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

Virtual-8086 Mode Exceptions

#UD	The LAR instruction cannot be executed in virtual-8086 mode.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#SS(0)	If the memory operand effective address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory operand effective address is in a non-canonical form.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and the memory operand effective address is unaligned while the current privilege level is 3.
#UD	If the LOCK prefix is used.

LDDQU—Load Unaligned Integer 128 Bits

Opcode/ Instruction	Op/ En	64/32-bit Mode	CPUID Feature Flag	Description
F2 0F F0 /r LDDQU *xmm1*, *mem*	RM	V/V	SSE3	Load unaligned data from *mem* and return double quadword in *xmm1*.
VEX.128.F2.0F.WIG F0 /r VLDDQU xmm1, m128	RM	V/V	AVX	Load unaligned packed integer values from mem to xmm1.
VEX.256.F2.0F.WIG F0 /r VLDDQU ymm1, m256	RM	V/V	AVX	Load unaligned packed integer values from mem to ymm1.

Instruction Operand Encoding

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

Description

The instruction is *functionally similar* to (V)MOVDQU ymm/xmm, m256/m128 for loading from memory. That is: 32/16 bytes of data starting at an address specified by the source memory operand (second operand) are fetched from memory and placed in a destination register (first operand). The source operand need not be aligned on a 32/16-byte boundary. Up to 64/32 bytes may be loaded from memory; this is implementation dependent.

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

Implementation Notes

- If the source is aligned to a 32/16-byte boundary, based on the implementation, the 32/16 bytes may be loaded more than once. For that reason, the usage of (V)LDDQU should be avoided when using uncached or write-combining (WC) memory regions. For uncached or WC memory regions, keep using (V)MOVDQU.
- This instruction is a replacement for (V)MOVDQU (load) in situations where cache line splits significantly affect performance. It should not be used in situations where store-load forwarding is performance critical. If performance of store-load forwarding is critical to the application, use (V)MOVDQA store-load pairs when data is 256/128-bit aligned or (V)MOVDQU store-load pairs when data is 256/128-bit unaligned.
- If the memory address is not aligned on 32/16-byte boundary, some implementations may load up to 64/32 bytes and return 32/16 bytes in the destination. Some processor implementations may issue multiple loads to access the appropriate 32/16 bytes. Developers of multi-threaded or multi-processor software should be aware that on these processors the loads will be performed in a non-atomic way.
- If alignment checking is enabled (CR0.AM = 1, RFLAGS.AC = 1, and CPL = 3), an alignment-check exception (#AC) may or may not be generated (depending on processor implementation) when the memory address is not aligned on an 8-byte boundary.

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

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

Operation

LDDQU (128-bit Legacy SSE version)

DEST[127:0] ← SRC[127:0]

DEST[MAXVL-1:128] (Unmodified)

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

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

Intel C/C++ Compiler Intrinsic Equivalent

LDDQU: __m128i _mm_lddqu_si128 (__m128i * p);

VLDDQU: __m256i _mm256_lddqu_si256 (__m256i * p);

Numeric Exceptions

None

Other Exceptions

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

LDMXCSR—Load MXCSR Register

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

Instruction Operand Encoding

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

Description

Loads the source operand into the MXCSR control/status register. The source operand is a 32-bit memory location. See "MXCSR Control and Status Register" in Chapter 10, of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*, for a description of the MXCSR register and its contents.

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

The default MXCSR value at reset is 1F80H.

If a (V)LDMXCSR instruction clears a SIMD floating-point exception mask bit and sets the corresponding exception flag bit, a SIMD floating-point exception will not be immediately generated. The exception will be generated only upon the execution of the next instruction that meets both conditions below:

- the instruction must operate on an XMM or YMM register operand,
- the instruction causes that particular SIMD floating-point exception to be reported.

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

If VLDMXCSR is encoded with VEX.L= 1, an attempt to execute the instruction encoded with VEX.L= 1 will cause an #UD exception.

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

Operation

MXCSR ← m32;

C/C++ Compiler Intrinsic Equivalent

_mm_setcsr(unsigned int i)

Numeric Exceptions

None

Other Exceptions

See Exceptions Type 5; additionally

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

LDS/LES/LFS/LGS/LSS—Load Far Pointer

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
C5 /r	LDS *r16,m16:16*	RM	Invalid	Valid	Load DS:*r16* with far pointer from memory.
C5 /r	LDS *r32,m16:32*	RM	Invalid	Valid	Load DS:*r32* with far pointer from memory.
0F B2 /r	LSS *r16,m16:16*	RM	Valid	Valid	Load SS:*r16* with far pointer from memory.
0F B2 /r	LSS *r32,m16:32*	RM	Valid	Valid	Load SS:*r32* with far pointer from memory.
REX + 0F B2 /r	LSS *r64,m16:64*	RM	Valid	N.E.	Load SS:*r64* with far pointer from memory.
C4 /r	LES *r16,m16:16*	RM	Invalid	Valid	Load ES:*r16* with far pointer from memory.
C4 /r	LES *r32,m16:32*	RM	Invalid	Valid	Load ES:*r32* with far pointer from memory.
0F B4 /r	LFS *r16,m16:16*	RM	Valid	Valid	Load FS:*r16* with far pointer from memory.
0F B4 /r	LFS *r32,m16:32*	RM	Valid	Valid	Load FS:*r32* with far pointer from memory.
REX + 0F B4 /r	LFS *r64,m16:64*	RM	Valid	N.E.	Load FS:*r64* with far pointer from memory.
0F B5 /r	LGS *r16,m16:16*	RM	Valid	Valid	Load GS:*r16* with far pointer from memory.
0F B5 /r	LGS *r32,m16:32*	RM	Valid	Valid	Load GS:*r32* with far pointer from memory.
REX + 0F B5 /r	LGS *r64,m16:64*	RM	Valid	N.E.	Load GS:*r64* with far pointer from memory.

Instruction Operand Encoding

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

Description

Loads a far pointer (segment selector and offset) from the second operand (source operand) into a segment register and the first operand (destination operand). The source operand specifies a 48-bit or a 32-bit pointer in memory depending on the current setting of the operand-size attribute (32 bits or 16 bits, respectively). The instruction opcode and the destination operand specify a segment register/general-purpose register pair. The 16-bit segment selector from the source operand is loaded into the segment register specified with the opcode (DS, SS, ES, FS, or GS). The 32-bit or 16-bit offset is loaded into the register specified with the destination operand.

If one of these instructions is executed in protected mode, additional information from the segment descriptor pointed to by the segment selector in the source operand is loaded in the hidden part of the selected segment register.

Also in protected mode, a NULL selector (values 0000 through 0003) can be loaded into DS, ES, FS, or GS registers without causing a protection exception. (Any subsequent reference to a segment whose corresponding segment register is loaded with a NULL selector, causes a general-protection exception (#GP) and no memory reference to the segment occurs.)

In 64-bit mode, the instruction's default operation size is 32 bits. Using a REX prefix in the form of REX.W promotes operation to specify a source operand referencing an 80-bit pointer (16-bit selector, 64-bit offset) in memory. Using a REX prefix in the form of REX.R permits access to additional registers (R8-R15). See the summary chart at the beginning of this section for encoding data and limits.

Operation

```
64-BIT_MODE
    IF SS is loaded
        THEN
            IF SegmentSelector = NULL and ( (RPL = 3) or
                    (RPL ≠ 3 and RPL ≠ CPL) )
                THEN #GP(0);
            ELSE IF descriptor is in non-canonical space
```

```
            THEN #GP(0); FI;
        ELSE IF Segment selector index is not within descriptor table limits
                or segment selector RPL ≠ CPL
                or access rights indicate nonwritable data segment
                or DPL ≠ CPL
            THEN #GP(selector); FI;
        ELSE IF Segment marked not present
            THEN #SS(selector); FI;
        FI;
        SS ← SegmentSelector(SRC);
        SS ← SegmentDescriptor([SRC]);
ELSE IF attempt to load DS, or ES
    THEN #UD;
ELSE IF FS, or GS is loaded with non-NULL segment selector
    THEN IF Segment selector index is not within descriptor table limits
        or access rights indicate segment neither data nor readable code segment
        or segment is data or nonconforming-code segment
        and ( RPL > DPL or CPL > DPL)
            THEN #GP(selector); FI;
        ELSE IF Segment marked not present
            THEN #NP(selector); FI;
        FI;
        SegmentRegister ← SegmentSelector(SRC) ;
        SegmentRegister ← SegmentDescriptor([SRC]);
    FI;
ELSE IF FS, or GS is loaded with a NULL selector:
    THEN
        SegmentRegister ← NULLSelector;
        SegmentRegister(DescriptorValidBit) ← 0; FI; (* Hidden flag;
            not accessible by software *)
FI;
DEST ← Offset(SRC);

PREOTECTED MODE OR COMPATIBILITY MODE;
IF SS is loaded
    THEN
        IF SegementSelector = NULL
            THEN #GP(0);
        ELSE IF Segment selector index is not within descriptor table limits
                or segment selector RPL ≠ CPL
                or access rights indicate nonwritable data segment
                or DPL ≠ CPL
            THEN #GP(selector); FI;
        ELSE IF Segment marked not present
            THEN #SS(selector); FI;
        FI;
        SS ← SegmentSelector(SRC);
        SS ← SegmentDescriptor([SRC]);
ELSE IF DS, ES, FS, or GS is loaded with non-NULL segment selector
    THEN IF Segment selector index is not within descriptor table limits
        or access rights indicate segment neither data nor readable code segment
        or segment is data or nonconforming-code segment
        and (RPL > DPL or CPL > DPL)
            THEN #GP(selector); FI;
```

```
            ELSE IF Segment marked not present
                THEN #NP(selector); FI;
            FI;
            SegmentRegister ← SegmentSelector(SRC) AND RPL;
            SegmentRegister ← SegmentDescriptor([SRC]);
        FI;
    ELSE IF DS, ES, FS, or GS is loaded with a NULL selector:
        THEN
            SegmentRegister ← NULLSelector;
            SegmentRegister(DescriptorValidBit) ← 0; FI; (* Hidden flag;
                not accessible by software *)
    FI;
    DEST ← Offset(SRC);

Real-Address or Virtual-8086 Mode
    SegmentRegister ← SegmentSelector(SRC); FI;
    DEST ← Offset(SRC);
```

Flags Affected

None

Protected Mode Exceptions

#UD	If source operand is not a memory location.
	If the LOCK prefix is used.
#GP(0)	If a NULL selector is loaded into the SS register.
	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
	If the DS, ES, FS, or GS register is used to access memory and it contains a NULL segment selector.
#GP(selector)	If the SS register is being loaded and any of the following is true: the segment selector index is not within the descriptor table limits, the segment selector RPL is not equal to CPL, the segment is a non-writable data segment, or DPL is not equal to CPL.
	If the DS, ES, FS, or GS register is being loaded with a non-NULL segment selector and any of the following is true: the segment selector index is not within descriptor table limits, the segment is neither a data nor a readable code segment, or the segment is a data or noncon-forming-code segment and both RPL and CPL are greater than DPL.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#SS(selector)	If the SS register is being loaded and the segment is marked not present.
#NP(selector)	If DS, ES, FS, or GS register is being loaded with a non-NULL segment selector and the segment is marked not present.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

Real-Address Mode Exceptions

#GP	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS	If a memory operand effective address is outside the SS segment limit.
#UD	If source operand is not a memory location.
	If the LOCK prefix is used.

Virtual-8086 Mode Exceptions

#UD	If source operand is not a memory location.
	If the LOCK prefix is used.
#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#GP(0)	If the memory address is in a non-canonical form.
	If a NULL selector is attempted to be loaded into the SS register in compatibility mode.
	If a NULL selector is attempted to be loaded into the SS register in CPL3 and 64-bit mode.
	If a NULL selector is attempted to be loaded into the SS register in non-CPL3 and 64-bit mode where its RPL is not equal to CPL.
#GP(Selector)	If the FS, or GS register is being loaded with a non-NULL segment selector and any of the following is true: the segment selector index is not within descriptor table limits, the memory address of the descriptor is non-canonical, the segment is neither a data nor a readable code segment, or the segment is a data or nonconforming-code segment and both RPL and CPL are greater than DPL.
	If the SS register is being loaded and any of the following is true: the segment selector index is not within the descriptor table limits, the memory address of the descriptor is non-canonical, the segment selector RPL is not equal to CPL, the segment is a nonwritable data segment, or DPL is not equal to CPL.
#SS(0)	If a memory operand effective address is non-canonical
#SS(Selector)	If the SS register is being loaded and the segment is marked not present.
#NP(selector)	If FS, or GS register is being loaded with a non-NULL segment selector and the segment is marked not present.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If source operand is not a memory location.
	If the LOCK prefix is used.

LEA—Load Effective Address

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
8D /r	LEA *r16,m*	RM	Valid	Valid	Store effective address for *m* in register *r16*.
8D /r	LEA *r32,m*	RM	Valid	Valid	Store effective address for *m* in register *r32*.
REX.W + 8D /r	LEA *r64,m*	RM	Valid	N.E.	Store effective address for *m* in register *r64*.

Instruction Operand Encoding

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

Description

Computes the effective address of the second operand (the source operand) and stores it in the first operand (destination operand). The source operand is a memory address (offset part) specified with one of the processors addressing modes; the destination operand is a general-purpose register. The address-size and operand-size attributes affect the action performed by this instruction, as shown in the following table. The operand-size attribute of the instruction is determined by the chosen register; the address-size attribute is determined by the attribute of the code segment.

Table 3-53. Non-64-bit Mode LEA Operation with Address and Operand Size Attributes

Operand Size	Address Size	Action Performed
16	16	16-bit effective address is calculated and stored in requested 16-bit register destination.
16	32	32-bit effective address is calculated. The lower 16 bits of the address are stored in the requested 16-bit register destination.
32	16	16-bit effective address is calculated. The 16-bit address is zero-extended and stored in the requested 32-bit register destination.
32	32	32-bit effective address is calculated and stored in the requested 32-bit register destination.

Different assemblers may use different algorithms based on the size attribute and symbolic reference of the source operand.

In 64-bit mode, the instruction's destination operand is governed by operand size attribute, the default operand size is 32 bits. Address calculation is governed by address size attribute, the default address size is 64-bits. In 64-bit mode, address size of 16 bits is not encodable. See Table 3-54.

Table 3-54. 64-bit Mode LEA Operation with Address and Operand Size Attributes

Operand Size	Address Size	Action Performed
16	32	32-bit effective address is calculated (using 67H prefix). The lower 16 bits of the address are stored in the requested 16-bit register destination (using 66H prefix).
16	64	64-bit effective address is calculated (default address size). The lower 16 bits of the address are stored in the requested 16-bit register destination (using 66H prefix).
32	32	32-bit effective address is calculated (using 67H prefix) and stored in the requested 32-bit register destination.
32	64	64-bit effective address is calculated (default address size) and the lower 32 bits of the address are stored in the requested 32-bit register destination.
64	32	32-bit effective address is calculated (using 67H prefix), zero-extended to 64-bits, and stored in the requested 64-bit register destination (using REX.W).
64	64	64-bit effective address is calculated (default address size) and all 64-bits of the address are stored in the requested 64-bit register destination (using REX.W).

Operation

```
IF OperandSize = 16 and AddressSize = 16
    THEN
        DEST ← EffectiveAddress(SRC); (* 16-bit address *)
    ELSE IF OperandSize = 16 and AddressSize = 32
        THEN
            temp ← EffectiveAddress(SRC); (* 32-bit address *)
            DEST ← temp[0:15]; (* 16-bit address *)
        FI;
    ELSE IF OperandSize = 32 and AddressSize = 16
        THEN
            temp ← EffectiveAddress(SRC); (* 16-bit address *)
            DEST ← ZeroExtend(temp); (* 32-bit address *)
        FI;
    ELSE IF OperandSize = 32 and AddressSize = 32
        THEN
            DEST ← EffectiveAddress(SRC); (* 32-bit address *)
        FI;
    ELSE IF OperandSize = 16 and AddressSize = 64
        THEN
            temp ← EffectiveAddress(SRC); (* 64-bit address *)
            DEST ← temp[0:15]; (* 16-bit address *)
        FI;
    ELSE IF OperandSize = 32 and AddressSize = 64
        THEN
            temp ← EffectiveAddress(SRC); (* 64-bit address *)
            DEST ← temp[0:31]; (* 16-bit address *)
        FI;
    ELSE IF OperandSize = 64 and AddressSize = 64
        THEN
            DEST ← EffectiveAddress(SRC); (* 64-bit address *)
        FI;
FI;
```

Flags Affected

None

Protected Mode Exceptions

#UD	If source operand is not a memory location. If the LOCK prefix is used.

Real-Address Mode Exceptions

Same exceptions as in protected mode.

Virtual-8086 Mode Exceptions

Same exceptions as in protected mode.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

Same exceptions as in protected mode.

LEAVE—High Level Procedure Exit

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
C9	LEAVE	ZO	Valid	Valid	Set SP to BP, then pop BP.
C9	LEAVE	ZO	N.E.	Valid	Set ESP to EBP, then pop EBP.
C9	LEAVE	ZO	Valid	N.E.	Set RSP to RBP, then pop RBP.

Instruction Operand Encoding

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

Description

Releases the stack frame set up by an earlier ENTER instruction. The LEAVE instruction copies the frame pointer (in the EBP register) into the stack pointer register (ESP), which releases the stack space allocated to the stack frame. The old frame pointer (the frame pointer for the calling procedure that was saved by the ENTER instruction) is then popped from the stack into the EBP register, restoring the calling procedure's stack frame.

A RET instruction is commonly executed following a LEAVE instruction to return program control to the calling procedure.

See "Procedure Calls for Block-Structured Languages" in Chapter 7 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*, for detailed information on the use of the ENTER and LEAVE instructions.

In 64-bit mode, the instruction's default operation size is 64 bits; 32-bit operation cannot be encoded. See the summary chart at the beginning of this section for encoding data and limits.

Operation

```
IF StackAddressSize = 32
    THEN
        ESP ← EBP;
    ELSE IF StackAddressSize = 64
        THEN RSP ← RBP; FI;
    ELSE IF StackAddressSize = 16
        THEN SP ← BP; FI;
FI;

IF OperandSize = 32
    THEN EBP ← Pop();
    ELSE IF OperandSize = 64
        THEN RBP ← Pop(); FI;
    ELSE IF OperandSize = 16
        THEN BP ← Pop(); FI;
FI;
```

Flags Affected

None

Protected Mode Exceptions

#SS(0)	If the EBP register points to a location that is not within the limits of the current stack segment.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If the LOCK prefix is used.

Real-Address Mode Exceptions

#GP	If the EBP register points to a location outside of the effective address space from 0 to FFFFH.
#UD	If the LOCK prefix is used.

Virtual-8086 Mode Exceptions

#GP(0)	If the EBP register points to a location outside of the effective address space from 0 to FFFFH.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.
#UD	If the LOCK prefix is used.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#SS(0)	If the stack address is in a non-canonical form.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If the LOCK prefix is used.

LFENCE—Load Fence

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
NP 0F AE E8	LFENCE	ZO	Valid	Valid	Serializes load operations.

Instruction Operand Encoding

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

Description

Performs a serializing operation on all load-from-memory instructions that were issued prior the LFENCE instruction. Specifically, LFENCE does not execute until all prior instructions have completed locally, and no later instruction begins execution until LFENCE completes. In particular, an instruction that loads from memory and that precedes an LFENCE receives data from memory prior to completion of the LFENCE. (An LFENCE that follows an instruction that stores to memory might complete **before** the data being stored have become globally visible.) Instructions following an LFENCE may be fetched from memory before the LFENCE, but they will not execute until the LFENCE completes.

Weakly ordered memory types can be used to achieve higher processor performance through such techniques as out-of-order issue and speculative reads. The degree to which a consumer of data recognizes or knows that the data is weakly ordered varies among applications and may be unknown to the producer of this data. The LFENCE instruction provides a performance-efficient way of ensuring load ordering between routines that produce weakly-ordered results and routines that consume that data.

Processors are free to fetch and cache data speculatively from regions of system memory that use the WB, WC, and WT memory types. This speculative fetching can occur at any time and is not tied to instruction execution. Thus, it is not ordered with respect to executions of the LFENCE instruction; data can be brought into the caches speculatively just before, during, or after the execution of an LFENCE instruction.

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

Specification of the instruction's opcode above indicates a ModR/M byte of E8. For this instruction, the processor ignores the r/m field of the ModR/M byte. Thus, LFENCE is encoded by any opcode of the form 0F AE Ex, where x is in the range 8-F.

Operation

```
Wait_On_Following_Instructions_Until(preceding_instructions_complete);
```

Intel C/C++ Compiler Intrinsic Equivalent

void _mm_lfence(void)

Exceptions (All Modes of Operation)

#UD	If CPUID.01H:EDX.SSE2[bit 26] = 0.
	If the LOCK prefix is used.

LGDT/LIDT—Load Global/Interrupt Descriptor Table Register

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
0F 01 /2	LGDT *m16&32*	M	N.E.	Valid	Load *m* into GDTR.
0F 01 /3	LIDT *m16&32*	M	N.E.	Valid	Load *m* into IDTR.
0F 01 /2	LGDT *m16&64*	M	Valid	N.E.	Load *m* into GDTR.
0F 01 /3	LIDT *m16&64*	M	Valid	N.E.	Load *m* into IDTR.

Instruction Operand Encoding

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

Description

Loads the values in the source operand into the global descriptor table register (GDTR) or the interrupt descriptor table register (IDTR). The source operand specifies a 6-byte memory location that contains the base address (a linear address) and the limit (size of table in bytes) of the global descriptor table (GDT) or the interrupt descriptor table (IDT). If operand-size attribute is 32 bits, a 16-bit limit (lower 2 bytes of the 6-byte data operand) and a 32-bit base address (upper 4 bytes of the data operand) are loaded into the register. If the operand-size attribute is 16 bits, a 16-bit limit (lower 2 bytes) and a 24-bit base address (third, fourth, and fifth byte) are loaded. Here, the high-order byte of the operand is not used and the high-order byte of the base address in the GDTR or IDTR is filled with zeros.

The LGDT and LIDT instructions are used only in operating-system software; they are not used in application programs. They are the only instructions that directly load a linear address (that is, not a segment-relative address) and a limit in protected mode. They are commonly executed in real-address mode to allow processor initialization prior to switching to protected mode.

In 64-bit mode, the instruction's operand size is fixed at 8+2 bytes (an 8-byte base and a 2-byte limit). See the summary chart at the beginning of this section for encoding data and limits.

See "SGDT—Store Global Descriptor Table Register" in Chapter 4, *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2B*, for information on storing the contents of the GDTR and IDTR.

Operation

```
IF Instruction is LIDT
    THEN
        IF OperandSize = 16
            THEN
                IDTR(Limit) ← SRC[0:15];
                IDTR(Base) ← SRC[16:47] AND 00FFFFFFH;
            ELSE IF 32-bit Operand Size
                THEN
                    IDTR(Limit) ← SRC[0:15];
                    IDTR(Base) ← SRC[16:47];
                FI;
            ELSE IF 64-bit Operand Size (* In 64-Bit Mode *)
                THEN
                    IDTR(Limit) ← SRC[0:15];
                    IDTR(Base) ← SRC[16:79];
                FI;
        FI;
    ELSE (* Instruction is LGDT *)
        IF OperandSize = 16
            THEN
                GDTR(Limit) ← SRC[0:15];
                GDTR(Base) ← SRC[16:47] AND 00FFFFFFH;
            ELSE IF 32-bit Operand Size
                THEN
                    GDTR(Limit) ← SRC[0:15];
                    GDTR(Base) ← SRC[16:47];
                FI;
            ELSE IF 64-bit Operand Size (* In 64-Bit Mode *)
                THEN
                    GDTR(Limit) ← SRC[0:15];
                    GDTR(Base) ← SRC[16:79];
                FI;
        FI;
FI;
```

Flags Affected

None

Protected Mode Exceptions

#UD	If the LOCK prefix is used.
#GP(0)	If the current privilege level is not 0.
	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
	If the DS, ES, FS, or GS register is used to access memory and it contains a NULL segment selector.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.

Real-Address Mode Exceptions

#UD	If the LOCK prefix is used.
#GP	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS	If a memory operand effective address is outside the SS segment limit.

Virtual-8086 Mode Exceptions

#UD	If the LOCK prefix is used.
#GP	If the current privilege level is not 0.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the current privilege level is not 0.
	If the memory address is in a non-canonical form.
#UD	If the LOCK prefix is used.
#PF(fault-code)	If a page fault occurs.

LLDT—Load Local Descriptor Table Register

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
0F 00 /2	LLDT *r/m*16	M	Valid	Valid	Load segment selector *r/m*16 into LDTR.

Instruction Operand Encoding

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

Description

Loads the source operand into the segment selector field of the local descriptor table register (LDTR). The source operand (a general-purpose register or a memory location) contains a segment selector that points to a local descriptor table (LDT). After the segment selector is loaded in the LDTR, the processor uses the segment selector to locate the segment descriptor for the LDT in the global descriptor table (GDT). It then loads the segment limit and base address for the LDT from the segment descriptor into the LDTR. The segment registers DS, ES, SS, FS, GS, and CS are not affected by this instruction, nor is the LDTR field in the task state segment (TSS) for the current task.

If bits 2-15 of the source operand are 0, LDTR is marked invalid and the LLDT instruction completes silently. However, all subsequent references to descriptors in the LDT (except by the LAR, VERR, VERW or LSL instructions) cause a general protection exception (#GP).

The operand-size attribute has no effect on this instruction.

The LLDT instruction is provided for use in operating-system software; it should not be used in application programs. This instruction can only be executed in protected mode or 64-bit mode.

In 64-bit mode, the operand size is fixed at 16 bits.

Operation

```
IF SRC(Offset) > descriptor table limit
    THEN #GP(segment selector); FI;

IF segment selector is valid

    Read segment descriptor;

    IF SegmentDescriptor(Type) ≠ LDT
        THEN #GP(segment selector); FI;
    IF segment descriptor is not present
        THEN #NP(segment selector); FI;

    LDTR(SegmentSelector) ← SRC;
    LDTR(SegmentDescriptor) ← GDTSegmentDescriptor;
ELSE LDTR ← INVALID
FI;
```

Flags Affected

None

Protected Mode Exceptions

#GP(0)	If the current privilege level is not 0.
	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
	If the DS, ES, FS, or GS register contains a NULL segment selector.
#GP(selector)	If the selector operand does not point into the Global Descriptor Table or if the entry in the GDT is not a Local Descriptor Table.
	Segment selector is beyond GDT limit.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#NP(selector)	If the LDT descriptor is not present.
#PF(fault-code)	If a page fault occurs.
#UD	If the LOCK prefix is used.

Real-Address Mode Exceptions

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

Virtual-8086 Mode Exceptions

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

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the current privilege level is not 0.
	If the memory address is in a non-canonical form.
#GP(selector)	If the selector operand does not point into the Global Descriptor Table or if the entry in the GDT is not a Local Descriptor Table.
	Segment selector is beyond GDT limit.
#NP(selector)	If the LDT descriptor is not present.
#PF(fault-code)	If a page fault occurs.
#UD	If the LOCK prefix is used.

LMSW—Load Machine Status Word

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
0F 01 /6	LMSW *r/m*16	M	Valid	Valid	Loads *r/m*16 in machine status word of CR0.

Instruction Operand Encoding

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

Description

Loads the source operand into the machine status word, bits 0 through 15 of register CR0. The source operand can be a 16-bit general-purpose register or a memory location. Only the low-order 4 bits of the source operand (which contains the PE, MP, EM, and TS flags) are loaded into CR0. The PG, CD, NW, AM, WP, NE, and ET flags of CR0 are not affected. The operand-size attribute has no effect on this instruction.

If the PE flag of the source operand (bit 0) is set to 1, the instruction causes the processor to switch to protected mode. While in protected mode, the LMSW instruction cannot be used to clear the PE flag and force a switch back to real-address mode.

The LMSW instruction is provided for use in operating-system software; it should not be used in application programs. In protected or virtual-8086 mode, it can only be executed at CPL 0.

This instruction is provided for compatibility with the Intel 286 processor; programs and procedures intended to run on IA-32 and Intel 64 processors beginning with Intel386 processors should use the MOV (control registers) instruction to load the whole CR0 register. The MOV CR0 instruction can be used to set and clear the PE flag in CR0, allowing a procedure or program to switch between protected and real-address modes.

This instruction is a serializing instruction.

This instruction's operation is the same in non-64-bit modes and 64-bit mode. Note that the operand size is fixed at 16 bits.

See "Changes to Instruction Behavior in VMX Non-Root Operation" in Chapter 25 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3C*, for more information about the behavior of this instruction in VMX non-root operation.

Operation

```
CR0[0:3] ← SRC[0:3];
```

Flags Affected

None

Protected Mode Exceptions

#GP(0)	If the current privilege level is not 0.
	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
	If the DS, ES, FS, or GS register is used to access memory and it contains a NULL segment selector.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#UD	If the LOCK prefix is used.

Real-Address Mode Exceptions

#GP	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#UD	If the LOCK prefix is used.

Virtual-8086 Mode Exceptions

#GP(0)	The LMSW instruction is not recognized in virtual-8086 mode.
#UD	If the LOCK prefix is used.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the current privilege level is not 0.
	If the memory address is in a non-canonical form.
#PF(fault-code)	If a page fault occurs.
#UD	If the LOCK prefix is used.

LOCK—Assert LOCK# Signal Prefix

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
F0	LOCK	ZO	Valid	Valid	Asserts LOCK# signal for duration of the accompanying instruction.

NOTES:

* See IA-32 Architecture Compatibility section below.

Instruction Operand Encoding

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

Description

Causes the processor's LOCK# signal to be asserted during execution of the accompanying instruction (turns the instruction into an atomic instruction). In a multiprocessor environment, the LOCK# signal ensures that the processor has exclusive use of any shared memory while the signal is asserted.

In most IA-32 and all Intel 64 processors, locking may occur without the LOCK# signal being asserted. See the "IA-32 Architecture Compatibility" section below for more details.

The LOCK prefix can be prepended only to the following instructions and only to those forms of the instructions where the destination operand is a memory operand: ADD, ADC, AND, BTC, BTR, BTS, CMPXCHG, CMPXCH8B, CMPXCHG16B, DEC, INC, NEG, NOT, OR, SBB, SUB, XOR, XADD, and XCHG. If the LOCK prefix is used with one of these instructions and the source operand is a memory operand, an undefined opcode exception (#UD) may be generated. An undefined opcode exception will also be generated if the LOCK prefix is used with any instruction not in the above list. The XCHG instruction always asserts the LOCK# signal regardless of the presence or absence of the LOCK prefix.

The LOCK prefix is typically used with the BTS instruction to perform a read-modify-write operation on a memory location in shared memory environment.

The integrity of the LOCK prefix is not affected by the alignment of the memory field. Memory locking is observed for arbitrarily misaligned fields.

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

IA-32 Architecture Compatibility

Beginning with the P6 family processors, when the LOCK prefix is prefixed to an instruction and the memory area being accessed is cached internally in the processor, the LOCK# signal is generally not asserted. Instead, only the processor's cache is locked. Here, the processor's cache coherency mechanism ensures that the operation is carried out atomically with regards to memory. See "Effects of a Locked Operation on Internal Processor Caches" in Chapter 8 of *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*, the for more information on locking of caches.

Operation

```
AssertLOCK#(DurationOfAccompaningInstruction);
```

Flags Affected

None

Protected Mode Exceptions

#UD	If the LOCK prefix is used with an instruction not listed: ADD, ADC, AND, BTC, BTR, BTS, CMPXCHG, CMPXCH8B, CMPXCHG16B, DEC, INC, NEG, NOT, OR, SBB, SUB, XOR, XADD, XCHG. Other exceptions can be generated by the instruction when the LOCK prefix is applied.

Real-Address Mode Exceptions

Same exceptions as in protected mode.

Virtual-8086 Mode Exceptions

Same exceptions as in protected mode.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

Same exceptions as in protected mode.

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

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
AC	LODS *m8*	ZO	Valid	Valid	For legacy mode, Load byte at address DS:(E)SI into AL. For 64-bit mode load byte at address (R)SI into AL.
AD	LODS *m16*	ZO	Valid	Valid	For legacy mode, Load word at address DS:(E)SI into AX. For 64-bit mode load word at address (R)SI into AX.
AD	LODS m32	ZO	Valid	Valid	For legacy mode, Load dword at address DS:(E)SI into EAX. For 64-bit mode load dword at address (R)SI into EAX.
REX.W + AD	LODS *m64*	ZO	Valid	N.E.	Load qword at address (R)SI into RAX.
AC	LODSB	ZO	Valid	Valid	For legacy mode, Load byte at address DS:(E)SI into AL. For 64-bit mode load byte at address (R)SI into AL.
AD	LODSW	ZO	Valid	Valid	For legacy mode, Load word at address DS:(E)SI into AX. For 64-bit mode load word at address (R)SI into AX.
AD	LODSD	ZO	Valid	Valid	For legacy mode, Load dword at address DS:(E)SI into EAX. For 64-bit mode load dword at address (R)SI into EAX.
REX.W + AD	LODSQ	ZO	Valid	N.E.	Load qword at address (R)SI into RAX.

Instruction Operand Encoding

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

Description

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

At the assembly-code level, two forms of this instruction are allowed: the "explicit-operands" form and the "no-operands" form. The explicit-operands form (specified with the LODS mnemonic) allows the source operand to be specified explicitly. Here, the source operand should be a symbol that indicates the size and location of the source value. The destination operand is then automatically selected to match the size of the source operand (the AL register for byte operands, AX for word operands, and EAX for doubleword operands). This explicit-operands form is provided to allow documentation; however, note that the documentation provided by this form can be misleading. That is, the source operand symbol must specify the correct **type** (size) of the operand (byte, word, or doubleword), but it does not have to specify the correct **location**. The location is always specified by the DS:(E)SI registers, which must be loaded correctly before the load string instruction is executed.

The no-operands form provides "short forms" of the byte, word, and doubleword versions of the LODS instructions. Here also DS:(E)SI is assumed to be the source operand and the AL, AX, or EAX register is assumed to be the destination operand. The size of the source and destination operands is selected with the mnemonic: LODSB (byte loaded into register AL), LODSW (word loaded into AX), or LODSD (doubleword loaded into EAX).

After the byte, word, or doubleword is transferred from the memory location into the AL, AX, or EAX register, the (E)SI register is incremented or decremented automatically according to the setting of the DF flag in the EFLAGS register. (If the DF flag is 0, the (E)SI register is incremented; if the DF flag is 1, the ESI register is decremented.) The (E)SI register is incremented or decremented by 1 for byte operations, by 2 for word operations, or by 4 for doubleword operations.

In 64-bit mode, use of the REX.W prefix promotes operation to 64 bits. LODS/LODSQ load the quadword at address (R)SI into RAX. The (R)SI register is then incremented or decremented automatically according to the setting of the DF flag in the EFLAGS register.

The LODS, LODSB, LODSW, and LODSD instructions can be preceded by the REP prefix for block loads of ECX bytes, words, or doublewords. More often, however, these instructions are used within a LOOP construct because further processing of the data moved into the register is usually necessary before the next transfer can be made. See "REP/REPE/REPZ /REPNE/REPNZ—Repeat String Operation Prefix" in Chapter 4 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2B*, for a description of the REP prefix.

Operation

```
IF AL ← SRC; (* Byte load *)
    THEN AL ← SRC; (* Byte load *)
        IF DF = 0
            THEN (E)SI ← (E)SI + 1;
            ELSE (E)SI ← (E)SI – 1;
        FI;
ELSE IF AX ← SRC; (* Word load *)
    THEN IF DF = 0
            THEN (E)SI ← (E)SI + 2;
            ELSE (E)SI ← (E)SI – 2;
        IF;
    FI;
ELSE IF EAX ← SRC; (* Doubleword load *)
    THEN IF DF = 0
            THEN (E)SI ← (E)SI + 4;
            ELSE (E)SI ← (E)SI – 4;
        FI;
    FI;
ELSE IF RAX ← SRC; (* Quadword load *)
    THEN IF DF = 0
            THEN (R)SI ← (R)SI + 8;
            ELSE (R)SI ← (R)SI – 8;
        FI;
    FI;
FI;
```

Flags Affected

None

Protected Mode Exceptions

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

Real-Address Mode Exceptions

#GP	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS	If a memory operand effective address is outside the SS segment limit.
#UD	If the LOCK prefix is used.

Virtual-8086 Mode Exceptions

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

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If the LOCK prefix is used.

LOOP/LOOP*cc*—Loop According to ECX Counter

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
E2 *cb*	LOOP *rel8*	D	Valid	Valid	Decrement count; jump short if count ≠ 0.
E1 *cb*	LOOPE *rel8*	D	Valid	Valid	Decrement count; jump short if count ≠ 0 and ZF = 1.
E0 *cb*	LOOPNE *rel8*	D	Valid	Valid	Decrement count; jump short if count ≠ 0 and ZF = 0.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
D	Offset	NA	NA	NA

Description

Performs a loop operation using the RCX, ECX or CX register as a counter (depending on whether address size is 64 bits, 32 bits, or 16 bits). Note that the LOOP instruction ignores REX.W; but 64-bit address size can be over-ridden using a 67H prefix.

Each time the LOOP instruction is executed, the count register is decremented, then checked for 0. If the count is 0, the loop is terminated and program execution continues with the instruction following the LOOP instruction. If the count is not zero, a near jump is performed to the destination (target) operand, which is presumably the instruction at the beginning of the loop.

The target instruction is specified with a relative offset (a signed offset relative to the current value of the instruction pointer in the IP/EIP/RIP register). This offset is generally specified as a label in assembly code, but at the machine code level, it is encoded as a signed, 8-bit immediate value, which is added to the instruction pointer. Offsets of −128 to +127 are allowed with this instruction.

Some forms of the loop instruction (LOOP*cc*) also accept the ZF flag as a condition for terminating the loop before the count reaches zero. With these forms of the instruction, a condition code (*cc*) is associated with each instruction to indicate the condition being tested for. Here, the LOOP*cc* instruction itself does not affect the state of the ZF flag; the ZF flag is changed by other instructions in the loop.

Operation

```
IF (AddressSize = 32)
    THEN Count is ECX;
ELSE IF (AddressSize = 64)
    Count is RCX;
ELSE Count is CX;
FI;

Count ← Count – 1;

IF Instruction is not LOOP
    THEN
        IF (Instruction ← LOOPE) or (Instruction ← LOOPZ)
            THEN IF (ZF = 1) and (Count ≠ 0)
                    THEN BranchCond ← 1;
                    ELSE BranchCond ← 0;
                FI;
            ELSE (Instruction = LOOPNE) or (Instruction = LOOPNZ)
                IF (ZF = 0 ) and (Count ≠ 0)
                    THEN BranchCond ← 1;
                    ELSE BranchCond ← 0;
```

```
                FI;
        FI;
    ELSE (* Instruction = LOOP *)
        IF (Count ≠ 0)
            THEN BranchCond ← 1;
            ELSE BranchCond ← 0;
        FI;
FI;

IF BranchCond = 1
    THEN
        IF OperandSize = 32
            THEN EIP ← EIP + SignExtend(DEST);
            ELSE IF OperandSize = 64
                THEN RIP ← RIP + SignExtend(DEST);
                FI;
            ELSE IF OperandSize = 16
                THEN EIP ← EIP AND 0000FFFFH;
                FI;
        FI;
        IF OperandSize = (32 or 64)
            THEN IF (R/E)IP < CS.Base or (R/E)IP > CS.Limit
                #GP; FI;
                FI;
        FI;
    ELSE
        Terminate loop and continue program execution at (R/E)IP;
FI;
```

Flags Affected

None

Protected Mode Exceptions

#GP(0)	If the offset being jumped to is beyond the limits of the CS segment.
#UD	If the LOCK prefix is used.

Real-Address Mode Exceptions

#GP	If the offset being jumped to is beyond the limits of the CS segment or is outside of the effective address space from 0 to FFFFH. This condition can occur if a 32-bit address size override prefix is used.
#UD	If the LOCK prefix is used.

Virtual-8086 Mode Exceptions

Same exceptions as in real address mode.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#GP(0)	If the offset being jumped to is in a non-canonical form.
#UD	If the LOCK prefix is used.

LSL—Load Segment Limit

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
0F 03 /r	LSL *r16, r16/m16*	RM	Valid	Valid	Load: *r16* ← segment limit, selector *r16/m16.*
0F 03 /r	LSL *r32, r32/m16**	RM	Valid	Valid	Load: *r32* ← segment limit, selector *r32/m16.*
REX.W + 0F 03 /r	LSL *r64, r32/m16**	RM	Valid	Valid	Load: *r64* ← segment limit, selector *r32/m16*

NOTES:

* For all loads (regardless of destination sizing), only bits 16-0 are used. Other bits are ignored.

Instruction Operand Encoding

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

Description

Loads the unscrambled segment limit from the segment descriptor specified with the second operand (source operand) into the first operand (destination operand) and sets the ZF flag in the EFLAGS register. The source operand (which can be a register or a memory location) contains the segment selector for the segment descriptor being accessed. The destination operand is a general-purpose register.

The processor performs access checks as part of the loading process. Once loaded in the destination register, software can compare the segment limit with the offset of a pointer.

The segment limit is a 20-bit value contained in bytes 0 and 1 and in the first 4 bits of byte 6 of the segment descriptor. If the descriptor has a byte granular segment limit (the granularity flag is set to 0), the destination operand is loaded with a byte granular value (byte limit). If the descriptor has a page granular segment limit (the granularity flag is set to 1), the LSL instruction will translate the page granular limit (page limit) into a byte limit before loading it into the destination operand. The translation is performed by shifting the 20-bit "raw" limit left 12 bits and filling the low-order 12 bits with 1s.

When the operand size is 32 bits, the 32-bit byte limit is stored in the destination operand. When the operand size is 16 bits, a valid 32-bit limit is computed; however, the upper 16 bits are truncated and only the low-order 16 bits are loaded into the destination operand.

This instruction performs the following checks before it loads the segment limit into the destination register:

- Checks that the segment selector is not NULL.
- Checks that the segment selector points to a descriptor that is within the limits of the GDT or LDT being accessed
- Checks that the descriptor type is valid for this instruction. All code and data segment descriptors are valid for (can be accessed with) the LSL instruction. The valid special segment and gate descriptor types are given in the following table.
- If the segment is not a conforming code segment, the instruction checks that the specified segment descriptor is visible at the CPL (that is, if the CPL and the RPL of the segment selector are less than or equal to the DPL of the segment selector).

If the segment descriptor cannot be accessed or is an invalid type for the instruction, the ZF flag is cleared and no value is loaded in the destination operand.

Table 3-55. Segment and Gate Descriptor Types

Type	Protected Mode		IA-32e Mode	
	Name	Valid	Name	Valid
0	Reserved	No	Upper 8 byte of a 16-Byte descriptor	Yes
1	Available 16-bit TSS	Yes	Reserved	No
2	LDT	Yes	LDT	Yes
3	Busy 16-bit TSS	Yes	Reserved	No
4	16-bit call gate	No	Reserved	No
5	16-bit/32-bit task gate	No	Reserved	No
6	16-bit interrupt gate	No	Reserved	No
7	16-bit trap gate	No	Reserved	No
8	Reserved	No	Reserved	No
9	Available 32-bit TSS	Yes	64-bit TSS	Yes
A	Reserved	No	Reserved	No
B	Busy 32-bit TSS	Yes	Busy 64-bit TSS	Yes
C	32-bit call gate	No	64-bit call gate	No
D	Reserved	No	Reserved	No
E	32-bit interrupt gate	No	64-bit interrupt gate	No
F	32-bit trap gate	No	64-bit trap gate	No

Operation

```
IF SRC(Offset) > descriptor table limit
    THEN ZF ← 0; FI;

Read segment descriptor;

IF SegmentDescriptor(Type) ≠ conforming code segment
and (CPL > DPL) OR (RPL > DPL)
or Segment type is not valid for instruction
        THEN
            ZF ← 0;
        ELSE
            temp ← SegmentLimit([SRC]);
            IF (G ← 1)
                THEN temp ← ShiftLeft(12, temp) OR 00000FFFH;
            ELSE IF OperandSize = 32
                THEN DEST ← temp; FI;
            ELSE IF OperandSize = 64 (* REX.W used *)
                THEN DEST (* Zero-extended *) ← temp; FI;
            ELSE (* OperandSize = 16 *)
                DEST ← temp AND FFFFH;
            FI;
FI;
```

Flags Affected

The ZF flag is set to 1 if the segment limit is loaded successfully; otherwise, it is set to 0.

Protected Mode Exceptions

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

Real-Address Mode Exceptions

#UD	The LSL instruction cannot be executed in real-address mode.

Virtual-8086 Mode Exceptions

#UD	The LSL instruction cannot be executed in virtual-8086 mode.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#SS(0)	If the memory operand effective address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory operand effective address is in a non-canonical form.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and the memory operand effective address is unaligned while the current privilege level is 3.
#UD	If the LOCK prefix is used.

LTR—Load Task Register

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
0F 00 /3	LTR *r/m*16	M	Valid	Valid	Load *r/m*16 into task register.

Instruction Operand Encoding

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

Description

Loads the source operand into the segment selector field of the task register. The source operand (a general-purpose register or a memory location) contains a segment selector that points to a task state segment (TSS). After the segment selector is loaded in the task register, the processor uses the segment selector to locate the segment descriptor for the TSS in the global descriptor table (GDT). It then loads the segment limit and base address for the TSS from the segment descriptor into the task register. The task pointed to by the task register is marked busy, but a switch to the task does not occur.

The LTR instruction is provided for use in operating-system software; it should not be used in application programs. It can only be executed in protected mode when the CPL is 0. It is commonly used in initialization code to establish the first task to be executed.

The operand-size attribute has no effect on this instruction.

In 64-bit mode, the operand size is still fixed at 16 bits. The instruction references a 16-byte descriptor to load the 64-bit base.

Operation

```
IF SRC is a NULL selector
    THEN #GP(0);

IF SRC(Offset) > descriptor table limit OR IF SRC(type) ≠ global
    THEN #GP(segment selector); FI;

Read segment descriptor;

IF segment descriptor is not for an available TSS
    THEN #GP(segment selector); FI;
IF segment descriptor is not present
    THEN #NP(segment selector); FI;

TSSsegmentDescriptor(busy) ← 1;
(* Locked read-modify-write operation on the entire descriptor when setting busy flag *)

TaskRegister(SegmentSelector) ← SRC;
TaskRegister(SegmentDescriptor) ← TSSSegmentDescriptor;
```

Flags Affected

None

Protected Mode Exceptions

#GP(0)	If the current privilege level is not 0.
	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
	If the source operand contains a NULL segment selector.
	If the DS, ES, FS, or GS register is used to access memory and it contains a NULL segment selector.
#GP(selector)	If the source selector points to a segment that is not a TSS or to one for a task that is already busy.
	If the selector points to LDT or is beyond the GDT limit.
#NP(selector)	If the TSS is marked not present.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#UD	If the LOCK prefix is used.

Real-Address Mode Exceptions

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

Virtual-8086 Mode Exceptions

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

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the current privilege level is not 0.
	If the memory address is in a non-canonical form.
	If the source operand contains a NULL segment selector.
#GP(selector)	If the source selector points to a segment that is not a TSS or to one for a task that is already busy.
	If the selector points to LDT or is beyond the GDT limit.
	If the descriptor type of the upper 8-byte of the 16-byte descriptor is non-zero.
#NP(selector)	If the TSS is marked not present.
#PF(fault-code)	If a page fault occurs.
#UD	If the LOCK prefix is used.

LZCNT— Count the Number of Leading Zero Bits

Opcode/Instruction	Op/ En	64/32 -bit Mode	CPUID Feature Flag	Description
F3 0F BD /r	RM	V/V	LZCNT	Count the number of leading zero bits in r/m16, return result in r16.
LZCNT r16, r/m16				
F3 0F BD /r	RM	V/V	LZCNT	Count the number of leading zero bits in r/m32, return result in r32.
LZCNT r32, r/m32				
F3 REX.W 0F BD /r	RM	V/N.E.	LZCNT	Count the number of leading zero bits in r/m64, return result in r64.
LZCNT r64, r/m64				

Instruction Operand Encoding

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

Description

Counts the number of leading most significant zero bits in a source operand (second operand) returning the result into a destination (first operand).

LZCNT differs from BSR. For example, LZCNT will produce the operand size when the input operand is zero. It should be noted that on processors that do not support LZCNT, the instruction byte encoding is executed as BSR.

In 64-bit mode 64-bit operand size requires REX.W=1.

Operation

```
temp ← OperandSize - 1
DEST ← 0
WHILE (temp >= 0) AND (Bit(SRC, temp) = 0)
DO
    temp ← temp - 1
    DEST ← DEST+ 1
OD

IF DEST = OperandSize
    CF ← 1
ELSE
    CF ← 0
FI

IF DEST = 0
    ZF ← 1
ELSE
    ZF ← 0
FI
```

Flags Affected

ZF flag is set to 1 in case of zero output (most significant bit of the source is set), and to 0 otherwise, CF flag is set to 1 if input was zero and cleared otherwise. OF, SF, PF and AF flags are undefined.

Intel C/C++ Compiler Intrinsic Equivalent

LZCNT: unsigned __int32 _lzcnt_u32(unsigned __int32 src);

LZCNT: unsigned __int64 _lzcnt_u64(unsigned __int64 src);

Protected Mode Exceptions

#GP(0)	For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments. If the DS, ES, FS, or GS register is used to access memory and it contains a null segment selector.
#SS(0)	For an illegal address in the SS segment.
#PF (fault-code)	For a page fault.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

Real-Address Mode Exceptions

#GP(0)	If any part of the operand lies outside of the effective address space from 0 to 0FFFFH.
#SS(0)	For an illegal address in the SS segment.

Virtual 8086 Mode Exceptions

#GP(0)	If any part of the operand lies outside of the effective address space from 0 to 0FFFFH.
#SS(0)	For an illegal address in the SS segment.
#PF (fault-code)	For a page fault.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

Compatibility Mode Exceptions

Same exceptions as in Protected Mode.

64-Bit Mode Exceptions

#GP(0)	If the memory address is in a non-canonical form.
#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#PF (fault-code)	For a page fault.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

8. Updates to Chapter 4, Volume 2B

Change bars show changes to Chapter 4 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2B:* Instruction Set Reference, M-U.

Change to this chapter: Updates to the following instructions: POPF/POPFD/POPFQ, PUSHF/PUSHFD, RDPID, RET, SGDT, SIDT, STI. Updated operand encoding table for instructions with tuple types, breaking out tuple types into a separate column. Corrected naming typos in operation section of various instructions.

CHAPTER 4
INSTRUCTION SET REFERENCE, M-U

4.1 IMM8 CONTROL BYTE OPERATION FOR PCMPESTRI / PCMPESTRM / PCMPISTRI / PCMPISTRM

The notations introduced in this section are referenced in the reference pages of PCMPESTRI, PCMPESTRM, PCMPISTRI, PCMPISTRM. The operation of the immediate control byte is common to these four string text processing instructions of SSE4.2. This section describes the common operations.

4.1.1 General Description

The operation of PCMPESTRI, PCMPESTRM, PCMPISTRI, PCMPISTRM is defined by the combination of the respective opcode and the interpretation of an immediate control byte that is part of the instruction encoding.

The opcode controls the relationship of input bytes/words to each other (determines whether the inputs terminated strings or whether lengths are expressed explicitly) as well as the desired output (index or mask).

The Imm8 Control Byte for PCMPESTRM/PCMPESTRI/PCMPISTRM/PCMPISTRI encodes a significant amount of programmable control over the functionality of those instructions. Some functionality is unique to each instruction while some is common across some or all of the four instructions. This section describes functionality which is common across the four instructions.

The arithmetic flags (ZF, CF, SF, OF, AF, PF) are set as a result of these instructions. However, the meanings of the flags have been overloaded from their typical meanings in order to provide additional information regarding the relationships of the two inputs.

PCMPxSTRx instructions perform arithmetic comparisons between all possible pairs of bytes or words, one from each packed input source operand. The boolean results of those comparisons are then aggregated in order to produce meaningful results. The Imm8 Control Byte is used to affect the interpretation of individual input elements as well as control the arithmetic comparisons used and the specific aggregation scheme.

Specifically, the Imm8 Control Byte consists of bit fields that control the following attributes:

- **Source data format —** Byte/word data element granularity, signed or unsigned elements
- **Aggregation operation —** Encodes the mode of per-element comparison operation and the aggregation of per-element comparisons into an intermediate result
- **Polarity —** Specifies intermediate processing to be performed on the intermediate result
- **Output selection —** Specifies final operation to produce the output (depending on index or mask) from the intermediate result

4.1.2 Source Data Format

Table 4-1. Source Data Format

Imm8[1:0]	Meaning	Description
00b	Unsigned bytes	Both 128-bit sources are treated as packed, unsigned bytes.
01b	Unsigned words	Both 128-bit sources are treated as packed, unsigned words.
10b	Signed bytes	Both 128-bit sources are treated as packed, signed bytes.
11b	Signed words	Both 128-bit sources are treated as packed, signed words.

If the Imm8 Control Byte has bit[0] cleared, each source contains 16 packed bytes. If the bit is set each source contains 8 packed words. If the Imm8 Control Byte has bit[1] cleared, each input contains unsigned data. If the bit is set each source contains signed data.

4.1.3 Aggregation Operation

Table 4-2. Aggregation Operation

Imm8[3:2]	Mode	Comparison
00b	Equal any	The arithmetic comparison is "equal."
01b	Ranges	Arithmetic comparison is "greater than or equal" between even indexed bytes/words of reg and each byte/word of reg/mem. Arithmetic comparison is "less than or equal" between odd indexed bytes/words of reg and each byte/word of reg/mem. (reg/mem[m] >= reg[n] for n = even, reg/mem[m] <= reg[n] for n = odd)
10b	Equal each	The arithmetic comparison is "equal."
11b	Equal ordered	The arithmetic comparison is "equal."

All 256 (64) possible comparisons are always performed. The individual Boolean results of those comparisons are referred by "BoolRes[*Reg/Mem element index, Reg element index*]." Comparisons evaluating to "True" are represented with a 1, False with a 0 (positive logic). The initial results are then aggregated into a 16-bit (8-bit) intermediate result (IntRes1) using one of the modes described in the table below, as determined by Imm8 Control Byte bit[3:2].

See Section 4.1.6 for a description of the overrideIfDataInvalid() function used in Table 4-3.

Table 4-3. Aggregation Operation

Mode	Pseudocode
Equal any (find characters from a set)	UpperBound = imm8[0] ? 7 : 15; IntRes1 = 0; For j = 0 to UpperBound, j++ For i = 0 to UpperBound, i++ IntRes1[j] OR= overrideIfDataInvalid(BoolRes[j,i])
Ranges (find characters from ranges)	UpperBound = imm8[0] ? 7 : 15; IntRes1 = 0; For j = 0 to UpperBound, j++ For i = 0 to UpperBound, i+=2 IntRes1[j] OR= (overrideIfDataInvalid(BoolRes[j,i]) AND overrideIfDataInvalid(BoolRes[j,i+1]))
Equal each (string compare)	UpperBound = imm8[0] ? 7 : 15; IntRes1 = 0; For i = 0 to UpperBound, i++ IntRes1[i] = overrideIfDataInvalid(BoolRes[i,i])
Equal ordered (substring search)	UpperBound = imm8[0] ? 7 :15; IntRes1 = imm8[0] ? FFH : FFFFH For j = 0 to UpperBound, j++ For i = 0 to UpperBound-j, k=j to UpperBound, k++, i++ IntRes1[j] AND= overrideIfDataInvalid(BoolRes[k,i])

4.1.4 Polarity

IntRes1 may then be further modified by performing a 1's complement, according to the value of the Imm8 Control Byte bit[4]. Optionally, a mask may be used such that only those IntRes1 bits which correspond to "valid" reg/mem input elements are complemented (note that the definition of a valid input element is dependant on the specific opcode and is defined in each opcode's description). The result of the possible negation is referred to as IntRes2.

Table 4-4. Polarity

Imm8[5:4]	Operation	Description
00b	Positive Polarity (+)	IntRes2 = IntRes1
01b	Negative Polarity (-)	IntRes2 = -1 XOR IntRes1
10b	Masked (+)	IntRes2 = IntRes1
11b	Masked (-)	IntRes2[i] = IntRes1[i] if reg/mem[i] invalid, else = ~IntRes1[i]

4.1.5 Output Selection

Table 4-5. Output Selection

Imm8[6]	Operation	Description
0b	Least significant index	The index returned to ECX is of the least significant set bit in IntRes2.
1b	Most significant index	The index returned to ECX is of the most significant set bit in IntRes2.

For PCMPESTRI/PCMPISTRI, the Imm8 Control Byte bit[6] is used to determine if the index is of the least significant or most significant bit of IntRes2.

Table 4-6. Output Selection

Imm8[6]	Operation	Description
0b	Bit mask	IntRes2 is returned as the mask to the least significant bits of XMM0 with zero extension to 128 bits.
1b	Byte/word mask	IntRes2 is expanded into a byte/word mask (based on imm8[1]) and placed in XMM0. The expansion is performed by replicating each bit into all of the bits of the byte/word of the same index.

Specifically for PCMPESTRM/PCMPISTRM, the Imm8 Control Byte bit[6] is used to determine if the mask is a 16 (8) bit mask or a 128 bit byte/word mask.

4.1.6 Valid/Invalid Override of Comparisons

PCMPxSTRx instructions allow for the possibility that an end-of-string (EOS) situation may occur within the 128-bit packed data value (see the instruction descriptions below for details). Any data elements on either source that are determined to be past the EOS are considered to be invalid, and the treatment of invalid data within a comparison pair varies depending on the aggregation function being performed.

In general, the individual comparison result for each element pair BoolRes[i.j] can be forced true or false if one or more elements in the pair are invalid. See Table 4-7.

Table 4-7. Comparison Result for Each Element Pair BoolRes[i.j]

xmm1 byte/ word	xmm2/ m128 byte/word	Imm8[3:2] = 00b (equal any)	Imm8[3:2] = 01b (ranges)	Imm8[3:2] = 10b (equal each)	Imm8[3:2] = 11b (equal ordered)
Invalid	Invalid	Force false	Force false	Force true	Force true
Invalid	Valid	Force false	Force false	Force false	Force true
Valid	Invalid	Force false	Force false	Force false	Force false
Valid	Valid	Do not force	Do not force	Do not force	Do not force

4.1.7 Summary of Im8 Control byte

Table 4-8. Summary of Imm8 Control Byte

Imm8	Description
-------0b	128-bit sources treated as 16 packed bytes.
-------1b	128-bit sources treated as 8 packed words.
------0-b	Packed bytes/words are unsigned.
------1-b	Packed bytes/words are signed.
----00--b	Mode is equal any.
----01--b	Mode is ranges.
----10--b	Mode is equal each.
----11--b	Mode is equal ordered.
---0----b	IntRes1 is unmodified.
---1----b	IntRes1 is negated (1's complement).
--0-----b	Negation of IntRes1 is for all 16 (8) bits.
--1-----b	Negation of IntRes1 is masked by reg/mem validity.
-0------b	Index of the least significant, set, bit is used (regardless of corresponding input element validity). IntRes2 is returned in least significant bits of XMM0.
-1------b	Index of the most significant, set, bit is used (regardless of corresponding input element validity). Each bit of IntRes2 is expanded to byte/word.
0-------b	This bit currently has no defined effect, should be 0.
1-------b	This bit currently has no defined effect, should be 0.

4.1.8 Diagram Comparison and Aggregation Process

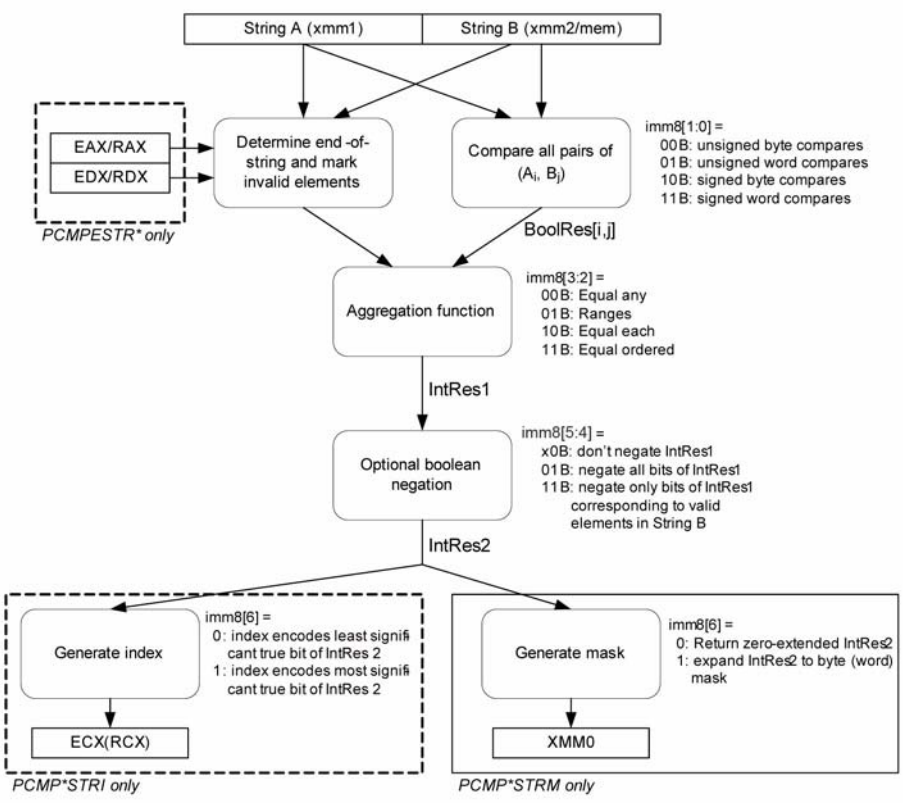


Figure 4-1. Operation of PCMPSTRx and PCMPESTRx

4.2 COMMON TRANSFORMATION AND PRIMITIVE FUNCTIONS FOR SHA1XXX AND SHA256XXX

The following primitive functions and transformations are used in the algorithmic descriptions of SHA1 and SHA256 instruction extensions SHA1NEXTE, SHA1RNDS4, SHA1MSG1, SHA1MSG2, SHA256RNDS4, SHA256MSG1 and SHA256MSG2. The operands of these primitives and transformation are generally 32-bit DWORD integers.

- f0(): A bit oriented logical operation that derives a new dword from three SHA1 state variables (dword). This function is used in SHA1 round 1 to 20 processing.

  f0(B,C,D) ← (B AND C) XOR ((NOT(B) AND D)

- f1(): A bit oriented logical operation that derives a new dword from three SHA1 state variables (dword). This function is used in SHA1 round 21 to 40 processing.

  f1(B,C,D) ← B XOR C XOR D

- f2(): A bit oriented logical operation that derives a new dword from three SHA1 state variables (dword). This function is used in SHA1 round 41 to 60 processing.

  f2(B,C,D) ← (B AND C) XOR (B AND D) XOR (C AND D)

- f3(): A bit oriented logical operation that derives a new dword from three SHA1 state variables (dword). This function is used in SHA1 round 61 to 80 processing. It is the same as f1().
  f3(B,C,D) ← B XOR C XOR D

- Ch(): A bit oriented logical operation that derives a new dword from three SHA256 state variables (dword).
  Ch(E,F,G) ← (E AND F) XOR ((NOT E) AND G)

- Maj(): A bit oriented logical operation that derives a new dword from three SHA256 state variables (dword).
  Maj(A,B,C) ← (A AND B) XOR (A AND C) XOR (B AND C)

ROR is rotate right operation
(A ROR N) ← A[N-1:0] || A[Width-1:N]

ROL is rotate left operation
(A ROL N) ← A ROR (Width-N)

SHR is the right shift operation
(A SHR N) ← ZEROES[N-1:0] || A[Width-1:N]

- Σ_0( ): A bit oriented logical and rotational transformation performed on a dword SHA256 state variable.
  Σ_0(A) ← (A ROR 2) XOR (A ROR 13) XOR (A ROR 22)

- Σ_1( ): A bit oriented logical and rotational transformation performed on a dword SHA256 state variable.
  Σ_1(E) ← (E ROR 6) XOR (E ROR 11) XOR (E ROR 25)

- σ_0( ): A bit oriented logical and rotational transformation performed on a SHA256 message dword used in the message scheduling.
  σ_0(W) ← (W ROR 7) XOR (W ROR 18) XOR (W SHR 3)

- σ_1( ): A bit oriented logical and rotational transformation performed on a SHA256 message dword used in the message scheduling.
  σ_1(W) ← (W ROR 17) XOR (W ROR 19) XOR (W SHR 10)

- K_i: SHA1 Constants dependent on immediate i.
  K0 = 0x5A827999
  K1 = 0x6ED9EBA1
  K2 = 0X8F1BBCDC
  K3 = 0xCA62C1D6

4.3 INSTRUCTIONS (M-U)

Chapter 4 continues an alphabetical discussion of Intel® 64 and IA-32 instructions (M-U). See also: Chapter 3, "Instruction Set Reference, A-L," in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A*, and Chapter 5, "Instruction Set Reference, V-Z," in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2C*.

MASKMOVDQU—Store Selected Bytes of Double Quadword

Opcode/ Instruction	Op/ En	64/32-bit Mode	CPUID Feature Flag	Description
66 0F F7 /r MASKMOVDQU *xmm1*, *xmm2*	RM	V/V	SSE2	Selectively write bytes from *xmm1* to memory location using the byte mask in *xmm2*. The default memory location is specified by DS:DI/EDI/RDI.
VEX.128.66.0F.WIG F7 /r VMASKMOVDQU *xmm1, xmm2*	RM	V/V	AVX	Selectively write bytes from *xmm1* to memory location using the byte mask in *xmm2*. The default memory location is specified by DS:DI/EDI/RDI.

Instruction Operand Encoding[1]

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

Description

Stores selected bytes from the source operand (first operand) into an 128-bit memory location. The mask operand (second operand) selects which bytes from the source operand are written to memory. The source and mask operands are XMM registers. The memory location specified by the effective address in the DI/EDI/RDI register (the default segment register is DS, but this may be overridden with a segment-override prefix). The memory location does not need to be aligned on a natural boundary. (The size of the store address depends on the address-size attribute.)

The most significant bit in each byte of the mask operand determines whether the corresponding byte in the source operand is written to the corresponding byte location in memory: 0 indicates no write and 1 indicates write.

The MASKMOVDQU instruction generates a non-temporal hint to the processor to minimize cache pollution. The non-temporal hint is implemented by using a write combining (WC) memory type protocol (see "Caching of Temporal vs. Non-Temporal Data" in Chapter 10, of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*). Because the WC protocol uses a weakly-ordered memory consistency model, a fencing operation implemented with the SFENCE or MFENCE instruction should be used in conjunction with MASKMOVDQU instructions if multiple processors might use different memory types to read/write the destination memory locations.

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

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

The MASKMOVDQU instruction can be used to improve performance of algorithms that need to merge data on a byte-by-byte basis. MASKMOVDQU should not cause a read for ownership; doing so generates unnecessary bandwidth since data is to be written directly using the byte-mask without allocating old data prior to the store.

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

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

If VMASKMOVDQU is encoded with VEX.L= 1, an attempt to execute the instruction encoded with VEX.L= 1 will cause an #UD exception.

1. ModRM.MOD = 011B required

Operation

```
IF (MASK[7] = 1)
    THEN DEST[DI/EDI] ← SRC[7:0] ELSE (* Memory location unchanged *); FI;
IF (MASK[15] = 1)
    THEN DEST[DI/EDI +1] ← SRC[15:8] ELSE (* Memory location unchanged *); FI;
    (* Repeat operation for 3rd through 14th bytes in source operand *)
IF (MASK[127] = 1)
    THEN DEST[DI/EDI +15] ← SRC[127:120] ELSE (* Memory location unchanged *); FI;
```

Intel C/C++ Compiler Intrinsic Equivalent

void _mm_maskmoveu_si128(__m128i d, __m128i n, char * p)

Other Exceptions

See Exceptions Type 4; additionally

#UD	If VEX.L= 1
	If VEX.vvvv ≠ 1111B.

MASKMOVQ—Store Selected Bytes of Quadword

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

Instruction Operand Encoding

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

Description

Stores selected bytes from the source operand (first operand) into a 64-bit memory location. The mask operand (second operand) selects which bytes from the source operand are written to memory. The source and mask operands are MMX technology registers. The memory location specified by the effective address in the DI/EDI/RDI register (the default segment register is DS, but this may be overridden with a segment-override prefix). The memory location does not need to be aligned on a natural boundary. (The size of the store address depends on the address-size attribute.)

The most significant bit in each byte of the mask operand determines whether the corresponding byte in the source operand is written to the corresponding byte location in memory: 0 indicates no write and 1 indicates write.

The MASKMOVQ instruction generates a non-temporal hint to the processor to minimize cache pollution. The non-temporal hint is implemented by using a write combining (WC) memory type protocol (see "Caching of Temporal vs. Non-Temporal Data" in Chapter 10, of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*). Because the WC protocol uses a weakly-ordered memory consistency model, a fencing operation implemented with the SFENCE or MFENCE instruction should be used in conjunction with MASKMOVQ instructions if multiple processors might use different memory types to read/write the destination memory locations.

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

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

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

The MASKMOVQ instruction can be used to improve performance for algorithms that need to merge data on a byte-by-byte basis. It should not cause a read for ownership; doing so generates unnecessary bandwidth since data is to be written directly using the byte-mask without allocating old data prior to the store.

In 64-bit mode, the memory address is specified by DS:RDI.

Operation

```
IF (MASK[7] = 1)
    THEN DEST[DI/EDI] ← SRC[7:0] ELSE (* Memory location unchanged *); FI;
IF (MASK[15] = 1)
    THEN DEST[DI/EDI +1] ← SRC[15:8] ELSE (* Memory location unchanged *); FI;
    (* Repeat operation for 3rd through 6th bytes in source operand *)
IF (MASK[63] = 1)
    THEN DEST[DI/EDI +15] ← SRC[63:56] ELSE (* Memory location unchanged *); FI;
```

Intel C/C++ Compiler Intrinsic Equivalent

void _mm_maskmove_si64(__m64d, __m64n, char * p)

Other Exceptions

See Table 22-8, "Exception Conditions for Legacy SIMD/MMX Instructions without FP Exception," in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*.

MAXPD—Maximum of Packed Double-Precision Floating-Point Values

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 5F /r MAXPD xmm1, xmm2/m128	A	V/V	SSE2	Return the maximum double-precision floating-point values between xmm1 and xmm2/m128.
VEX.NDS.128.66.0F.WIG 5F /r VMAXPD xmm1, xmm2, xmm3/m128	B	V/V	AVX	Return the maximum double-precision floating-point values between xmm2 and xmm3/m128.
VEX.NDS.256.66.0F.WIG 5F /r VMAXPD ymm1, ymm2, ymm3/m256	B	V/V	AVX	Return the maximum packed double-precision floating-point values between ymm2 and ymm3/m256.
EVEX.NDS.128.66.0F.W1 5F /r VMAXPD xmm1 {k1}{z}, xmm2, xmm3/m128/m64bcst	C	V/V	AVX512VL AVX512F	Return the maximum packed double-precision floating-point values between xmm2 and xmm3/m128/m64bcst and store result in xmm1 subject to writemask k1.
EVEX.NDS.256.66.0F.W1 5F /r VMAXPD ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst	C	V/V	AVX512VL AVX512F	Return the maximum packed double-precision floating-point values between ymm2 and ymm3/m256/m64bcst and store result in ymm1 subject to writemask k1.
EVEX.NDS.512.66.0F.W1 5F /r VMAXPD zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst{sae}	C	V/V	AVX512F	Return the maximum packed double-precision floating-point values between zmm2 and zmm3/m512/m64bcst and store result in zmm1 subject to writemask k1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	NA	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
B	NA	ModRM:reg (w)	VEX.vvvv	ModRM:r/m (r)	NA
C	Full Vector	ModRM:reg (w)	EVEX.vvvv	ModRM:r/m (r)	NA

Description

Performs a SIMD compare of the packed double-precision floating-point values in the first source operand and the second source operand and returns the maximum value for each pair of values to the destination operand.

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

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

EVEX encoded versions: The first source operand (the second operand) is a ZMM/YMM/XMM register. The second source operand can be a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 64-bit memory location. The destination operand is a ZMM/YMM/XMM register conditionally updated with writemask k1.

VEX.256 encoded version: The first source operand is a YMM register. The second source operand can be a YMM register or a 256-bit memory location. The destination operand is a YMM register. The upper bits (MAXVL-1:256) of the corresponding ZMM register destination are zeroed.

VEX.128 encoded version: The first source operand is a XMM register. The second source operand can be a XMM register or a 128-bit memory location. The destination operand is a XMM register. The upper bits (MAXVL-1:128) of the corresponding ZMM register destination are zeroed.

128-bit Legacy SSE version: The second source can be an XMM register or an 128-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (MAXVL-1:128) of the corresponding ZMM register destination are unmodified.

Operation

```
MAX(SRC1, SRC2)
{
    IF ((SRC1 = 0.0) and (SRC2 = 0.0)) THEN DEST ←SRC2;
        ELSE IF (SRC1 = SNaN) THEN DEST ←SRC2; FI;
        ELSE IF (SRC2 = SNaN) THEN DEST ←SRC2; FI;
        ELSE IF (SRC1 > SRC2) THEN DEST ←SRC1;
        ELSE DEST ←SRC2;
    FI;
}
```

VMAXPD (EVEX encoded versions)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j ← 0 TO KL-1
    i ← j * 64
    IF k1[j] OR *no writemask*
        THEN
            IF (EVEX.b = 1) AND (SRC2 *is memory*)
                THEN
                    DEST[i+63:i] ← MAX(SRC1[i+63:i], SRC2[63:0])
                ELSE
                    DEST[i+63:i] ← MAX(SRC1[i+63:i], SRC2[i+63:i])
            FI;
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE  DEST[i+63:i] ← 0              ; zeroing-masking
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0
```

VMAXPD (VEX.256 encoded version)

```
DEST[63:0] ←MAX(SRC1[63:0], SRC2[63:0])
DEST[127:64] ←MAX(SRC1[127:64], SRC2[127:64])
DEST[191:128] ←MAX(SRC1[191:128], SRC2[191:128])
DEST[255:192] ←MAX(SRC1[255:192], SRC2[255:192])
DEST[MAXVL-1:256] ←0
```

VMAXPD (VEX.128 encoded version)

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

MAXPD (128-bit Legacy SSE version)
DEST[63:0] ←MAX(DEST[63:0], SRC[63:0])
DEST[127:64] ←MAX(DEST[127:64], SRC[127:64])
DEST[MAXVL-1:128] (Unmodified)

Intel C/C++ Compiler Intrinsic Equivalent

VMAXPD __m512d _mm512_max_pd(__m512d a, __m512d b);
VMAXPD __m512d _mm512_mask_max_pd(__m512d s, __mmask8 k, __m512d a, __m512d b,);
VMAXPD __m512d _mm512_maskz_max_pd(__mmask8 k, __m512d a, __m512d b);
VMAXPD __m512d _mm512_max_round_pd(__m512d a, __m512d b, int);
VMAXPD __m512d _mm512_mask_max_round_pd(__m512d s, __mmask8 k, __m512d a, __m512d b, int);
VMAXPD __m512d _mm512_maskz_max_round_pd(__mmask8 k, __m512d a, __m512d b, int);
VMAXPD __m256d _mm256_mask_max_pd(__m5256d s, __mmask8 k, __m256d a, __m256d b);
VMAXPD __m256d _mm256_maskz_max_pd(__mmask8 k, __m256d a, __m256d b);
VMAXPD __m128d _mm_mask_max_pd(__m128d s, __mmask8 k, __m128d a, __m128d b);
VMAXPD __m128d _mm_maskz_max_pd(__mmask8 k, __m128d a, __m128d b);
VMAXPD __m256d _mm256_max_pd (__m256d a, __m256d b);
(V)MAXPD __m128d _mm_max_pd (__m128d a, __m128d b);

SIMD Floating-Point Exceptions

Invalid (including QNaN Source Operand), Denormal

Other Exceptions

Non-EVEX-encoded instruction, see Exceptions Type 2.
EVEX-encoded instruction, see Exceptions Type E2.

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

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F 5F /r MAXPS xmm1, xmm2/m128	A	V/V	SSE	Return the maximum single-precision floating-point values between xmm1 and xmm2/mem.
VEX.NDS.128.0F.WIG 5F /r VMAXPS xmm1, xmm2, xmm3/m128	B	V/V	AVX	Return the maximum single-precision floating-point values between xmm2 and xmm3/mem.
VEX.NDS.256.0F.WIG 5F /r VMAXPS ymm1, ymm2, ymm3/m256	B	V/V	AVX	Return the maximum single-precision floating-point values between ymm2 and ymm3/mem.
EVEX.NDS.128.0F.W0 5F /r VMAXPS xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst	C	V/V	AVX512VL AVX512F	Return the maximum packed single-precision floating-point values between xmm2 and xmm3/m128/m32bcst and store result in xmm1 subject to writemask k1.
EVEX.NDS.256.0F.W0 5F /r VMAXPS ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst	C	V/V	AVX512VL AVX512F	Return the maximum packed single-precision floating-point values between ymm2 and ymm3/m256/m32bcst and store result in ymm1 subject to writemask k1.
EVEX.NDS.512.0F.W0 5F /r VMAXPS zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst{sae}	C	V/V	AVX512F	Return the maximum packed single-precision floating-point values between zmm2 and zmm3/m512/m32bcst and store result in zmm1 subject to writemask k1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	NA	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
B	NA	ModRM:reg (w)	VEX.vvvv	ModRM:r/m (r)	NA
C	Full Vector	ModRM:reg (w)	EVEX.vvvv	ModRM:r/m (r)	NA

Description

Performs a SIMD compare of the packed single-precision floating-point values in the first source operand and the second source operand and returns the maximum value for each pair of values to the destination operand.

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

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

EVEX encoded versions: The first source operand (the second operand) is a ZMM/YMM/XMM register. The second source operand can be a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 32-bit memory location. The destination operand is a ZMM/YMM/XMM register conditionally updated with writemask k1.

VEX.256 encoded version: The first source operand is a YMM register. The second source operand can be a YMM register or a 256-bit memory location. The destination operand is a YMM register. The upper bits (MAXVL-1:256) of the corresponding ZMM register destination are zeroed.

VEX.128 encoded version: The first source operand is a XMM register. The second source operand can be a XMM register or a 128-bit memory location. The destination operand is a XMM register. The upper bits (MAXVL-1:128) of the corresponding ZMM register destination are zeroed.

128-bit Legacy SSE version: The second source can be an XMM register or an 128-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (MAXVL-1:128) of the corresponding ZMM register destination are unmodified.

Operation

```
MAX(SRC1, SRC2)
{
    IF ((SRC1 = 0.0) and (SRC2 = 0.0)) THEN DEST ←SRC2;
        ELSE IF (SRC1 = SNaN) THEN DEST ←SRC2; FI;
        ELSE IF (SRC2 = SNaN) THEN DEST ←SRC2; FI;
        ELSE IF (SRC1 > SRC2) THEN DEST ←SRC1;
        ELSE DEST ←SRC2;
    FI;
}
```

VMAXPS (EVEX encoded versions)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j ← 0 TO KL-1
    i ← j * 32
    IF k1[j] OR *no writemask*
        THEN
            IF (EVEX.b = 1) AND (SRC2 *is memory*)
                THEN
                    DEST[i+31:i] ← MAX(SRC1[i+31:i], SRC2[31:0])
                ELSE
                    DEST[i+31:i] ← MAX(SRC1[i+31:i], SRC2[i+31:i])
            FI;
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE  DEST[i+31:i] ← 0              ; zeroing-masking
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0
```

VMAXPS (VEX.256 encoded version)

```
DEST[31:0] ←MAX(SRC1[31:0], SRC2[31:0])
DEST[63:32] ←MAX(SRC1[63:32], SRC2[63:32])
DEST[95:64] ←MAX(SRC1[95:64], SRC2[95:64])
DEST[127:96] ←MAX(SRC1[127:96], SRC2[127:96])
DEST[159:128] ←MAX(SRC1[159:128], SRC2[159:128])
DEST[191:160] ←MAX(SRC1[191:160], SRC2[191:160])
DEST[223:192] ←MAX(SRC1[223:192], SRC2[223:192])
DEST[255:224] ←MAX(SRC1[255:224], SRC2[255:224])
DEST[MAXVL-1:256] ←0
```

VMAXPS (VEX.128 encoded version)

```
DEST[31:0] ←MAX(SRC1[31:0], SRC2[31:0])
DEST[63:32] ←MAX(SRC1[63:32], SRC2[63:32])
DEST[95:64] ←MAX(SRC1[95:64], SRC2[95:64])
DEST[127:96] ←MAX(SRC1[127:96], SRC2[127:96])
DEST[MAXVL-1:128] ←0
```

MAXPS (128-bit Legacy SSE version)
DEST[31:0] ←MAX(DEST[31:0], SRC[31:0])
DEST[63:32] ←MAX(DEST[63:32], SRC[63:32])
DEST[95:64] ←MAX(DEST[95:64], SRC[95:64])
DEST[127:96] ←MAX(DEST[127:96], SRC[127:96])
DEST[MAXVL-1:128] (Unmodified)

Intel C/C++ Compiler Intrinsic Equivalent

VMAXPS __m512 _mm512_max_ps(__m512 a, __m512 b);
VMAXPS __m512 _mm512_mask_max_ps(__m512 s, __mmask16 k, __m512 a, __m512 b);
VMAXPS __m512 _mm512_maskz_max_ps(__mmask16 k, __m512 a, __m512 b);
VMAXPS __m512 _mm512_max_round_ps(__m512 a, __m512 b, int);
VMAXPS __m512 _mm512_mask_max_round_ps(__m512 s, __mmask16 k, __m512 a, __m512 b, int);
VMAXPS __m512 _mm512_maskz_max_round_ps(__mmask16 k, __m512 a, __m512 b, int);
VMAXPS __m256 _mm256_mask_max_ps(__m256 s, __mmask8 k, __m256 a, __m256 b);
VMAXPS __m256 _mm256_maskz_max_ps(__mmask8 k, __m256 a, __m256 b);
VMAXPS __m128 _mm_mask_max_ps(__m128 s, __mmask8 k, __m128 a, __m128 b);
VMAXPS __m128 _mm_maskz_max_ps(__mmask8 k, __m128 a, __m128 b);
VMAXPS __m256 _mm256_max_ps (__m256 a, __m256 b);
MAXPS __m128 _mm_max_ps (__m128 a, __m128 b);

SIMD Floating-Point Exceptions

Invalid (including QNaN Source Operand), Denormal

Other Exceptions

Non-EVEX-encoded instruction, see Exceptions Type 2.
EVEX-encoded instruction, see Exceptions Type E2.

MAXSD—Return Maximum Scalar Double-Precision Floating-Point Value

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
F2 0F 5F /r MAXSD xmm1, xmm2/m64	A	V/V	SSE2	Return the maximum scalar double-precision floating-point value between xmm2/m64 and xmm1.
VEX.NDS.LIG.F2.0F.WIG 5F /r VMAXSD xmm1, xmm2, xmm3/m64	B	V/V	AVX	Return the maximum scalar double-precision floating-point value between xmm3/m64 and xmm2.
EVEX.NDS.LIG.F2.0F.W1 5F /r VMAXSD xmm1 {k1}{z}, xmm2, xmm3/m64{sae}	C	V/V	AVX512F	Return the maximum scalar double-precision floating-point value between xmm3/m64 and xmm2.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	NA	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
B	NA	ModRM:reg (w)	VEX.vvvv	ModRM:r/m (r)	NA
C	Tuple1 Scalar	ModRM:reg (w)	EVEX.vvvv	ModRM:r/m (r)	NA

Description

Compares the low double-precision floating-point values in the first source operand and the second source operand, and returns the maximum value to the low quadword of the destination operand. The second source operand can be an XMM register or a 64-bit memory location. The first source and destination operands are XMM registers. When the second source operand is a memory operand, only 64 bits are accessed.

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

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

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

VEX.128 and EVEX encoded version: Bits (127:64) of the XMM register destination are copied from corresponding bits in the first source operand. Bits (MAXVL-1:128) of the destination register are zeroed.

EVEX encoded version: The low quadword element of the destination operand is updated according to the writemask.

Software should ensure VMAXSD is encoded with VEX.L=0. Encoding VMAXSD with VEX.L=1 may encounter unpredictable behavior across different processor generations.

Operation

```
MAX(SRC1, SRC2)
{
    IF ((SRC1 = 0.0) and (SRC2 = 0.0)) THEN DEST ←SRC2;
        ELSE IF (SRC1 = SNaN) THEN DEST ←SRC2; FI;
        ELSE IF (SRC2 = SNaN) THEN DEST ←SRC2; FI;
        ELSE IF (SRC1 > SRC2) THEN DEST ←SRC1;
        ELSE DEST ←SRC2;
    FI;
}
```

VMAXSD (EVEX encoded version)

```
IF k1[0] or *no writemask*
    THEN      DEST[63:0] ← MAX(SRC1[63:0], SRC2[63:0])
    ELSE
        IF *merging-masking*                    ; merging-masking
            THEN *DEST[63:0] remains unchanged*
            ELSE                                ; zeroing-masking
                DEST[63:0] ← 0
        FI;
FI;
DEST[127:64] ← SRC1[127:64]
DEST[MAXVL-1:128] ← 0
```

VMAXSD (VEX.128 encoded version)

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

MAXSD (128-bit Legacy SSE version)

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

Intel C/C++ Compiler Intrinsic Equivalent

```
VMAXSD __m128d _mm_max_round_sd( __m128d a, __m128d b, int);
VMAXSD __m128d _mm_mask_max_round_sd(__m128d s, __mmask8 k, __m128d a, __m128d b, int);
VMAXSD __m128d _mm_maskz_max_round_sd( __mmask8 k, __m128d a, __m128d b, int);
MAXSD __m128d _mm_max_sd(__m128d a, __m128d b)
```

SIMD Floating-Point Exceptions

Invalid (Including QNaN Source Operand), Denormal

Other Exceptions

Non-EVEX-encoded instruction, see Exceptions Type 3.

EVEX-encoded instruction, see Exceptions Type E3.

MAXSS—Return Maximum Scalar Single-Precision Floating-Point Value

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
F3 0F 5F /r MAXSS xmm1, xmm2/m32	A	V/V	SSE	Return the maximum scalar single-precision floating-point value between xmm2/m32 and xmm1.
VEX.NDS.LIG.F3.0F.WIG 5F /r VMAXSS xmm1, xmm2, xmm3/m32	B	V/V	AVX	Return the maximum scalar single-precision floating-point value between xmm3/m32 and xmm2.
EVEX.NDS.LIG.F3.0F.W0 5F /r VMAXSS xmm1 {k1}{z}, xmm2, xmm3/m32{sae}	C	V/V	AVX512F	Return the maximum scalar single-precision floating-point value between xmm3/m32 and xmm2.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	NA	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
B	NA	ModRM:reg (w)	VEX.vvvv	ModRM:r/m (r)	NA
C	Tuple1 Scalar	ModRM:reg (w)	EVEX.vvvv	ModRM:r/m (r)	NA

Description

Compares the low single-precision floating-point values in the first source operand and the second source operand, and returns the maximum value to the low doubleword of the destination operand.

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

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

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

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

VEX.128 and EVEX encoded version: The first source operand is an xmm register encoded by VEX.vvvv. Bits (127:32) of the XMM register destination are copied from corresponding bits in the first source operand. Bits (MAXVL:128) of the destination register are zeroed.

EVEX encoded version: The low doubleword element of the destination operand is updated according to the writemask.

Software should ensure VMAXSS is encoded with VEX.L=0. Encoding VMAXSS with VEX.L=1 may encounter unpredictable behavior across different processor generations.

Operation

```
MAX(SRC1, SRC2)
{
    IF ((SRC1 = 0.0) and (SRC2 = 0.0)) THEN DEST ←SRC2;
        ELSE IF (SRC1 = SNaN) THEN DEST ←SRC2; FI;
        ELSE IF (SRC2 = SNaN) THEN DEST ←SRC2; FI;
        ELSE IF (SRC1 > SRC2) THEN DEST ←SRC1;
        ELSE DEST ←SRC2;
    FI;
}
```

VMAXSS (EVEX encoded version)

```
IF k1[0] or *no writemask*
    THEN      DEST[31:0] ← MAX(SRC1[31:0], SRC2[31:0])
    ELSE
        IF *merging-masking*                    ; merging-masking
            THEN *DEST[31:0] remains unchanged*
            ELSE                                ; zeroing-masking
                THEN DEST[31:0] ← 0
        FI;
FI;
DEST[127:32] ← SRC1[127:32]
DEST[MAXVL-1:128] ← 0
```

VMAXSS (VEX.128 encoded version)

```
DEST[31:0] ←MAX(SRC1[31:0], SRC2[31:0])
DEST[127:32] ←SRC1[127:32]
DEST[MAXVL-1:128] ←0
```

MAXSS (128-bit Legacy SSE version)

```
DEST[31:0] ←MAX(DEST[31:0], SRC[31:0])
DEST[MAXVL-1:32] (Unmodified)
```

Intel C/C++ Compiler Intrinsic Equivalent

VMAXSS __m128 _mm_max_round_ss(__m128 a, __m128 b, int);
VMAXSS __m128 _mm_mask_max_round_ss(__m128 s, __mmask8 k, __m128 a, __m128 b, int);
VMAXSS __m128 _mm_maskz_max_round_ss(__mmask8 k, __m128 a, __m128 b, int);
MAXSS __m128 _mm_max_ss(__m128 a, __m128 b)

SIMD Floating-Point Exceptions

Invalid (Including QNaN Source Operand), Denormal

Other Exceptions

Non-EVEX-encoded instruction, see Exceptions Type 3.
EVEX-encoded instruction, see Exceptions Type E3.

MFENCE—Memory Fence

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
NP 0F AE F0	MFENCE	ZO	Valid	Valid	Serializes load and store operations.

Instruction Operand Encoding

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

Description

Performs a serializing operation on all load-from-memory and store-to-memory instructions that were issued prior the MFENCE instruction. This serializing operation guarantees that every load and store instruction that precedes the MFENCE instruction in program order becomes globally visible before any load or store instruction that follows the MFENCE instruction.[1] The MFENCE instruction is ordered with respect to all load and store instructions, other MFENCE instructions, any LFENCE and SFENCE instructions, and any serializing instructions (such as the CPUID instruction). MFENCE does not serialize the instruction stream.

Weakly ordered memory types can be used to achieve higher processor performance through such techniques as out-of-order issue, speculative reads, write-combining, and write-collapsing. The degree to which a consumer of data recognizes or knows that the data is weakly ordered varies among applications and may be unknown to the producer of this data. The MFENCE instruction provides a performance-efficient way of ensuring load and store ordering between routines that produce weakly-ordered results and routines that consume that data.

Processors are free to fetch and cache data speculatively from regions of system memory that use the WB, WC, and WT memory types. This speculative fetching can occur at any time and is not tied to instruction execution. Thus, it is not ordered with respect to executions of the MFENCE instruction; data can be brought into the caches speculatively just before, during, or after the execution of an MFENCE instruction.

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

Specification of the instruction's opcode above indicates a ModR/M byte of F0. For this instruction, the processor ignores the r/m field of the ModR/M byte. Thus, MFENCE is encoded by any opcode of the form 0F AE Fx, where x is in the range 0-7.

Operation

```
Wait_On_Following_Loads_And_Stores_Until(preceding_loads_and_stores_globally_visible);
```

Intel C/C++ Compiler Intrinsic Equivalent

void _mm_mfence(void)

Exceptions (All Modes of Operation)

#UD	If CPUID.01H:EDX.SSE2[bit 26] = 0.
	If the LOCK prefix is used.

1. A load instruction is considered to become globally visible when the value to be loaded into its destination register is determined.

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

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 5D /r MINPD xmm1, xmm2/m128	A	V/V	SSE2	Return the minimum double-precision floating-point values between xmm1 and xmm2/mem
VEX.NDS.128.66.0F.WIG 5D /r VMINPD xmm1, xmm2, xmm3/m128	B	V/V	AVX	Return the minimum double-precision floating-point values between xmm2 and xmm3/mem.
VEX.NDS.256.66.0F.WIG 5D /r VMINPD ymm1, ymm2, ymm3/m256	B	V/V	AVX	Return the minimum packed double-precision floating-point values between ymm2 and ymm3/mem.
EVEX.NDS.128.66.0F.W1 5D /r VMINPD xmm1 {k1}{z}, xmm2, xmm3/m128/m64bcst	C	V/V	AVX512VL AVX512F	Return the minimum packed double-precision floating-point values between xmm2 and xmm3/m128/m64bcst and store result in xmm1 subject to writemask k1.
EVEX.NDS.256.66.0F.W1 5D /r VMINPD ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst	C	V/V	AVX512VL AVX512F	Return the minimum packed double-precision floating-point values between ymm2 and ymm3/m256/m64bcst and store result in ymm1 subject to writemask k1.
EVEX.NDS.512.66.0F.W1 5D /r VMINPD zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst{sae}	C	V/V	AVX512F	Return the minimum packed double-precision floating-point values between zmm2 and zmm3/m512/m64bcst and store result in zmm1 subject to writemask k1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	NA	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
B	NA	ModRM:reg (w)	VEX.vvvv	ModRM:r/m (r)	NA
C	Full Vector	ModRM:reg (w)	EVEX.vvvv	ModRM:r/m (r)	NA

Description

Performs a SIMD compare of the packed double-precision floating-point values in the first source operand and the second source operand and returns the minimum value for each pair of values to the destination operand.

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

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

EVEX encoded versions: The first source operand (the second operand) is a ZMM/YMM/XMM register. The second source operand can be a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 64-bit memory location. The destination operand is a ZMM/YMM/XMM register conditionally updated with writemask k1.

VEX.256 encoded version: The first source operand is a YMM register. The second source operand can be a YMM register or a 256-bit memory location. The destination operand is a YMM register. The upper bits (MAXVL-1:256) of the corresponding ZMM register destination are zeroed.

VEX.128 encoded version: The first source operand is a XMM register. The second source operand can be a XMM register or a 128-bit memory location. The destination operand is a XMM register. The upper bits (MAXVL-1:128) of the corresponding ZMM register destination are zeroed.

128-bit Legacy SSE version: The second source can be an XMM register or an 128-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (MAXVL-1:128) of the corresponding ZMM register destination are unmodified.

Operation

```
MIN(SRC1, SRC2)
{
    IF ((SRC1 = 0.0) and (SRC2 = 0.0)) THEN DEST ←SRC2;
        ELSE IF (SRC1 = SNaN) THEN DEST ←SRC2; FI;
        ELSE IF (SRC2 = SNaN) THEN DEST ←SRC2; FI;
        ELSE IF (SRC1 < SRC2) THEN DEST ←SRC1;
        ELSE DEST ←SRC2;
    FI;
}
```

VMINPD (EVEX encoded version)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j ← 0 TO KL-1
    i ← j * 64
    IF k1[j] OR *no writemask*
        THEN
            IF (EVEX.b = 1) AND (SRC2 *is memory*)
                THEN
                    DEST[i+63:i] ← MIN(SRC1[i+63:i], SRC2[63:0])
                ELSE
                    DEST[i+63:i] ← MIN(SRC1[i+63:i], SRC2[i+63:i])
            FI;
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE  DEST[i+63:i] ← 0              ; zeroing-masking
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0
```

VMINPD (VEX.256 encoded version)

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

VMINPD (VEX.128 encoded version)

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

MINPD (128-bit Legacy SSE version)

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

Intel C/C++ Compiler Intrinsic Equivalent

VMINPD __m512d _mm512_min_pd(__m512d a, __m512d b);
VMINPD __m512d _mm512_mask_min_pd(__m512d s, __mmask8 k, __m512d a, __m512d b);
VMINPD __m512d _mm512_maskz_min_pd(__mmask8 k, __m512d a, __m512d b);
VMINPD __m512d _mm512_min_round_pd(__m512d a, __m512d b, int);
VMINPD __m512d _mm512_mask_min_round_pd(__m512d s, __mmask8 k, __m512d a, __m512d b, int);
VMINPD __m512d _mm512_maskz_min_round_pd(__mmask8 k, __m512d a, __m512d b, int);
VMINPD __m256d _mm256_mask_min_pd(__m256d s, __mmask8 k, __m256d a, __m256d b);
VMINPD __m256d _mm256_maskz_min_pd(__mmask8 k, __m256d a, __m256d b);
VMINPD __m128d _mm_mask_min_pd(__m128d s, __mmask8 k, __m128d a, __m128d b);
VMINPD __m128d _mm_maskz_min_pd(__mmask8 k, __m128d a, __m128d b);
VMINPD __m256d _mm256_min_pd (__m256d a, __m256d b);
MINPD __m128d _mm_min_pd (__m128d a, __m128d b);

SIMD Floating-Point Exceptions

Invalid (including QNaN Source Operand), Denormal

Other Exceptions

Non-EVEX-encoded instruction, see Exceptions Type 2.
EVEX-encoded instruction, see Exceptions Type E2.

MINPS—Minimum of Packed Single-Precision Floating-Point Values

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F 5D /r MINPS xmm1, xmm2/m128	A	V/V	SSE	Return the minimum single-precision floating-point values between xmm1 and xmm2/mem.
VEX.NDS.128.0F.WIG 5D /r VMINPS xmm1, xmm2, xmm3/m128	B	V/V	AVX	Return the minimum single-precision floating-point values between xmm2 and xmm3/mem.
VEX.NDS.256.0F.WIG 5D /r VMINPS ymm1, ymm2, ymm3/m256	B	V/V	AVX	Return the minimum single double-precision floating-point values between ymm2 and ymm3/mem.
EVEX.NDS.128.0F.W0 5D /r VMINPS xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst	C	V/V	AVX512VL AVX512F	Return the minimum packed single-precision floating-point values between xmm2 and xmm3/m128/m32bcst and store result in xmm1 subject to writemask k1.
EVEX.NDS.256.0F.W0 5D /r VMINPS ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst	C	V/V	AVX512VL AVX512F	Return the minimum packed single-precision floating-point values between ymm2 and ymm3/m256/m32bcst and store result in ymm1 subject to writemask k1.
EVEX.NDS.512.0F.W0 5D /r VMINPS zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst{sae}	C	V/V	AVX512F	Return the minimum packed single-precision floating-point values between zmm2 and zmm3/m512/m32bcst and store result in zmm1 subject to writemask k1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	NA	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
B	NA	ModRM:reg (w)	VEX.vvvv	ModRM:r/m (r)	NA
C	Full Vector	ModRM:reg (w)	EVEX.vvvv	ModRM:r/m (r)	NA

Description

Performs a SIMD compare of the packed single-precision floating-point values in the first source operand and the second source operand and returns the minimum value for each pair of values to the destination operand.

If the values being compared are both 0.0s (of either sign), the value in the second operand (source operand) is returned. If a value in the second operand is an SNaN, then SNaN is forwarded unchanged to the destination (that is, a QNaN version of the SNaN is not returned).

If only one value is a NaN (SNaN or QNaN) for this instruction, the second operand (source operand), either a NaN or a valid floating-point value, is written to the result. If instead of this behavior, it is required that the NaN source operand (from either the first or second operand) be returned, the action of MINPS can be emulated using a sequence of instructions, such as, a comparison followed by AND, ANDN and OR.

EVEX encoded versions: The first source operand (the second operand) is a ZMM/YMM/XMM register. The second source operand can be a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 32-bit memory location. The destination operand is a ZMM/YMM/XMM register conditionally updated with writemask k1.

VEX.256 encoded version: The first source operand is a YMM register. The second source operand can be a YMM register or a 256-bit memory location. The destination operand is a YMM register. The upper bits (MAXVL-1:256) of the corresponding ZMM register destination are zeroed.

VEX.128 encoded version: The first source operand is a XMM register. The second source operand can be a XMM register or a 128-bit memory location. The destination operand is a XMM register. The upper bits (MAXVL-1:128) of the corresponding ZMM register destination are zeroed.

128-bit Legacy SSE version: The second source can be an XMM register or an 128-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (MAXVL-1:128) of the corresponding ZMM register destination are unmodified.

Operation

```
MIN(SRC1, SRC2)
{
    IF ((SRC1 = 0.0) and (SRC2 = 0.0)) THEN DEST ←SRC2;
        ELSE IF (SRC1 = SNaN) THEN DEST ←SRC2; FI;
        ELSE IF (SRC2 = SNaN) THEN DEST ←SRC2; FI;
        ELSE IF (SRC1 < SRC2) THEN DEST ←SRC1;
        ELSE DEST ←SRC2;
    FI;
}
```

VMINPS (EVEX encoded version)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j ← 0 TO KL-1
    i ← j * 32
    IF k1[j] OR *no writemask*
        THEN
            IF (EVEX.b = 1) AND (SRC2 *is memory*)
                THEN
                    DEST[i+31:i] ← MIN(SRC1[i+31:i], SRC2[31:0])
                ELSE
                    DEST[i+31:i] ← MIN(SRC1[i+31:i], SRC2[i+31:i])
            FI;
            ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE  DEST[i+31:i] ← 0              ; zeroing-masking
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0
```

VMINPS (VEX.256 encoded version)

```
DEST[31:0] ←MIN(SRC1[31:0], SRC2[31:0])
DEST[63:32] ←MIN(SRC1[63:32], SRC2[63:32])
DEST[95:64] ←MIN(SRC1[95:64], SRC2[95:64])
DEST[127:96] ←MIN(SRC1[127:96], SRC2[127:96])
DEST[159:128] ←MIN(SRC1[159:128], SRC2[159:128])
DEST[191:160] ←MIN(SRC1[191:160], SRC2[191:160])
DEST[223:192] ←MIN(SRC1[223:192], SRC2[223:192])
DEST[255:224] ←MIN(SRC1[255:224], SRC2[255:224])
```

VMINPS (VEX.128 encoded version)

```
DEST[31:0] ←MIN(SRC1[31:0], SRC2[31:0])
DEST[63:32] ←MIN(SRC1[63:32], SRC2[63:32])
DEST[95:64] ←MIN(SRC1[95:64], SRC2[95:64])
DEST[127:96] ←MIN(SRC1[127:96], SRC2[127:96])
DEST[MAXVL-1:128] ←0
```

MINPS (128-bit Legacy SSE version)
DEST[31:0] ←MIN(SRC1[31:0], SRC2[31:0])
DEST[63:32] ←MIN(SRC1[63:32], SRC2[63:32])
DEST[95:64] ←MIN(SRC1[95:64], SRC2[95:64])
DEST[127:96] ←MIN(SRC1[127:96], SRC2[127:96])
DEST[MAXVL-1:128] (Unmodified)

Intel C/C++ Compiler Intrinsic Equivalent

VMINPS __m512 _mm512_min_ps(__m512 a, __m512 b);
VMINPS __m512 _mm512_mask_min_ps(__m512 s, __mmask16 k, __m512 a, __m512 b);
VMINPS __m512 _mm512_maskz_min_ps(__mmask16 k, __m512 a, __m512 b);
VMINPS __m512 _mm512_min_round_ps(__m512 a, __m512 b, int);
VMINPS __m512 _mm512_mask_min_round_ps(__m512 s, __mmask16 k, __m512 a, __m512 b, int);
VMINPS __m512 _mm512_maskz_min_round_ps(__mmask16 k, __m512 a, __m512 b, int);
VMINPS __m256 _mm256_mask_min_ps(__m256 s, __mmask8 k, __m256 a, __m256 b);
VMINPS __m256 _mm256_maskz_min_ps(__mmask8 k, __m256 a, __m25 b);
VMINPS __m128 _mm_mask_min_ps(__m128 s, __mmask8 k, __m128 a, __m128 b);
VMINPS __m128 _mm_maskz_min_ps(__mmask8 k, __m128 a, __m128 b);
VMINPS __m256 _mm256_min_ps (__m256 a, __m256 b);
MINPS __m128 _mm_min_ps (__m128 a, __m128 b);

SIMD Floating-Point Exceptions

Invalid (including QNaN Source Operand), Denormal

Other Exceptions

Non-EVEX-encoded instruction, see Exceptions Type 2.
EVEX-encoded instruction, see Exceptions Type E2.

MINSD—Return Minimum Scalar Double-Precision Floating-Point Value

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
F2 0F 5D /r MINSD xmm1, xmm2/m64	A	V/V	SSE2	Return the minimum scalar double-precision floating-point value between xmm2/m64 and xmm1.
VEX.NDS.LIG.F2.0F.WIG 5D /r VMINSD xmm1, xmm2, xmm3/m64	B	V/V	AVX	Return the minimum scalar double-precision floating-point value between xmm3/m64 and xmm2.
EVEX.NDS.LIG.F2.0F.W1 5D /r VMINSD xmm1 {k1}{z}, xmm2, xmm3/m64{sae}	C	V/V	AVX512F	Return the minimum scalar double-precision floating-point value between xmm3/m64 and xmm2.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	NA	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
B	NA	ModRM:reg (w)	VEX.vvvv	ModRM:r/m (r)	NA
C	Tuple1 Scalar	ModRM:reg (w)	EVEX.vvvv	ModRM:r/m (r)	NA

Description

Compares the low double-precision floating-point values in the first source operand and the second source operand, and returns the minimum value to the low quadword of the destination operand. When the source operand is a memory operand, only the 64 bits are accessed.

If the values being compared are both 0.0s (of either sign), the value in the second source operand is returned. If a value in the second source operand is an SNaN, then SNaN is returned unchanged to the destination (that is, a QNaN version of the SNaN is not returned).

If only one value is a NaN (SNaN or QNaN) for this instruction, the second source operand, either a NaN or a valid floating-point value, is written to the result. If instead of this behavior, it is required that the NaN source operand (from either the first or second source) be returned, the action of MINSD can be emulated using a sequence of instructions, such as, a comparison followed by AND, ANDN and OR.

The second source operand can be an XMM register or a 64-bit memory location. The first source and destination operands are XMM registers.

128-bit Legacy SSE version: The destination and first source operand are the same. Bits (MAXVL-1:64) of the corresponding destination register remain unchanged.

VEX.128 and EVEX encoded version: Bits (127:64) of the XMM register destination are copied from corresponding bits in the first source operand. Bits (MAXVL-1:128) of the destination register are zeroed.

EVEX encoded version: The low quadword element of the destination operand is updated according to the writemask.

Software should ensure VMINSD is encoded with VEX.L=0. Encoding VMINSD with VEX.L=1 may encounter unpredictable behavior across different processor generations.

Operation

```
MIN(SRC1, SRC2)
{
    IF ((SRC1 = 0.0) and (SRC2 = 0.0)) THEN DEST ←SRC2;
        ELSE IF (SRC1 = SNaN) THEN DEST ←SRC2; FI;
        ELSE IF (SRC2 = SNaN) THEN DEST ←SRC2; FI;
        ELSE IF (SRC1 < SRC2) THEN DEST ←SRC1;
        ELSE DEST ←SRC2;
    FI;
}
```

MINSD (EVEX encoded version)

```
IF k1[0] or *no writemask*
    THEN      DEST[63:0] ← MIN(SRC1[63:0], SRC2[63:0])
    ELSE
        IF *merging-masking*                    ; merging-masking
            THEN *DEST[63:0] remains unchanged*
            ELSE                                ; zeroing-masking
                THEN DEST[63:0] ← 0
        FI;
FI;
DEST[127:64] ← SRC1[127:64]
DEST[MAXVL-1:128] ← 0
```

MINSD (VEX.128 encoded version)

```
DEST[63:0] ←MIN(SRC1[63:0], SRC2[63:0])
DEST[127:64] ←SRC1[127:64]
DEST[MAXVL-1:128] ←0
```

MINSD (128-bit Legacy SSE version)

```
DEST[63:0] ←MIN(SRC1[63:0], SRC2[63:0])
DEST[MAXVL-1:64] (Unmodified)
```

Intel C/C++ Compiler Intrinsic Equivalent

VMINSD __m128d _mm_min_round_sd(__m128d a, __m128d b, int);
VMINSD __m128d _mm_mask_min_round_sd(__m128d s, __mmask8 k, __m128d a, __m128d b, int);
VMINSD __m128d _mm_maskz_min_round_sd(__mmask8 k, __m128d a, __m128d b, int);
MINSD __m128d _mm_min_sd(__m128d a, __m128d b)

SIMD Floating-Point Exceptions

Invalid (including QNaN Source Operand), Denormal

Other Exceptions

Non-EVEX-encoded instruction, see Exceptions Type 3.
EVEX-encoded instruction, see Exceptions Type E3.

MINSS—Return Minimum Scalar Single-Precision Floating-Point Value

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
F3 0F 5D /r MINSS xmm1,xmm2/m32	A	V/V	SSE	Return the minimum scalar single-precision floating-point value between xmm2/m32 and xmm1.
VEX.NDS.LIG.F3.0F.WIG 5D /r VMINSS xmm1,xmm2, xmm3/m32	B	V/V	AVX	Return the minimum scalar single-precision floating-point value between xmm3/m32 and xmm2.
EVEX.NDS.LIG.F3.0F.W0 5D /r VMINSS xmm1 {k1}{z}, xmm2, xmm3/m32{sae}	C	V/V	AVX512F	Return the minimum scalar single-precision floating-point value between xmm3/m32 and xmm2.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	NA	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
B	NA	ModRM:reg (w)	VEX.vvvv	ModRM:r/m (r)	NA
C	Tuple1 Scalar	ModRM:reg (w)	EVEX.vvvv	ModRM:r/m (r)	NA

Description

Compares the low single-precision floating-point values in the first source operand and the second source operand and returns the minimum value to the low doubleword of the destination operand.

If the values being compared are both 0.0s (of either sign), the value in the second source operand is returned. If a value in the second operand is an SNaN, that SNaN is returned unchanged to the destination (that is, a QNaN version of the SNaN is not returned).

If only one value is a NaN (SNaN or QNaN) for this instruction, the second source operand, either a NaN or a valid floating-point value, is written to the result. If instead of this behavior, it is required that the NaN in either source operand be returned, the action of MINSD can be emulated using a sequence of instructions, such as, a comparison followed by AND, ANDN and OR.

The second source operand can be an XMM register or a 32-bit memory location. The first source and destination operands are XMM registers.

128-bit Legacy SSE version: The destination and first source operand are the same. Bits (MAXVL:32) of the corresponding destination register remain unchanged.

VEX.128 and EVEX encoded version: The first source operand is an xmm register encoded by (E)VEX.vvvv. Bits (127:32) of the XMM register destination are copied from corresponding bits in the first source operand. Bits (MAXVL-1:128) of the destination register are zeroed.

EVEX encoded version: The low doubleword element of the destination operand is updated according to the writemask.

Software should ensure VMINSS is encoded with VEX.L=0. Encoding VMINSS with VEX.L=1 may encounter unpredictable behavior across different processor generations.

Operation

```
MIN(SRC1, SRC2)
{
    IF ((SRC1 = 0.0) and (SRC2 = 0.0)) THEN DEST ←SRC2;
        ELSE IF (SRC1 = SNaN) THEN DEST ←SRC2; FI;
        ELSE IF (SRC2 = SNaN) THEN DEST ←SRC2; FI;
        ELSE IF (SRC1 < SRC2) THEN DEST ←SRC1;
        ELSE DEST ←SRC2;
    FI;
}
```

MINSS (EVEX encoded version)

```
IF k1[0] or *no writemask*
    THEN      DEST[31:0] ← MIN(SRC1[31:0], SRC2[31:0])
    ELSE
        IF *merging-masking*                    ; merging-masking
            THEN *DEST[31:0] remains unchanged*
            ELSE                                ; zeroing-masking
                THEN DEST[31:0] ← 0
        FI;
FI;
DEST[127:32] ← SRC1[127:32]
DEST[MAXVL-1:128] ← 0
```

VMINSS (VEX.128 encoded version)

```
DEST[31:0] ←MIN(SRC1[31:0], SRC2[31:0])
DEST[127:32] ←SRC1[127:32]
DEST[MAXVL-1:128] ←0
```

MINSS (128-bit Legacy SSE version)

```
DEST[31:0] ←MIN(SRC1[31:0], SRC2[31:0])
DEST[MAXVL-1:128] (Unmodified)
```

Intel C/C++ Compiler Intrinsic Equivalent

VMINSS __m128 _mm_min_round_ss(__m128 a, __m128 b, int);
VMINSS __m128 _mm_mask_min_round_ss(__m128 s, __mmask8 k, __m128 a, __m128 b, int);
VMINSS __m128 _mm_maskz_min_round_ss(__mmask8 k, __m128 a, __m128 b, int);
MINSS __m128 _mm_min_ss(__m128 a, __m128 b)

SIMD Floating-Point Exceptions

Invalid (Including QNaN Source Operand), Denormal

Other Exceptions

Non-EVEX-encoded instruction, see Exceptions Type 2.
EVEX-encoded instruction, see Exceptions Type E2.

MONITOR—Set Up Monitor Address

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
0F 01 C8	MONITOR	ZO	Valid	Valid	Sets up a linear address range to be monitored by hardware and activates the monitor. The address range should be a write-back memory caching type. The address is DS:RAX/EAX/AX.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
ZO	NA	NA	NA	NA

Description

The MONITOR instruction arms address monitoring hardware using an address specified in EAX (the address range that the monitoring hardware checks for store operations can be determined by using CPUID). A store to an address within the specified address range triggers the monitoring hardware. The state of monitor hardware is used by MWAIT.

The address is specified in RAX/EAX/AX and the size is based on the effective address size of the encoded instruction. By default, the DS segment is used to create a linear address that is monitored. Segment overrides can be used.

ECX and EDX are also used. They communicate other information to MONITOR. ECX specifies optional extensions. EDX specifies optional hints; it does not change the architectural behavior of the instruction. For the Pentium 4 processor (family 15, model 3), no extensions or hints are defined. Undefined hints in EDX are ignored by the processor; undefined extensions in ECX raises a general protection fault.

The address range must use memory of the write-back type. Only write-back memory will correctly trigger the monitoring hardware. Additional information on determining what address range to use in order to prevent false wake-ups is described in Chapter 8, "Multiple-Processor Management" of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*.

The MONITOR instruction is ordered as a load operation with respect to other memory transactions. The instruction is subject to the permission checking and faults associated with a byte load. Like a load, MONITOR sets the A-bit but not the D-bit in page tables.

CPUID.01H:ECX.MONITOR[bit 3] indicates the availability of MONITOR and MWAIT in the processor. When set, MONITOR may be executed only at privilege level 0 (use at any other privilege level results in an invalid-opcode exception). The operating system or system BIOS may disable this instruction by using the IA32_MISC_ENABLE MSR; disabling MONITOR clears the CPUID feature flag and causes execution to generate an invalid-opcode exception.

The instruction's operation is the same in non-64-bit modes and 64-bit mode.

Operation

MONITOR sets up an address range for the monitor hardware using the content of EAX (RAX in 64-bit mode) as an effective address and puts the monitor hardware in armed state. Always use memory of the write-back caching type. A store to the specified address range will trigger the monitor hardware. The content of ECX and EDX are used to communicate other information to the monitor hardware.

Intel C/C++ Compiler Intrinsic Equivalent

MONITOR: void _mm_monitor(void const *p, unsigned extensions,unsigned hints)

Numeric Exceptions

None

Protected Mode Exceptions

#GP(0)	If the value in EAX is outside the CS, DS, ES, FS, or GS segment limit.
	If the DS, ES, FS, or GS register is used to access memory and it contains a NULL segment selector.
	If ECX ≠ 0.
#SS(0)	If the value in EAX is outside the SS segment limit.
#PF(fault-code)	For a page fault.
#UD	If CPUID.01H:ECX.MONITOR[bit 3] = 0.
	If current privilege level is not 0.

Real Address Mode Exceptions

#GP	If the CS, DS, ES, FS, or GS register is used to access memory and the value in EAX is outside of the effective address space from 0 to FFFFH.
	If ECX ≠ 0.
#SS	If the SS register is used to access memory and the value in EAX is outside of the effective address space from 0 to FFFFH.
#UD	If CPUID.01H:ECX.MONITOR[bit 3] = 0.

Virtual 8086 Mode Exceptions

#UD	The MONITOR instruction is not recognized in virtual-8086 mode (even if CPUID.01H:ECX.MONITOR[bit 3] = 1).

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#GP(0)	If the linear address of the operand in the CS, DS, ES, FS, or GS segment is in a non-canonical form.
	If RCX ≠ 0.
#SS(0)	If the SS register is used to access memory and the value in EAX is in a non-canonical form.
#PF(fault-code)	For a page fault.
#UD	If the current privilege level is not 0.
	If CPUID.01H:ECX.MONITOR[bit 3] = 0.

MOV—Move

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
88 /r	MOV *r/m8,r8*	MR	Valid	Valid	Move *r8* to *r/m8.*
REX + 88 /r	MOV *r/m8***,r8****	MR	Valid	N.E.	Move *r8* to *r/m8.*
89 /r	MOV *r/m16,r16*	MR	Valid	Valid	Move *r16* to *r/m16.*
89 /r	MOV *r/m32,r32*	MR	Valid	Valid	Move *r32* to *r/m32.*
REX.W + 89 /r	MOV *r/m64,r64*	MR	Valid	N.E.	Move *r64* to *r/m64.*
8A /r	MOV *r8,r/m8*	RM	Valid	Valid	Move *r/m8* to *r8.*
REX + 8A /r	MOV *r8***,r/m8****	RM	Valid	N.E.	Move *r/m8* to *r8.*
8B /r	MOV *r16,r/m16*	RM	Valid	Valid	Move *r/m16* to *r16.*
8B /r	MOV *r32,r/m32*	RM	Valid	Valid	Move *r/m32* to *r32.*
REX.W + 8B /r	MOV *r64,r/m64*	RM	Valid	N.E.	Move *r/m64* to *r64.*
8C /r	MOV *r/m16,Sreg***	MR	Valid	Valid	Move segment register to *r/m16.*
REX.W + 8C /r	MOV *r/m64,Sreg***	MR	Valid	Valid	Move zero extended 16-bit segment register to *r/m64.*
8E /r	MOV *Sreg,r/m16***	RM	Valid	Valid	Move *r/m16* to segment register.
REX.W + 8E /r	MOV *Sreg,r/m64***	RM	Valid	Valid	Move *lower 16 bits of r/m64* to segment register.
A0	MOV AL,*moffs8**	FD	Valid	Valid	Move byte at (*seg:offset*) to AL.
REX.W + A0	MOV AL,*moffs8**	FD	Valid	N.E.	Move byte at (*offset*) to AL.
A1	MOV AX,*moffs16**	FD	Valid	Valid	Move word at (*seg:offset*) to AX.
A1	MOV EAX,*moffs32**	FD	Valid	Valid	Move doubleword at (*seg:offset*) to EAX.
REX.W + A1	MOV RAX,*moffs64**	FD	Valid	N.E.	Move quadword at (*offset*) to RAX.
A2	MOV *moffs8*,AL	TD	Valid	Valid	Move AL to (*seg:offset*).
REX.W + A2	MOV *moffs8****,AL	TD	Valid	N.E.	Move AL to (*offset*).
A3	MOV *moffs16**,AX	TD	Valid	Valid	Move AX to (*seg:offset*).
A3	MOV *moffs32**,EAX	TD	Valid	Valid	Move EAX to (*seg:offset*).
REX.W + A3	MOV *moffs64**,RAX	TD	Valid	N.E.	Move RAX to (*offset*).
B0+ *rb ib*	MOV *r8, imm8*	OI	Valid	Valid	Move *imm8* to *r8.*
REX + B0+ *rb ib*	MOV *r8***, imm8*	OI	Valid	N.E.	Move *imm8* to *r8.*
B8+ *rw iw*	MOV *r16, imm16*	OI	Valid	Valid	Move *imm16* to *r16.*
B8+ *rd id*	MOV *r32, imm32*	OI	Valid	Valid	Move *imm32* to *r32.*
REX.W + B8+ *rd io*	MOV *r64, imm64*	OI	Valid	N.E.	Move *imm64* to *r64.*
C6 */0 ib*	MOV *r/m8, imm8*	MI	Valid	Valid	Move *imm8* to *r/m8.*
REX + C6 */0 ib*	MOV *r/m8***, imm8*	MI	Valid	N.E.	Move *imm8* to *r/m8.*
C7 */0 iw*	MOV *r/m16, imm16*	MI	Valid	Valid	Move *imm16* to *r/m16.*
C7 */0 id*	MOV *r/m32, imm32*	MI	Valid	Valid	Move *imm32* to *r/m32.*
REX.W + C7 */0 id*	MOV *r/m64, imm32*	MI	Valid	N.E.	Move *imm32 sign extended to 64-bits* to *r/m64.*

NOTES:

* The *moffs8*, *moffs16*, *moffs32* and *moffs64* operands specify a simple offset relative to the segment base, where 8, 16, 32 and 64 refer to the size of the data. The address-size attribute of the instruction determines the size of the offset, either 16, 32 or 64 bits.

** In 32-bit mode, the assembler may insert the 16-bit operand-size prefix with this instruction (see the following "Description" section for further information).

***In 64-bit mode, *r/m8* can not be encoded to access the following byte registers if a REX prefix is used: AH, BH, CH, DH.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
MR	ModRM:r/m (w)	ModRM:reg (r)	NA	NA
RM	ModRM:reg (w)	ModRM:r/m (r)	NA	NA
FD	AL/AX/EAX/RAX	Moffs	NA	NA
TD	Moffs (w)	AL/AX/EAX/RAX	NA	NA
OI	opcode + rd (w)	imm8/16/32/64	NA	NA
MI	ModRM:r/m (w)	imm8/16/32/64	NA	NA

Description

Copies the second operand (source operand) to the first operand (destination operand). The source operand can be an immediate value, general-purpose register, segment register, or memory location; the destination register can be a general-purpose register, segment register, or memory location. Both operands must be the same size, which can be a byte, a word, a doubleword, or a quadword.

The MOV instruction cannot be used to load the CS register. Attempting to do so results in an invalid opcode exception (#UD). To load the CS register, use the far JMP, CALL, or RET instruction.

If the destination operand is a segment register (DS, ES, FS, GS, or SS), the source operand must be a valid segment selector. In protected mode, moving a segment selector into a segment register automatically causes the segment descriptor information associated with that segment selector to be loaded into the hidden (shadow) part of the segment register. While loading this information, the segment selector and segment descriptor information is validated (see the "Operation" algorithm below). The segment descriptor data is obtained from the GDT or LDT entry for the specified segment selector.

A NULL segment selector (values 0000-0003) can be loaded into the DS, ES, FS, and GS registers without causing a protection exception. However, any subsequent attempt to reference a segment whose corresponding segment register is loaded with a NULL value causes a general protection exception (#GP) and no memory reference occurs.

Loading the SS register with a MOV instruction inhibits all interrupts until after the execution of the next instruction. This operation allows a stack pointer to be loaded into the ESP register with the next instruction (MOV ESP, **stack-pointer value**) before an interrupt occurs[1]. Be aware that the LSS instruction offers a more efficient method of loading the SS and ESP registers.

When executing MOV Reg, Sreg, the processor copies the content of Sreg to the 16 least significant bits of the general-purpose register. The upper bits of the destination register are zero for most IA-32 processors (Pentium

1. If a code instruction breakpoint (for debug) is placed on an instruction located immediately after a MOV SS instruction, the breakpoint may not be triggered. However, in a sequence of instructions that load the SS register, only the first instruction in the sequence is guaranteed to delay an interrupt.

   In the following sequence, interrupts may be recognized before MOV ESP, EBP executes:

   MOV SS, EDX
   MOV SS, EAX
   MOV ESP, EBP

Pro processors and later) and all Intel 64 processors, with the exception that bits 31:16 are undefined for Intel Quark X1000 processors, Pentium and earlier processors.

In 64-bit mode, the instruction's default operation size is 32 bits. Use of the REX.R prefix permits access to additional registers (R8-R15). Use of the REX.W prefix promotes operation to 64 bits. See the summary chart at the beginning of this section for encoding data and limits.

Operation

```
DEST ← SRC;
```

Loading a segment register while in protected mode results in special checks and actions, as described in the following listing. These checks are performed on the segment selector and the segment descriptor to which it points.

```
IF SS is loaded
    THEN
        IF segment selector is NULL
            THEN #GP(0); FI;
        IF segment selector index is outside descriptor table limits
        or segment selector's RPL ≠ CPL
        or segment is not a writable data segment
        or DPL ≠ CPL
            THEN #GP(selector); FI;
        IF segment not marked present
            THEN #SS(selector);
            ELSE
                SS ← segment selector;
                SS ← segment descriptor; FI;
FI;

IF DS, ES, FS, or GS is loaded with non-NULL selector
THEN
    IF segment selector index is outside descriptor table limits
    or segment is not a data or readable code segment
    or ((segment is a data or nonconforming code segment)
    or ((RPL > DPL) and (CPL > DPL))
        THEN #GP(selector); FI;
    IF segment not marked present
        THEN #NP(selector);
        ELSE
            SegmentRegister ← segment selector;
            SegmentRegister ← segment descriptor; FI;
FI;

IF DS, ES, FS, or GS is loaded with NULL selector
    THEN
        SegmentRegister ← segment selector;
        SegmentRegister ← segment descriptor;
FI;
```

Flags Affected

None

Protected Mode Exceptions

#GP(0)	If attempt is made to load SS register with NULL segment selector.
	If the destination operand is in a non-writable segment.
	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
	If the DS, ES, FS, or GS register contains a NULL segment selector.
#GP(selector)	If segment selector index is outside descriptor table limits.
	If the SS register is being loaded and the segment selector's RPL and the segment descriptor's DPL are not equal to the CPL.
	If the SS register is being loaded and the segment pointed to is a non-writable data segment.
	If the DS, ES, FS, or GS register is being loaded and the segment pointed to is not a data or readable code segment.
	If the DS, ES, FS, or GS register is being loaded and the segment pointed to is a data or nonconforming code segment, but both the RPL and the CPL are greater than the DPL.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#SS(selector)	If the SS register is being loaded and the segment pointed to is marked not present.
#NP	If the DS, ES, FS, or GS register is being loaded and the segment pointed to is marked not present.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If attempt is made to load the CS register.
	If the LOCK prefix is used.

Real-Address Mode Exceptions

#GP	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS	If a memory operand effective address is outside the SS segment limit.
#UD	If attempt is made to load the CS register.
	If the LOCK prefix is used.

Virtual-8086 Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.
#UD	If attempt is made to load the CS register.
	If the LOCK prefix is used.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#GP(0)	If the memory address is in a non-canonical form.
	If an attempt is made to load SS register with NULL segment selector when CPL = 3.
	If an attempt is made to load SS register with NULL segment selector when CPL < 3 and CPL ≠ RPL.
#GP(selector)	If segment selector index is outside descriptor table limits.
	If the memory access to the descriptor table is non-canonical.
	If the SS register is being loaded and the segment selector's RPL and the segment descriptor’s DPL are not equal to the CPL.
	If the SS register is being loaded and the segment pointed to is a nonwritable data segment.
	If the DS, ES, FS, or GS register is being loaded and the segment pointed to is not a data or readable code segment.
	If the DS, ES, FS, or GS register is being loaded and the segment pointed to is a data or nonconforming code segment, but both the RPL and the CPL are greater than the DPL.
#SS(0)	If the stack address is in a non-canonical form.
#SS(selector)	If the SS register is being loaded and the segment pointed to is marked not present.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If attempt is made to load the CS register.
	If the LOCK prefix is used.

MOV—Move to/from Control Registers

Opcode/ Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
0F 20/*r* MOV *r32*, CR0–CR7	MR	N.E.	Valid	Move control register to *r32*.
0F 20/*r* MOV *r64*, CR0–CR7	MR	Valid	N.E.	Move extended control register to *r64*.
REX.R + 0F 20 /0 MOV *r64*, CR8	MR	Valid	N.E.	Move extended CR8 to *r64*.[1]
0F 22 /*r* MOV CR0–CR7, *r32*	RM	N.E.	Valid	Move *r32* to control register.
0F 22 /*r* MOV CR0–CR7, *r64*	RM	Valid	N.E.	Move *r64* to extended control register.
REX.R + 0F 22 /0 MOV CR8, *r64*	RM	Valid	N.E.	Move *r64* to extended CR8.[1]

NOTE:

1. MOV CR* instructions, except for MOV CR8, are serializing instructions. MOV CR8 is not architecturally defined as a serializing instruction. For more information, see Chapter 8 in *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
MR	ModRM:r/m (w)	ModRM:reg (r)	NA	NA
RM	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

Description

Moves the contents of a control register (CR0, CR2, CR3, CR4, or CR8) to a general-purpose register or the contents of a general purpose register to a control register. The operand size for these instructions is always 32 bits in non-64-bit modes, regardless of the operand-size attribute. (See "Control Registers" in Chapter 2 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*, for a detailed description of the flags and fields in the control registers.) This instruction can be executed only when the current privilege level is 0.

At the opcode level, the *reg* field within the ModR/M byte specifies which of the control registers is loaded or read. The 2 bits in the *mod* field are ignored. The *r/m* field specifies the general-purpose register loaded or read. Attempts to reference CR1, CR5, CR6, CR7, and CR9–CR15 result in undefined opcode (#UD) exceptions.

When loading control registers, programs should not attempt to change the reserved bits; that is, always set reserved bits to the value previously read. An attempt to change CR4's reserved bits will cause a general protection fault. Reserved bits in CR0 and CR3 remain clear after any load of those registers; attempts to set them have no impact. On Pentium 4, Intel Xeon and P6 family processors, CR0.ET remains set after any load of CR0; attempts to clear this bit have no impact.

In certain cases, these instructions have the side effect of invalidating entries in the TLBs and the paging-structure caches. See Section 4.10.4.1, "Operations that Invalidate TLBs and Paging-Structure Caches," in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A* for details.

The following side effects are implementation-specific for the Pentium 4, Intel Xeon, and P6 processor family: when modifying PE or PG in register CR0, or PSE or PAE in register CR4, all TLB entries are flushed, including global entries. Software should not depend on this functionality in all Intel 64 or IA-32 processors.

In 64-bit mode, the instruction's default operation size is 64 bits. The REX.R prefix must be used to access CR8. Use of REX.B permits access to additional registers (R8-R15). Use of the REX.W prefix or 66H prefix is ignored. Use of

the REX.R prefix to specify a register other than CR8 causes an invalid-opcode exception. See the summary chart at the beginning of this section for encoding data and limits.

If CR4.PCIDE = 1, bit 63 of the source operand to MOV to CR3 determines whether the instruction invalidates entries in the TLBs and the paging-structure caches (see Section 4.10.4.1, "Operations that Invalidate TLBs and Paging-Structure Caches," in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*). The instruction does not modify bit 63 of CR3, which is reserved and always 0.

See "Changes to Instruction Behavior in VMX Non-Root Operation" in Chapter 25 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3C*, for more information about the behavior of this instruction in VMX non-root operation.

Operation

```
DEST ← SRC;
```

Flags Affected

The OF, SF, ZF, AF, PF, and CF flags are undefined.

Protected Mode Exceptions

#GP(0)	If the current privilege level is not 0.
	If an attempt is made to write invalid bit combinations in CR0 (such as setting the PG flag to 1 when the PE flag is set to 0, or setting the CD flag to 0 when the NW flag is set to 1).
	If an attempt is made to write a 1 to any reserved bit in CR4.
	If an attempt is made to write 1 to CR4.PCIDE.
	If any of the reserved bits are set in the page-directory pointers table (PDPT) and the loading of a control register causes the PDPT to be loaded into the processor.
#UD	If the LOCK prefix is used.
	If an attempt is made to access CR1, CR5, CR6, or CR7.

Real-Address Mode Exceptions

#GP	If an attempt is made to write a 1 to any reserved bit in CR4.
	If an attempt is made to write 1 to CR4.PCIDE.
	If an attempt is made to write invalid bit combinations in CR0 (such as setting the PG flag to 1 when the PE flag is set to 0).
#UD	If the LOCK prefix is used.
	If an attempt is made to access CR1, CR5, CR6, or CR7.

Virtual-8086 Mode Exceptions

#GP(0)	These instructions cannot be executed in virtual-8086 mode.

Compatibility Mode Exceptions

#GP(0)	If the current privilege level is not 0.
	If an attempt is made to write invalid bit combinations in CR0 (such as setting the PG flag to 1 when the PE flag is set to 0, or setting the CD flag to 0 when the NW flag is set to 1).
	If an attempt is made to change CR4.PCIDE from 0 to 1 while CR3[11:0] ≠ 000H.
	If an attempt is made to clear CR0.PG[bit 31] while CR4.PCIDE = 1.
	If an attempt is made to write a 1 to any reserved bit in CR3.
	If an attempt is made to leave IA-32e mode by clearing CR4.PAE[bit 5].
#UD	If the LOCK prefix is used.
	If an attempt is made to access CR1, CR5, CR6, or CR7.

64-Bit Mode Exceptions

#GP(0)	If the current privilege level is not 0.
	If an attempt is made to write invalid bit combinations in CR0 (such as setting the PG flag to 1 when the PE flag is set to 0, or setting the CD flag to 0 when the NW flag is set to 1).
	If an attempt is made to change CR4.PCIDE from 0 to 1 while CR3[11:0] ≠ 000H.
	If an attempt is made to clear CR0.PG[bit 31].
	If an attempt is made to write a 1 to any reserved bit in CR4.
	If an attempt is made to write a 1 to any reserved bit in CR8.
	If an attempt is made to write a 1 to any reserved bit in CR3.
	If an attempt is made to leave IA-32e mode by clearing CR4.PAE[bit 5].
#UD	If the LOCK prefix is used.
	If an attempt is made to access CR1, CR5, CR6, or CR7.
	If the REX.R prefix is used to specify a register other than CR8.

MOV—Move to/from Debug Registers

Opcode/ Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
0F 21/*r* MOV *r32*, DR0–DR7	MR	N.E.	Valid	Move debug register to *r32*.
0F 21/*r* MOV *r64*, DR0–DR7	MR	Valid	N.E.	Move extended debug register to *r64*.
0F 23 /*r* MOV DR0–DR7, *r32*	RM	N.E.	Valid	Move *r32* to debug register.
0F 23 /*r* MOV DR0–DR7, *r64*	RM	Valid	N.E.	Move *r64* to extended debug register.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
MR	ModRM:r/m (w)	ModRM:reg (r)	NA	NA
RM	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

Description

Moves the contents of a debug register (DR0, DR1, DR2, DR3, DR4, DR5, DR6, or DR7) to a general-purpose register or vice versa. The operand size for these instructions is always 32 bits in non-64-bit modes, regardless of the operand-size attribute. (See Section 17.2, "Debug Registers", of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*, for a detailed description of the flags and fields in the debug registers.)

The instructions must be executed at privilege level 0 or in real-address mode.

When the debug extension (DE) flag in register CR4 is clear, these instructions operate on debug registers in a manner that is compatible with Intel386 and Intel486 processors. In this mode, references to DR4 and DR5 refer to DR6 and DR7, respectively. When the DE flag in CR4 is set, attempts to reference DR4 and DR5 result in an undefined opcode (#UD) exception. (The CR4 register was added to the IA-32 Architecture beginning with the Pentium processor.)

At the opcode level, the *reg* field within the ModR/M byte specifies which of the debug registers is loaded or read. The two bits in the *mod* field are ignored. The *r/m* field specifies the general-purpose register loaded or read.

In 64-bit mode, the instruction's default operation size is 64 bits. Use of the REX.B prefix permits access to additional registers (R8–R15). Use of the REX.W or 66H prefix is ignored. Use of the REX.R prefix causes an invalid-opcode exception. See the summary chart at the beginning of this section for encoding data and limits.

Operation

```
IF ((DE = 1) and (SRC or DEST = DR4 or DR5))
    THEN
        #UD;
    ELSE
        DEST ← SRC;

FI;
```

Flags Affected

The OF, SF, ZF, AF, PF, and CF flags are undefined.

Protected Mode Exceptions

#GP(0)	If the current privilege level is not 0.
#UD	If CR4.DE[bit 3] = 1 (debug extensions) and a MOV instruction is executed involving DR4 or DR5.
	If the LOCK prefix is used.
#DB	If any debug register is accessed while the DR7.GD[bit 13] = 1.

Real-Address Mode Exceptions

#UD	If CR4.DE[bit 3] = 1 (debug extensions) and a MOV instruction is executed involving DR4 or DR5.
	If the LOCK prefix is used.
#DB	If any debug register is accessed while the DR7.GD[bit 13] = 1.

Virtual-8086 Mode Exceptions

#GP(0)	The debug registers cannot be loaded or read when in virtual-8086 mode.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#GP(0)	If the current privilege level is not 0.
	If an attempt is made to write a 1 to any of bits 63:32 in DR6.
	If an attempt is made to write a 1 to any of bits 63:32 in DR7.
#UD	If CR4.DE[bit 3] = 1 (debug extensions) and a MOV instruction is executed involving DR4 or DR5.
	If the LOCK prefix is used.
	If the REX.R prefix is used.
#DB	If any debug register is accessed while the DR7.GD[bit 13] = 1.

MOVAPD—Move Aligned Packed Double-Precision Floating-Point Values

Opcode/ Instruction	Op/En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 28 /r MOVAPD xmm1, xmm2/m128	A	V/V	SSE2	Move aligned packed double-precision floating-point values from xmm2/mem to xmm1.
66 0F 29 /r MOVAPD xmm2/m128, xmm1	B	V/V	SSE2	Move aligned packed double-precision floating-point values from xmm1 to xmm2/mem.
VEX.128.66.0F.WIG 28 /r VMOVAPD xmm1, xmm2/m128	A	V/V	AVX	Move aligned packed double-precision floating-point values from xmm2/mem to xmm1.
VEX.128.66.0F.WIG 29 /r VMOVAPD xmm2/m128, xmm1	B	V/V	AVX	Move aligned packed double-precision floating-point values from xmm1 to xmm2/mem.
VEX.256.66.0F.WIG 28 /r VMOVAPD ymm1, ymm2/m256	A	V/V	AVX	Move aligned packed double-precision floating-point values from ymm2/mem to ymm1.
VEX.256.66.0F.WIG 29 /r VMOVAPD ymm2/m256, ymm1	B	V/V	AVX	Move aligned packed double-precision floating-point values from ymm1 to ymm2/mem.
EVEX.128.66.0F.W1 28 /r VMOVAPD xmm1 {k1}{z}, xmm2/m128	C	V/V	AVX512VL AVX512F	Move aligned packed double-precision floating-point values from xmm2/m128 to xmm1 using writemask k1.
EVEX.256.66.0F.W1 28 /r VMOVAPD ymm1 {k1}{z}, ymm2/m256	C	V/V	AVX512VL AVX512F	Move aligned packed double-precision floating-point values from ymm2/m256 to ymm1 using writemask k1.
EVEX.512.66.0F.W1 28 /r VMOVAPD zmm1 {k1}{z}, zmm2/m512	C	V/V	AVX512F	Move aligned packed double-precision floating-point values from zmm2/m512 to zmm1 using writemask k1.
EVEX.128.66.0F.W1 29 /r VMOVAPD xmm2/m128 {k1}{z}, xmm1	D	V/V	AVX512VL AVX512F	Move aligned packed double-precision floating-point values from xmm1 to xmm2/m128 using writemask k1.
EVEX.256.66.0F.W1 29 /r VMOVAPD ymm2/m256 {k1}{z}, ymm1	D	V/V	AVX512VL AVX512F	Move aligned packed double-precision floating-point values from ymm1 to ymm2/m256 using writemask k1.
EVEX.512.66.0F.W1 29 /r VMOVAPD zmm2/m512 {k1}{z}, zmm1	D	V/V	AVX512F	Move aligned packed double-precision floating-point values from zmm1 to zmm2/m512 using writemask k1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	NA	ModRM:reg (w)	ModRM:r/m (r)	NA	NA
B	NA	ModRM:r/m (w)	ModRM:reg (r)	NA	NA
C	Full Vector Mem	ModRM:reg (w)	ModRM:r/m (r)	NA	NA
D	Full Vector Mem	ModRM:r/m (w)	ModRM:reg (r)	NA	NA

Description

Moves 2, 4 or 8 double-precision floating-point values from the source operand (second operand) to the destination operand (first operand). This instruction can be used to load an XMM, YMM or ZMM register from an 128-bit, 256-bit or 512-bit memory location, to store the contents of an XMM, YMM or ZMM register into a 128-bit, 256-bit or 512-bit memory location, or to move data between two XMM, two YMM or two ZMM registers.

When the source or destination operand is a memory operand, the operand must be aligned on a 16-byte (128-bit versions), 32-byte (256-bit version) or 64-byte (EVEX.512 encoded version) boundary or a general-protection exception (#GP) will be generated. For EVEX encoded versions, the operand must be aligned to the size of the memory operand. To move double-precision floating-point values to and from unaligned memory locations, use the VMOVUPD instruction.

Note: VEX.vvvv and EVEX.vvvv are reserved and must be 1111b otherwise instructions will #UD.

EVEX.512 encoded version:

Moves 512 bits of packed double-precision floating-point values from the source operand (second operand) to the destination operand (first operand). This instruction can be used to load a ZMM register from a 512-bit float64 memory location, to store the contents of a ZMM register into a 512-bit float64 memory location, or to move data between two ZMM registers. When the source or destination operand is a memory operand, the operand must be aligned on a 64-byte boundary or a general-protection exception (#GP) will be generated. To move single-precision floating-point values to and from unaligned memory locations, use the VMOVUPD instruction.

VEX.256 and EVEX.256 encoded versions:

Moves 256 bits of packed double-precision floating-point values from the source operand (second operand) to the destination operand (first operand). This instruction can be used to load a YMM register from a 256-bit memory location, to store the contents of a YMM register into a 256-bit memory location, or to move data between two YMM registers. When the source or destination operand is a memory operand, the operand must be aligned on a 32-byte boundary or a general-protection exception (#GP) will be generated. To move double-precision floating-point values to and from unaligned memory locations, use the VMOVUPD instruction.

128-bit versions:

Moves 128 bits of packed double-precision floating-point values from the source operand (second operand) to the destination operand (first operand). This instruction can be used to load an XMM register from a 128-bit memory location, to store the contents of an XMM register into a 128-bit memory location, or to move data between two XMM registers. When the source or destination operand is a memory operand, the operand must be aligned on a 16-byte boundary or a general-protection exception (#GP) will be generated. To move single-precision floating-point values to and from unaligned memory locations, use the VMOVUPD instruction.

128-bit Legacy SSE version: Bits (MAXVL-1:128) of the corresponding ZMM destination register remain unchanged.

(E)VEX.128 encoded version: Bits (MAXVL-1:128) of the destination ZMM register destination are zeroed.

Operation

VMOVAPD (EVEX encoded versions, register-copy form)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j ← 0 TO KL-1
    i ← j * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+63:i] ← SRC[i+63:i]
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE  DEST[i+63:i] ← 0              ; zeroing-masking
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0
```

VMOVAPD (EVEX encoded versions, store-form)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j ← 0 TO KL-1
    i ← j * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+63:i]← SRC[i+63:i]
        ELSE
        ELSE *DEST[i+63:i] remains unchanged*          ; merging-masking

    FI;
ENDFOR;
```

VMOVAPD (EVEX encoded versions, load-form)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j ← 0 TO KL-1
    i ← j * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+63:i] ← SRC[i+63:i]
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE  DEST[i+63:i] ← 0              ; zeroing-masking
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0
```

VMOVAPD (VEX.256 encoded version, load - and register copy)

```
DEST[255:0] ← SRC[255:0]
DEST[MAXVL-1:256] ← 0
```

VMOVAPD (VEX.256 encoded version, store-form)

```
DEST[255:0] ← SRC[255:0]
```

VMOVAPD (VEX.128 encoded version, load - and register copy)

```
DEST[127:0] ← SRC[127:0]
DEST[MAXVL-1:128] ← 0
```

MOVAPD (128-bit load- and register-copy- form Legacy SSE version)

```
DEST[127:0] ← SRC[127:0]
DEST[MAXVL-1:128] (Unmodified)
```

(V)MOVAPD (128-bit store-form version)

```
DEST[127:0] ← SRC[127:0]
```

Intel C/C++ Compiler Intrinsic Equivalent

VMOVAPD __m512d _mm512_load_pd(void * m);
VMOVAPD __m512d _mm512_mask_load_pd(__m512d s, __mmask8 k, void * m);
VMOVAPD __m512d _mm512_maskz_load_pd(__mmask8 k, void * m);
VMOVAPD void _mm512_store_pd(void * d, __m512d a);
VMOVAPD void _mm512_mask_store_pd(void * d, __mmask8 k, __m512d a);
VMOVAPD __m256d _mm256_mask_load_pd(__m256d s, __mmask8 k, void * m);
VMOVAPD __m256d _mm256_maskz_load_pd(__mmask8 k, void * m);
VMOVAPD void _mm256_mask_store_pd(void * d, __mmask8 k, __m256d a);
VMOVAPD __m128d _mm_mask_load_pd(__m128d s, __mmask8 k, void * m);
VMOVAPD __m128d _mm_maskz_load_pd(__mmask8 k, void * m);
VMOVAPD void _mm_mask_store_pd(void * d, __mmask8 k, __m128d a);
MOVAPD __m256d _mm256_load_pd (double * p);
MOVAPD void _mm256_store_pd(double * p, __m256d a);
MOVAPD __m128d _mm_load_pd (double * p);
MOVAPD void _mm_store_pd(double * p, __m128d a);

SIMD Floating-Point Exceptions

None

Other Exceptions

Non-EVEX-encoded instruction, see Exceptions Type1.SSE2;
EVEX-encoded instruction, see Exceptions Type E1.

#UD	If EVEX.vvvv != 1111B or VEX.vvvv != 1111B.

MOVAPS—Move Aligned Packed Single-Precision Floating-Point Values

Opcode/ Instruction	Op/En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F 28 /r MOVAPS xmm1, xmm2/m128	A	V/V	SSE	Move aligned packed single-precision floating-point values from xmm2/mem to xmm1.
NP 0F 29 /r MOVAPS xmm2/m128, xmm1	B	V/V	SSE	Move aligned packed single-precision floating-point values from xmm1 to xmm2/mem.
VEX.128.0F.WIG 28 /r VMOVAPS xmm1, xmm2/m128	A	V/V	AVX	Move aligned packed single-precision floating-point values from xmm2/mem to xmm1.
VEX.128.0F.WIG 29 /r VMOVAPS xmm2/m128, xmm1	B	V/V	AVX	Move aligned packed single-precision floating-point values from xmm1 to xmm2/mem.
VEX.256.0F.WIG 28 /r VMOVAPS ymm1, ymm2/m256	A	V/V	AVX	Move aligned packed single-precision floating-point values from ymm2/mem to ymm1.
VEX.256.0F.WIG 29 /r VMOVAPS ymm2/m256, ymm1	B	V/V	AVX	Move aligned packed single-precision floating-point values from ymm1 to ymm2/mem.
EVEX.128.0F.W0 28 /r VMOVAPS xmm1 {k1}{z}, xmm2/m128	C	V/V	AVX512VL AVX512F	Move aligned packed single-precision floating-point values from xmm2/m128 to xmm1 using writemask k1.
EVEX.256.0F.W0 28 /r VMOVAPS ymm1 {k1}{z}, ymm2/m256	C	V/V	AVX512VL AVX512F	Move aligned packed single-precision floating-point values from ymm2/m256 to ymm1 using writemask k1.
EVEX.512.0F.W0 28 /r VMOVAPS zmm1 {k1}{z}, zmm2/m512	C	V/V	AVX512F	Move aligned packed single-precision floating-point values from zmm2/m512 to zmm1 using writemask k1.
EVEX.128.0F.W0 29 /r VMOVAPS xmm2/m128 {k1}{z}, xmm1	D	V/V	AVX512VL AVX512F	Move aligned packed single-precision floating-point values from xmm1 to xmm2/m128 using writemask k1.
EVEX.256.0F.W0 29 /r VMOVAPS ymm2/m256 {k1}{z}, ymm1	D	V/V	AVX512VL AVX512F	Move aligned packed single-precision floating-point values from ymm1 to ymm2/m256 using writemask k1.
EVEX.512.0F.W0 29 /r VMOVAPS zmm2/m512 {k1}{z}, zmm1	D	V/V	AVX512F	Move aligned packed single-precision floating-point values from zmm1 to zmm2/m512 using writemask k1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	NA	ModRM:reg (w)	ModRM:r/m (r)	NA	NA
B	NA	ModRM:r/m (w)	ModRM:reg (r)	NA	NA
C	Full Vector Mem	ModRM:reg (w)	ModRM:r/m (r)	NA	NA
D	Full Vector Mem	ModRM:r/m (w)	ModRM:reg (r)	NA	NA

Description

Moves 4, 8 or 16 single-precision floating-point values from the source operand (second operand) to the destination operand (first operand). This instruction can be used to load an XMM, YMM or ZMM register from an 128-bit, 256-bit or 512-bit memory location, to store the contents of an XMM, YMM or ZMM register into a 128-bit, 256-bit or 512-bit memory location, or to move data between two XMM, two YMM or two ZMM registers.

When the source or destination operand is a memory operand, the operand must be aligned on a 16-byte (128-bit version), 32-byte (VEX.256 encoded version) or 64-byte (EVEX.512 encoded version) boundary or a general-protection exception (#GP) will be generated. For EVEX.512 encoded versions, the operand must be aligned to the size of the memory operand. To move single-precision floating-point values to and from unaligned memory locations, use the VMOVUPS instruction.

Note: VEX.vvvv and EVEX.vvvv are reserved and must be 1111b otherwise instructions will #UD.

EVEX.512 encoded version:

Moves 512 bits of packed single-precision floating-point values from the source operand (second operand) to the destination operand (first operand). This instruction can be used to load a ZMM register from a 512-bit float32 memory location, to store the contents of a ZMM register into a float32 memory location, or to move data between two ZMM registers. When the source or destination operand is a memory operand, the operand must be aligned on a 64-byte boundary or a general-protection exception (#GP) will be generated. To move single-precision floating-point values to and from unaligned memory locations, use the VMOVUPS instruction.

VEX.256 and EVEX.256 encoded version:

Moves 256 bits of packed single-precision floating-point values from the source operand (second operand) to the destination operand (first operand). This instruction can be used to load a YMM register from a 256-bit memory location, to store the contents of a YMM register into a 256-bit memory location, or to move data between two YMM registers. When the source or destination operand is a memory operand, the operand must be aligned on a 32-byte boundary or a general-protection exception (#GP) will be generated.

128-bit versions:

Moves 128 bits of packed single-precision floating-point values from the source operand (second operand) to the destination operand (first operand). This instruction can be used to load an XMM register from a 128-bit memory location, to store the contents of an XMM register into a 128-bit memory location, or to move data between two XMM registers. When the source or destination operand is a memory operand, the operand must be aligned on a 16-byte boundary or a general-protection exception (#GP) will be generated. To move single-precision floating-point values to and from unaligned memory locations, use the VMOVUPS instruction.

128-bit Legacy SSE version: Bits (MAXVL-1:128) of the corresponding ZMM destination register remain unchanged.

(E)VEX.128 encoded version: Bits (MAXVL-1:128) of the destination ZMM register are zeroed.

Operation

VMOVAPS (EVEX encoded versions, register-copy form)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j ← 0 TO KL-1
    i ← j * 32
    IF k1[j] OR *no writemask*
        THEN DEST[i+31:i] ← SRC[i+31:i]
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE  DEST[i+31:i] ← 0              ; zeroing-masking
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0
```

VMOVAPS (EVEX encoded versions, store-form)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j ← 0 TO KL-1
    i ← j * 32
    IF k1[j] OR *no writemask*
        THEN DEST[i+31:i]←
            SRC[i+31:i]
        ELSE *DEST[i+31:i] remains unchanged*          ; merging-masking
    FI;
ENDFOR;
```

VMOVAPS (EVEX encoded versions, load-form)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j ← 0 TO KL-1
    i ← j * 32
    IF k1[j] OR *no writemask*
        THEN DEST[i+31:i] ← SRC[i+31:i]
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE  DEST[i+31:i] ← 0              ; zeroing-masking
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0
```

VMOVAPS (VEX.256 encoded version, load - and register copy)

```
DEST[255:0] ← SRC[255:0]
DEST[MAXVL-1:256] ← 0
```

VMOVAPS (VEX.256 encoded version, store-form)

```
DEST[255:0] ← SRC[255:0]
```

VMOVAPS (VEX.128 encoded version, load - and register copy)

```
DEST[127:0] ← SRC[127:0]
DEST[MAXVL-1:128] ← 0
```

MOVAPS (128-bit load- and register-copy- form Legacy SSE version)

```
DEST[127:0] ← SRC[127:0]
DEST[MAXVL-1:128] (Unmodified)
```

(V)MOVAPS (128-bit store-form version)

```
DEST[127:0] ← SRC[127:0]
```

Intel C/C++ Compiler Intrinsic Equivalent

VMOVAPS __m512 _mm512_load_ps(void * m);
VMOVAPS __m512 _mm512_mask_load_ps(__m512 s, __mmask16 k, void * m);
VMOVAPS __m512 _mm512_maskz_load_ps(__mmask16 k, void * m);
VMOVAPS void _mm512_store_ps(void * d, __m512 a);
VMOVAPS void _mm512_mask_store_ps(void * d, __mmask16 k, __m512 a);
VMOVAPS __m256 _mm256_mask_load_ps(__m256 a, __mmask8 k, void * s);
VMOVAPS __m256 _mm256_maskz_load_ps(__mmask8 k, void * s);
VMOVAPS void _mm256_mask_store_ps(void * d, __mmask8 k, __m256 a);
VMOVAPS __m128 _mm_mask_load_ps(__m128 a, __mmask8 k, void * s);
VMOVAPS __m128 _mm_maskz_load_ps(__mmask8 k, void * s);
VMOVAPS void _mm_mask_store_ps(void * d, __mmask8 k, __m128 a);
MOVAPS __m256 _mm256_load_ps (float * p);
MOVAPS void _mm256_store_ps(float * p, __m256 a);
MOVAPS __m128 _mm_load_ps (float * p);
MOVAPS void _mm_store_ps(float * p, __m128 a);

SIMD Floating-Point Exceptions

None

Other Exceptions

Non-EVEX-encoded instruction, see Exceptions Type1.SSE; additionally

#UD	If VEX.vvvv != 1111B.

EVEX-encoded instruction, see Exceptions Type E1.

MOVBE—Move Data After Swapping Bytes

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
0F 38 F0 /r	MOVBE *r16,* m16	RM	Valid	Valid	Reverse byte order in *m16 and move to r16.*
0F 38 F0 /r	MOVBE *r32,* m32	RM	Valid	Valid	Reverse byte order in *m32 and move to r32.*
REX.W + 0F 38 F0 /r	MOVBE *r64,* m64	RM	Valid	N.E.	Reverse byte order in *m64 and move to r64.*
0F 38 F1 /r	MOVBE *m16,* r16	MR	Valid	Valid	Reverse byte order in *r16 and move to m16.*
0F 38 F1 /r	MOVBE *m32,* r32	MR	Valid	Valid	Reverse byte order in *r32 and move to m32.*
REX.W + 0F 38 F1 /r	MOVBE *m64,* r64	MR	Valid	N.E.	Reverse byte order in *r64 and move to m64.*

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (w)	ModRM:r/m (r)	NA	NA
MR	ModRM:r/m (w)	ModRM:reg (r)	NA	NA

Description

Performs a byte swap operation on the data copied from the second operand (source operand) and store the result in the first operand (destination operand). The source operand can be a general-purpose register, or memory location; the destination register can be a general-purpose register, or a memory location; however, both operands can not be registers, and only one operand can be a memory location. Both operands must be the same size, which can be a word, a doubleword or quadword.

The MOVBE instruction is provided for swapping the bytes on a read from memory or on a write to memory; thus providing support for converting little-endian values to big-endian format and vice versa.

In 64-bit mode, the instruction's default operation size is 32 bits. Use of the REX.R prefix permits access to additional registers (R8-R15). Use of the REX.W prefix promotes operation to 64 bits. See the summary chart at the beginning of this section for encoding data and limits.

Operation

```
TEMP ← SRC

IF ( OperandSize = 16)
    THEN
        DEST[7:0] ← TEMP[15:8];
        DEST[15:8] ← TEMP[7:0];
    ELES IF   ( OperandSize = 32)
        DEST[7:0] ← TEMP[31:24];
        DEST[15:8] ← TEMP[23:16];
        DEST[23:16] ← TEMP[15:8];
        DEST[31:23] ← TEMP[7:0];
    ELSE IF ( OperandSize = 64)
        DEST[7:0] ← TEMP[63:56];
        DEST[15:8] ← TEMP[55:48];
        DEST[23:16] ← TEMP[47:40];
        DEST[31:24] ← TEMP[39:32];
        DEST[39:32] ← TEMP[31:24];
        DEST[47:40] ← TEMP[23:16];
        DEST[55:48] ← TEMP[15:8];
        DEST[63:56] ← TEMP[7:0];

FI;
```

Flags Affected

None

Protected Mode Exceptions

#GP(0)	If the destination operand is in a non-writable segment.
	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
	If the DS, ES, FS, or GS register contains a NULL segment selector.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If CPUID.01H:ECX.MOVBE[bit 22] = 0.
	If the LOCK prefix is used.
	If REP (F3H) prefix is used.

Real-Address Mode Exceptions

#GP	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS	If a memory operand effective address is outside the SS segment limit.
#UD	If CPUID.01H:ECX.MOVBE[bit 22] = 0.
	If the LOCK prefix is used.
	If REP (F3H) prefix is used.

Virtual-8086 Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If CPUID.01H:ECX.MOVBE[bit 22] = 0.
	If the LOCK prefix is used.
	If REP (F3H) prefix is used.
	If REPNE (F2H) prefix is used and CPUID.01H:ECX.SSE4_2[bit 20] = 0.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#GP(0)	If the memory address is in a non-canonical form.
#SS(0)	If the stack address is in a non-canonical form.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If CPUID.01H:ECX.MOVBE[bit 22] = 0.
	If the LOCK prefix is used.
	If REP (F3H) prefix is used.

MOVD/MOVQ—Move Doubleword/Move Quadword

Opcode/ Instruction	Op/ En	64/32-bit Mode	CPUID Feature Flag	Description
NP 0F 6E /r MOVD *mm, r/m32*	A	V/V	MMX	Move doubleword from *r/m32* to *mm*.
NP REX.W + 0F 6E /r MOVQ *mm, r/m64*	A	V/N.E.	MMX	Move quadword from *r/m64* to *mm*.
NP 0F 7E /r MOVD *r/m32, mm*	B	V/V	MMX	Move doubleword from *mm* to *r/m32*.
NP REX.W + 0F 7E /r MOVQ *r/m64, mm*	B	V/N.E.	MMX	Move quadword from *mm* to *r/m64*.
66 0F 6E /r MOVD *xmm, r/m32*	A	V/V	SSE2	Move doubleword from *r/m32* to *xmm*.
66 REX.W 0F 6E /r MOVQ *xmm, r/m64*	A	V/N.E.	SSE2	Move quadword from *r/m64* to *xmm*.
66 0F 7E /r MOVD *r/m32, xmm*	B	V/V	SSE2	Move doubleword from *xmm* register to *r/m32*.
66 REX.W 0F 7E /r MOVQ *r/m64, xmm*	B	V/N.E.	SSE2	Move quadword from *xmm* register to *r/m64*.
VEX.128.66.0F.W0 6E / VMOVD *xmm1, r32/m32*	A	V/V	AVX	Move doubleword from *r/m32* to *xmm1*.
VEX.128.66.0F.W1 6E /r VMOVQ *xmm1, r64/m64*	A	V/N.E[1].	AVX	Move quadword from *r/m64* to *xmm1*.
VEX.128.66.0F.W0 7E /r VMOVD *r32/m32, xmm1*	B	V/V	AVX	Move doubleword from *xmm1* register to *r/m32*.
VEX.128.66.0F.W1 7E /r VMOVQ *r64/m64, xmm1*	B	V/N.E[1].	AVX	Move quadword from *xmm1* register to *r/m64*.
EVEX.128.66.0F.W0 6E /r VMOVD xmm1, r32/m32	C	V/V	AVX512F	Move doubleword from r/m32 to xmm1.
EVEX.128.66.0F.W1 6E /r VMOVQ xmm1, r64/m64	C	V/N.E.[1]	AVX512F	Move quadword from r/m64 to xmm1.
EVEX.128.66.0F.W0 7E /r VMOVD r32/m32, xmm1	D	V/V	AVX512F	Move doubleword from xmm1 register to r/m32.
EVEX.128.66.0F.W1 7E /r VMOVQ r64/m64, xmm1	D	V/N.E.[1]	AVX512F	Move quadword from xmm1 register to r/m64.

NOTES:

1. For this specific instruction, VEX.W/EVEX.W in non-64 bit is ignored; the instructions behaves as if the W0 version is used.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	NA	ModRM:reg (w)	ModRM:r/m (r)	NA	NA
B	NA	ModRM:r/m (w)	ModRM:reg (r)	NA	NA
C	Tuple1 Scalar	ModRM:reg (w)	ModRM:r/m (r)	NA	NA
D	Tuple1 Scalar	ModRM:r/m (w)	ModRM:reg (r)	NA	NA

Description

Copies a doubleword from the source operand (second operand) to the destination operand (first operand). The source and destination operands can be general-purpose registers, MMX technology registers, XMM registers, or 32-bit memory locations. This instruction can be used to move a doubleword to and from the low doubleword of an MMX technology register and a general-purpose register or a 32-bit memory location, or to and from the low doubleword of an XMM register and a general-purpose register or a 32-bit memory location. The instruction cannot be used to transfer data between MMX technology registers, between XMM registers, between general-purpose registers, or between memory locations.

When the destination operand is an MMX technology register, the source operand is written to the low doubleword of the register, and the register is zero-extended to 64 bits. When the destination operand is an XMM register, the source operand is written to the low doubleword of the register, and the register is zero-extended to 128 bits.

In 64-bit mode, the instruction's default operation size is 32 bits. Use of the REX.R prefix permits access to additional registers (R8-R15). Use of the REX.W prefix promotes operation to 64 bits. See the summary chart at the beginning of this section for encoding data and limits.

MOVD/Q with XMM destination:

Moves a dword/qword integer from the source operand and stores it in the low 32/64-bits of the destination XMM register. The upper bits of the destination are zeroed. The source operand can be a 32/64-bit register or 32/64-bit memory location.

128-bit Legacy SSE version: Bits (MAXVL-1:128) of the corresponding YMM destination register remain unchanged. Qword operation requires the use of REX.W=1.

VEX.128 encoded version: Bits (MAXVL-1:128) of the destination register are zeroed. Qword operation requires the use of VEX.W=1.

EVEX.128 encoded version: Bits (MAXVL-1:128) of the destination register are zeroed. Qword operation requires the use of EVEX.W=1.

MOVD/Q with 32/64 reg/mem destination:

Stores the low dword/qword of the source XMM register to 32/64-bit memory location or general-purpose register. Qword operation requires the use of REX.W=1, VEX.W=1, or EVEX.W=1.

Note: VEX.vvvv and EVEX.vvvv are reserved and must be 1111b otherwise instructions will #UD.

If VMOVD or VMOVQ is encoded with VEX.L= 1, an attempt to execute the instruction encoded with VEX.L= 1 will cause an #UD exception.

Operation

MOVD (when destination operand is MMX technology register)

```
DEST[31:0] ← SRC;
DEST[63:32] ← 00000000H;
```

MOVD (when destination operand is XMM register)

```
DEST[31:0] ← SRC;
DEST[127:32] ← 000000000000000000000000H;
DEST[MAXVL-1:128] (Unmodified)
```

MOVD (when source operand is MMX technology or XMM register)
DEST ← SRC[31:0];

VMOVD (VEX-encoded version when destination is an XMM register)
DEST[31:0] ← SRC[31:0]
DEST[MAXVL-1:32] ← 0

MOVQ (when destination operand is XMM register)
DEST[63:0] ← SRC[63:0];
DEST[127:64] ← 0000000000000000H;
DEST[MAXVL-1:128] (Unmodified)

MOVQ (when destination operand is r/m64)
DEST[63:0] ← SRC[63:0];

MOVQ (when source operand is XMM register or r/m64)
DEST ← SRC[63:0];

VMOVQ (VEX-encoded version when destination is an XMM register)
DEST[63:0] ← SRC[63:0]
DEST[MAXVL-1:64] ← 0

VMOVD (EVEX-encoded version when destination is an XMM register)
DEST[31:0] ← SRC[31:0]
DEST[MAXVL-1:32] ← 0

VMOVQ (EVEX-encoded version when destination is an XMM register)
DEST[63:0] ← SRC[63:0]
DEST[MAXVL-1:64] ← 0

Intel C/C++ Compiler Intrinsic Equivalent

MOVD: __m64 _mm_cvtsi32_si64 (int i)
MOVD: int _mm_cvtsi64_si32 (__m64m)
MOVD: __m128i _mm_cvtsi32_si128 (int a)
MOVD: int _mm_cvtsi128_si32 (__m128i a)
MOVQ: __int64 _mm_cvtsi128_si64(__m128i);
MOVQ: __m128i _mm_cvtsi64_si128(__int64);
VMOVD __m128i _mm_cvtsi32_si128(int);
VMOVD int _mm_cvtsi128_si32(__m128i);
VMOVQ __m128i _mm_cvtsi64_si128 (__int64);
VMOVQ __int64 _mm_cvtsi128_si64(__m128i);
VMOVQ __m128i _mm_loadl_epi64(__m128i * s);
VMOVQ void _mm_storel_epi64(__m128i * d, __m128i s);

Flags Affected

None

SIMD Floating-Point Exceptions

None

Other Exceptions

Non-EVEX-encoded instruction, see Exceptions Type 5.

EVEX-encoded instruction, see Exceptions Type E9NF.

#UD	If VEX.L = 1.
	If VEX.vvvv != 1111B or EVEX.vvvv != 1111B.

MOVDDUP—Replicate Double FP Values

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
F2 0F 12 /r MOVDDUP xmm1, xmm2/m64	A	V/V	SSE3	Move double-precision floating-point value from xmm2/m64 and duplicate into xmm1.
VEX.128.F2.0F.WIG 12 /r VMOVDDUP xmm1, xmm2/m64	A	V/V	AVX	Move double-precision floating-point value from xmm2/m64 and duplicate into xmm1.
VEX.256.F2.0F.WIG 12 /r VMOVDDUP ymm1, ymm2/m256	A	V/V	AVX	Move even index double-precision floating-point values from ymm2/mem and duplicate each element into ymm1.
EVEX.128.F2.0F.W1 12 /r VMOVDDUP xmm1 {k1}{z}, xmm2/m64	B	V/V	AVX512VL AVX512F	Move double-precision floating-point value from xmm2/m64 and duplicate each element into xmm1 subject to writemask k1.
EVEX.256.F2.0F.W1 12 /r VMOVDDUP ymm1 {k1}{z}, ymm2/m256	B	V/V	AVX512VL AVX512F	Move even index double-precision floating-point values from ymm2/m256 and duplicate each element into ymm1 subject to writemask k1.
EVEX.512.F2.0F.W1 12 /r VMOVDDUP zmm1 {k1}{z}, zmm2/m512	B	V/V	AVX512F	Move even index double-precision floating-point values from zmm2/m512 and duplicate each element into zmm1 subject to writemask k1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	NA	ModRM:reg (w)	ModRM:r/m (r)	NA	NA
B	MOVDDUP	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

Description

For 256-bit or higher versions: Duplicates even-indexed double-precision floating-point values from the source operand (the second operand) and into adjacent pair and store to the destination operand (the first operand).

For 128-bit versions: Duplicates the low double-precision floating-point value from the source operand (the second operand) and store to the destination operand (the first operand).

128-bit Legacy SSE version: Bits (MAXVL-1:128) of the corresponding destination register are unchanged. The source operand is XMM register or a 64-bit memory location.

VEX.128 and EVEX.128 encoded version: Bits (MAXVL-1:128) of the destination register are zeroed. The source operand is XMM register or a 64-bit memory location. The destination is updated conditionally under the writemask for EVEX version.

VEX.256 and EVEX.256 encoded version: Bits (MAXVL-1:256) of the destination register are zeroed. The source operand is YMM register or a 256-bit memory location. The destination is updated conditionally under the writemask for EVEX version.

EVEX.512 encoded version: The destination is updated according to the writemask. The source operand is ZMM register or a 512-bit memory location.

Note: VEX.vvvv and EVEX.vvvv are reserved and must be 1111b otherwise instructions will #UD.

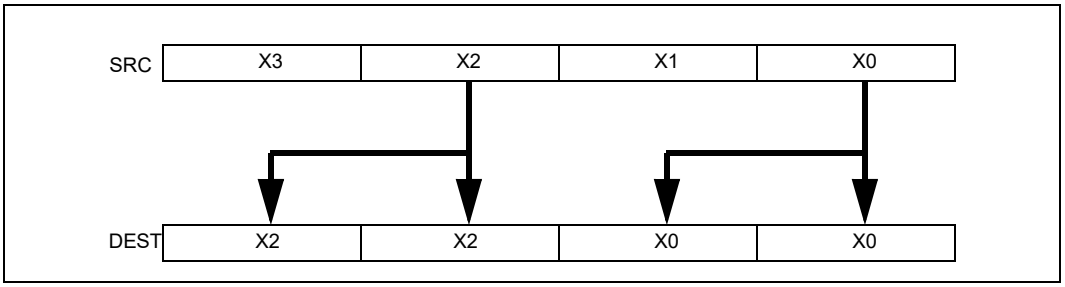


Figure 4-2. VMOVDDUP Operation

Operation

VMOVDDUP (EVEX encoded versions)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
TMP_SRC[63:0] ← SRC[63:0]
TMP_SRC[127:64] ← SRC[63:0]
IF VL >= 256
    TMP_SRC[191:128] ← SRC[191:128]
    TMP_SRC[255:192] ← SRC[191:128]
FI;
IF VL >= 512
    TMP_SRC[319:256] ← SRC[319:256]
    TMP_SRC[383:320] ← SRC[319:256]
    TMP_SRC[477:384] ← SRC[477:384]
    TMP_SRC[511:484] ← SRC[477:384]
FI;
FOR j ← 0 TO KL-1
    i ← j * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+63:i] ← TMP_SRC[i+63:i]
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE                            ; zeroing-masking
                    DEST[i+63:i] ← 0            ; zeroing-masking
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0
```

VMOVDDUP (VEX.256 encoded version)

```
DEST[63:0] ←SRC[63:0]
DEST[127:64] ←SRC[63:0]
DEST[191:128] ←SRC[191:128]
DEST[255:192] ←SRC[191:128]
DEST[MAXVL-1:256] ←0
```

VMOVDDUP (VEX.128 encoded version)

```
DEST[63:0] ←SRC[63:0]
DEST[127:64] ←SRC[63:0]
DEST[MAXVL-1:128] ←0
```

MOVDDUP (128-bit Legacy SSE version)
DEST[63:0] ←SRC[63:0]
DEST[127:64] ←SRC[63:0]
DEST[MAXVL-1:128] (Unmodified)

Intel C/C++ Compiler Intrinsic Equivalent

VMOVDDUP __m512d _mm512_movedup_pd(__m512d a);
VMOVDDUP __m512d _mm512_mask_movedup_pd(__m512d s, __mmask8 k, __m512d a);
VMOVDDUP __m512d _mm512_maskz_movedup_pd(__mmask8 k, __m512d a);
VMOVDDUP __m256d _mm256_mask_movedup_pd(__m256d s, __mmask8 k, __m256d a);
VMOVDDUP __m256d _mm256_maskz_movedup_pd(__mmask8 k, __m256d a);
VMOVDDUP __m128d _mm_mask_movedup_pd(__m128d s, __mmask8 k, __m128d a);
VMOVDDUP __m128d _mm_maskz_movedup_pd(__mmask8 k, __m128d a);
MOVDDUP __m256d _mm256_movedup_pd (__m256d a);
MOVDDUP __m128d _mm_movedup_pd (__m128d a);

SIMD Floating-Point Exceptions

None

Other Exceptions

Non-EVEX-encoded instruction, see Exceptions Type 5;
EVEX-encoded instruction, see Exceptions Type E5NF.

#UD	If EVEX.vvvv != 1111B or VEX.vvvv != 1111B.

MOVDQA,VMOVDQA32/64—Move Aligned Packed Integer Values

Opcode/ Instruction	Op/En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 6F /r MOVDQA xmm1, xmm2/m128	A	V/V	SSE2	Move aligned packed integer values from xmm2/mem to xmm1.
66 0F 7F /r MOVDQA xmm2/m128, xmm1	B	V/V	SSE2	Move aligned packed integer values from xmm1 to xmm2/mem.
VEX.128.66.0F.WIG 6F /r VMOVDQA xmm1, xmm2/m128	A	V/V	AVX	Move aligned packed integer values from xmm2/mem to xmm1.
VEX.128.66.0F.WIG 7F /r VMOVDQA xmm2/m128, xmm1	B	V/V	AVX	Move aligned packed integer values from xmm1 to xmm2/mem.
VEX.256.66.0F.WIG 6F /r VMOVDQA ymm1, ymm2/m256	A	V/V	AVX	Move aligned packed integer values from ymm2/mem to ymm1.
VEX.256.66.0F.WIG 7F /r VMOVDQA ymm2/m256, ymm1	B	V/V	AVX	Move aligned packed integer values from ymm1 to ymm2/mem.
EVEX.128.66.0F.W0 6F /r VMOVDQA32 xmm1 {k1}{z}, xmm2/m128	C	V/V	AVX512VL AVX512F	Move aligned packed doubleword integer values from xmm2/m128 to xmm1 using writemask k1.
EVEX.256.66.0F.W0 6F /r VMOVDQA32 ymm1 {k1}{z}, ymm2/m256	C	V/V	AVX512VL AVX512F	Move aligned packed doubleword integer values from ymm2/m256 to ymm1 using writemask k1.
EVEX.512.66.0F.W0 6F /r VMOVDQA32 zmm1 {k1}{z}, zmm2/m512	C	V/V	AVX512F	Move aligned packed doubleword integer values from zmm2/m512 to zmm1 using writemask k1.
EVEX.128.66.0F.W0 7F /r VMOVDQA32 xmm2/m128 {k1}{z}, xmm1	D	V/V	AVX512VL AVX512F	Move aligned packed doubleword integer values from xmm1 to xmm2/m128 using writemask k1.
EVEX.256.66.0F.W0 7F /r VMOVDQA32 ymm2/m256 {k1}{z}, ymm1	D	V/V	AVX512VL AVX512F	Move aligned packed doubleword integer values from ymm1 to ymm2/m256 using writemask k1.
EVEX.512.66.0F.W0 7F /r VMOVDQA32 zmm2/m512 {k1}{z}, zmm1	D	V/V	AVX512F	Move aligned packed doubleword integer values from zmm1 to zmm2/m512 using writemask k1.
EVEX.128.66.0F.W1 6F /r VMOVDQA64 xmm1 {k1}{z}, xmm2/m128	C	V/V	AVX512VL AVX512F	Move aligned quadword integer values from xmm2/m128 to xmm1 using writemask k1.
EVEX.256.66.0F.W1 6F /r VMOVDQA64 ymm1 {k1}{z}, ymm2/m256	C	V/V	AVX512VL AVX512F	Move aligned quadword integer values from ymm2/m256 to ymm1 using writemask k1.
EVEX.512.66.0F.W1 6F /r VMOVDQA64 zmm1 {k1}{z}, zmm2/m512	C	V/V	AVX512F	Move aligned packed quadword integer values from zmm2/m512 to zmm1 using writemask k1.
EVEX.128.66.0F.W1 7F /r VMOVDQA64 xmm2/m128 {k1}{z}, xmm1	D	V/V	AVX512VL AVX512F	Move aligned packed quadword integer values from xmm1 to xmm2/m128 using writemask k1.
EVEX.256.66.0F.W1 7F /r VMOVDQA64 ymm2/m256 {k1}{z}, ymm1	D	V/V	AVX512VL AVX512F	Move aligned packed quadword integer values from ymm1 to ymm2/m256 using writemask k1.
EVEX.512.66.0F.W1 7F /r VMOVDQA64 zmm2/m512 {k1}{z}, zmm1	D	V/V	AVX512F	Move aligned packed quadword integer values from zmm1 to zmm2/m512 using writemask k1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	NA	ModRM:reg (w)	ModRM:r/m (r)	NA	NA
B	NA	ModRM:r/m (w)	ModRM:reg (r)	NA	NA
C	Full Vector Mem	ModRM:reg (w)	ModRM:r/m (r)	NA	NA
D	Full Vector Mem	ModRM:r/m (w)	ModRM:reg (r)	NA	NA

Description

Note: VEX.vvvv and EVEX.vvvv are reserved and must be 1111b otherwise instructions will #UD.

EVEX encoded versions:

Moves 128, 256 or 512 bits of packed doubleword/quadword integer values from the source operand (the second operand) to the destination operand (the first operand). This instruction can be used to load a vector register from an int32/int64 memory location, to store the contents of a vector register into an int32/int64 memory location, or to move data between two ZMM registers. When the source or destination operand is a memory operand, the operand must be aligned on a 16 (EVEX.128)/32(EVEX.256)/64(EVEX.512)-byte boundary or a general-protection exception (#GP) will be generated. To move integer data to and from unaligned memory locations, use the VMOVDQU instruction.

The destination operand is updated at 32-bit (VMOVDQA32) or 64-bit (VMOVDQA64) granularity according to the writemask.

VEX.256 encoded version:

Moves 256 bits of packed integer values from the source operand (second operand) to the destination operand (first operand). This instruction can be used to load a YMM register from a 256-bit memory location, to store the contents of a YMM register into a 256-bit memory location, or to move data between two YMM registers.

When the source or destination operand is a memory operand, the operand must be aligned on a 32-byte boundary or a general-protection exception (#GP) will be generated. To move integer data to and from unaligned memory locations, use the VMOVDQU instruction. Bits (MAXVL-1:256) of the destination register are zeroed.

128-bit versions:

Moves 128 bits of packed integer values from the source operand (second operand) to the destination operand (first operand). This instruction can be used to load an XMM register from a 128-bit memory location, to store the contents of an XMM register into a 128-bit memory location, or to move data between two XMM registers.

When the source or destination operand is a memory operand, the operand must be aligned on a 16-byte boundary or a general-protection exception (#GP) will be generated. To move integer data to and from unaligned memory locations, use the VMOVDQU instruction.

128-bit Legacy SSE version: Bits (MAXVL-1:128) of the corresponding ZMM destination register remain unchanged.

VEX.128 encoded version: Bits (MAXVL-1:128) of the destination register are zeroed.

Operation

VMOVDQA32 (EVEX encoded versions, register-copy form)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j ← 0 TO KL-1
    i ← j * 32
    IF k1[j] OR *no writemask*
        THEN DEST[i+31:i] ← SRC[i+31:i]
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE  DEST[i+31:i] ← 0          ; zeroing-masking
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0
```

VMOVDQA32 (EVEX encoded versions, store-form)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j ← 0 TO KL-1
    i ← j * 32
    IF k1[j] OR *no writemask*
        THEN DEST[i+31:i]← SRC[i+31:i]
        ELSE *DEST[i+31:i] remains unchanged*       ; merging-masking
    FI;
ENDFOR;
```

VMOVDQA32 (EVEX encoded versions, load-form)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j ← 0 TO KL-1
    i ← j * 32
    IF k1[j] OR *no writemask*
        THEN DEST[i+31:i] ← SRC[i+31:i]
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE  DEST[i+31:i] ← 0          ; zeroing-masking
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0
```

VMOVDQA64 (EVEX encoded versions, register-copy form)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j ← 0 TO KL-1
    i ← j * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+63:i] ← SRC[i+63:i]
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE  DEST[i+63:i] ← 0              ; zeroing-masking
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0
```

VMOVDQA64 (EVEX encoded versions, store-form)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j ← 0 TO KL-1
    i ← j * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+63:i]← SRC[i+63:i]
        ELSE *DEST[i+63:i] remains unchanged*          ; merging-masking
    FI;
ENDFOR;
```

VMOVDQA64 (EVEX encoded versions, load-form)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j ← 0 TO KL-1
    i ← j * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+63:i] ← SRC[i+63:i]
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE  DEST[i+63:i] ← 0              ; zeroing-masking
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0
```

VMOVDQA (VEX.256 encoded version, load - and register copy)

```
DEST[255:0] ← SRC[255:0]
DEST[MAXVL-1:256] ← 0
```

VMOVDQA (VEX.256 encoded version, store-form)

```
DEST[255:0] ← SRC[255:0]
```

VMOVDQA (VEX.128 encoded version)

```
DEST[127:0] ← SRC[127:0]
DEST[MAXVL-1:128] ← 0
```

VMOVDQA (128-bit load- and register-copy- form Legacy SSE version)

```
DEST[127:0] ← SRC[127:0]
DEST[MAXVL-1:128] (Unmodified)
```

(V)MOVDQA (128-bit store-form version)
DEST[127:0] ← SRC[127:0]

Intel C/C++ Compiler Intrinsic Equivalent

VMOVDQA32 __m512i _mm512_load_epi32(void * sa);
VMOVDQA32 __m512i _mm512_mask_load_epi32(__m512i s, __mmask16 k, void * sa);
VMOVDQA32 __m512i _mm512_maskz_load_epi32(__mmask16 k, void * sa);
VMOVDQA32 void _mm512_store_epi32(void * d, __m512i a);
VMOVDQA32 void _mm512_mask_store_epi32(void * d, __mmask16 k, __m512i a);
VMOVDQA32 __m256i _mm256_mask_load_epi32(__m256i s, __mmask8 k, void * sa);
VMOVDQA32 __m256i _mm256_maskz_load_epi32(__mmask8 k, void * sa);
VMOVDQA32 void _mm256_store_epi32(void * d, __m256i a);
VMOVDQA32 void _mm256_mask_store_epi32(void * d, __mmask8 k, __m256i a);
VMOVDQA32 __m128i _mm_mask_load_epi32(__m128i s, __mmask8 k, void * sa);
VMOVDQA32 __m128i _mm_maskz_load_epi32(__mmask8 k, void * sa);
VMOVDQA32 void _mm_store_epi32(void * d, __m128i a);
VMOVDQA32 void _mm_mask_store_epi32(void * d, __mmask8 k, __m128i a);
VMOVDQA64 __m512i _mm512_load_epi64(void * sa);
VMOVDQA64 __m512i _mm512_mask_load_epi64(__m512i s, __mmask8 k, void * sa);
VMOVDQA64 __m512i _mm512_maskz_load_epi64(__mmask8 k, void * sa);
VMOVDQA64 void _mm512_store_epi64(void * d, __m512i a);
VMOVDQA64 void _mm512_mask_store_epi64(void * d, __mmask8 k, __m512i a);
VMOVDQA64 __m256i _mm256_mask_load_epi64(__m256i s, __mmask8 k, void * sa);
VMOVDQA64 __m256i _mm256_maskz_load_epi64(__mmask8 k, void * sa);
VMOVDQA64 void _mm256_store_epi64(void * d, __m256i a);
VMOVDQA64 void _mm256_mask_store_epi64(void * d, __mmask8 k, __m256i a);
VMOVDQA64 __m128i _mm_mask_load_epi64(__m128i s, __mmask8 k, void * sa);
VMOVDQA64 __m128i _mm_maskz_load_epi64(__mmask8 k, void * sa);
VMOVDQA64 void _mm_store_epi64(void * d, __m128i a);
VMOVDQA64 void _mm_mask_store_epi64(void * d, __mmask8 k, __m128i a);
MOVDQA void __m256i _mm256_load_si256 (__m256i * p);
MOVDQA _mm256_store_si256(_m256i *p, __m256i a);
MOVDQA __m128i _mm_load_si128 (__m128i * p);
MOVDQA void _mm_store_si128(__m128i *p, __m128i a);

SIMD Floating-Point Exceptions

None

Other Exceptions

Non-EVEX-encoded instruction, see Exceptions Type1.SSE2;
EVEX-encoded instruction, see Exceptions Type E1.

#UD	If EVEX.vvvv != 1111B or VEX.vvvv != 1111B.

MOVDQU,VMOVDQU8/16/32/64—Move Unaligned Packed Integer Values

Opcode/ Instruction	Op/En	64/32 bit Mode Support	CPUID Feature Flag	Description
F3 0F 6F /r MOVDQU xmm1, xmm2/m128	A	V/V	SSE2	Move unaligned packed integer values from xmm2/m128 to xmm1.
F3 0F 7F /r MOVDQU xmm2/m128, xmm1	B	V/V	SSE2	Move unaligned packed integer values from xmm1 to xmm2/m128.
VEX.128.F3.0F.WIG 6F /r VMOVDQU xmm1, xmm2/m128	A	V/V	AVX	Move unaligned packed integer values from xmm2/m128 to xmm1.
VEX.128.F3.0F.WIG 7F /r VMOVDQU xmm2/m128, xmm1	B	V/V	AVX	Move unaligned packed integer values from xmm1 to xmm2/m128.
VEX.256.F3.0F.WIG 6F /r VMOVDQU ymm1, ymm2/m256	A	V/V	AVX	Move unaligned packed integer values from ymm2/m256 to ymm1.
VEX.256.F3.0F.WIG 7F /r VMOVDQU ymm2/m256, ymm1	B	V/V	AVX	Move unaligned packed integer values from ymm1 to ymm2/m256.
EVEX.128.F2.0F.W0 6F /r VMOVDQU8 xmm1 {k1}{z}, xmm2/m128	C	V/V	AVX512VL AVX512BW	Move unaligned packed byte integer values from xmm2/m128 to xmm1 using writemask k1.
EVEX.256.F2.0F.W0 6F /r VMOVDQU8 ymm1 {k1}{z}, ymm2/m256	C	V/V	AVX512VL AVX512BW	Move unaligned packed byte integer values from ymm2/m256 to ymm1 using writemask k1.
EVEX.512.F2.0F.W0 6F /r VMOVDQU8 zmm1 {k1}{z}, zmm2/m512	C	V/V	AVX512BW	Move unaligned packed byte integer values from zmm2/m512 to zmm1 using writemask k1.
EVEX.128.F2.0F.W0 7F /r VMOVDQU8 xmm2/m128 {k1}{z}, xmm1	D	V/V	AVX512VL AVX512BW	Move unaligned packed byte integer values from xmm1 to xmm2/m128 using writemask k1.
EVEX.256.F2.0F.W0 7F /r VMOVDQU8 ymm2/m256 {k1}{z}, ymm1	D	V/V	AVX512VL AVX512BW	Move unaligned packed byte integer values from ymm1 to ymm2/m256 using writemask k1.
EVEX.512.F2.0F.W0 7F /r VMOVDQU8 zmm2/m512 {k1}{z}, zmm1	D	V/V	AVX512BW	Move unaligned packed byte integer values from zmm1 to zmm2/m512 using writemask k1.
EVEX.128.F2.0F.W1 6F /r VMOVDQU16 xmm1 {k1}{z}, xmm2/m128	C	V/V	AVX512VL AVX512BW	Move unaligned packed word integer values from xmm2/m128 to xmm1 using writemask k1.
EVEX.256.F2.0F.W1 6F /r VMOVDQU16 ymm1 {k1}{z}, ymm2/m256	C	V/V	AVX512VL AVX512BW	Move unaligned packed word integer values from ymm2/m256 to ymm1 using writemask k1.
EVEX.512.F2.0F.W1 6F /r VMOVDQU16 zmm1 {k1}{z}, zmm2/m512	C	V/V	AVX512BW	Move unaligned packed word integer values from zmm2/m512 to zmm1 using writemask k1.
EVEX.128.F2.0F.W1 7F /r VMOVDQU16 xmm2/m128 {k1}{z}, xmm1	D	V/V	AVX512VL AVX512BW	Move unaligned packed word integer values from xmm1 to xmm2/m128 using writemask k1.
EVEX.256.F2.0F.W1 7F /r VMOVDQU16 ymm2/m256 {k1}{z}, ymm1	D	V/V	AVX512VL AVX512BW	Move unaligned packed word integer values from ymm1 to ymm2/m256 using writemask k1.
EVEX.512.F2.0F.W1 7F /r VMOVDQU16 zmm2/m512 {k1}{z}, zmm1	D	V/V	AVX512BW	Move unaligned packed word integer values from zmm1 to zmm2/m512 using writemask k1.
EVEX.128.F3.0F.W0 6F /r VMOVDQU32 xmm1 {k1}{z}, xmm2/mm128	C	V/V	AVX512VL AVX512F	Move unaligned packed doubleword integer values from xmm2/m128 to xmm1 using writemask k1.

Opcode/ Instruction	Op/En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.256.F3.0F.W0 6F /r VMOVDQU32 ymm1 {k1}{z}, ymm2/m256	C	V/V	AVX512VL AVX512F	Move unaligned packed doubleword integer values from ymm2/m256 to ymm1 using writemask k1.
EVEX.512.F3.0F.W0 6F /r VMOVDQU32 zmm1 {k1}{z}, zmm2/m512	C	V/V	AVX512F	Move unaligned packed doubleword integer values from zmm2/m512 to zmm1 using writemask k1.
EVEX.128.F3.0F.W0 7F /r VMOVDQU32 xmm2/m128 {k1}{z}, xmm1	D	V/V	AVX512VL AVX512F	Move unaligned packed doubleword integer values from xmm1 to xmm2/m128 using writemask k1.
EVEX.256.F3.0F.W0 7F /r VMOVDQU32 ymm2/m256 {k1}{z}, ymm1	D	V/V	AVX512VL AVX512F	Move unaligned packed doubleword integer values from ymm1 to ymm2/m256 using writemask k1.
EVEX.512.F3.0F.W0 7F /r VMOVDQU32 zmm2/m512 {k1}{z}, zmm1	D	V/V	AVX512F	Move unaligned packed doubleword integer values from zmm1 to zmm2/m512 using writemask k1.
EVEX.128.F3.0F.W1 6F /r VMOVDQU64 xmm1 {k1}{z}, xmm2/m128	C	V/V	AVX512VL AVX512F	Move unaligned packed quadword integer values from xmm2/m128 to xmm1 using writemask k1.
EVEX.256.F3.0F.W1 6F /r VMOVDQU64 ymm1 {k1}{z}, ymm2/m256	C	V/V	AVX512VL AVX512F	Move unaligned packed quadword integer values from ymm2/m256 to ymm1 using writemask k1.
EVEX.512.F3.0F.W1 6F /r VMOVDQU64 zmm1 {k1}{z}, zmm2/m512	C	V/V	AVX512F	Move unaligned packed quadword integer values from zmm2/m512 to zmm1 using writemask k1.
EVEX.128.F3.0F.W1 7F /r VMOVDQU64 xmm2/m128 {k1}{z}, xmm1	D	V/V	AVX512VL AVX512F	Move unaligned packed quadword integer values from xmm1 to xmm2/m128 using writemask k1.
EVEX.256.F3.0F.W1 7F /r VMOVDQU64 ymm2/m256 {k1}{z}, ymm1	D	V/V	AVX512VL AVX512F	Move unaligned packed quadword integer values from ymm1 to ymm2/m256 using writemask k1.
EVEX.512.F3.0F.W1 7F /r VMOVDQU64 zmm2/m512 {k1}{z}, zmm1	D	V/V	AVX512F	Move unaligned packed quadword integer values from zmm1 to zmm2/m512 using writemask k1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	NA	ModRM:reg (w)	ModRM:r/m (r)	NA	NA
B	NA	ModRM:r/m (w)	ModRM:reg (r)	NA	NA
C	Full Vector Mem	ModRM:reg (w)	ModRM:r/m (r)	NA	NA
D	Full Vector Mem	ModRM:r/m (w)	ModRM:reg (r)	NA	NA

Description

Note: VEX.vvvv and EVEX.vvvv are reserved and must be 1111b otherwise instructions will #UD.

EVEX encoded versions:

Moves 128, 256 or 512 bits of packed byte/word/doubleword/quadword integer values from the source operand (the second operand) to the destination operand (first operand). This instruction can be used to load a vector register from a memory location, to store the contents of a vector register into a memory location, or to move data between two vector registers.

The destination operand is updated at 8-bit (VMOVDQU8), 16-bit (VMOVDQU16), 32-bit (VMOVDQU32), or 64-bit (VMOVDQU64) granularity according to the writemask.

VEX.256 encoded version:

Moves 256 bits of packed integer values from the source operand (second operand) to the destination operand (first operand). This instruction can be used to load a YMM register from a 256-bit memory location, to store the contents of a YMM register into a 256-bit memory location, or to move data between two YMM registers.

Bits (MAXVL-1:256) of the destination register are zeroed.

128-bit versions:

Moves 128 bits of packed integer values from the source operand (second operand) to the destination operand (first operand). This instruction can be used to load an XMM register from a 128-bit memory location, to store the contents of an XMM register into a 128-bit memory location, or to move data between two XMM registers.

128-bit Legacy SSE version: Bits (MAXVL-1:128) of the corresponding destination register remain unchanged.

When the source or destination operand is a memory operand, the operand may be unaligned to any alignment without causing a general-protection exception (#GP) to be generated

VEX.128 encoded version: Bits (MAXVL-1:128) of the destination register are zeroed.

Operation

VMOVDQU8 (EVEX encoded versions, register-copy form)

```
(KL, VL) = (16, 128), (32, 256), (64, 512)
FOR j ← 0 TO KL-1
    i ← j * 8
    IF k1[j] OR *no writemask*
        THEN DEST[i+7:i] ← SRC[i+7:i]
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+7:i] remains unchanged*
                ELSE  DEST[i+7:i] ← 0               ; zeroing-masking
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0
```

VMOVDQU8 (EVEX encoded versions, store-form)

```
(KL, VL) = (16, 128), (32, 256), (64, 512)
FOR j ← 0 TO KL-1
    i ← j * 8
    IF k1[j] OR *no writemask*
        THEN DEST[i+7:i]←
            SRC[i+7:i]
        ELSE *DEST[i+7:i] remains unchanged*          ; merging-masking
    FI;
ENDFOR;
```

VMOVDQU8 (EVEX encoded versions, load-form)

```
(KL, VL) = (16, 128), (32, 256), (64, 512)
FOR j ← 0 TO KL-1
    i ← j * 8
    IF k1[j] OR *no writemask*
        THEN DEST[i+7:i] ← SRC[i+7:i]
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+7:i] remains unchanged*
                ELSE  DEST[i+7:i] ← 0               ; zeroing-masking
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0
```

VMOVDQU16 (EVEX encoded versions, register-copy form)

```
(KL, VL) = (8, 128), (16, 256), (32, 512)
FOR j ← 0 TO KL-1
    i ← j * 16
    IF k1[j] OR *no writemask*
        THEN DEST[i+15:i] ← SRC[i+15:i]
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+15:i] remains unchanged*
                ELSE  DEST[i+15:i] ← 0              ; zeroing-masking
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0
```

VMOVDQU16 (EVEX encoded versions, store-form)

```
(KL, VL) = (8, 128), (16, 256), (32, 512)
FOR j ← 0 TO KL-1
    i ← j * 16
    IF k1[j] OR *no writemask*
        THEN DEST[i+15:i]←
            SRC[i+15:i]
        ELSE *DEST[i+15:i] remains unchanged*        ; merging-masking
    FI;
ENDFOR;
```

VMOVDQU16 (EVEX encoded versions, load-form)

```
(KL, VL) = (8, 128), (16, 256), (32, 512)
FOR j ← 0 TO KL-1
    i ← j * 16
    IF k1[j] OR *no writemask*
        THEN DEST[i+15:i] ← SRC[i+15:i]
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+15:i] remains unchanged*
                ELSE  DEST[i+15:i] ← 0              ; zeroing-masking
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0
```

VMOVDQU32 (EVEX encoded versions, register-copy form)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j ← 0 TO KL-1
    i ← j * 32
    IF k1[j] OR *no writemask*
        THEN DEST[i+31:i] ← SRC[i+31:i]
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE  DEST[i+31:i] ← 0              ; zeroing-masking
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0
```

VMOVDQU32 (EVEX encoded versions, store-form)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j ← 0 TO KL-1
    i ← j * 32
    IF k1[j] OR *no writemask*
        THEN DEST[i+31:i]←
            SRC[i+31:i]
        ELSE *DEST[i+31:i] remains unchanged*        ; merging-masking
    FI;
ENDFOR;
```

VMOVDQU32 (EVEX encoded versions, load-form)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j ← 0 TO KL-1
    i ← j * 32
    IF k1[j] OR *no writemask*
        THEN DEST[i+31:i] ← SRC[i+31:i]
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE  DEST[i+31:i] ← 0              ; zeroing-masking
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0
```

VMOVDQU64 (EVEX encoded versions, register-copy form)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j ← 0 TO KL-1
    i ← j * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+63:i] ← SRC[i+63:i]
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE  DEST[i+63:i] ← 0              ; zeroing-masking
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0
```

VMOVDQU64 (EVEX encoded versions, store-form)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j ← 0 TO KL-1
    i ← j * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+63:i]← SRC[i+63:i]
        ELSE *DEST[i+63:i] remains unchanged*        ; merging-masking

    FI;
ENDFOR;
```

VMOVDQU64 (EVEX encoded versions, load-form)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j ← 0 TO KL-1
    i ← j * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+63:i] ← SRC[i+63:i]
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE  DEST[i+63:i] ← 0              ; zeroing-masking
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0
```

VMOVDQU (VEX.256 encoded version, load - and register copy)

```
DEST[255:0] ← SRC[255:0]
DEST[MAXVL-1:256] ← 0
```

VMOVDQU (VEX.256 encoded version, store-form)

```
DEST[255:0] ← SRC[255:0]
```

VMOVDQU (VEX.128 encoded version)

```
DEST[127:0] ← SRC[127:0]
DEST[MAXVL-1:128] ← 0
```

VMOVDQU (128-bit load- and register-copy- form Legacy SSE version)

```
DEST[127:0] ← SRC[127:0]
DEST[MAXVL-1:128] (Unmodified)
```

(V)MOVDQU (128-bit store-form version)

```
DEST[127:0] ← SRC[127:0]
```

Intel C/C++ Compiler Intrinsic Equivalent

VMOVDQU16 __m512i _mm512_mask_loadu_epi16(__m512i s, __mmask32 k, void * sa);
VMOVDQU16 __m512i _mm512_maskz_loadu_epi16(__mmask32 k, void * sa);
VMOVDQU16 void _mm512_mask_storeu_epi16(void * d, __mmask32 k, __m512i a);
VMOVDQU16 __m256i _mm256_mask_loadu_epi16(__m256i s, __mmask16 k, void * sa);
VMOVDQU16 __m256i _mm256_maskz_loadu_epi16(__mmask16 k, void * sa);
VMOVDQU16 void _mm256_mask_storeu_epi16(void * d, __mmask16 k, __m256i a);
VMOVDQU16 void _mm256_maskz_storeu_epi16(void * d, __mmask16 k);
VMOVDQU16 __m128i _mm_mask_loadu_epi16(__m128i s, __mmask8 k, void * sa);
VMOVDQU16 __m128i _mm_maskz_loadu_epi16(__mmask8 k, void * sa);
VMOVDQU16 void _mm_mask_storeu_epi16(void * d, __mmask8 k, __m128i a);
VMOVDQU32 __m512i _mm512_loadu_epi32(void * sa);
VMOVDQU32 __m512i _mm512_mask_loadu_epi32(__m512i s, __mmask16 k, void * sa);
VMOVDQU32 __m512i _mm512_maskz_loadu_epi32(__mmask16 k, void * sa);
VMOVDQU32 void _mm512_storeu_epi32(void * d, __m512i a);
VMOVDQU32 void _mm512_mask_storeu_epi32(void * d, __mmask16 k, __m512i a);
VMOVDQU32 __m256i _mm256_mask_loadu_epi32(__m256i s, __mmask8 k, void * sa);
VMOVDQU32 __m256i _mm256_maskz_loadu_epi32(__mmask8 k, void * sa);
VMOVDQU32 void _mm256_storeu_epi32(void * d, __m256i a);
VMOVDQU32 void _mm256_mask_storeu_epi32(void * d, __mmask8 k, __m256i a);
VMOVDQU32 __m128i _mm_mask_loadu_epi32(__m128i s, __mmask8 k, void * sa);

VMOVDQU32 __m128i _mm_maskz_loadu_epi32(__mmask8 k, void * sa);
VMOVDQU32 void _mm_storeu_epi32(void * d, __m128i a);
VMOVDQU32 void _mm_mask_storeu_epi32(void * d, __mmask8 k, __m128i a);
VMOVDQU64 __m512i _mm512_loadu_epi64(void * sa);
VMOVDQU64 __m512i _mm512_mask_loadu_epi64(__m512i s, __mmask8 k, void * sa);
VMOVDQU64 __m512i _mm512_maskz_loadu_epi64(__mmask8 k, void * sa);
VMOVDQU64 void _mm512_storeu_epi64(void * d, __m512i a);
VMOVDQU64 void _mm512_mask_storeu_epi64(void * d, __mmask8 k, __m512i a);
VMOVDQU64 __m256i _mm256_mask_loadu_epi64(__m256i s, __mmask8 k, void * sa);
VMOVDQU64 __m256i _mm256_maskz_loadu_epi64(__mmask8 k, void * sa);
VMOVDQU64 void _mm256_storeu_epi64(void * d, __m256i a);
VMOVDQU64 void _mm256_mask_storeu_epi64(void * d, __mmask8 k, __m256i a);
VMOVDQU64 __m128i _mm_mask_loadu_epi64(__m128i s, __mmask8 k, void * sa);
VMOVDQU64 __m128i _mm_maskz_loadu_epi64(__mmask8 k, void * sa);
VMOVDQU64 void _mm_storeu_epi64(void * d, __m128i a);
VMOVDQU64 void _mm_mask_storeu_epi64(void * d, __mmask8 k, __m128i a);
VMOVDQU8 __m512i _mm512_mask_loadu_epi8(__m512i s, __mmask64 k, void * sa);
VMOVDQU8 __m512i _mm512_maskz_loadu_epi8(__mmask64 k, void * sa);
VMOVDQU8 void _mm512_mask_storeu_epi8(void * d, __mmask64 k, __m512i a);
VMOVDQU8 __m256i _mm256_mask_loadu_epi8(__m256i s, __mmask32 k, void * sa);
VMOVDQU8 __m256i _mm256_maskz_loadu_epi8(__mmask32 k, void * sa);
VMOVDQU8 void _mm256_mask_storeu_epi8(void * d, __mmask32 k, __m256i a);
VMOVDQU8 void _mm256_maskz_storeu_epi8(void * d, __mmask32 k);
VMOVDQU8 __m128i _mm_mask_loadu_epi8(__m128i s, __mmask16 k, void * sa);
VMOVDQU8 __m128i _mm_maskz_loadu_epi8(__mmask16 k, void * sa);
VMOVDQU8 void _mm_mask_storeu_epi8(void * d, __mmask16 k, __m128i a);
MOVDQU __m256i _mm256_loadu_si256 (__m256i * p);
MOVDQU _mm256_storeu_si256(_m256i *p, __m256i a);
MOVDQU __m128i _mm_loadu_si128 (__m128i * p);
MOVDQU _mm_storeu_si128(__m128i *p, __m128i a);

SIMD Floating-Point Exceptions

None

Other Exceptions

Non-EVEX-encoded instruction, see Exceptions Type 4;

EVEX-encoded instruction, see Exceptions Type E4.nb.

#UD	If EVEX.vvvv != 1111B or VEX.vvvv != 1111B.

MOVDQ2Q—Move Quadword from XMM to MMX Technology Register

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
F2 0F D6 /r	MOVDQ2Q *mm, xmm*	RM	Valid	Valid	Move low quadword from *xmm* to *mmx* register.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

Description

Moves the low quadword from the source operand (second operand) to the destination operand (first operand). The source operand is an XMM register and the destination operand is an MMX technology register.

This instruction causes a transition from x87 FPU to MMX technology operation (that is, the x87 FPU top-of-stack pointer is set to 0 and the x87 FPU tag word is set to all 0s [valid]). If this instruction is executed while an x87 FPU floating-point exception is pending, the exception is handled before the MOVDQ2Q instruction is executed.

In 64-bit mode, use of the REX.R prefix permits this instruction to access additional registers (XMM8-XMM15).

Operation

```
DEST ← SRC[63:0];
```

Intel C/C++ Compiler Intrinsic Equivalent

MOVDQ2Q: __m64 _mm_movepi64_pi64 (__m128i a)

SIMD Floating-Point Exceptions

None.

Protected Mode Exceptions

#NM	If CR0.TS[bit 3] = 1.
#UD	If CR0.EM[bit 2] = 1.
	If CR4.OSFXSR[bit 9] = 0.
	If CPUID.01H:EDX.SSE2[bit 26] = 0.
	If the LOCK prefix is used.
#MF	If there is a pending x87 FPU exception.

Real-Address Mode Exceptions

Same exceptions as in protected mode.

Virtual-8086 Mode Exceptions

Same exceptions as in protected mode.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

Same exceptions as in protected mode.

MOVHLPS—Move Packed Single-Precision Floating-Point Values High to Low

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F 12 /r MOVHLPS xmm1, xmm2	RM	V/V	SSE	Move two packed single-precision floating-point values from high quadword of xmm2 to low quadword of xmm1.
VEX.NDS.128.0F.WIG 12 /r VMOVHLPS xmm1, xmm2, xmm3	RVM	V/V	AVX	Merge two packed single-precision floating-point values from high quadword of xmm3 and low quadword of xmm2.
EVEX.NDS.128.0F.W0 12 /r VMOVHLPS xmm1, xmm2, xmm3	RVM	V/V	AVX512F	Merge two packed single-precision floating-point values from high quadword of xmm3 and low quadword of xmm2.

Instruction Operand Encoding[1]

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (w)	ModRM:r/m (r)	NA	NA
RVM	ModRM:reg (w)	vvvv (r)	ModRM:r/m (r)	NA

Description

This instruction cannot be used for memory to register moves.

128-bit two-argument form:

Moves two packed single-precision floating-point values from the high quadword of the second XMM argument (second operand) to the low quadword of the first XMM register (first argument). The quadword at bits 127:64 of the destination operand is left unchanged. Bits (MAXVL-1:128) of the corresponding destination register remain unchanged.

128-bit and EVEX three-argument form

Moves two packed single-precision floating-point values from the high quadword of the third XMM argument (third operand) to the low quadword of the destination (first operand). Copies the high quadword from the second XMM argument (second operand) to the high quadword of the destination (first operand). Bits (MAXVL-1:128) of the corresponding destination register are zeroed.

If VMOVHLPS is encoded with VEX.L or EVEX.L'L= 1, an attempt to execute the instruction encoded with VEX.L or EVEX.L'L= 1 will cause an #UD exception.

Operation

MOVHLPS (128-bit two-argument form)

```
DEST[63:0] ← SRC[127:64]
DEST[MAXVL-1:64] (Unmodified)
```

VMOVHLPS (128-bit three-argument form - VEX & EVEX)

```
DEST[63:0] ← SRC2[127:64]
DEST[127:64] ← SRC1[127:64]
DEST[MAXVL-1:128] ← 0
```

Intel C/C++ Compiler Intrinsic Equivalent

MOVHLPS __m128 _mm_movehl_ps(__m128 a, __m128 b)

SIMD Floating-Point Exceptions

None

1. ModRM.MOD = 011B required

Other Exceptions

Non-EVEX-encoded instruction, see Exceptions Type 7; additionally

#UD	If VEX.L = 1.

EVEX-encoded instruction, see Exceptions Type E7NM.128.

MOVHPD—Move High Packed Double-Precision Floating-Point Value

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 16 /r MOVHPD xmm1, m64	A	V/V	SSE2	Move double-precision floating-point value from m64 to high quadword of xmm1.
VEX.NDS.128.66.0F.WIG 16 /r VMOVHPD xmm2, xmm1, m64	B	V/V	AVX	Merge double-precision floating-point value from m64 and the low quadword of xmm1.
EVEX.NDS.128.66.0F.W1 16 /r VMOVHPD xmm2, xmm1, m64	D	V/V	AVX512F	Merge double-precision floating-point value from m64 and the low quadword of xmm1.
66 0F 17 /r MOVHPD m64, xmm1	C	V/V	SSE2	Move double-precision floating-point value from high quadword of xmm1 to m64.
VEX.128.66.0F.WIG 17 /r VMOVHPD m64, xmm1	C	V/V	AVX	Move double-precision floating-point value from high quadword of xmm1 to m64.
EVEX.128.66.0F.W1 17 /r VMOVHPD m64, xmm1	E	V/V	AVX512F	Move double-precision floating-point value from high quadword of xmm1 to m64.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	NA	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
B	NA	ModRM:reg (w)	VEX.vvvv	ModRM:r/m (r)	NA
C	NA	ModRM:r/m (w)	ModRM:reg (r)	NA	NA
D	Tuple1 Scalar	ModRM:reg (w)	EVEX.vvvv	ModRM:r/m (r)	NA
E	Tuple1 Scalar	ModRM:r/m (w)	ModRM:reg (r)	NA	NA

Description

This instruction cannot be used for register to register or memory to memory moves.

128-bit Legacy SSE load:

Moves a double-precision floating-point value from the source 64-bit memory operand and stores it in the high 64-bits of the destination XMM register. The lower 64bits of the XMM register are preserved. Bits (MAXVL-1:128) of the corresponding destination register are preserved.

VEX.128 & EVEX encoded load:

Loads a double-precision floating-point value from the source 64-bit memory operand (the third operand) and stores it in the upper 64-bits of the destination XMM register (first operand). The low 64-bits from the first source operand (second operand) are copied to the low 64-bits of the destination. Bits (MAXVL-1:128) of the corresponding destination register are zeroed.

128-bit store:

Stores a double-precision floating-point value from the high 64-bits of the XMM register source (second operand) to the 64-bit memory location (first operand).

Note: VMOVHPD (store) (VEX.128.66.0F 17 /r) is legal and has the same behavior as the existing 66 0F 17 store. For VMOVHPD (store) VEX.vvvv and EVEX.vvvv are reserved and must be 1111b otherwise instruction will #UD.

If VMOVHPD is encoded with VEX.L or EVEX.L'L= 1, an attempt to execute the instruction encoded with VEX.L or EVEX.L'L= 1 will cause an #UD exception.

Operation

MOVHPD (128-bit Legacy SSE load)

```
DEST[63:0] (Unmodified)
DEST[127:64] ← SRC[63:0]
DEST[MAXVL-1:128] (Unmodified)
```

VMOVHPD (VEX.128 & EVEX encoded load)

```
DEST[63:0] ← SRC1[63:0]
DEST[127:64] ← SRC2[63:0]
DEST[MAXVL-1:128] ← 0
```

VMOVHPD (store)

```
DEST[63:0] ← SRC[127:64]
```

Intel C/C++ Compiler Intrinsic Equivalent

MOVHPD __m128d _mm_loadh_pd (__m128d a, double *p)
MOVHPD void _mm_storeh_pd (double *p, __m128d a)

SIMD Floating-Point Exceptions

None

Other Exceptions

Non-EVEX-encoded instruction, see Exceptions Type 5; additionally

#UD	If VEX.L = 1.

EVEX-encoded instruction, see Exceptions Type E9NF.

MOVHPS—Move High Packed Single-Precision Floating-Point Values

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F 16 /r MOVHPS xmm1, m64	A	V/V	SSE	Move two packed single-precision floating-point values from m64 to high quadword of xmm1.
VEX.NDS.128.0F.WIG 16 /r VMOVHPS xmm2, xmm1, m64	B	V/V	AVX	Merge two packed single-precision floating-point values from m64 and the low quadword of xmm1.
EVEX.NDS.128.0F.W0 16 /r VMOVHPS xmm2, xmm1, m64	D	V/V	AVX512F	Merge two packed single-precision floating-point values from m64 and the low quadword of xmm1.
NP 0F 17 /r MOVHPS m64, xmm1	C	V/V	SSE	Move two packed single-precision floating-point values from high quadword of xmm1 to m64.
VEX.128.0F.WIG 17 /r VMOVHPS m64, xmm1	C	V/V	AVX	Move two packed single-precision floating-point values from high quadword of xmm1 to m64.
EVEX.128.0F.W0 17 /r VMOVHPS m64, xmm1	E	V/V	AVX512F	Move two packed single-precision floating-point values from high quadword of xmm1 to m64.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	NA	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
B	NA	ModRM:reg (w)	VEX.vvvv	ModRM:r/m (r)	NA
C	NA	ModRM:r/m (w)	ModRM:reg (r)	NA	NA
D	Tuple2	ModRM:reg (w)	EVEX.vvvv	ModRM:r/m (r)	NA
E	Tuple2	ModRM:r/m (w)	ModRM:reg (r)	NA	NA

Description

This instruction cannot be used for register to register or memory to memory moves.

128-bit Legacy SSE load:

Moves two packed single-precision floating-point values from the source 64-bit memory operand and stores them in the high 64-bits of the destination XMM register. The lower 64bits of the XMM register are preserved. Bits (MAXVL-1:128) of the corresponding destination register are preserved.

VEX.128 & EVEX encoded load:

Loads two single-precision floating-point values from the source 64-bit memory operand (the third operand) and stores it in the upper 64-bits of the destination XMM register (first operand). The low 64-bits from the first source operand (the second operand) are copied to the lower 64-bits of the destination. Bits (MAXVL-1:128) of the corresponding destination register are zeroed.

128-bit store:

Stores two packed single-precision floating-point values from the high 64-bits of the XMM register source (second operand) to the 64-bit memory location (first operand).

Note: VMOVHPS (store) (VEX.NDS.128.0F 17 /r) is legal and has the same behavior as the existing 0F 17 store. For VMOVHPS (store) VEX.vvvv and EVEX.vvvv are reserved and must be 1111b otherwise instruction will #UD.

If VMOVHPS is encoded with VEX.L or EVEX.L'L= 1, an attempt to execute the instruction encoded with VEX.L or EVEX.L'L= 1 will cause an #UD exception.

Operation

MOVHPS (128-bit Legacy SSE load)

```
DEST[63:0] (Unmodified)
DEST[127:64] ← SRC[63:0]
DEST[MAXVL-1:128] (Unmodified)
```

VMOVHPS (VEX.128 and EVEX encoded load)

```
DEST[63:0] ← SRC1[63:0]
DEST[127:64] ← SRC2[63:0]
DEST[MAXVL-1:128] ← 0
```

VMOVHPS (store)

```
DEST[63:0] ← SRC[127:64]
```

Intel C/C++ Compiler Intrinsic Equivalent

MOVHPS __m128 _mm_loadh_pi (__m128 a, __m64 *p)
MOVHPS void _mm_storeh_pi (__m64 *p, __m128 a)

SIMD Floating-Point Exceptions

None

Other Exceptions

Non-EVEX-encoded instruction, see Exceptions Type 5; additionally

#UD	If VEX.L = 1.

EVEX-encoded instruction, see Exceptions Type E9NF.

MOVLHPS—Move Packed Single-Precision Floating-Point Values Low to High

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F 16 /r MOVLHPS xmm1, xmm2	RM	V/V	SSE	Move two packed single-precision floating-point values from low quadword of xmm2 to high quadword of xmm1.
VEX.NDS.128.0F.WIG 16 /r VMOVLHPS xmm1, xmm2, xmm3	RVM	V/V	AVX	Merge two packed single-precision floating-point values from low quadword of xmm3 and low quadword of xmm2.
EVEX.NDS.128.0F.W0 16 /r VMOVLHPS xmm1, xmm2, xmm3	RVM	V/V	AVX512F	Merge two packed single-precision floating-point values from low quadword of xmm3 and low quadword of xmm2.

Instruction Operand Encoding[1]

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (w)	ModRM:r/m (r)	NA	NA
RVM	ModRM:reg (w)	vvvv (r)	ModRM:r/m (r)	NA

Description

This instruction cannot be used for memory to register moves.

128-bit two-argument form:

Moves two packed single-precision floating-point values from the low quadword of the second XMM argument (second operand) to the high quadword of the first XMM register (first argument). The low quadword of the destination operand is left unchanged. Bits (MAXVL-1:128) of the corresponding destination register are unmodified.

128-bit three-argument forms:

Moves two packed single-precision floating-point values from the low quadword of the third XMM argument (third operand) to the high quadword of the destination (first operand). Copies the low quadword from the second XMM argument (second operand) to the low quadword of the destination (first operand). Bits (MAXVL-1:128) of the corresponding destination register are zeroed.

If VMOVLHPS is encoded with VEX.L or EVEX.L'L= 1, an attempt to execute the instruction encoded with VEX.L or EVEX.L'L= 1 will cause an #UD exception.

Operation

MOVLHPS (128-bit two-argument form)

```
DEST[63:0] (Unmodified)
DEST[127:64] ← SRC[63:0]
DEST[MAXVL-1:128] (Unmodified)
```

VMOVLHPS (128-bit three-argument form - VEX & EVEX)

```
DEST[63:0] ← SRC1[63:0]
DEST[127:64] ← SRC2[63:0]
DEST[MAXVL-1:128] ← 0
```

Intel C/C++ Compiler Intrinsic Equivalent

MOVLHPS __m128 _mm_movelh_ps(__m128 a, __m128 b)

SIMD Floating-Point Exceptions

None

1. ModRM.MOD = 011B required

Other Exceptions

Non-EVEX-encoded instruction, see Exceptions Type 7; additionally

#UD	If VEX.L = 1.

EVEX-encoded instruction, see Exceptions Type E7NM.128.

MOVLPD—Move Low Packed Double-Precision Floating-Point Value

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 12 /r MOVLPD xmm1, m64	A	V/V	SSE2	Move double-precision floating-point value from m64 to low quadword of xmm1.
VEX.NDS.128.66.0F.WIG 12 /r VMOVLPD xmm2, xmm1, m64	B	V/V	AVX	Merge double-precision floating-point value from m64 and the high quadword of xmm1.
EVEX.NDS.128.66.0F.W1 12 /r VMOVLPD xmm2, xmm1, m64	D	V/V	AVX512F	Merge double-precision floating-point value from m64 and the high quadword of xmm1.
66 0F 13/r MOVLPD m64, xmm1	C	V/V	SSE2	Move double-precision floating-point value from low quadword of xmm1 to m64.
VEX.128.66.0F.WIG 13/r VMOVLPD m64, xmm1	C	V/V	AVX	Move double-precision floating-point value from low quadword of xmm1 to m64.
EVEX.128.66.0F.W1 13/r VMOVLPD m64, xmm1	E	V/V	AVX512F	Move double-precision floating-point value from low quadword of xmm1 to m64.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	NA	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
B	NA	ModRM:r/m (r)	VEX.vvvv	ModRM:r/m (r)	NA
C	NA	ModRM:r/m (w)	ModRM:reg (r)	NA	NA
D	Tuple1 Scalar	ModRM:reg (w)	EVEX.vvvv	ModRM:r/m (r)	NA
E	Tuple1 Scalar	ModRM:r/m (w)	ModRM:reg (r)	NA	NA

Description

This instruction cannot be used for register to register or memory to memory moves.

128-bit Legacy SSE load:

Moves a double-precision floating-point value from the source 64-bit memory operand and stores it in the low 64-bits of the destination XMM register. The upper 64bits of the XMM register are preserved. Bits (MAXVL-1:128) of the corresponding destination register are preserved.

VEX.128 & EVEX encoded load:

Loads a double-precision floating-point value from the source 64-bit memory operand (third operand), merges it with the upper 64-bits of the first source XMM register (second operand), and stores it in the low 128-bits of the destination XMM register (first operand). Bits (MAXVL-1:128) of the corresponding destination register are zeroed.

128-bit store:

Stores a double-precision floating-point value from the low 64-bits of the XMM register source (second operand) to the 64-bit memory location (first operand).

Note: VMOVLPD (store) (VEX.128.66.0F 13 /r) is legal and has the same behavior as the existing 66 0F 13 store. For VMOVLPD (store) VEX.vvvv and EVEX.vvvv are reserved and must be 1111b otherwise instruction will #UD.

If VMOVLPD is encoded with VEX.L or EVEX.L'L= 1, an attempt to execute the instruction encoded with VEX.L or EVEX.L'L= 1 will cause an #UD exception.

Operation

MOVLPD (128-bit Legacy SSE load)

```
DEST[63:0] ← SRC[63:0]
DEST[MAXVL-1:64] (Unmodified)
```

VMOVLPD (VEX.128 & EVEX encoded load)
DEST[63:0] ← SRC2[63:0]
DEST[127:64] ← SRC1[127:64]
DEST[MAXVL-1:128] ← 0

VMOVLPD (store)
DEST[63:0] ← SRC[63:0]

Intel C/C++ Compiler Intrinsic Equivalent

MOVLPD __m128d _mm_loadl_pd (__m128d a, double *p)
MOVLPD void _mm_storel_pd (double *p, __m128d a)

SIMD Floating-Point Exceptions

None

Other Exceptions

Non-EVEX-encoded instruction, see Exceptions Type 5; additionally

#UD	If VEX.L = 1.

EVEX-encoded instruction, see Exceptions Type E9NF.

MOVLPS—Move Low Packed Single-Precision Floating-Point Values

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F 12 /r MOVLPS xmm1, m64	A	V/V	SSE	Move two packed single-precision floating-point values from m64 to low quadword of xmm1.
VEX.NDS.128.0F.WIG 12 /r VMOVLPS xmm2, xmm1, m64	B	V/V	AVX	Merge two packed single-precision floating-point values from m64 and the high quadword of xmm1.
EVEX.NDS.128.0F.W0 12 /r VMOVLPS xmm2, xmm1, m64	D	V/V	AVX512F	Merge two packed single-precision floating-point values from m64 and the high quadword of xmm1.
0F 13/r MOVLPS m64, xmm1	C	V/V	SSE	Move two packed single-precision floating-point values from low quadword of xmm1 to m64.
VEX.128.0F.WIG 13/r VMOVLPS m64, xmm1	C	V/V	AVX	Move two packed single-precision floating-point values from low quadword of xmm1 to m64.
EVEX.128.0F.W0 13/r VMOVLPS m64, xmm1	E	V/V	AVX512F	Move two packed single-precision floating-point values from low quadword of xmm1 to m64.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	NA	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
B	NA	ModRM:reg (w)	VEX.vvvv	ModRM:r/m (r)	NA
C	NA	ModRM:r/m (w)	ModRM:reg (r)	NA	NA
D	Tuple2	ModRM:reg (w)	EVEX.vvvv	ModRM:r/m (r)	NA
E	Tuple2	ModRM:r/m (w)	ModRM:reg (r)	NA	NA

Description

This instruction cannot be used for register to register or memory to memory moves.

128-bit Legacy SSE load:

Moves two packed single-precision floating-point values from the source 64-bit memory operand and stores them in the low 64-bits of the destination XMM register. The upper 64bits of the XMM register are preserved. Bits (MAXVL-1:128) of the corresponding destination register are preserved.

VEX.128 & EVEX encoded load:

Loads two packed single-precision floating-point values from the source 64-bit memory operand (the third operand), merges them with the upper 64-bits of the first source operand (the second operand), and stores them in the low 128-bits of the destination register (the first operand). Bits (MAXVL-1:128) of the corresponding destination register are zeroed.

128-bit store:

Loads two packed single-precision floating-point values from the low 64-bits of the XMM register source (second operand) to the 64-bit memory location (first operand).

Note: VMOVLPS (store) (VEX.128.0F 13 /r) is legal and has the same behavior as the existing 0F 13 store. For VMOVLPS (store) VEX.vvvv and EVEX.vvvv are reserved and must be 1111b otherwise instruction will #UD.

If VMOVLPS is encoded with VEX.L or EVEX.L'L= 1, an attempt to execute the instruction encoded with VEX.L or EVEX.L'L= 1 will cause an #UD exception.

Operation

MOVLPS (128-bit Legacy SSE load)

```
DEST[63:0] ← SRC[63:0]
DEST[MAXVL-1:64] (Unmodified)
```

VMOVLPS (VEX.128 & EVEX encoded load)

```
DEST[63:0] ← SRC2[63:0]
DEST[127:64] ← SRC1[127:64]
DEST[MAXVL-1:128] ← 0
```

VMOVLPS (store)

```
DEST[63:0] ← SRC[63:0]
```

Intel C/C++ Compiler Intrinsic Equivalent

MOVLPS __m128 _mm_loadl_pi (__m128 a, __m64 *p)
MOVLPS void _mm_storel_pi (__m64 *p, __m128 a)

SIMD Floating-Point Exceptions

None

Other Exceptions

Non-EVEX-encoded instruction, see Exceptions Type 5; additionally

#UD	If VEX.L = 1.

EVEX-encoded instruction, see Exceptions Type E9NF.

MOVMSKPD—Extract Packed Double-Precision Floating-Point Sign Mask

Opcode/ Instruction	Op/ En	64/32-bit Mode	CPUID Feature Flag	Description
66 0F 50 /r MOVMSKPD *reg, xmm*	RM	V/V	SSE2	Extract 2-bit sign mask from *xmm* and store in *reg*. The upper bits of *r32* or *r64* are filled with zeros.
VEX.128.66.0F.WIG 50 /r VMOVMSKPD *reg, xmm2*	RM	V/V	AVX	Extract 2-bit sign mask from *xmm2* and store in reg. The upper bits of *r32* or *r64* are zeroed.
VEX.256.66.0F.WIG 50 /r VMOVMSKPD *reg, ymm2*	RM	V/V	AVX	Extract 4-bit sign mask from *ymm2* and store in reg. The upper bits of *r32* or *r64* are zeroed.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

Description

Extracts the sign bits from the packed double-precision floating-point values in the source operand (second operand), formats them into a 2-bit mask, and stores the mask in the destination operand (first operand). The source operand is an XMM register, and the destination operand is a general-purpose register. The mask is stored in the 2 low-order bits of the destination operand. Zero-extend the upper bits of the destination.

In 64-bit mode, the instruction can access additional registers (XMM8-XMM15, R8-R15) when used with a REX.R prefix. The default operand size is 64-bit in 64-bit mode.

128-bit versions: The source operand is a YMM register. The destination operand is a general purpose register.

VEX.256 encoded version: The source operand is a YMM register. The destination operand is a general purpose register.

Note: In VEX-encoded versions, VEX.vvvv is reserved and must be 1111b, otherwise instructions will #UD.

Operation

(V)MOVMSKPD (128-bit versions)

```
DEST[0] ← SRC[63]
DEST[1] ← SRC[127]
IF DEST = r32
    THEN DEST[31:2] ← 0;
    ELSE DEST[63:2] ← 0;
FI
```

VMOVMSKPD (VEX.256 encoded version)

```
DEST[0] ← SRC[63]
DEST[1] ← SRC[127]
DEST[2] ← SRC[191]
DEST[3] ← SRC[255]
IF DEST = r32
    THEN DEST[31:4] ← 0;
    ELSE DEST[63:4] ← 0;
FI
```

Intel C/C++ Compiler Intrinsic Equivalent

MOVMSKPD: int _mm_movemask_pd (__m128d a)

VMOVMSKPD: _mm256_movemask_pd(__m256d a)

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type 7; additionally

#UD	If VEX.vvvv ≠ 1111B.

MOVMSKPS—Extract Packed Single-Precision Floating-Point Sign Mask

Opcode/ Instruction	Op/ En	64/32-bit Mode	CPUID Feature Flag	Description
NP 0F 50 /r MOVMSKPS *reg, xmm*	RM	V/V	SSE	Extract 4-bit sign mask from *xmm* and store in *reg*. The upper bits of *r32* or *r64* are filled with zeros.
VEX.128.0F.WIG 50 /r VMOVMSKPS *reg, xmm2*	RM	V/V	AVX	Extract 4-bit sign mask from *xmm2* and store in reg. The upper bits of *r32* or *r64* are zeroed.
VEX.256.0F.WIG 50 /r VMOVMSKPS *reg, ymm2*	RM	V/V	AVX	Extract 8-bit sign mask from ymm2 and store in reg. The upper bits of *r32* or *r64* are zeroed.

Instruction Operand Encoding[1]

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

Description

Extracts the sign bits from the packed single-precision floating-point values in the source operand (second operand), formats them into a 4- or 8-bit mask, and stores the mask in the destination operand (first operand). The source operand is an XMM or YMM register, and the destination operand is a general-purpose register. The mask is stored in the 4 or 8 low-order bits of the destination operand. The upper bits of the destination operand beyond the mask are filled with zeros.

In 64-bit mode, the instruction can access additional registers (XMM8-XMM15, R8-R15) when used with a REX.R prefix. The default operand size is 64-bit in 64-bit mode.

128-bit versions: The source operand is a YMM register. The destination operand is a general purpose register.

VEX.256 encoded version: The source operand is a YMM register. The destination operand is a general purpose register.

Note: In VEX-encoded versions, VEX.vvvv is reserved and must be 1111b, otherwise instructions will #UD.

Operation

```
DEST[0] ← SRC[31];
DEST[1] ← SRC[63];
DEST[2] ← SRC[95];
DEST[3] ← SRC[127];

IF DEST = r32
    THEN DEST[31:4] ← ZeroExtend;
    ELSE DEST[63:4] ← ZeroExtend;
FI;
```

1. ModRM.MOD = 011B required

(V)MOVMSKPS (128-bit version)

```
DEST[0] ← SRC[31]
DEST[1] ← SRC[63]
DEST[2] ← SRC[95]
DEST[3] ← SRC[127]
IF DEST = r32
    THEN DEST[31:4] ← 0;
    ELSE DEST[63:4] ← 0;
FI
```

VMOVMSKPS (VEX.256 encoded version)

```
DEST[0] ← SRC[31]
DEST[1] ← SRC[63]
DEST[2] ← SRC[95]
DEST[3] ← SRC[127]
DEST[4] ← SRC[159]
DEST[5] ← SRC[191]
DEST[6] ← SRC[223]
DEST[7] ← SRC[255]
IF DEST = r32
    THEN DEST[31:8] ← 0;
    ELSE DEST[63:8] ← 0;
FI
```

Intel C/C++ Compiler Intrinsic Equivalent

int _mm_movemask_ps(__m128 a)

int _mm256_movemask_ps(__m256 a)

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Exceptions Type 7; additionally

#UD	If VEX.vvvv ≠ 1111B.

MOVNTDQA—Load Double Quadword Non-Temporal Aligned Hint

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 38 2A /r MOVNTDQA xmm1, m128	A	V/V	SSE4_1	Move double quadword from m128 to xmm1 using non-temporal hint if WC memory type.
VEX.128.66.0F38.WIG 2A /r VMOVNTDQA xmm1, m128	A	V/V	AVX	Move double quadword from m128 to xmm using non-temporal hint if WC memory type.
VEX.256.66.0F38.WIG 2A /r VMOVNTDQA ymm1, m256	A	V/V	AVX2	Move 256-bit data from m256 to ymm using non-temporal hint if WC memory type.
EVEX.128.66.0F38.W0 2A /r VMOVNTDQA xmm1, m128	B	V/V	AVX512VL AVX512F	Move 128-bit data from m128 to xmm using non-temporal hint if WC memory type.
EVEX.256.66.0F38.W0 2A /r VMOVNTDQA ymm1, m256	B	V/V	AVX512VL AVX512F	Move 256-bit data from m256 to ymm using non-temporal hint if WC memory type.
EVEX.512.66.0F38.W0 2A /r VMOVNTDQA zmm1, m512	B	V/V	AVX512F	Move 512-bit data from m512 to zmm using non-temporal hint if WC memory type.

Instruction Operand Encoding[1]

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	NA	ModRM:reg (w)	ModRM:r/m (r)	NA	NA
B	Full Vector Mem	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

Description

MOVNTDQA loads a double quadword from the source operand (second operand) to the destination operand (first operand) using a non-temporal hint if the memory source is WC (write combining) memory type. For WC memory type, the nontemporal hint may be implemented by loading a temporary internal buffer with the equivalent of an aligned cache line without filling this data to the cache. Any memory-type aliased lines in the cache will be snooped and flushed. Subsequent MOVNTDQA reads to unread portions of the WC cache line will receive data from the temporary internal buffer if data is available. The temporary internal buffer may be flushed by the processor at any time for any reason, for example:

- A load operation other than a MOVNTDQA which references memory already resident in a temporary internal buffer.
- A non-WC reference to memory already resident in a temporary internal buffer.
- Interleaving of reads and writes to a single temporary internal buffer.
- Repeated (V)MOVNTDQA loads of a particular 16-byte item in a streaming line.
- Certain micro-architectural conditions including resource shortages, detection of

a mis-speculation condition, and various fault conditions

The non-temporal hint is implemented by using a write combining (WC) memory type protocol when reading the data from memory. Using this protocol, the processor

does not read the data into the cache hierarchy, nor does it fetch the corresponding cache line from memory into the cache hierarchy. The memory type of the region being read can override the non-temporal hint, if the memory address specified for the non-temporal read is not a WC memory region. Information on non-temporal reads and writes can be found in "Caching of Temporal vs. Non-Temporal Data" in Chapter 10 in the Intel® 64 and IA-32 Architecture Software Developer's Manual, Volume 3A.

Because the WC protocol uses a weakly-ordered memory consistency model, a fencing operation implemented with a MFENCE instruction should be used in conjunction with MOVNTDQA instructions if multiple processors might use different memory types for the referenced memory locations or to synchronize reads of a processor with writes by other agents in the system. A processor's implementation of the streaming load hint does not override the effective memory type, but the implementation of the hint is processor dependent. For example, a processor implementa-

1. ModRM.MOD = 011B required

tion may choose to ignore the hint and process the instruction as a normal MOVDQA for any memory type. Alternatively, another implementation may optimize cache reads generated by MOVNTDQA on WB memory type to reduce cache evictions.

The 128-bit (V)MOVNTDQA addresses must be 16-byte aligned or the instruction will cause a #GP.

The 256-bit VMOVNTDQA addresses must be 32-byte aligned or the instruction will cause a #GP.

The 512-bit VMOVNTDQA addresses must be 64-byte aligned or the instruction will cause a #GP.

Operation

MOVNTDQA (128bit- Legacy SSE form)

```
DEST ←SRC
DEST[MAXVL-1:128] (Unmodified)
```

VMOVNTDQA (VEX.128 and EVEX.128 encoded form)

```
DEST ← SRC
DEST[MAXVL-1:128] ← 0
```

VMOVNTDQA (VEX.256 and EVEX.256 encoded forms)

```
DEST[255:0] ← SRC[255:0]
DEST[MAXVL-1:256] ← 0
```

VMOVNTDQA (EVEX.512 encoded form)

```
DEST[511:0] ← SRC[511:0]
DEST[MAXVL-1:512] ← 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VMOVNTDQA __m512i _mm512_stream_load_si512(void * p);
MOVNTDQA __m128i _mm_stream_load_si128 (__m128i *p);
VMOVNTDQA __m256i _mm_stream_load_si256 (__m256i *p);

SIMD Floating-Point Exceptions

None

Other Exceptions

Non-EVEX-encoded instruction, see Exceptions Type1;

EVEX-encoded instruction, see Exceptions Type E1NF.

#UD	If VEX.vvvv != 1111B or EVEX.vvvv != 1111B.

MOVNTDQ—Store Packed Integers Using Non-Temporal Hint

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F E7 /r MOVNTDQ m128, xmm1	A	V/V	SSE2	Move packed integer values in xmm1 to m128 using non-temporal hint.
VEX.128.66.0F.WIG E7 /r VMOVNTDQ m128, xmm1	A	V/V	AVX	Move packed integer values in xmm1 to m128 using non-temporal hint.
VEX.256.66.0F.WIG E7 /r VMOVNTDQ m256, ymm1	A	V/V	AVX	Move packed integer values in ymm1 to m256 using non-temporal hint.
EVEX.128.66.0F.W0 E7 /r VMOVNTDQ m128, xmm1	B	V/V	AVX512VL AVX512F	Move packed integer values in xmm1 to m128 using non-temporal hint.
EVEX.256.66.0F.W0 E7 /r VMOVNTDQ m256, ymm1	B	V/V	AVX512VL AVX512F	Move packed integer values in zmm1 to m256 using non-temporal hint.
EVEX.512.66.0F.W0 E7 /r VMOVNTDQ m512, zmm1	B	V/V	AVX512F	Move packed integer values in zmm1 to m512 using non-temporal hint.

Instruction Operand Encoding[1]

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	NA	ModRM:r/m (w)	ModRM:reg (r)	NA	NA
B	Full Vector Mem	ModRM:r/m (w)	ModRM:reg (r)	NA	NA

Description

Moves the packed integers in the source operand (second operand) to the destination operand (first operand) using a non-temporal hint to prevent caching of the data during the write to memory. The source operand is an XMM register, YMM register or ZMM register, which is assumed to contain integer data (packed bytes, words, double-words, or quadwords). The destination operand is a 128-bit, 256-bit or 512-bit memory location. The memory operand must be aligned on a 16-byte (128-bit version), 32-byte (VEX.256 encoded version) or 64-byte (512-bit version) boundary otherwise a general-protection exception (#GP) will be generated.

The non-temporal hint is implemented by using a write combining (WC) memory type protocol when writing the data to memory. Using this protocol, the processor does not write the data into the cache hierarchy, nor does it fetch the corresponding cache line from memory into the cache hierarchy. The memory type of the region being written to can override the non-temporal hint, if the memory address specified for the non-temporal store is in an uncacheable (UC) or write protected (WP) memory region. For more information on non-temporal stores, see "Caching of Temporal vs. Non-Temporal Data" in Chapter 10 in the IA-32 Intel Architecture Software Developer's Manual, Volume 1.

Because the WC protocol uses a weakly-ordered memory consistency model, a fencing operation implemented with the SFENCE or MFENCE instruction should be used in conjunction with VMOVNTDQ instructions if multiple processors might use different memory types to read/write the destination memory locations.

Note: VEX.vvvv and EVEX.vvvv are reserved and must be 1111b, VEX.L must be 0; otherwise instructions will #UD.

Operation

VMOVNTDQ(EVEX encoded versions)

```
VL = 128, 256, 512
DEST[VL-1:0] ← SRC[VL-1:0]
DEST[MAXVL-1:VL] ← 0
```

1. ModRM.MOD = 011B required

MOVNTDQ (Legacy and VEX versions)
DEST ← SRC

Intel C/C++ Compiler Intrinsic Equivalent

VMOVNTDQ void _mm512_stream_si512(void * p, __m512i a);
VMOVNTDQ void _mm256_stream_si256 (__m256i * p, __m256i a);
MOVNTDQ void _mm_stream_si128 (__m128i * p, __m128i a);

SIMD Floating-Point Exceptions

None

Other Exceptions

Non-EVEX-encoded instruction, see Exceptions Type1.SSE2;
EVEX-encoded instruction, see Exceptions Type E1NF.

#UD	If VEX.vvvv != 1111B or EVEX.vvvv != 1111B.

MOVNTI—Store Doubleword Using Non-Temporal Hint

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
NP 0F C3 /r	MOVNTI *m32, r32*	MR	Valid	Valid	Move doubleword from *r32* to *m32* using non-temporal hint.
NP REX.W + 0F C3 /r	MOVNTI *m64, r64*	MR	Valid	N.E.	Move quadword from *r64* to *m64* using non-temporal hint.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
MR	ModRM:r/m (w)	ModRM:reg (r)	NA	NA

Description

Moves the doubleword integer in the source operand (second operand) to the destination operand (first operand) using a non-temporal hint to minimize cache pollution during the write to memory. The source operand is a general-purpose register. The destination operand is a 32-bit memory location.

The non-temporal hint is implemented by using a write combining (WC) memory type protocol when writing the data to memory. Using this protocol, the processor does not write the data into the cache hierarchy, nor does it fetch the corresponding cache line from memory into the cache hierarchy. The memory type of the region being written to can override the non-temporal hint, if the memory address specified for the non-temporal store is in an uncacheable (UC) or write protected (WP) memory region. For more information on non-temporal stores, see "Caching of Temporal vs. Non-Temporal Data" in Chapter 10 in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*.

Because the WC protocol uses a weakly-ordered memory consistency model, a fencing operation implemented with the SFENCE or MFENCE instruction should be used in conjunction with MOVNTI instructions if multiple processors might use different memory types to read/write the destination memory locations.

In 64-bit mode, the instruction's default operation size is 32 bits. Use of the REX.R prefix permits access to additional registers (R8-R15). Use of the REX.W prefix promotes operation to 64 bits. See the summary chart at the beginning of this section for encoding data and limits.

Operation

```
DEST ← SRC;
```

Intel C/C++ Compiler Intrinsic Equivalent

MOVNTI: void _mm_stream_si32 (int *p, int a)

MOVNTI: void _mm_stream_si64(__int64 *p, __int64 a)

SIMD Floating-Point Exceptions

None.

Protected Mode Exceptions

#GP(0)	For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.
#SS(0)	For an illegal address in the SS segment.
#PF(fault-code)	For a page fault.
#UD	If CPUID.01H:EDX.SSE2[bit 26] = 0.
	If the LOCK prefix is used.

Real-Address Mode Exceptions

#GP	If any part of the operand lies outside the effective address space from 0 to FFFFH.
#UD	If CPUID.01H:EDX.SSE2[bit 26] = 0.
	If the LOCK prefix is used.

Virtual-8086 Mode Exceptions

Same exceptions as in real address mode.

#PF(fault-code)	For a page fault.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form.
#PF(fault-code)	For a page fault.
#UD	If CPUID.01H:EDX.SSE2[bit 26] = 0.
	If the LOCK prefix is used.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

MOVNTPD—Store Packed Double-Precision Floating-Point Values Using Non-Temporal Hint

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 2B /r MOVNTPD m128, xmm1	A	V/V	SSE2	Move packed double-precision values in xmm1 to m128 using non-temporal hint.
VEX.128.66.0F.WIG 2B /r VMOVNTPD m128, xmm1	A	V/V	AVX	Move packed double-precision values in xmm1 to m128 using non-temporal hint.
VEX.256.66.0F.WIG 2B /r VMOVNTPD m256, ymm1	A	V/V	AVX	Move packed double-precision values in ymm1 to m256 using non-temporal hint.
EVEX.128.66.0F.W1 2B /r VMOVNTPD m128, xmm1	B	V/V	AVX512VL AVX512F	Move packed double-precision values in xmm1 to m128 using non-temporal hint.
EVEX.256.66.0F.W1 2B /r VMOVNTPD m256, ymm1	B	V/V	AVX512VL AVX512F	Move packed double-precision values in ymm1 to m256 using non-temporal hint.
EVEX.512.66.0F.W1 2B /r VMOVNTPD m512, zmm1	B	V/V	AVX512F	Move packed double-precision values in zmm1 to m512 using non-temporal hint.

Instruction Operand Encoding[1]

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	NA	ModRM:r/m (w)	ModRM:reg (r)	NA	NA
B	Full Vector Mem	ModRM:r/m (w)	ModRM:reg (r)	NA	NA

Description

Moves the packed double-precision floating-point values in the source operand (second operand) to the destination operand (first operand) using a non-temporal hint to prevent caching of the data during the write to memory. The source operand is an XMM register, YMM register or ZMM register, which is assumed to contain packed double-precision, floating-pointing data. The destination operand is a 128-bit, 256-bit or 512-bit memory location. The memory operand must be aligned on a 16-byte (128-bit version), 32-byte (VEX.256 encoded version) or 64-byte (EVEX.512 encoded version) boundary otherwise a general-protection exception (#GP) will be generated.

The non-temporal hint is implemented by using a write combining (WC) memory type protocol when writing the data to memory. Using this protocol, the processor does not write the data into the cache hierarchy, nor does it fetch the corresponding cache line from memory into the cache hierarchy. The memory type of the region being written to can override the non-temporal hint, if the memory address specified for the non-temporal store is in an uncacheable (UC) or write protected (WP) memory region. For more information on non-temporal stores, see "Caching of Temporal vs. Non-Temporal Data" in Chapter 10 in the IA-32 Intel Architecture Software Developer's Manual, Volume 1.

Because the WC protocol uses a weakly-ordered memory consistency model, a fencing operation implemented with the SFENCE or MFENCE instruction should be used in conjunction with MOVNTPD instructions if multiple processors might use different memory types to read/write the destination memory locations.

Note: VEX.vvvv and EVEX.vvvv are reserved and must be 1111b, VEX.L must be 0; otherwise instructions will #UD.

Operation

VMOVNTPD (EVEX encoded versions)

```
VL = 128, 256, 512
DEST[VL-1:0] ← SRC[VL-1:0]
DEST[MAXVL-1:VL] ← 0
```

1. ModRM.MOD = 011B required

MOVNTPD (Legacy and VEX versions)
DEST ← SRC

Intel C/C++ Compiler Intrinsic Equivalent

VMOVNTPD void _mm512_stream_pd(double * p, __m512d a);
VMOVNTPD void _mm256_stream_pd (double * p, __m256d a);
MOVNTPD void _mm_stream_pd (double * p, __m128d a);

SIMD Floating-Point Exceptions

None

Other Exceptions

Non-EVEX-encoded instruction, see Exceptions Type1.SSE2;
EVEX-encoded instruction, see Exceptions Type E1NF.

#UD	If VEX.vvvv != 1111B or EVEX.vvvv != 1111B.

MOVNTPS—Store Packed Single-Precision Floating-Point Values Using Non-Temporal Hint

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F 2B /r MOVNTPS m128, xmm1	A	V/V	SSE	Move packed single-precision values xmm1 to mem using non-temporal hint.
VEX.128.0F.WIG 2B /r VMOVNTPS m128, xmm1	A	V/V	AVX	Move packed single-precision values xmm1 to mem using non-temporal hint.
VEX.256.0F.WIG 2B /r VMOVNTPS m256, ymm1	A	V/V	AVX	Move packed single-precision values ymm1 to mem using non-temporal hint.
EVEX.128.0F.W0 2B /r VMOVNTPS m128, xmm1	B	V/V	AVX512VL AVX512F	Move packed single-precision values in xmm1 to m128 using non-temporal hint.
EVEX.256.0F.W0 2B /r VMOVNTPS m256, ymm1	B	V/V	AVX512VL AVX512F	Move packed single-precision values in ymm1 to m256 using non-temporal hint.
EVEX.512.0F.W0 2B /r VMOVNTPS m512, zmm1	B	V/V	AVX512F	Move packed single-precision values in zmm1 to m512 using non-temporal hint.

Instruction Operand Encoding[1]

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	NA	ModRM:r/m (w)	ModRM:reg (r)	NA	NA
B	Full Vector Mem	ModRM:r/m (w)	ModRM:reg (r)	NA	NA

Description

Moves the packed single-precision floating-point values in the source operand (second operand) to the destination operand (first operand) using a non-temporal hint to prevent caching of the data during the write to memory. The source operand is an XMM register, YMM register or ZMM register, which is assumed to contain packed single-precision, floating-pointing. The destination operand is a 128-bit, 256-bit or 512-bit memory location. The memory operand must be aligned on a 16-byte (128-bit version), 32-byte (VEX.256 encoded version) or 64-byte (EVEX.512 encoded version) boundary otherwise a general-protection exception (#GP) will be generated.

The non-temporal hint is implemented by using a write combining (WC) memory type protocol when writing the data to memory. Using this protocol, the processor does not write the data into the cache hierarchy, nor does it fetch the corresponding cache line from memory into the cache hierarchy. The memory type of the region being written to can override the non-temporal hint, if the memory address specified for the non-temporal store is in an uncacheable (UC) or write protected (WP) memory region. For more information on non-temporal stores, see "Caching of Temporal vs. Non-Temporal Data" in Chapter 10 in the IA-32 Intel Architecture Software Developer's Manual, Volume 1.

Because the WC protocol uses a weakly-ordered memory consistency model, a fencing operation implemented with the SFENCE or MFENCE instruction should be used in conjunction with MOVNTPS instructions if multiple processors might use different memory types to read/write the destination memory locations.

Note: VEX.vvvv and EVEX.vvvv are reserved and must be 1111b otherwise instructions will #UD.

Operation

VMOVNTPS (EVEX encoded versions)

VL = 128, 256, 512

DEST[VL-1:0] ← SRC[VL-1:0]

DEST[MAXVL-1:VL] ← 0

1. ModRM.MOD = 011B required

MOVNTPS
DEST ← SRC

Intel C/C++ Compiler Intrinsic Equivalent

VMOVNTPS void _mm512_stream_ps(float * p, __m512d a);
MOVNTPS void _mm_stream_ps (float * p, __m128d a);
VMOVNTPS void _mm256_stream_ps (float * p, __m256 a);

SIMD Floating-Point Exceptions

None

Other Exceptions

Non-EVEX-encoded instruction, see Exceptions Type1.SSE; additionally
EVEX-encoded instruction, see Exceptions Type E1NF.

#UD	If VEX.vvvv != 1111B or EVEX.vvvv != 1111B.

MOVNTQ—Store of Quadword Using Non-Temporal Hint

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
NP 0F E7 /r	MOVNTQ *m64, mm*	MR	Valid	Valid	Move quadword from *mm* to *m64* using non-temporal hint.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
MR	ModRM:r/m (w)	ModRM:reg (r)	NA	NA

Description

Moves the quadword in the source operand (second operand) to the destination operand (first operand) using a non-temporal hint to minimize cache pollution during the write to memory. The source operand is an MMX technology register, which is assumed to contain packed integer data (packed bytes, words, or doublewords). The destination operand is a 64-bit memory location.

The non-temporal hint is implemented by using a write combining (WC) memory type protocol when writing the data to memory. Using this protocol, the processor does not write the data into the cache hierarchy, nor does it fetch the corresponding cache line from memory into the cache hierarchy. The memory type of the region being written to can override the non-temporal hint, if the memory address specified for the non-temporal store is in an uncacheable (UC) or write protected (WP) memory region. For more information on non-temporal stores, see "Caching of Temporal vs. Non-Temporal Data" in Chapter 10 in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*.

Because the WC protocol uses a weakly-ordered memory consistency model, a fencing operation implemented with the SFENCE or MFENCE instruction should be used in conjunction with MOVNTQ instructions if multiple processors might use different memory types to read/write the destination memory locations.

This instruction's operation is the same in non-64-bit modes and 64-bit mode.

Operation

```
DEST ← SRC;
```

Intel C/C++ Compiler Intrinsic Equivalent

MOVNTQ: void _mm_stream_pi(__m64 * p, __m64 a)

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Table 22-8, "Exception Conditions for Legacy SIMD/MMX Instructions without FP Exception," in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*.

MOVQ—Move Quadword

Opcode/ Instruction	Op/ En	64/32-bit Mode	CPUID Feature Flag	Description
NP 0F 6F /r MOVQ *mm, mm/m64*	A	V/V	MMX	Move quadword from *mm/m64* to *mm*.
NP 0F 7F /r MOVQ *mm/m64, mm*	B	V/V	MMX	Move quadword from *mm* to *mm/m64*.
F3 0F 7E /r MOVQ *xmm1, xmm2/m64*	A	V/V	SSE2	Move quadword from *xmm2/mem64* to *xmm1*.
VEX.128.F3.0F.WIG 7E /r VMOVQ *xmm1, xmm2/m64*	A	V/V	AVX	Move quadword from *xmm2* to *xmm1*.
EVEX.128.F3.0F.W1 7E /r VMOVQ xmm1, xmm2/m64	C	V/V	AVX512F	Move quadword from xmm2/m64 to xmm1.
66 0F D6 /r MOVQ *xmm2/m64, xmm1*	B	V/V	SSE2	Move quadword from *xmm1* to *xmm2/mem64*.
VEX.128.66.0F.WIG D6 /r VMOVQ *xmm1/m64, xmm2*	B	V/V	AVX	Move quadword from *xmm2* register to *xmm1/m64*.
EVEX.128.66.0F.W1 D6 /r VMOVQ xmm1/m64, xmm2	D	V/V	AVX512F	Move quadword from xmm2 register to xmm1/m64.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	NA	ModRM:reg (w)	ModRM:r/m (r)	NA	NA
B	NA	ModRM:r/m (w)	ModRM:reg (r)	NA	NA
C	Tuple1 Scalar	ModRM:reg (w)	ModRM:r/m (r)	NA	NA
D	Tuple1 Scalar	ModRM:r/m (w)	ModRM:reg (r)	NA	NA

Description

Copies a quadword from the source operand (second operand) to the destination operand (first operand). The source and destination operands can be MMX technology registers, XMM registers, or 64-bit memory locations. This instruction can be used to move a quadword between two MMX technology registers or between an MMX technology register and a 64-bit memory location, or to move data between two XMM registers or between an XMM register and a 64-bit memory location. The instruction cannot be used to transfer data between memory locations.

When the source operand is an XMM register, the low quadword is moved; when the destination operand is an XMM register, the quadword is stored to the low quadword of the register, and the high quadword is cleared to all 0s.

In 64-bit mode and if not encoded using VEX/EVEX, use of the REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

Note: VEX.vvvv and EVEX.vvvv are reserved and must be 1111b, otherwise instructions will #UD.

If VMOVQ is encoded with VEX.L= 1, an attempt to execute the instruction encoded with VEX.L= 1 will cause an #UD exception.

Operation

MOVQ instruction when operating on MMX technology registers and memory locations

```
DEST ← SRC;
```

MOVQ instruction when source and destination operands are XMM registers

```
DEST[63:0] ← SRC[63:0];
DEST[127:64] ← 0000000000000000H;
```

MOVQ instruction when source operand is XMM register and destination
operand is memory location:

```
DEST ← SRC[63:0];
```

MOVQ instruction when source operand is memory location and destination
operand is XMM register:

```
DEST[63:0] ← SRC;
DEST[127:64] ← 0000000000000000H;
```

VMOVQ (VEX.NDS.128.F3.0F 7E) with XMM register source and destination

```
DEST[63:0] ← SRC[63:0]
DEST[MAXVL-1:64] ← 0
```

VMOVQ (VEX.128.66.0F D6) with XMM register source and destination

```
DEST[63:0] ← SRC[63:0]
DEST[MAXVL-1:64] ← 0
```

VMOVQ (7E - EVEX encoded version) with XMM register source and destination

```
DEST[63:0] ← SRC[63:0]
DEST[MAXVL-1:64] ← 0
```

VMOVQ (D6 - EVEX encoded version) with XMM register source and destination

```
DEST[63:0] ← SRC[63:0]
DEST[MAXVL-1:64] ← 0
```

VMOVQ (7E) with memory source

```
DEST[63:0] ← SRC[63:0]
DEST[MAXVL-1:64] ← 0
```

VMOVQ (7E - EVEX encoded version) with memory source

```
DEST[63:0] ← SRC[63:0]
DEST[:MAXVL-1:64] ← 0
```

VMOVQ (D6) with memory dest

```
DEST[63:0] ← SRC2[63:0]
```

Flags Affected

None.

Intel C/C++ Compiler Intrinsic Equivalent

VMOVQ __m128i _mm_loadu_si64(void * s);
VMOVQ void _mm_storeu_si64(void * d, __m128i s);

MOVQ m128i _mm_mov_epi64(__m128i a)

SIMD Floating-Point Exceptions

None

Other Exceptions

See Table 22-8, "Exception Conditions for Legacy SIMD/MMX Instructions without FP Exception," in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3B*.

MOVQ2DQ—Move Quadword from MMX Technology to XMM Register

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
F3 0F D6 /r	MOVQ2DQ *xmm, mm*	RM	Valid	Valid	Move quadword from *mmx* to low quadword of *xmm*.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

Description

Moves the quadword from the source operand (second operand) to the low quadword of the destination operand (first operand). The source operand is an MMX technology register and the destination operand is an XMM register.

This instruction causes a transition from x87 FPU to MMX technology operation (that is, the x87 FPU top-of-stack pointer is set to 0 and the x87 FPU tag word is set to all 0s [valid]). If this instruction is executed while an x87 FPU floating-point exception is pending, the exception is handled before the MOVQ2DQ instruction is executed.

In 64-bit mode, use of the REX.R prefix permits this instruction to access additional registers (XMM8-XMM15).

Operation

```
DEST[63:0] ← SRC[63:0];
DEST[127:64] ← 0000000000000000H;
```

Intel C/C++ Compiler Intrinsic Equivalent

MOVQ2DQ: __128i _mm_movpi64_pi64 (__m64 a)

SIMD Floating-Point Exceptions

None.

Protected Mode Exceptions

#NM	If CR0.TS[bit 3] = 1.
#UD	If CR0.EM[bit 2] = 1.
	If CR4.OSFXSR[bit 9] = 0.
	If CPUID.01H:EDX.SSE2[bit 26] = 0.
	If the LOCK prefix is used.
#MF	If there is a pending x87 FPU exception.

Real-Address Mode Exceptions

Same exceptions as in protected mode.

Virtual-8086 Mode Exceptions

Same exceptions as in protected mode.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

Same exceptions as in protected mode.

MOVS/MOVSB/MOVSW/MOVSD/MOVSQ—Move Data from String to String

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
A4	MOVS *m8, m8*	ZO	Valid	Valid	For legacy mode, Move byte from address DS:(E)SI to ES:(E)DI. For 64-bit mode move byte from address (R|E)SI to (R|E)DI.
A5	MOVS *m16, m16*	ZO	Valid	Valid	For legacy mode, move word from address DS:(E)SI to ES:(E)DI. For 64-bit mode move word at address (R|E)SI to (R|E)DI.
A5	MOVS *m32, m32*	ZO	Valid	Valid	For legacy mode, move dword from address DS:(E)SI to ES:(E)DI. For 64-bit mode move dword from address (R|E)SI to (R|E)DI.
REX.W + A5	MOVS *m64, m64*	ZO	Valid	N.E.	Move qword from address (R|E)SI to (R|E)DI.
A4	MOVSB	ZO	Valid	Valid	For legacy mode, Move byte from address DS:(E)SI to ES:(E)DI. For 64-bit mode move byte from address (R|E)SI to (R|E)DI.
A5	MOVSW	ZO	Valid	Valid	For legacy mode, move word from address DS:(E)SI to ES:(E)DI. For 64-bit mode move word at address (R|E)SI to (R|E)DI.
A5	MOVSD	ZO	Valid	Valid	For legacy mode, move dword from address DS:(E)SI to ES:(E)DI. For 64-bit mode move dword from address (R|E)SI to (R|E)DI.
REX.W + A5	MOVSQ	ZO	Valid	N.E.	Move qword from address (R|E)SI to (R|E)DI.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
ZO	NA	NA	NA	NA

Description

Moves the byte, word, or doubleword specified with the second operand (source operand) to the location specified with the first operand (destination operand). Both the source and destination operands are located in memory. The address of the source operand is read from the DS:ESI or the DS:SI registers (depending on the address-size attribute of the instruction, 32 or 16, respectively). The address of the destination operand is read from the ES:EDI or the ES:DI registers (again depending on the address-size attribute of the instruction). The DS segment may be overridden with a segment override prefix, but the ES segment cannot be overridden.

At the assembly-code level, two forms of this instruction are allowed: the "explicit-operands" form and the "no-operands" form. The explicit-operands form (specified with the MOVS mnemonic) allows the source and destination operands to be specified explicitly. Here, the source and destination operands should be symbols that indicate the size and location of the source value and the destination, respectively. This explicit-operands form is provided to allow documentation; however, note that the documentation provided by this form can be misleading. That is, the source and destination operand symbols must specify the correct **type** (size) of the operands (bytes, words, or doublewords), but they do not have to specify the correct **location**. The locations of the source and destination operands are always specified by the DS:(E)SI and ES:(E)DI registers, which must be loaded correctly before the move string instruction is executed.

The no-operands form provides "short forms" of the byte, word, and doubleword versions of the MOVS instructions. Here also DS:(E)SI and ES:(E)DI are assumed to be the source and destination operands, respectively. The size of the source and destination operands is selected with the mnemonic: MOVSB (byte move), MOVSW (word move), or MOVSD (doubleword move).

After the move operation, the (E)SI and (E)DI registers are incremented or decremented automatically according to the setting of the DF flag in the EFLAGS register. (If the DF flag is 0, the (E)SI and (E)DI register are incre-

mented; if the DF flag is 1, the (E)SI and (E)DI registers are decremented.) The registers are incremented or decremented by 1 for byte operations, by 2 for word operations, or by 4 for doubleword operations.

NOTE

To improve performance, more recent processors support modifications to the processor's operation during the string store operations initiated with MOVS and MOVSB. See Section 7.3.9.3 in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1* for additional information on fast-string operation.

The MOVS, MOVSB, MOVSW, and MOVSD instructions can be preceded by the REP prefix (see "REP/REPE/REPZ /REPNE/REPNZ—Repeat String Operation Prefix" for a description of the REP prefix) for block moves of ECX bytes, words, or doublewords.

In 64-bit mode, the instruction's default address size is 64 bits, 32-bit address size is supported using the prefix 67H. The 64-bit addresses are specified by RSI and RDI; 32-bit address are specified by ESI and EDI. Use of the REX.W prefix promotes doubleword operation to 64 bits. See the summary chart at the beginning of this section for encoding data and limits.

Operation

```
DEST ← SRC;

Non-64-bit Mode:

IF (Byte move)
    THEN IF DF = 0
        THEN
            (E)SI ← (E)SI + 1;
            (E)DI ← (E)DI + 1;
        ELSE
            (E)SI ← (E)SI – 1;
            (E)DI ← (E)DI – 1;
        FI;
    ELSE IF (Word move)
        THEN IF DF = 0
            (E)SI ← (E)SI + 2;
            (E)DI ← (E)DI + 2;
            FI;
        ELSE
            (E)SI ← (E)SI – 2;
            (E)DI ← (E)DI – 2;
        FI;
    ELSE IF (Doubleword move)
        THEN IF DF = 0
            (E)SI ← (E)SI + 4;
            (E)DI ← (E)DI + 4;
            FI;
        ELSE
            (E)SI ← (E)SI – 4;
            (E)DI ← (E)DI – 4;
        FI;
FI;
64-bit Mode:

IF (Byte move)
    THEN IF DF = 0
        THEN
```

```
            (R|E)SI ← (R|E)SI + 1;
            (R|E)DI ← (R|E)DI + 1;
        ELSE
            (R|E)SI ← (R|E)SI – 1;
            (R|E)DI ← (R|E)DI – 1;
        FI;
    ELSE IF (Word move)
        THEN IF DF = 0
            (R|E)SI ← (R|E)SI + 2;
            (R|E)DI ← (R|E)DI + 2;
            FI;
        ELSE
            (R|E)SI ← (R|E)SI – 2;
            (R|E)DI ← (R|E)DI – 2;
        FI;
    ELSE IF (Doubleword move)
        THEN IF DF = 0
            (R|E)SI ← (R|E)SI + 4;
            (R|E)DI ← (R|E)DI + 4;
            FI;
        ELSE
            (R|E)SI ← (R|E)SI – 4;
            (R|E)DI ← (R|E)DI – 4;
        FI;
    ELSE IF (Quadword move)
        THEN IF DF = 0
            (R|E)SI ← (R|E)SI + 8;
            (R|E)DI ← (R|E)DI + 8;
            FI;
        ELSE
            (R|E)SI ← (R|E)SI – 8;
            (R|E)DI ← (R|E)DI – 8;
        FI;
FI;
```

Flags Affected

None

Protected Mode Exceptions

#GP(0)	If the destination is located in a non-writable segment.
	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
	If the DS, ES, FS, or GS register contains a NULL segment selector.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If the LOCK prefix is used.

Real-Address Mode Exceptions

#GP	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS	If a memory operand effective address is outside the SS segment limit.
#UD	If the LOCK prefix is used.

Virtual-8086 Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.
#UD	If the LOCK prefix is used.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If the LOCK prefix is used.

MOVSD—Move or Merge Scalar Double-Precision Floating-Point Value

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
F2 0F 10 /r MOVSD xmm1, xmm2	A	V/V	SSE2	Move scalar double-precision floating-point value from xmm2 to xmm1 register.
F2 0F 10 /r MOVSD xmm1, m64	A	V/V	SSE2	Load scalar double-precision floating-point value from m64 to xmm1 register.
F2 0F 11 /r MOVSD xmm1/m64, xmm2	C	V/V	SSE2	Move scalar double-precision floating-point value from xmm2 register to xmm1/m64.
VEX.NDS.LIG.F2.0F.WIG 10 /r VMOVSD xmm1, xmm2, xmm3	B	V/V	AVX	Merge scalar double-precision floating-point value from xmm2 and xmm3 to xmm1 register.
VEX.LIG.F2.0F.WIG 10 /r VMOVSD xmm1, m64	D	V/V	AVX	Load scalar double-precision floating-point value from m64 to xmm1 register.
VEX.NDS.LIG.F2.0F.WIG 11 /r VMOVSD xmm1, xmm2, xmm3	E	V/V	AVX	Merge scalar double-precision floating-point value from xmm2 and xmm3 registers to xmm1.
VEX.LIG.F2.0F.WIG 11 /r VMOVSD m64, xmm1	C	V/V	AVX	Store scalar double-precision floating-point value from xmm1 register to m64.
EVEX.NDS.LIG.F2.0F.W1 10 /r VMOVSD xmm1 {k1}{z}, xmm2, xmm3	B	V/V	AVX512F	Merge scalar double-precision floating-point value from xmm2 and xmm3 registers to xmm1 under writemask k1.
EVEX.LIG.F2.0F.W1 10 /r VMOVSD xmm1 {k1}{z}, m64	F	V/V	AVX512F	Load scalar double-precision floating-point value from m64 to xmm1 register under writemask k1.
EVEX.NDS.LIG.F2.0F.W1 11 /r VMOVSD xmm1 {k1}{z}, xmm2, xmm3	E	V/V	AVX512F	Merge scalar double-precision floating-point value from xmm2 and xmm3 registers to xmm1 under writemask k1.
EVEX.LIG.F2.0F.W1 11 /r VMOVSD m64 {k1}, xmm1	G	V/V	AVX512F	Store scalar double-precision floating-point value from xmm1 register to m64 under writemask k1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	NA	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
B	NA	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA
C	NA	ModRM:r/m (w)	ModRM:reg (r)	NA	NA
D	NA	ModRM:reg (w)	ModRM:r/m (r)	NA	NA
E	NA	ModRM:r/m (w)	vvvv (r)	ModRM:reg (r)	NA
F	Tuple1 Scalar	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
G	Tuple1 Scalar	ModRM:r/m (w)	ModRM:reg (r)	NA	NA

Description

Moves a scalar double-precision floating-point value from the source operand (second operand) to the destination operand (first operand). The source and destination operands can be XMM registers or 64-bit memory locations. This instruction can be used to move a double-precision floating-point value to and from the low quadword of an XMM register and a 64-bit memory location, or to move a double-precision floating-point value between the low quadwords of two XMM registers. The instruction cannot be used to transfer data between memory locations.

Legacy version: When the source and destination operands are XMM registers, bits MAXVL:64 of the destination operand remains unchanged. When the source operand is a memory location and destination operand is an XMM registers, the quadword at bits 127:64 of the destination operand is cleared to all 0s, bits MAXVL:128 of the destination operand remains unchanged.

VEX and EVEX encoded register-register syntax: Moves a scalar double-precision floating-point value from the second source operand (the third operand) to the low quadword element of the destination operand (the first operand). Bits 127:64 of the destination operand are copied from the first source operand (the second operand). Bits (MAXVL-1:128) of the corresponding destination register are zeroed.

VEX and EVEX encoded memory store syntax: When the source operand is a memory location and destination operand is an XMM registers, bits MAXVL:64 of the destination operand is cleared to all 0s.

EVEX encoded versions: The low quadword of the destination is updated according to the writemask.

Note: For VMOVSD (memory store and load forms), VEX.vvvv and EVEX.vvvv are reserved and must be 1111b, otherwise instruction will #UD.

Operation

VMOVSD (EVEX.NDS.LIG.F2.0F 10 /r: VMOVSD xmm1, m64 with support for 32 registers)

```
IF k1[0] or *no writemask*
    THEN    DEST[63:0] ← SRC[63:0]
    ELSE
        IF *merging-masking*                ; merging-masking
            THEN *DEST[63:0] remains unchanged*
            ELSE                            ; zeroing-masking
                THEN DEST[63:0] ← 0
        FI;
FI;
DEST[MAXVL-1:64] ← 0
```

VMOVSD (EVEX.NDS.LIG.F2.0F 11 /r: VMOVSD m64, xmm1 with support for 32 registers)

```
IF k1[0] or *no writemask*
    THEN    DEST[63:0] ← SRC[63:0]
    ELSE    *DEST[63:0] remains unchanged*      ; merging-masking
FI;
```

VMOVSD (EVEX.NDS.LIG.F2.0F 11 /r: VMOVSD xmm1, xmm2, xmm3)

```
IF k1[0] or *no writemask*
    THEN    DEST[63:0] ← SRC2[63:0]
    ELSE
        IF *merging-masking*                ; merging-masking
            THEN *DEST[63:0] remains unchanged*
            ELSE                            ; zeroing-masking
                THEN DEST[63:0] ← 0
        FI;
FI;
DEST[127:64] ← SRC1[127:64]
DEST[MAXVL-1:128] ← 0
```

MOVSD (128-bit Legacy SSE version: MOVSD XMM1, XMM2)
DEST[63:0] ←SRC[63:0]
DEST[MAXVL-1:64] (Unmodified)

VMOVSD (VEX.NDS.128.F2.0F 11 /r: VMOVSD xmm1, xmm2, xmm3)
DEST[63:0] ←SRC2[63:0]
DEST[127:64] ←SRC1[127:64]
DEST[MAXVL-1:128] ←0

VMOVSD (VEX.NDS.128.F2.0F 10 /r: VMOVSD xmm1, xmm2, xmm3)
DEST[63:0] ←SRC2[63:0]
DEST[127:64] ←SRC1[127:64]
DEST[MAXVL-1:128] ←0

VMOVSD (VEX.NDS.128.F2.0F 10 /r: VMOVSD xmm1, m64)
DEST[63:0] ←SRC[63:0]
DEST[MAXVL-1:64] ←0

MOVSD/VMOVSD (128-bit versions: MOVSD m64, xmm1 or VMOVSD m64, xmm1)
DEST[63:0] ←SRC[63:0]

MOVSD (128-bit Legacy SSE version: MOVSD XMM1, m64)
DEST[63:0] ←SRC[63:0]
DEST[127:64] ←0
DEST[MAXVL-1:128] (Unmodified)

Intel C/C++ Compiler Intrinsic Equivalent

VMOVSD __m128d _mm_mask_load_sd(__m128d s, __mmask8 k, double * p);
VMOVSD __m128d _mm_maskz_load_sd(__mmask8 k, double * p);
VMOVSD __m128d _mm_mask_move_sd(__m128d sh, __mmask8 k, __m128d sl, __m128d a);
VMOVSD __m128d _mm_maskz_move_sd(__mmask8 k, __m128d s, __m128d a);
VMOVSD void _mm_mask_store_sd(double * p, __mmask8 k, __m128d s);
MOVSD __m128d _mm_load_sd (double *p)
MOVSD void _mm_store_sd (double *p, __m128d a)
MOVSD __m128d _mm_move_sd (__m128d a, __m128d b)

SIMD Floating-Point Exceptions

None

Other Exceptions

Non-EVEX-encoded instruction, see Exceptions Type 5; additionally

#UD	If VEX.vvvv != 1111B.

EVEX-encoded instruction, see Exceptions Type E10.

MOVSHDUP—Replicate Single FP Values

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
F3 0F 16 /r MOVSHDUP xmm1, xmm2/m128	A	V/V	SSE3	Move odd index single-precision floating-point values from xmm2/mem and duplicate each element into xmm1.
VEX.128.F3.0F.WIG 16 /r VMOVSHDUP xmm1, xmm2/m128	A	V/V	AVX	Move odd index single-precision floating-point values from xmm2/mem and duplicate each element into xmm1.
VEX.256.F3.0F.WIG 16 /r VMOVSHDUP ymm1, ymm2/m256	A	V/V	AVX	Move odd index single-precision floating-point values from ymm2/mem and duplicate each element into ymm1.
EVEX.128.F3.0F.W0 16 /r VMOVSHDUP xmm1 {k1}{z}, xmm2/m128	B	V/V	AVX512VL AVX512F	Move odd index single-precision floating-point values from xmm2/m128 and duplicate each element into xmm1 under writemask.
EVEX.256.F3.0F.W0 16 /r VMOVSHDUP ymm1 {k1}{z}, ymm2/m256	B	V/V	AVX512VL AVX512F	Move odd index single-precision floating-point values from ymm2/m256 and duplicate each element into ymm1 under writemask.
EVEX.512.F3.0F.W0 16 /r VMOVSHDUP zmm1 {k1}{z}, zmm2/m512	B	V/V	AVX512F	Move odd index single-precision floating-point values from zmm2/m512 and duplicate each element into zmm1 under writemask.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	NA	ModRM:reg (w)	ModRM:r/m (r)	NA	NA
B	Full Vector Mem	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

Description

Duplicates odd-indexed single-precision floating-point values from the source operand (the second operand) to adjacent element pair in the destination operand (the first operand). See Figure 4-3. The source operand is an XMM, YMM or ZMM register or 128, 256 or 512-bit memory location and the destination operand is an XMM, YMM or ZMM register.

128-bit Legacy SSE version: Bits (MAXVL-1:128) of the corresponding destination register remain unchanged.

VEX.128 encoded version: Bits (MAXVL-1:128) of the destination register are zeroed.

VEX.256 encoded version: Bits (MAXVL-1:256) of the destination register are zeroed.

EVEX encoded version: The destination operand is updated at 32-bit granularity according to the writemask.

Note: VEX.vvvv and EVEX.vvvv are reserved and must be 1111b otherwise instructions will #UD.

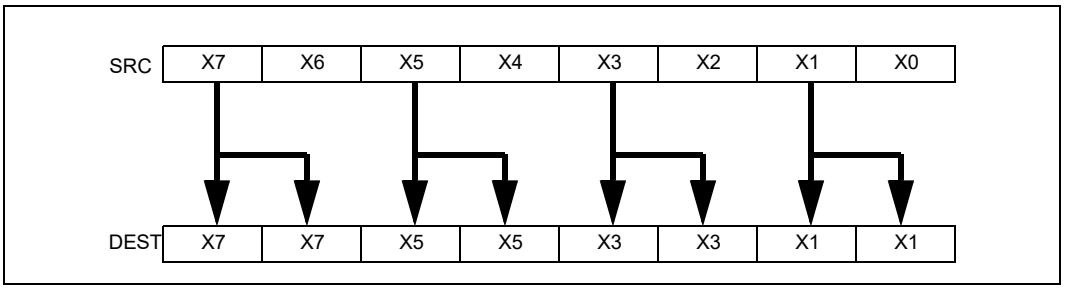


Figure 4-3. MOVSHDUP Operation

Operation

VMOVSHDUP (EVEX encoded versions)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
TMP_SRC[31:0] ← SRC[63:32]
TMP_SRC[63:32] ← SRC[63:32]
TMP_SRC[95:64] ← SRC[127:96]
TMP_SRC[127:96] ← SRC[127:96]
IF VL >= 256
    TMP_SRC[159:128] ← SRC[191:160]
    TMP_SRC[191:160] ← SRC[191:160]
    TMP_SRC[223:192] ← SRC[255:224]
    TMP_SRC[255:224] ← SRC[255:224]
FI;
IF VL >= 512
    TMP_SRC[287:256] ← SRC[319:288]
    TMP_SRC[319:288] ← SRC[319:288]
    TMP_SRC[351:320] ← SRC[383:352]
    TMP_SRC[383:352] ← SRC[383:352]
    TMP_SRC[415:384] ← SRC[447:416]
    TMP_SRC[447:416] ← SRC[447:416]
    TMP_SRC[479:448] ← SRC[511:480]
    TMP_SRC[511:480] ← SRC[511:480]
FI;
FOR j ← 0 TO KL-1
    i ← j * 32
    IF k1[j] OR *no writemask*
        THEN DEST[i+31:i] ← TMP_SRC[i+31:i]
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE                                ; zeroing-masking
                    DEST[i+31:i] ← 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0
```

VMOVSHDUP (VEX.256 encoded version)

```
DEST[31:0] ← SRC[63:32]
DEST[63:32] ← SRC[63:32]
DEST[95:64] ← SRC[127:96]
DEST[127:96] ← SRC[127:96]
DEST[159:128] ← SRC[191:160]
DEST[191:160] ← SRC[191:160]
DEST[223:192] ← SRC[255:224]
DEST[255:224] ← SRC[255:224]
DEST[MAXVL-1:256] ← 0
```

VMOVSHDUP (VEX.128 encoded version)

```
DEST[31:0] ← SRC[63:32]
DEST[63:32] ← SRC[63:32]
DEST[95:64] ← SRC[127:96]
DEST[127:96] ← SRC[127:96]
DEST[MAXVL-1:128] ← 0
```

MOVSHDUP (128-bit Legacy SSE version)
DEST[31:0] ←SRC[63:32]
DEST[63:32] ←SRC[63:32]
DEST[95:64] ←SRC[127:96]
DEST[127:96] ←SRC[127:96]
DEST[MAXVL-1:128] (Unmodified)

Intel C/C++ Compiler Intrinsic Equivalent

VMOVSHDUP __m512 _mm512_movehdup_ps(__m512 a);
VMOVSHDUP __m512 _mm512_mask_movehdup_ps(__m512 s, __mmask16 k, __m512 a);
VMOVSHDUP __m512 _mm512_maskz_movehdup_ps(__mmask16 k, __m512 a);
VMOVSHDUP __m256 _mm256_mask_movehdup_ps(__m256 s, __mmask8 k, __m256 a);
VMOVSHDUP __m256 _mm256_maskz_movehdup_ps(__mmask8 k, __m256 a);
VMOVSHDUP __m128 _mm_mask_movehdup_ps(__m128 s, __mmask8 k, __m128 a);
VMOVSHDUP __m128 _mm_maskz_movehdup_ps(__mmask8 k, __m128 a);
VMOVSHDUP __m256 _mm256_movehdup_ps (__m256 a);
VMOVSHDUP __m128 _mm_movehdup_ps (__m128 a);

SIMD Floating-Point Exceptions

None

Other Exceptions

Non-EVEX-encoded instruction, see Exceptions Type 4;
EVEX-encoded instruction, see Exceptions Type E4NF.nb.

#UD	If EVEX.vvvv != 1111B or VEX.vvvv != 1111B.

MOVSLDUP—Replicate Single FP Values

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
F3 0F 12 /r MOVSLDUP xmm1, xmm2/m128	A	V/V	SSE3	Move even index single-precision floating-point values from xmm2/mem and duplicate each element into xmm1.
VEX.128.F3.0F.WIG 12 /r VMOVSLDUP xmm1, xmm2/m128	A	V/V	AVX	Move even index single-precision floating-point values from xmm2/mem and duplicate each element into xmm1.
VEX.256.F3.0F.WIG 12 /r VMOVSLDUP ymm1, ymm2/m256	A	V/V	AVX	Move even index single-precision floating-point values from ymm2/mem and duplicate each element into ymm1.
EVEX.128.F3.0F.W0 12 /r VMOVSLDUP xmm1 {k1}{z}, xmm2/m128	B	V/V	AVX512VL AVX512F	Move even index single-precision floating-point values from xmm2/m128 and duplicate each element into xmm1 under writemask.
EVEX.256.F3.0F.W0 12 /r VMOVSLDUP ymm1 {k1}{z}, ymm2/m256	B	V/V	AVX512VL AVX512F	Move even index single-precision floating-point values from ymm2/m256 and duplicate each element into ymm1 under writemask.
EVEX.512.F3.0F.W0 12 /r VMOVSLDUP zmm1 {k1}{z}, zmm2/m512	B	V/V	AVX512F	Move even index single-precision floating-point values from zmm2/m512 and duplicate each element into zmm1 under writemask.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	NA	ModRM:reg (w)	ModRM:r/m (r)	NA	NA
B	Full Vector Mem	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

Description

Duplicates even-indexed single-precision floating-point values from the source operand (the second operand). See Figure 4-4. The source operand is an XMM, YMM or ZMM register or 128, 256 or 512-bit memory location and the destination operand is an XMM, YMM or ZMM register.

128-bit Legacy SSE version: Bits (MAXVL-1:128) of the corresponding destination register remain unchanged.

VEX.128 encoded version: Bits (MAXVL-1:128) of the destination register are zeroed.

VEX.256 encoded version: Bits (MAXVL-1:256) of the destination register are zeroed.

EVEX encoded version: The destination operand is updated at 32-bit granularity according to the writemask.

Note: VEX.vvvv and EVEX.vvvv are reserved and must be 1111b otherwise instructions will #UD.

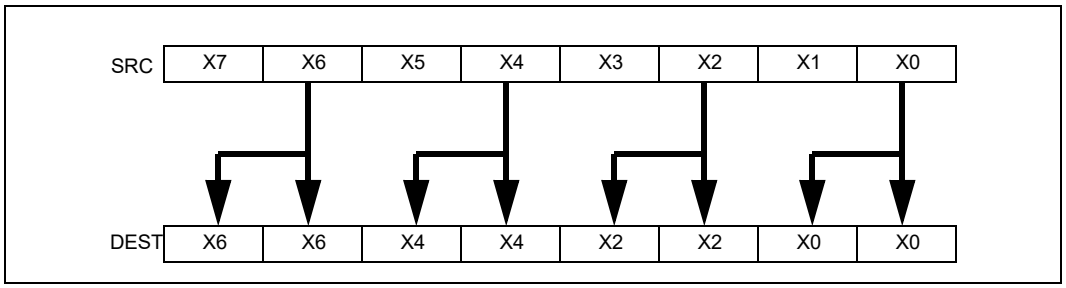


Figure 4-4. MOVSLDUP Operation

Operation

VMOVSLDUP (EVEX encoded versions)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
TMP_SRC[31:0] ← SRC[31:0]
TMP_SRC[63:32] ← SRC[31:0]
TMP_SRC[95:64] ← SRC[95:64]
TMP_SRC[127:96] ← SRC[95:64]
IF VL >= 256
    TMP_SRC[159:128] ← SRC[159:128]
    TMP_SRC[191:160] ← SRC[159:128]
    TMP_SRC[223:192] ← SRC[223:192]
    TMP_SRC[255:224] ← SRC[223:192]
FI;
IF VL >= 512
    TMP_SRC[287:256] ← SRC[287:256]
    TMP_SRC[319:288] ← SRC[287:256]
    TMP_SRC[351:320] ← SRC[351:320]
    TMP_SRC[383:352] ← SRC[351:320]
    TMP_SRC[415:384] ← SRC[415:384]
    TMP_SRC[447:416] ← SRC[415:384]
    TMP_SRC[479:448] ← SRC[479:448]
    TMP_SRC[511:480] ← SRC[479:448]
FI;
FOR j ← 0 TO KL-1
    i ← j * 32
    IF k1[j] OR *no writemask*
        THEN DEST[i+31:i] ← TMP_SRC[i+31:i]
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE                                ; zeroing-masking
                    DEST[i+31:i] ← 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0
```

VMOVSLDUP (VEX.256 encoded version)

```
DEST[31:0] ← SRC[31:0]
DEST[63:32] ← SRC[31:0]
DEST[95:64] ← SRC[95:64]
DEST[127:96] ← SRC[95:64]
DEST[159:128] ← SRC[159:128]
DEST[191:160] ← SRC[159:128]
DEST[223:192] ← SRC[223:192]
DEST[255:224] ← SRC[223:192]
DEST[MAXVL-1:256] ← 0
```

VMOVSLDUP (VEX.128 encoded version)

```
DEST[31:0] ← SRC[31:0]
DEST[63:32] ← SRC[31:0]
DEST[95:64] ← SRC[95:64]
DEST[127:96] ← SRC[95:64]
DEST[MAXVL-1:128] ← 0
```

MOVSLDUP (128-bit Legacy SSE version)
DEST[31:0] ←SRC[31:0]
DEST[63:32] ←SRC[31:0]
DEST[95:64] ←SRC[95:64]
DEST[127:96] ←SRC[95:64]
DEST[MAXVL-1:128] (Unmodified)

Intel C/C++ Compiler Intrinsic Equivalent

VMOVSLDUP __m512 _mm512_moveldup_ps(__m512 a);
VMOVSLDUP __m512 _mm512_mask_moveldup_ps(__m512 s, __mmask16 k, __m512 a);
VMOVSLDUP __m512 _mm512_maskz_moveldup_ps(__mmask16 k, __m512 a);
VMOVSLDUP __m256 _mm256_mask_moveldup_ps(__m256 s, __mmask8 k, __m256 a);
VMOVSLDUP __m256 _mm256_maskz_moveldup_ps(__mmask8 k, __m256 a);
VMOVSLDUP __m128 _mm_mask_moveldup_ps(__m128 s, __mmask8 k, __m128 a);
VMOVSLDUP __m128 _mm_maskz_moveldup_ps(__mmask8 k, __m128 a);
VMOVSLDUP __m256 _mm256_moveldup_ps (__m256 a);
VMOVSLDUP __m128 _mm_moveldup_ps (__m128 a);

SIMD Floating-Point Exceptions

None

Other Exceptions

Non-EVEX-encoded instruction, see Exceptions Type 4;
EVEX-encoded instruction, see Exceptions Type E4NF.nb.

#UD	If EVEX.vvvv != 1111B or VEX.vvvv != 1111B.

MOVSS—Move or Merge Scalar Single-Precision Floating-Point Value

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
F3 0F 10 /r MOVSS xmm1, xmm2	A	V/V	SSE	Merge scalar single-precision floating-point value from xmm2 to xmm1 register.
F3 0F 10 /r MOVSS xmm1, m32	A	V/V	SSE	Load scalar single-precision floating-point value from m32 to xmm1 register.
VEX.NDS.LIG.F3.0F.WIG 10 /r VMOVSS xmm1, xmm2, xmm3	B	V/V	AVX	Merge scalar single-precision floating-point value from xmm2 and xmm3 to xmm1 register
VEX.LIG.F3.0F.WIG 10 /r VMOVSS xmm1, m32	D	V/V	AVX	Load scalar single-precision floating-point value from m32 to xmm1 register.
F3 0F 11 /r MOVSS xmm2/m32, xmm1	C	V/V	SSE	Move scalar single-precision floating-point value from xmm1 register to xmm2/m32.
VEX.NDS.LIG.F3.0F.WIG 11 /r VMOVSS xmm1, xmm2, xmm3	E	V/V	AVX	Move scalar single-precision floating-point value from xmm2 and xmm3 to xmm1 register.
VEX.LIG.F3.0F.WIG 11 /r VMOVSS m32, xmm1	C	V/V	AVX	Move scalar single-precision floating-point value from xmm1 register to m32.
EVEX.NDS.LIG.F3.0F.W0 10 /r VMOVSS xmm1 {k1}{z}, xmm2, xmm3	B	V/V	AVX512F	Move scalar single-precision floating-point value from xmm2 and xmm3 to xmm1 register under writemask k1.
EVEX.LIG.F3.0F.W0 10 /r VMOVSS xmm1 {k1}{z}, m32	F	V/V	AVX512F	Move scalar single-precision floating-point values from m32 to xmm1 under writemask k1.
EVEX.NDS.LIG.F3.0F.W0 11 /r VMOVSS xmm1 {k1}{z}, xmm2, xmm3	E	V/V	AVX512F	Move scalar single-precision floating-point value from xmm2 and xmm3 to xmm1 register under writemask k1.
EVEX.LIG.F3.0F.W0 11 /r VMOVSS m32 {k1}, xmm1	G	V/V	AVX512F	Move scalar single-precision floating-point values from xmm1 to m32 under writemask k1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	NA	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
B	NA	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA
C	NA	ModRM:r/m (w)	ModRM:reg (r)	NA	NA
D	NA	ModRM:reg (w)	ModRM:r/m (r)	NA	NA
E	NA	ModRM:r/m (w)	vvvv (r)	ModRM:reg (r)	NA
F	Tuple1 Scalar	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
G	Tuple1 Scalar	ModRM:r/m (w)	ModRM:reg (r)	NA	NA

Description

Moves a scalar single-precision floating-point value from the source operand (second operand) to the destination operand (first operand). The source and destination operands can be XMM registers or 32-bit memory locations. This instruction can be used to move a single-precision floating-point value to and from the low doubleword of an XMM register and a 32-bit memory location, or to move a single-precision floating-point value between the low doublewords of two XMM registers. The instruction cannot be used to transfer data between memory locations.

Legacy version: When the source and destination operands are XMM registers, bits (MAXVL-1:32) of the corresponding destination register are unmodified. When the source operand is a memory location and destination operand is an XMM registers, Bits (127:32) of the destination operand is cleared to all 0s, bits MAXVL:128 of the destination operand remains unchanged.

VEX and EVEX encoded register-register syntax: Moves a scalar single-precision floating-point value from the second source operand (the third operand) to the low doubleword element of the destination operand (the first operand). Bits 127:32 of the destination operand are copied from the first source operand (the second operand). Bits (MAXVL-1:128) of the corresponding destination register are zeroed.

VEX and EVEX encoded memory load syntax: When the source operand is a memory location and destination operand is an XMM registers, bits MAXVL:32 of the destination operand is cleared to all 0s.

EVEX encoded versions: The low doubleword of the destination is updated according to the writemask.

Note: For memory store form instruction "VMOVSS m32, xmm1", VEX.vvvv is reserved and must be 1111b otherwise instruction will #UD. For memory store form instruction "VMOVSS mv {k1}, xmm1", EVEX.vvvv is reserved and must be 1111b otherwise instruction will #UD.

Software should ensure VMOVSS is encoded with VEX.L=0. Encoding VMOVSS with VEX.L=1 may encounter unpredictable behavior across different processor generations.

Operation

VMOVSS (EVEX.NDS.LIG.F3.0F.W0 11 /r when the source operand is memory and the destination is an XMM register)

```
IF k1[0] or *no writemask*
    THEN    DEST[31:0] ← SRC[31:0]
    ELSE
        IF *merging-masking*                ; merging-masking
            THEN *DEST[31:0] remains unchanged*
            ELSE                            ; zeroing-masking
                THEN DEST[31:0] ← 0
        FI;
FI;
DEST[MAXVL-1:32] ← 0
```

VMOVSS (EVEX.NDS.LIG.F3.0F.W0 10 /r when the source operand is an XMM register and the destination is memory)

```
IF k1[0] or *no writemask*
    THEN    DEST[31:0] ← SRC[31:0]
    ELSE    *DEST[31:0] remains unchanged*          ; merging-masking
FI;
```

VMOVSS (EVEX.NDS.LIG.F3.0F.W0 10/11 /r where the source and destination are XMM registers)

```
IF k1[0] or *no writemask*
    THEN      DEST[31:0] ← SRC2[31:0]
    ELSE
          IF *merging-masking*                    ; merging-masking
               THEN *DEST[31:0] remains unchanged*
               ELSE                               ; zeroing-masking
                    THEN DEST[31:0] ← 0
          FI;
FI;
DEST[127:32] ← SRC1[127:32]
DEST[MAXVL-1:128] ← 0
```

MOVSS (Legacy SSE version when the source and destination operands are both XMM registers)

```
DEST[31:0] ←SRC[31:0]
DEST[MAXVL-1:32] (Unmodified)
```

VMOVSS (VEX.NDS.128.F3.0F 11 /r where the destination is an XMM register)

```
DEST[31:0] ←SRC2[31:0]
DEST[127:32] ←SRC1[127:32]
DEST[MAXVL-1:128] ←0
```

VMOVSS (VEX.NDS.128.F3.0F 10 /r where the source and destination are XMM registers)

```
DEST[31:0] ←SRC2[31:0]
DEST[127:32] ←SRC1[127:32]
DEST[MAXVL-1:128] ←0
```

VMOVSS (VEX.NDS.128.F3.0F 10 /r when the source operand is memory and the destination is an XMM register)

```
DEST[31:0] ←SRC[31:0]
DEST[MAXVL-1:32] ←0
```

MOVSS/VMOVSS (when the source operand is an XMM register and the destination is memory)

```
DEST[31:0] ←SRC[31:0]
```

MOVSS (Legacy SSE version when the source operand is memory and the destination is an XMM register)

```
DEST[31:0] ←SRC[31:0]
DEST[127:32] ←0
DEST[MAXVL-1:128] (Unmodified)
```

Intel C/C++ Compiler Intrinsic Equivalent

VMOVSS __m128 _mm_mask_load_ss(__m128 s, __mmask8 k, float * p);
VMOVSS __m128 _mm_maskz_load_ss(__mmask8 k, float * p);
VMOVSS __m128 _mm_mask_move_ss(__m128 sh, __mmask8 k, __m128 sl, __m128 a);
VMOVSS __m128 _mm_maskz_move_ss(__mmask8 k, __m128 s, __m128 a);
VMOVSS void _mm_mask_store_ss(float * p, __mmask8 k, __m128 a);
MOVSS __m128 _mm_load_ss(float * p)
MOVSS void_mm_store_ss(float * p, __m128 a)
MOVSS __m128 _mm_move_ss(__m128 a, __m128 b)

SIMD Floating-Point Exceptions

None

Other Exceptions

Non-EVEX-encoded instruction, see Exceptions Type 5; additionally

#UD	If VEX.vvvv != 1111B.

EVEX-encoded instruction, see Exceptions Type E10.

MOVSX/MOVSXD—Move with Sign-Extension

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
0F BE /r	MOVSX *r16, r/m8*	RM	Valid	Valid	Move byte to word with sign-extension.
0F BE /r	MOVSX *r32, r/m8*	RM	Valid	Valid	Move byte to doubleword with sign-extension.
REX + 0F BE /r	MOVSX *r64, r/m8**	RM	Valid	N.E.	Move byte to quadword with sign-extension.
0F BF /r	MOVSX *r32, r/m16*	RM	Valid	Valid	Move word to doubleword, with sign-extension.
REX.W + 0F BF /r	MOVSX *r64, r/m16*	RM	Valid	N.E.	Move word to quadword with sign-extension.
REX.W** + 63 /r	MOVSXD *r64, r/m32*	RM	Valid	N.E.	Move doubleword to quadword with sign-extension.

NOTES:

* In 64-bit mode, r/m8 can not be encoded to access the following byte registers if a REX prefix is used: AH, BH, CH, DH.

** The use of MOVSXD without REX.W in 64-bit mode is discouraged. Regular MOV should be used instead of using MOVSXD without REX.W.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

Description

Copies the contents of the source operand (register or memory location) to the destination operand (register) and sign extends the value to 16 or 32 bits (see Figure 7-6 in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*). The size of the converted value depends on the operand-size attribute.

In 64-bit mode, the instruction's default operation size is 32 bits. Use of the REX.R prefix permits access to additional registers (R8-R15). Use of the REX.W prefix promotes operation to 64 bits. See the summary chart at the beginning of this section for encoding data and limits.

Operation

```
DEST ← SignExtend(SRC);
```

Flags Affected

None.

Protected Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit. If the DS, ES, FS, or GS register contains a NULL segment selector.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If the LOCK prefix is used.

Real-Address Mode Exceptions

#GP	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS	If a memory operand effective address is outside the SS segment limit.
#UD	If the LOCK prefix is used.

Virtual-8086 Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#UD	If the LOCK prefix is used.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If the LOCK prefix is used.

MOVUPD—Move Unaligned Packed Double-Precision Floating-Point Values

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 10 /r MOVUPD xmm1, xmm2/m128	A	V/V	SSE2	Move unaligned packed double-precision floating-point from xmm2/mem to xmm1.
66 0F 11 /r MOVUPD xmm2/m128, xmm1	B	V/V	SSE2	Move unaligned packed double-precision floating-point from xmm1 to xmm2/mem.
VEX.128.66.0F.WIG 10 /r VMOVUPD xmm1, xmm2/m128	A	V/V	AVX	Move unaligned packed double-precision floating-point from xmm2/mem to xmm1.
VEX.128.66.0F.WIG 11 /r VMOVUPD xmm2/m128, xmm1	B	V/V	AVX	Move unaligned packed double-precision floating-point from xmm1 to xmm2/mem.
VEX.256.66.0F.WIG 10 /r VMOVUPD ymm1, ymm2/m256	A	V/V	AVX	Move unaligned packed double-precision floating-point from ymm2/mem to ymm1.
VEX.256.66.0F.WIG 11 /r VMOVUPD ymm2/m256, ymm1	B	V/V	AVX	Move unaligned packed double-precision floating-point from ymm1 to ymm2/mem.
EVEX.128.66.0F.W1 10 /r VMOVUPD xmm1 {k1}{z}, xmm2/m128	C	V/V	AVX512VL AVX512F	Move unaligned packed double-precision floating-point from xmm2/m128 to xmm1 using writemask k1.
EVEX.128.66.0F.W1 11 /r VMOVUPD xmm2/m128 {k1}{z}, xmm1	D	V/V	AVX512VL AVX512F	Move unaligned packed double-precision floating-point from xmm1 to xmm2/m128 using writemask k1.
EVEX.256.66.0F.W1 10 /r VMOVUPD ymm1 {k1}{z}, ymm2/m256	C	V/V	AVX512VL AVX512F	Move unaligned packed double-precision floating-point from ymm2/m256 to ymm1 using writemask k1.
EVEX.256.66.0F.W1 11 /r VMOVUPD ymm2/m256 {k1}{z}, ymm1	D	V/V	AVX512VL AVX512F	Move unaligned packed double-precision floating-point from ymm1 to ymm2/m256 using writemask k1.
EVEX.512.66.0F.W1 10 /r VMOVUPD zmm1 {k1}{z}, zmm2/m512	C	V/V	AVX512F	Move unaligned packed double-precision floating-point values from zmm2/m512 to zmm1 using writemask k1.
EVEX.512.66.0F.W1 11 /r VMOVUPD zmm2/m512 {k1}{z}, zmm1	D	V/V	AVX512F	Move unaligned packed double-precision floating-point values from zmm1 to zmm2/m512 using writemask k1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	NA	ModRM:reg (w)	ModRM:r/m (r)	NA	NA
B	NA	ModRM:r/m (w)	ModRM:reg (r)	NA	NA
C	Full Vector Mem	ModRM:reg (w)	ModRM:r/m (r)	NA	NA
D	Full Vector Mem	ModRM:r/m (w)	ModRM:reg (r)	NA	NA

Description

Note: VEX.vvvv and EVEX.vvvv is reserved and must be 1111b otherwise instructions will #UD.

EVEX.512 encoded version:

Moves 512 bits of packed double-precision floating-point values from the source operand (second operand) to the destination operand (first operand). This instruction can be used to load a ZMM register from a float64 memory location, to store the contents of a ZMM register into a memory. The destination operand is updated according to the writemask.

VEX.256 encoded version:

Moves 256 bits of packed double-precision floating-point values from the source operand (second operand) to the destination operand (first operand). This instruction can be used to load a YMM register from a 256-bit memory location, to store the contents of a YMM register into a 256-bit memory location, or to move data between two YMM registers. Bits (MAXVL-1:256) of the destination register are zeroed.

128-bit versions:

Moves 128 bits of packed double-precision floating-point values from the source operand (second operand) to the destination operand (first operand). This instruction can be used to load an XMM register from a 128-bit memory location, to store the contents of an XMM register into a 128-bit memory location, or to move data between two XMM registers.

128-bit Legacy SSE version: Bits (MAXVL-1:128) of the corresponding destination register remain unchanged.

When the source or destination operand is a memory operand, the operand may be unaligned on a 16-byte boundary without causing a general-protection exception (#GP) to be generated

VEX.128 and EVEX.128 encoded versions: Bits (MAXVL-1:128) of the destination register are zeroed.

Operation

VMOVUPD (EVEX encoded versions, register-copy form)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j ← 0 TO KL-1
    i ← j * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+63:i] ← SRC[i+63:i]
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE  DEST[i+63:i] ← 0              ; zeroing-masking
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0
```

VMOVUPD (EVEX encoded versions, store-form)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j ← 0 TO KL-1
    i ← j * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+63:i]← SRC[i+63:i]
        ELSE *DEST[i+63:i] remains unchanged*          ; merging-masking

    FI;
ENDFOR;
```

VMOVUPD (EVEX encoded versions, load-form)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j ← 0 TO KL-1
    i ← j * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+63:i] ← SRC[i+63:i]
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE  DEST[i+63:i] ← 0              ; zeroing-masking
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0
```

```
VMOVUPD (VEX.256 encoded version, load - and register copy)
DEST[255:0] ← SRC[255:0]
DEST[MAXVL-1:256] ← 0
```

VMOVUPD (VEX.256 encoded version, store-form)

```
DEST[255:0] ← SRC[255:0]
```

VMOVUPD (VEX.128 encoded version)

```
DEST[127:0] ← SRC[127:0]
DEST[MAXVL-1:128] ← 0
```

MOVUPD (128-bit load- and register-copy- form Legacy SSE version)

```
DEST[127:0] ← SRC[127:0]
DEST[MAXVL-1:128] (Unmodified)
```

(V)MOVUPD (128-bit store-form version)

```
DEST[127:0] ← SRC[127:0]
```

Intel C/C++ Compiler Intrinsic Equivalent

VMOVUPD __m512d _mm512_loadu_pd(void * s);
VMOVUPD __m512d _mm512_mask_loadu_pd(__m512d a, __mmask8 k, void * s);
VMOVUPD __m512d _mm512_maskz_loadu_pd(__mmask8 k, void * s);
VMOVUPD void _mm512_storeu_pd(void * d, __m512d a);
VMOVUPD void _mm512_mask_storeu_pd(void * d, __mmask8 k, __m512d a);
VMOVUPD __m256d _mm256_mask_loadu_pd(__m256d s, __mmask8 k, void * m);
VMOVUPD __m256d _mm256_maskz_loadu_pd(__mmask8 k, void * m);
VMOVUPD void _mm256_mask_storeu_pd(void * d, __mmask8 k, __m256d a);
VMOVUPD __m128d _mm_mask_loadu_pd(__m128d s, __mmask8 k, void * m);
VMOVUPD __m128d _mm_maskz_loadu_pd(__mmask8 k, void * m);
VMOVUPD void _mm_mask_storeu_pd(void * d, __mmask8 k, __m128d a);
MOVUPD __m256d _mm256_loadu_pd (double * p);
MOVUPD void _mm256_storeu_pd(double *p, __m256d a);
MOVUPD __m128d _mm_loadu_pd (double * p);
MOVUPD void _mm_storeu_pd(double *p, __m128d a);

SIMD Floating-Point Exceptions

None

Other Exceptions

Non-EVEX-encoded instruction, see Exceptions Type 4.
Note treatment of #AC varies; additionally

#UD	If VEX.vvvv != 1111B.

EVEX-encoded instruction, see Exceptions Type E4.nb.

MOVUPS—Move Unaligned Packed Single-Precision Floating-Point Values

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F 10 /r MOVUPS xmm1, xmm2/m128	A	V/V	SSE	Move unaligned packed single-precision floating-point from xmm2/mem to xmm1.
NP 0F 11 /r MOVUPS xmm2/m128, xmm1	B	V/V	SSE	Move unaligned packed single-precision floating-point from xmm1 to xmm2/mem.
VEX.128.0F.WIG 10 /r VMOVUPS xmm1, xmm2/m128	A	V/V	AVX	Move unaligned packed single-precision floating-point from xmm2/mem to xmm1.
VEX.128.0F 11.WIG /r VMOVUPS xmm2/m128, xmm1	B	V/V	AVX	Move unaligned packed single-precision floating-point from xmm1 to xmm2/mem.
VEX.256.0F 10.WIG /r VMOVUPS ymm1, ymm2/m256	A	V/V	AVX	Move unaligned packed single-precision floating-point from ymm2/mem to ymm1.
VEX.256.0F 11.WIG /r VMOVUPS ymm2/m256, ymm1	B	V/V	AVX	Move unaligned packed single-precision floating-point from ymm1 to ymm2/mem.
EVEX.128.0F.W0 10 /r VMOVUPS xmm1 {k1}{z}, xmm2/m128	C	V/V	AVX512VL AVX512F	Move unaligned packed single-precision floating-point values from xmm2/m128 to xmm1 using writemask k1.
EVEX.256.0F.W0 10 /r VMOVUPS ymm1 {k1}{z}, ymm2/m256	C	V/V	AVX512VL AVX512F	Move unaligned packed single-precision floating-point values from ymm2/m256 to ymm1 using writemask k1.
EVEX.512.0F.W0 10 /r VMOVUPS zmm1 {k1}{z}, zmm2/m512	C	V/V	AVX512F	Move unaligned packed single-precision floating-point values from zmm2/m512 to zmm1 using writemask k1.
EVEX.128.0F.W0 11 /r VMOVUPS xmm2/m128 {k1}{z}, xmm1	D	V/V	AVX512VL AVX512F	Move unaligned packed single-precision floating-point values from xmm1 to xmm2/m128 using writemask k1.
EVEX.256.0F.W0 11 /r VMOVUPS ymm2/m256 {k1}{z}, ymm1	D	V/V	AVX512VL AVX512F	Move unaligned packed single-precision floating-point values from ymm1 to ymm2/m256 using writemask k1.
EVEX.512.0F.W0 11 /r VMOVUPS zmm2/m512 {k1}{z}, zmm1	D	V/V	AVX512F	Move unaligned packed single-precision floating-point values from zmm1 to zmm2/m512 using writemask k1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	NA	ModRM:reg (w)	ModRM:r/m (r)	NA	NA
B	NA	ModRM:r/m (w)	ModRM:reg (r)	NA	NA
C	Full Vector Mem	ModRM:reg (w)	ModRM:r/m (r)	NA	NA
D	Full Vector Mem	ModRM:r/m (w)	ModRM:reg (r)	NA	NA

Description

Note: VEX.vvvv and EVEX.vvvv is reserved and must be 1111b otherwise instructions will #UD.

EVEX.512 encoded version:

Moves 512 bits of packed single-precision floating-point values from the source operand (second operand) to the destination operand (first operand). This instruction can be used to load a ZMM register from a 512-bit float32 memory location, to store the contents of a ZMM register into memory. The destination operand is updated according to the writemask.

VEX.256 and EVEX.256 encoded versions:

Moves 256 bits of packed single-precision floating-point values from the source operand (second operand) to the destination operand (first operand). This instruction can be used to load a YMM register from a 256-bit memory location, to store the contents of a YMM register into a 256-bit memory location, or to move data between two YMM registers. Bits (MAXVL-1:256) of the destination register are zeroed.

128-bit versions:

Moves 128 bits of packed single-precision floating-point values from the source operand (second operand) to the destination operand (first operand). This instruction can be used to load an XMM register from a 128-bit memory location, to store the contents of an XMM register into a 128-bit memory location, or to move data between two XMM registers.

128-bit Legacy SSE version: Bits (MAXVL-1:128) of the corresponding destination register remain unchanged.

When the source or destination operand is a memory operand, the operand may be unaligned without causing a general-protection exception (#GP) to be generated.

VEX.128 and EVEX.128 encoded versions: Bits (MAXVL-1:128) of the destination register are zeroed.

Operation

VMOVUPS (EVEX encoded versions, register-copy form)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j ← 0 TO KL-1
    i ← j * 32
    IF k1[j] OR *no writemask*
        THEN DEST[i+31:i] ← SRC[i+31:i]
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE  DEST[i+31:i] ← 0          ; zeroing-masking
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0
```

VMOVUPS (EVEX encoded versions, store-form)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j ← 0 TO KL-1
    i ← j * 32
    IF k1[j] OR *no writemask*
        THEN DEST[i+31:i]← SRC[i+31:i]
        ELSE *DEST[i+31:i] remains unchanged*        ; merging-masking
    FI;
ENDFOR;
```

VMOVUPS (EVEX encoded versions, load-form)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j ← 0 TO KL-1
    i ← j * 32
    IF k1[j] OR *no writemask*
        THEN DEST[i+31:i] ← SRC[i+31:i]
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE  DEST[i+31:i] ← 0              ; zeroing-masking
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0
```

VMOVUPS (VEX.256 encoded version, load - and register copy)

```
DEST[255:0] ← SRC[255:0]
DEST[MAXVL-1:256] ← 0
```

VMOVUPS (VEX.256 encoded version, store-form)

```
DEST[255:0] ← SRC[255:0]
```

VMOVUPS (VEX.128 encoded version)

```
DEST[127:0] ← SRC[127:0]
DEST[MAXVL-1:128] ← 0
```

MOVUPS (128-bit load- and register-copy- form Legacy SSE version)

```
DEST[127:0] ← SRC[127:0]
DEST[MAXVL-1:128] (Unmodified)
```

(V)MOVUPS (128-bit store-form version)

```
DEST[127:0] ← SRC[127:0]
```

Intel C/C++ Compiler Intrinsic Equivalent

VMOVUPS __m512 _mm512_loadu_ps(void * s);
VMOVUPS __m512 _mm512_mask_loadu_ps(__m512 a, __mmask16 k, void * s);
VMOVUPS __m512 _mm512_maskz_loadu_ps(__mmask16 k, void * s);
VMOVUPS void _mm512_storeu_ps(void * d, __m512 a);
VMOVUPS void _mm512_mask_storeu_ps(void * d, __mmask8 k, __m512 a);
VMOVUPS __m256 _mm256_mask_loadu_ps(__m256 a, __mmask8 k, void * s);
VMOVUPS __m256 _mm256_maskz_loadu_ps(__mmask8 k, void * s);
VMOVUPS void _mm256_mask_storeu_ps(void * d, __mmask8 k, __m256 a);
VMOVUPS __m128 _mm_mask_loadu_ps(__m128 a, __mmask8 k, void * s);
VMOVUPS __m128 _mm_maskz_loadu_ps(__mmask8 k, void * s);
VMOVUPS void _mm_mask_storeu_ps(void * d, __mmask8 k, __m128 a);
MOVUPS __m256 _mm256_loadu_ps (float * p);
MOVUPS void _mm256 _storeu_ps(float *p, __m256 a);
MOVUPS __m128 _mm_loadu_ps (float * p);
MOVUPS void _mm_storeu_ps(float *p, __m128 a);

SIMD Floating-Point Exceptions

None

Other Exceptions

Non-EVEX-encoded instruction, see Exceptions Type 4.
Note treatment of #AC varies;
EVEX-encoded instruction, see Exceptions Type E4.nb.

#UD	If EVEX.vvvv != 1111B or VEX.vvvv != 1111B.

MOVZX—Move with Zero-Extend

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
0F B6 /r	MOVZX *r16, r/m8*	RM	Valid	Valid	Move byte to word with zero-extension.
0F B6 /r	MOVZX *r32, r/m8*	RM	Valid	Valid	Move byte to doubleword, zero-extension.
REX.W + 0F B6 /r	MOVZX *r64, r/m8**	RM	Valid	N.E.	Move byte to quadword, zero-extension.
0F B7 /r	MOVZX *r32, r/m16*	RM	Valid	Valid	Move word to doubleword, zero-extension.
REX.W + 0F B7 /r	MOVZX *r64, r/m16*	RM	Valid	N.E.	Move word to quadword, zero-extension.

NOTES:

* In 64-bit mode, r/m8 can not be encoded to access the following byte registers if the REX prefix is used: AH, BH, CH, DH.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

Description

Copies the contents of the source operand (register or memory location) to the destination operand (register) and zero extends the value. The size of the converted value depends on the operand-size attribute.

In 64-bit mode, the instruction's default operation size is 32 bits. Use of the REX.R prefix permits access to additional registers (R8-R15). Use of the REX.W prefix promotes operation to 64 bit operands. See the summary chart at the beginning of this section for encoding data and limits.

Operation

```
DEST ← ZeroExtend(SRC);
```

Flags Affected

None.

Protected Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit. If the DS, ES, FS, or GS register contains a NULL segment selector.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If the LOCK prefix is used.

Real-Address Mode Exceptions

#GP	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS	If a memory operand effective address is outside the SS segment limit.
#UD	If the LOCK prefix is used.

Virtual-8086 Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.
#UD	If the LOCK prefix is used.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If the LOCK prefix is used.

MPSADBW — Compute Multiple Packed Sums of Absolute Difference

Opcode/ Instruction	Op/ En	64/32-bit Mode	CPUID Feature Flag	Description
66 0F 3A 42 /r ib MPSADBW *xmm1, xmm2/m128, imm8*	RMI	V/V	SSE4_1	Sums absolute 8-bit integer difference of adjacent groups of 4 byte integers in *xmm1* and *xmm2/m128* and writes the results in *xmm1*. Starting offsets within *xmm1* and *xmm2/m128* are determined by *imm8*.
VEX.NDS.128.66.0F3A.WIG 42 /r ib VMPSADBW *xmm1, xmm2, xmm3/m128, imm8*	RVMI	V/V	AVX	Sums absolute 8-bit integer difference of adjacent groups of 4 byte integers in *xmm2* and *xmm3/m128* and writes the results in *xmm1*. Starting offsets within *xmm2* and *xmm3/m128* are determined by *imm8*.
VEX.NDS.256.66.0F3A.WIG 42 /r ib VMPSADBW *ymm1, ymm2, ymm3/m256, imm8*	RVMI	V/V	AVX2	Sums absolute 8-bit integer difference of adjacent groups of 4 byte integers in *xmm2* and *ymm3/m128* and writes the results in *ymm1*. Starting offsets within *ymm2* and *xmm3/m128* are determined by *imm8*.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RMI	ModRM:reg (r, w)	ModRM:r/m (r)	imm8	NA
RVMI	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	imm8

Description

(V)MPSADBW calculates packed word results of sum-absolute-difference (SAD) of unsigned bytes from two blocks of 32-bit dword elements, using two select fields in the immediate byte to select the offsets of the two blocks within the first source operand and the second operand. Packed SAD word results are calculated within each 128-bit lane. Each SAD word result is calculated between a stationary block_2 (whose offset within the second source operand is selected by a two bit select control, multiplied by 32 bits) and a sliding block_1 at consecutive byte-granular position within the first source operand. The offset of the first 32-bit block of block_1 is selectable using a one bit select control, multiplied by 32 bits.

128-bit Legacy SSE version: Imm8[1:0]*32 specifies the bit offset of block_2 within the second source operand. Imm[2]*32 specifies the initial bit offset of the block_1 within the first source operand. The first source operand and destination operand are the same. The first source and destination operands are XMM registers. The second source operand is either an XMM register or a 128-bit memory location. Bits (MAXVL-1:128) of the corresponding YMM destination register remain unchanged. Bits 7:3 of the immediate byte are ignored.

VEX.128 encoded version: Imm8[1:0]*32 specifies the bit offset of block_2 within the second source operand. Imm[2]*32 specifies the initial bit offset of the block_1 within the first source operand. The first source and destination operands are XMM registers. The second source operand is either an XMM register or a 128-bit memory location. Bits (127:128) of the corresponding YMM register are zeroed. Bits 7:3 of the immediate byte are ignored.

VEX.256 encoded version: The sum-absolute-difference (SAD) operation is repeated 8 times for MPSADW between the same block_2 (fixed offset within the second source operand) and a variable block_1 (offset is shifted by 8 bits for each SAD operation) in the first source operand. Each 16-bit result of eight SAD operations between block_2 and block_1 is written to the respective word in the lower 128 bits of the destination operand.

Additionally, VMPSADBW performs another eight SAD operations on block_4 of the second source operand and block_3 of the first source operand. (Imm8[4:3]*32 + 128) specifies the bit offset of block_4 within the second source operand. (Imm[5]*32+128) specifies the initial bit offset of the block_3 within the first source operand. Each 16-bit result of eight SAD operations between block_4 and block_3 is written to the respective word in the upper 128 bits of the destination operand.

The first source operand is a YMM register. The second source register can be a YMM register or a 256-bit memory location. The destination operand is a YMM register. Bits 7:6 of the immediate byte are ignored.

Note: If VMPSADBW is encoded with VEX.L= 1, an attempt to execute the instruction encoded with VEX.L= 1 will cause an #UD exception.

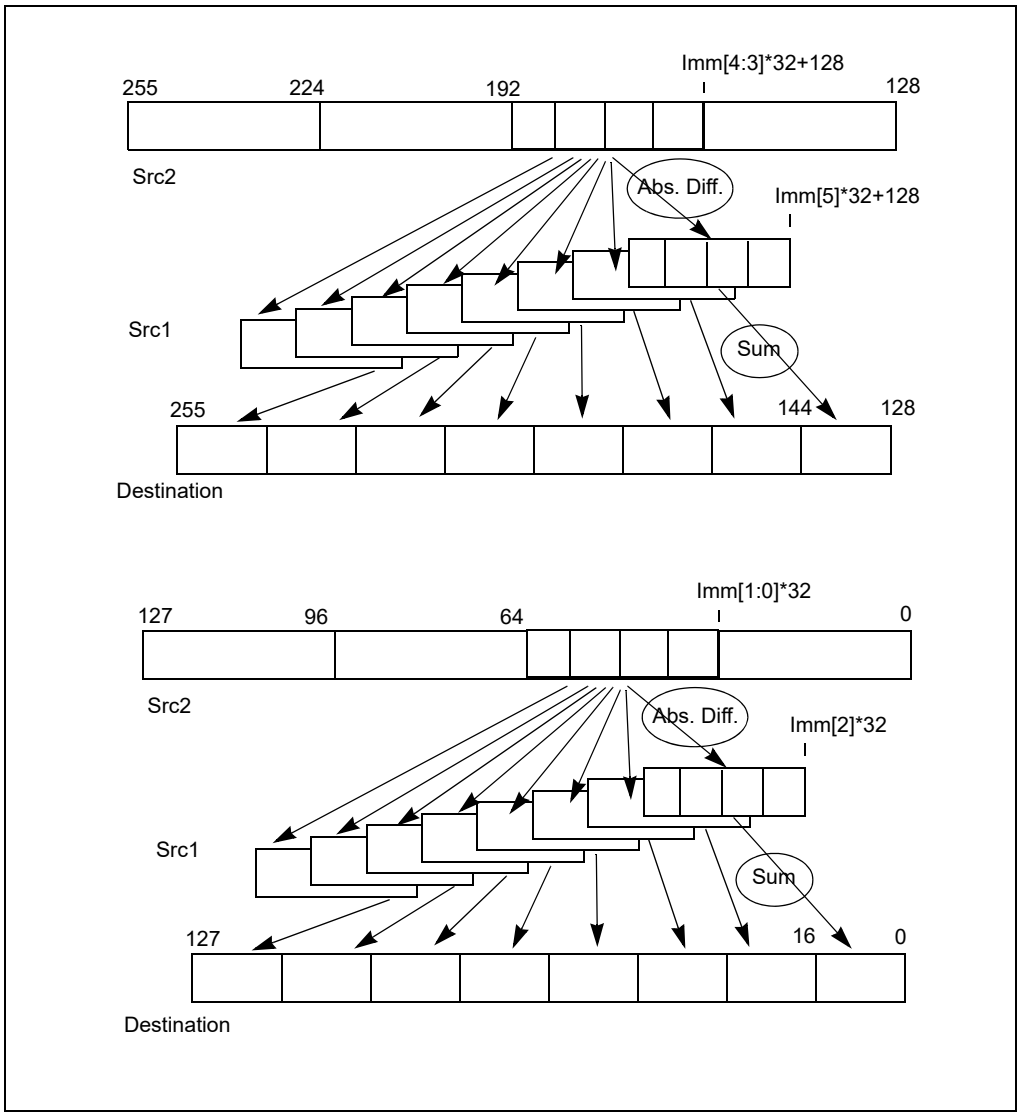


Figure 4-5. 256-bit VMPSADBW Operation

Operation

VMPSADBW (VEX.256 encoded version)

```
BLK2_OFFSET ← imm8[1:0]*32
BLK1_OFFSET ← imm8[2]*32
SRC1_BYTE0 ← SRC1[BLK1_OFFSET+7:BLK1_OFFSET]
SRC1_BYTE1 ← SRC1[BLK1_OFFSET+15:BLK1_OFFSET+8]
SRC1_BYTE2 ← SRC1[BLK1_OFFSET+23:BLK1_OFFSET+16]
SRC1_BYTE3 ← SRC1[BLK1_OFFSET+31:BLK1_OFFSET+24]
SRC1_BYTE4 ←SRC1[BLK1_OFFSET+39:BLK1_OFFSET+32]
SRC1_BYTE5 ← SRC1[BLK1_OFFSET+47:BLK1_OFFSET+40]
SRC1_BYTE6 ← SRC1[BLK1_OFFSET+55:BLK1_OFFSET+48]
SRC1_BYTE7 ← SRC1[BLK1_OFFSET+63:BLK1_OFFSET+56]
SRC1_BYTE8 ← SRC1[BLK1_OFFSET+71:BLK1_OFFSET+64]
SRC1_BYTE9 ← SRC1[BLK1_OFFSET+79:BLK1_OFFSET+72]
SRC1_BYTE10 ← SRC1[BLK1_OFFSET+87:BLK1_OFFSET+80]
SRC2_BYTE0 ←SRC2[BLK2_OFFSET+7:BLK2_OFFSET]
SRC2_BYTE1 ← SRC2[BLK2_OFFSET+15:BLK2_OFFSET+8]
SRC2_BYTE2 ← SRC2[BLK2_OFFSET+23:BLK2_OFFSET+16]
SRC2_BYTE3 ← SRC2[BLK2_OFFSET+31:BLK2_OFFSET+24]

TEMP0 ← ABS(SRC1_BYTE0 - SRC2_BYTE0)
TEMP1 ← ABS(SRC1_BYTE1 - SRC2_BYTE1)
TEMP2 ← ABS(SRC1_BYTE2 - SRC2_BYTE2)
TEMP3 ← ABS(SRC1_BYTE3 - SRC2_BYTE3)
DEST[15:0] ← TEMP0 + TEMP1 + TEMP2 + TEMP3

TEMP0 ← ABS(SRC1_BYTE1 - SRC2_BYTE0)
TEMP1 ← ABS(SRC1_BYTE2 - SRC2_BYTE1)
TEMP2 ← ABS(SRC1_BYTE3 - SRC2_BYTE2)
TEMP3 ← ABS(SRC1_BYTE4 - SRC2_BYTE3)
DEST[31:16] ← TEMP0 + TEMP1 + TEMP2 + TEMP3

TEMP0 ← ABS(SRC1_BYTE2 - SRC2_BYTE0)
TEMP1 ← ABS(SRC1_BYTE3 - SRC2_BYTE1)
TEMP2 ← ABS(SRC1_BYTE4 - SRC2_BYTE2)
TEMP3 ← ABS(SRC1_BYTE5 - SRC2_BYTE3)
DEST[47:32] ← TEMP0 + TEMP1 + TEMP2 + TEMP3

TEMP0 ← ABS(SRC1_BYTE3 - SRC2_BYTE0)
TEMP1 ← ABS(SRC1_BYTE4 - SRC2_BYTE1)
TEMP2 ← ABS(SRC1_BYTE5 - SRC2_BYTE2)
TEMP3 ← ABS(SRC1_BYTE6 - SRC2_BYTE3)
DEST[63:48] ← TEMP0 + TEMP1 + TEMP2 + TEMP3

TEMP0 ← ABS(SRC1_BYTE4 - SRC2_BYTE0)
TEMP1 ← ABS(SRC1_BYTE5 - SRC2_BYTE1)
TEMP2 ← ABS(SRC1_BYTE6 - SRC2_BYTE2)
TEMP3 ← ABS(SRC1_BYTE7 - SRC2_BYTE3)
DEST[79:64] ← TEMP0 + TEMP1 + TEMP2 + TEMP3
```

```
TEMP0 ← ABS(SRC1_BYTE5 - SRC2_BYTE0)
TEMP1 ← ABS(SRC1_BYTE6 - SRC2_BYTE1)
TEMP2 ← ABS(SRC1_BYTE7 - SRC2_BYTE2)
TEMP3 ← ABS(SRC1_BYTE8 - SRC2_BYTE3)
DEST[95:80] ← TEMP0 + TEMP1 + TEMP2 + TEMP3

TEMP0 ← ABS(SRC1_BYTE6 - SRC2_BYTE0)
TEMP1 ← ABS(SRC1_BYTE7 - SRC2_BYTE1)
TEMP2 ← ABS(SRC1_BYTE8 - SRC2_BYTE2)
TEMP3 ← ABS(SRC1_BYTE9 - SRC2_BYTE3)
DEST[111:96] ← TEMP0 + TEMP1 + TEMP2 + TEMP3

TEMP0 ← ABS(SRC1_BYTE7 - SRC2_BYTE0)
TEMP1 ← ABS(SRC1_BYTE8 - SRC2_BYTE1)
TEMP2 ← ABS(SRC1_BYTE9 - SRC2_BYTE2)
TEMP3 ← ABS(SRC1_BYTE10 - SRC2_BYTE3)
DEST[127:112] ← TEMP0 + TEMP1 + TEMP2 + TEMP3

BLK2_OFFSET ← imm8[4:3]*32 + 128
BLK1_OFFSET ← imm8[5]*32 + 128
SRC1_BYTE0 ← SRC1[BLK1_OFFSET+7:BLK1_OFFSET]
SRC1_BYTE1 ← SRC1[BLK1_OFFSET+15:BLK1_OFFSET+8]
SRC1_BYTE2 ← SRC1[BLK1_OFFSET+23:BLK1_OFFSET+16]
SRC1_BYTE3 ← SRC1[BLK1_OFFSET+31:BLK1_OFFSET+24]
SRC1_BYTE4 ← SRC1[BLK1_OFFSET+39:BLK1_OFFSET+32]
SRC1_BYTE5 ← SRC1[BLK1_OFFSET+47:BLK1_OFFSET+40]
SRC1_BYTE6 ← SRC1[BLK1_OFFSET+55:BLK1_OFFSET+48]
SRC1_BYTE7 ← SRC1[BLK1_OFFSET+63:BLK1_OFFSET+56]
SRC1_BYTE8 ← SRC1[BLK1_OFFSET+71:BLK1_OFFSET+64]
SRC1_BYTE9 ← SRC1[BLK1_OFFSET+79:BLK1_OFFSET+72]
SRC1_BYTE10 ← SRC1[BLK1_OFFSET+87:BLK1_OFFSET+80]

SRC2_BYTE0 ←SRC2[BLK2_OFFSET+7:BLK2_OFFSET]
SRC2_BYTE1 ← SRC2[BLK2_OFFSET+15:BLK2_OFFSET+8]
SRC2_BYTE2 ← SRC2[BLK2_OFFSET+23:BLK2_OFFSET+16]
SRC2_BYTE3 ← SRC2[BLK2_OFFSET+31:BLK2_OFFSET+24]

TEMP0 ← ABS(SRC1_BYTE0 - SRC2_BYTE0)
TEMP1 ← ABS(SRC1_BYTE1 - SRC2_BYTE1)
TEMP2 ← ABS(SRC1_BYTE2 - SRC2_BYTE2)
TEMP3 ← ABS(SRC1_BYTE3 - SRC2_BYTE3)
DEST[143:128] ← TEMP0 + TEMP1 + TEMP2 + TEMP3

TEMP0 ←ABS(SRC1_BYTE1 - SRC2_BYTE0)
TEMP1 ← ABS(SRC1_BYTE2 - SRC2_BYTE1)
TEMP2 ← ABS(SRC1_BYTE3 - SRC2_BYTE2)
TEMP3 ← ABS(SRC1_BYTE4 - SRC2_BYTE3)
DEST[159:144] ← TEMP0 + TEMP1 + TEMP2 + TEMP3

TEMP0 ← ABS(SRC1_BYTE2 - SRC2_BYTE0)
TEMP1 ← ABS(SRC1_BYTE3 - SRC2_BYTE1)
TEMP2 ← ABS(SRC1_BYTE4 - SRC2_BYTE2)
TEMP3 ← ABS(SRC1_BYTE5 - SRC2_BYTE3)
DEST[175:160] ← TEMP0 + TEMP1 + TEMP2 + TEMP3
```

TEMP0 ←ABS(SRC1_BYTE3 - SRC2_BYTE0)
TEMP1 ← ABS(SRC1_BYTE4 - SRC2_BYTE1)
TEMP2 ← ABS(SRC1_BYTE5 - SRC2_BYTE2)
TEMP3 ← ABS(SRC1_BYTE6 - SRC2_BYTE3)
DEST[191:176] ← TEMP0 + TEMP1 + TEMP2 + TEMP3

TEMP0 ← ABS(SRC1_BYTE4 - SRC2_BYTE0)
TEMP1 ← ABS(SRC1_BYTE5 - SRC2_BYTE1)
TEMP2 ← ABS(SRC1_BYTE6 - SRC2_BYTE2)
TEMP3 ← ABS(SRC1_BYTE7 - SRC2_BYTE3)
DEST[207:192] ← TEMP0 + TEMP1 + TEMP2 + TEMP3

TEMP0 ← ABS(SRC1_BYTE5 - SRC2_BYTE0)
TEMP1 ← ABS(SRC1_BYTE6 - SRC2_BYTE1)
TEMP2 ← ABS(SRC1_BYTE7 - SRC2_BYTE2)
TEMP3 ← ABS(SRC1_BYTE8 - SRC2_BYTE3)
DEST[223:208] ← TEMP0 + TEMP1 + TEMP2 + TEMP3

TEMP0 ← ABS(SRC1_BYTE6 - SRC2_BYTE0)
TEMP1 ← ABS(SRC1_BYTE7 - SRC2_BYTE1)
TEMP2 ← ABS(SRC1_BYTE8 - SRC2_BYTE2)
TEMP3 ← ABS(SRC1_BYTE9 - SRC2_BYTE3)
DEST[239:224] ← TEMP0 + TEMP1 + TEMP2 + TEMP3

TEMP0 ← ABS(SRC1_BYTE7 - SRC2_BYTE0)
TEMP1 ← ABS(SRC1_BYTE8 - SRC2_BYTE1)
TEMP2 ← ABS(SRC1_BYTE9 - SRC2_BYTE2)
TEMP3 ← ABS(SRC1_BYTE10 - SRC2_BYTE3)
DEST[255:240] ← TEMP0 + TEMP1 + TEMP2 + TEMP3

VMPSADBW (VEX.128 encoded version)
BLK2_OFFSET ← imm8[1:0]*32
BLK1_OFFSET ← imm8[2]*32
SRC1_BYTE0 ← SRC1[BLK1_OFFSET+7:BLK1_OFFSET]
SRC1_BYTE1 ← SRC1[BLK1_OFFSET+15:BLK1_OFFSET+8]
SRC1_BYTE2 ← SRC1[BLK1_OFFSET+23:BLK1_OFFSET+16]
SRC1_BYTE3 ← SRC1[BLK1_OFFSET+31:BLK1_OFFSET+24]
SRC1_BYTE4 ← SRC1[BLK1_OFFSET+39:BLK1_OFFSET+32]
SRC1_BYTE5 ← SRC1[BLK1_OFFSET+47:BLK1_OFFSET+40]
SRC1_BYTE6 ← SRC1[BLK1_OFFSET+55:BLK1_OFFSET+48]
SRC1_BYTE7 ← SRC1[BLK1_OFFSET+63:BLK1_OFFSET+56]
SRC1_BYTE8 ← SRC1[BLK1_OFFSET+71:BLK1_OFFSET+64]
SRC1_BYTE9 ← SRC1[BLK1_OFFSET+79:BLK1_OFFSET+72]
SRC1_BYTE10 ← SRC1[BLK1_OFFSET+87:BLK1_OFFSET+80]

SRC2_BYTE0 ←SRC2[BLK2_OFFSET+7:BLK2_OFFSET]
SRC2_BYTE1 ← SRC2[BLK2_OFFSET+15:BLK2_OFFSET+8]
SRC2_BYTE2 ← SRC2[BLK2_OFFSET+23:BLK2_OFFSET+16]
SRC2_BYTE3 ← SRC2[BLK2_OFFSET+31:BLK2_OFFSET+24]

```
TEMP0 ← ABS(SRC1_BYTE0 - SRC2_BYTE0)
TEMP1 ← ABS(SRC1_BYTE1 - SRC2_BYTE1)
TEMP2 ← ABS(SRC1_BYTE2 - SRC2_BYTE2)
TEMP3 ← ABS(SRC1_BYTE3 - SRC2_BYTE3)
DEST[15:0] ← TEMP0 + TEMP1 + TEMP2 + TEMP3

TEMP0 ← ABS(SRC1_BYTE1 - SRC2_BYTE0)
TEMP1 ← ABS(SRC1_BYTE2 - SRC2_BYTE1)
TEMP2 ← ABS(SRC1_BYTE3 - SRC2_BYTE2)
TEMP3 ← ABS(SRC1_BYTE4 - SRC2_BYTE3)
DEST[31:16] ← TEMP0 + TEMP1 + TEMP2 + TEMP3

TEMP0 ← ABS(SRC1_BYTE2 - SRC2_BYTE0)
TEMP1 ← ABS(SRC1_BYTE3 - SRC2_BYTE1)
TEMP2 ← ABS(SRC1_BYTE4 - SRC2_BYTE2)
TEMP3 ← ABS(SRC1_BYTE5 - SRC2_BYTE3)
DEST[47:32] ← TEMP0 + TEMP1 + TEMP2 + TEMP3

TEMP0 ← ABS(SRC1_BYTE3 - SRC2_BYTE0)
TEMP1 ← ABS(SRC1_BYTE4 - SRC2_BYTE1)
TEMP2 ← ABS(SRC1_BYTE5 - SRC2_BYTE2)
TEMP3 ← ABS(SRC1_BYTE6 - SRC2_BYTE3)
DEST[63:48] ← TEMP0 + TEMP1 + TEMP2 + TEMP3

TEMP0 ← ABS(SRC1_BYTE4 - SRC2_BYTE0)
TEMP1 ← ABS(SRC1_BYTE5 - SRC2_BYTE1)
TEMP2 ← ABS(SRC1_BYTE6 - SRC2_BYTE2)
TEMP3 ← ABS(SRC1_BYTE7 - SRC2_BYTE3)
DEST[79:64] ← TEMP0 + TEMP1 + TEMP2 + TEMP3

TEMP0 ← ABS(SRC1_BYTE5 - SRC2_BYTE0)
TEMP1 ← ABS(SRC1_BYTE6 - SRC2_BYTE1)
TEMP2 ← ABS(SRC1_BYTE7 - SRC2_BYTE2)
TEMP3 ← ABS(SRC1_BYTE8 - SRC2_BYTE3)
DEST[95:80] ← TEMP0 + TEMP1 + TEMP2 + TEMP3

TEMP0 ← ABS(SRC1_BYTE6 - SRC2_BYTE0)
TEMP1 ← ABS(SRC1_BYTE7 - SRC2_BYTE1)
TEMP2 ← ABS(SRC1_BYTE8 - SRC2_BYTE2)
TEMP3 ← ABS(SRC1_BYTE9 - SRC2_BYTE3)
DEST[111:96] ← TEMP0 + TEMP1 + TEMP2 + TEMP3

TEMP0 ← ABS(SRC1_BYTE7 - SRC2_BYTE0)
TEMP1 ← ABS(SRC1_BYTE8 - SRC2_BYTE1)
TEMP2 ← ABS(SRC1_BYTE9 - SRC2_BYTE2)
TEMP3 ← ABS(SRC1_BYTE10 - SRC2_BYTE3)
DEST[127:112] ← TEMP0 + TEMP1 + TEMP2 + TEMP3
DEST[MAXVL-1:128] ← 0
```

MPSADBW (128-bit Legacy SSE version)

```
SRC_OFFSET ← imm8[1:0]*32
DEST_OFFSET ← imm8[2]*32
DEST_BYTE0 ← DEST[DEST_OFFSET+7:DEST_OFFSET]
DEST_BYTE1 ← DEST[DEST_OFFSET+15:DEST_OFFSET+8]
DEST_BYTE2 ← DEST[DEST_OFFSET+23:DEST_OFFSET+16]
DEST_BYTE3 ← DEST[DEST_OFFSET+31:DEST_OFFSET+24]
DEST_BYTE4 ← DEST[DEST_OFFSET+39:DEST_OFFSET+32]
DEST_BYTE5 ← DEST[DEST_OFFSET+47:DEST_OFFSET+40]
DEST_BYTE6 ← DEST[DEST_OFFSET+55:DEST_OFFSET+48]
DEST_BYTE7 ← DEST[DEST_OFFSET+63:DEST_OFFSET+56]
DEST_BYTE8 ← DEST[DEST_OFFSET+71:DEST_OFFSET+64]
DEST_BYTE9 ← DEST[DEST_OFFSET+79:DEST_OFFSET+72]
DEST_BYTE10 ← DEST[DEST_OFFSET+87:DEST_OFFSET+80]

SRC_BYTE0 ← SRC[SRC_OFFSET+7:SRC_OFFSET]
SRC_BYTE1 ← SRC[SRC_OFFSET+15:SRC_OFFSET+8]
SRC_BYTE2 ← SRC[SRC_OFFSET+23:SRC_OFFSET+16]
SRC_BYTE3 ← SRC[SRC_OFFSET+31:SRC_OFFSET+24]

TEMP0 ← ABS( DEST_BYTE0 - SRC_BYTE0)
TEMP1 ← ABS( DEST_BYTE1 - SRC_BYTE1)
TEMP2 ← ABS( DEST_BYTE2 - SRC_BYTE2)
TEMP3 ← ABS( DEST_BYTE3 - SRC_BYTE3)
DEST[15:0] ← TEMP0 + TEMP1 + TEMP2 + TEMP3

TEMP0 ← ABS( DEST_BYTE1 - SRC_BYTE0)
TEMP1 ← ABS( DEST_BYTE2 - SRC_BYTE1)
TEMP2 ← ABS( DEST_BYTE3 - SRC_BYTE2)
TEMP3 ← ABS( DEST_BYTE4 - SRC_BYTE3)
DEST[31:16] ← TEMP0 + TEMP1 + TEMP2 + TEMP3

TEMP0 ← ABS( DEST_BYTE2 - SRC_BYTE0)
TEMP1 ← ABS( DEST_BYTE3 - SRC_BYTE1)
TEMP2 ← ABS( DEST_BYTE4 - SRC_BYTE2)
TEMP3 ← ABS( DEST_BYTE5 - SRC_BYTE3)
DEST[47:32] ← TEMP0 + TEMP1 + TEMP2 + TEMP3

TEMP0 ← ABS( DEST_BYTE3 - SRC_BYTE0)
TEMP1 ← ABS( DEST_BYTE4 - SRC_BYTE1)
TEMP2 ← ABS( DEST_BYTE5 - SRC_BYTE2)
TEMP3 ← ABS( DEST_BYTE6 - SRC_BYTE3)
DEST[63:48] ← TEMP0 + TEMP1 + TEMP2 + TEMP3

TEMP0 ← ABS( DEST_BYTE4 - SRC_BYTE0)
TEMP1 ← ABS( DEST_BYTE5 - SRC_BYTE1)
TEMP2 ← ABS( DEST_BYTE6 - SRC_BYTE2)
TEMP3 ← ABS( DEST_BYTE7 - SRC_BYTE3)
DEST[79:64] ← TEMP0 + TEMP1 + TEMP2 + TEMP3
```

TEMP0 ← ABS(DEST_BYTE5 - SRC_BYTE0)
TEMP1 ← ABS(DEST_BYTE6 - SRC_BYTE1)
TEMP2 ← ABS(DEST_BYTE7 - SRC_BYTE2)
TEMP3 ← ABS(DEST_BYTE8 - SRC_BYTE3)
DEST[95:80] ← TEMP0 + TEMP1 + TEMP2 + TEMP3

TEMP0 ← ABS(DEST_BYTE6 - SRC_BYTE0)
TEMP1 ← ABS(DEST_BYTE7 - SRC_BYTE1)
TEMP2 ← ABS(DEST_BYTE8 - SRC_BYTE2)
TEMP3 ← ABS(DEST_BYTE9 - SRC_BYTE3)
DEST[111:96] ← TEMP0 + TEMP1 + TEMP2 + TEMP3

TEMP0 ← ABS(DEST_BYTE7 - SRC_BYTE0)
TEMP1 ← ABS(DEST_BYTE8 - SRC_BYTE1)
TEMP2 ← ABS(DEST_BYTE9 - SRC_BYTE2)
TEMP3 ← ABS(DEST_BYTE10 - SRC_BYTE3)
DEST[127:112] ← TEMP0 + TEMP1 + TEMP2 + TEMP3
DEST[MAXVL-1:128] (Unmodified)

Intel C/C++ Compiler Intrinsic Equivalent

(V)MPSADBW: __m128i _mm_mpsadbw_epu8 (__m128i s1, __m128i s2, const int mask);

VMPSADBW: __m256i _mm256_mpsadbw_epu8 (__m256i s1, __m256i s2, const int mask);

Flags Affected

None

Other Exceptions

See Exceptions Type 4; additionally

#UD	If VEX.L = 1.

MUL—Unsigned Multiply

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
F6 /4	MUL *r/m8*	M	Valid	Valid	Unsigned multiply (AX ← AL ∗ *r/m8*).
REX + F6 /4	MUL *r/m8**	M	Valid	N.E.	Unsigned multiply (AX ← AL ∗ *r/m8*).
F7 /4	MUL *r/m16*	M	Valid	Valid	Unsigned multiply (DX:AX ← AX ∗ *r/m16*).
F7 /4	MUL *r/m32*	M	Valid	Valid	Unsigned multiply (EDX:EAX ← EAX ∗ *r/m32*).
REX.W + F7 /4	MUL *r/m64*	M	Valid	N.E.	Unsigned multiply (RDX:RAX ← RAX ∗ *r/m64).*

NOTES:

* In 64-bit mode, r/m8 can not be encoded to access the following byte registers if a REX prefix is used: AH, BH, CH, DH.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
M	ModRM:r/m (r)	NA	NA	NA

Description

Performs an unsigned multiplication of the first operand (destination operand) and the second operand (source operand) and stores the result in the destination operand. The destination operand is an implied operand located in register AL, AX or EAX (depending on the size of the operand); the source operand is located in a general-purpose register or a memory location. The action of this instruction and the location of the result depends on the opcode and the operand size as shown in Table 4-9.

The result is stored in register AX, register pair DX:AX, or register pair EDX:EAX (depending on the operand size), with the high-order bits of the product contained in register AH, DX, or EDX, respectively. If the high-order bits of the product are 0, the CF and OF flags are cleared; otherwise, the flags are set.

In 64-bit mode, the instruction's default operation size is 32 bits. Use of the REX.R prefix permits access to additional registers (R8-R15). Use of the REX.W prefix promotes operation to 64 bits.

See the summary chart at the beginning of this section for encoding data and limits.

Table 4-9. MUL Results

Operand Size	Source 1	Source 2	Destination
Byte	AL	r/m8	AX
Word	AX	r/m16	DX:AX
Doubleword	EAX	r/m32	EDX:EAX
Quadword	RAX	r/m64	RDX:RAX

Operation

```
IF (Byte operation)
    THEN
        AX ← AL * SRC;
    ELSE (* Word or doubleword operation *)
        IF OperandSize = 16
            THEN
                DX:AX ← AX * SRC;
            ELSE IF OperandSize = 32
                THEN EDX:EAX ← EAX * SRC; FI;
            ELSE (* OperandSize = 64 *)
                RDX:RAX ← RAX * SRC;
        FI;
FI;
```

Flags Affected

The OF and CF flags are set to 0 if the upper half of the result is 0; otherwise, they are set to 1. The SF, ZF, AF, and PF flags are undefined.

Protected Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit. If the DS, ES, FS, or GS register contains a NULL segment selector.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If the LOCK prefix is used.

Real-Address Mode Exceptions

#GP	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS	If a memory operand effective address is outside the SS segment limit.
#UD	If the LOCK prefix is used.

Virtual-8086 Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.
#UD	If the LOCK prefix is used.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

MULPD—Multiply Packed Double-Precision Floating-Point Values

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 59 /r MULPD xmm1, xmm2/m128	A	V/V	SSE2	Multiply packed double-precision floating-point values in xmm2/m128 with xmm1 and store result in xmm1.
VEX.NDS.128.66.0F.WIG 59 /r VMULPD xmm1,xmm2, xmm3/m128	B	V/V	AVX	Multiply packed double-precision floating-point values in xmm3/m128 with xmm2 and store result in xmm1.
VEX.NDS.256.66.0F.WIG 59 /r VMULPD ymm1, ymm2, ymm3/m256	B	V/V	AVX	Multiply packed double-precision floating-point values in ymm3/m256 with ymm2 and store result in ymm1.
EVEX.NDS.128.66.0F.W1 59 /r VMULPD xmm1 {k1}{z}, xmm2, xmm3/m128/m64bcst	C	V/V	AVX512VL AVX512F	Multiply packed double-precision floating-point values from xmm3/m128/m64bcst to xmm2 and store result in xmm1.
EVEX.NDS.256.66.0F.W1 59 /r VMULPD ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst	C	V/V	AVX512VL AVX512F	Multiply packed double-precision floating-point values from ymm3/m256/m64bcst to ymm2 and store result in ymm1.
EVEX.NDS.512.66.0F.W1 59 /r VMULPD zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst{er}	C	V/V	AVX512F	Multiply packed double-precision floating-point values in zmm3/m512/m64bcst with zmm2 and store result in zmm1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	NA	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
B	NA	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA
C	Full Vector	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA

Description

Multiply packed double-precision floating-point values from the first source operand with corresponding values in the second source operand, and stores the packed double-precision floating-point results in the destination operand.

EVEX encoded versions: The first source operand (the second operand) is a ZMM/YMM/XMM register. The second source operand can be a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 64-bit memory location. The destination operand is a ZMM/YMM/XMM register conditionally updated with writemask k1.

VEX.256 encoded version: The first source operand is a YMM register. The second source operand can be a YMM register or a 256-bit memory location. The destination operand is a YMM register. Bits (MAXVL-1:256) of the corresponding destination ZMM register are zeroed.

VEX.128 encoded version: The first source operand is a XMM register. The second source operand can be a XMM register or a 128-bit memory location. The destination operand is a XMM register. The upper bits (MAXVL-1:128) of the destination YMM register destination are zeroed.

128-bit Legacy SSE version: The second source can be an XMM register or an 128-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (MAXVL-1:128) of the corresponding ZMM register destination are unmodified.

Operation

VMULPD (EVEX encoded versions)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
IF (VL = 512) AND (EVEX.b = 1) AND SRC2 *is a register*
    THEN
        SET_RM(EVEX.RC);
    ELSE
        SET_RM(MXCSR.RM);
FI;
FOR j ← 0 TO KL-1
    i ← j * 64
    IF k1[j] OR *no writemask*
        THEN
            IF (EVEX.b = 1) AND (SRC2 *is memory*)
                THEN
                    DEST[i+63:i] ← SRC1[i+63:i] * SRC2[63:0]
                ELSE
                    DEST[i+63:i] ← SRC1[i+63:i] * SRC2[i+63:i]
            FI;
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE                                ; zeroing-masking
                    DEST[i+63:i] ← 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0
```

VMULPD (VEX.256 encoded version)

```
DEST[63:0] ←SRC1[63:0] * SRC2[63:0]
DEST[127:64] ←SRC1[127:64] * SRC2[127:64]
DEST[191:128] ←SRC1[191:128] * SRC2[191:128]
DEST[255:192] ←SRC1[255:192] * SRC2[255:192]
DEST[MAXVL-1:256] ←0;
```

.

VMULPD (VEX.128 encoded version)

```
DEST[63:0] ←SRC1[63:0] * SRC2[63:0]
DEST[127:64] ←SRC1[127:64] * SRC2[127:64]
DEST[MAXVL-1:128] ←0
```

MULPD (128-bit Legacy SSE version)

```
DEST[63:0] ←DEST[63:0] * SRC[63:0]
DEST[127:64] ←DEST[127:64] * SRC[127:64]
DEST[MAXVL-1:128] (Unmodified)
```

Intel C/C++ Compiler Intrinsic Equivalent

VMULPD __m512d _mm512_mul_pd(__m512d a, __m512d b);
VMULPD __m512d _mm512_mask_mul_pd(__m512d s, __mmask8 k, __m512d a, __m512d b);
VMULPD __m512d _mm512_maskz_mul_pd(__mmask8 k, __m512d a, __m512d b);
VMULPD __m512d _mm512_mul_round_pd(__m512d a, __m512d b, int);
VMULPD __m512d _mm512_mask_mul_round_pd(__m512d s, __mmask8 k, __m512d a, __m512d b, int);
VMULPD __m512d _mm512_maskz_mul_round_pd(__mmask8 k, __m512d a, __m512d b, int);
VMULPD __m256d _mm256_mul_pd (__m256d a, __m256d b);
MULPD __m128d _mm_mul_pd (__m128d a, __m128d b);

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal

Other Exceptions

Non-EVEX-encoded instruction, see Exceptions Type 2.
EVEX-encoded instruction, see Exceptions Type E2.

MULPS—Multiply Packed Single-Precision Floating-Point Values

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F 59 /r MULPS xmm1, xmm2/m128	A	V/V	SSE	Multiply packed single-precision floating-point values in xmm2/m128 with xmm1 and store result in xmm1.
VEX.NDS.128.0F.WIG 59 /r VMULPS xmm1,xmm2, xmm3/m128	B	V/V	AVX	Multiply packed single-precision floating-point values in xmm3/m128 with xmm2 and store result in xmm1.
VEX.NDS.256.0F.WIG 59 /r VMULPS ymm1, ymm2, ymm3/m256	B	V/V	AVX	Multiply packed single-precision floating-point values in ymm3/m256 with ymm2 and store result in ymm1.
EVEX.NDS.128.0F.W0 59 /r VMULPS xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst	C	V/V	AVX512VL AVX512F	Multiply packed single-precision floating-point values from xmm3/m128/m32bcst to xmm2 and store result in xmm1.
EVEX.NDS.256.0F.W0 59 /r VMULPS ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst	C	V/V	AVX512VL AVX512F	Multiply packed single-precision floating-point values from ymm3/m256/m32bcst to ymm2 and store result in ymm1.
EVEX.NDS.512.0F.W0 59 /r VMULPS zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst {er}	C	V/V	AVX512F	Multiply packed single-precision floating-point values in zmm3/m512/m32bcst with zmm2 and store result in zmm1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	NA	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
B	NA	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA
C	Full Vector	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA

Description

Multiply the packed single-precision floating-point values from the first source operand with the corresponding values in the second source operand, and stores the packed double-precision floating-point results in the destination operand.

EVEX encoded versions: The first source operand (the second operand) is a ZMM/YMM/XMM register. The second source operand can be a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 32-bit memory location. The destination operand is a ZMM/YMM/XMM register conditionally updated with writemask k1.

VEX.256 encoded version: The first source operand is a YMM register. The second source operand can be a YMM register or a 256-bit memory location. The destination operand is a YMM register. Bits (MAXVL-1:256) of the corresponding destination ZMM register are zeroed.

VEX.128 encoded version: The first source operand is a XMM register. The second source operand can be a XMM register or a 128-bit memory location. The destination operand is a XMM register. The upper bits (MAXVL-1:128) of the destination YMM register destination are zeroed.

128-bit Legacy SSE version: The second source can be an XMM register or an 128-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (MAXVL-1:128) of the corresponding ZMM register destination are unmodified.

Operation

VMULPS (EVEX encoded version)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
IF (VL = 512) AND (EVEX.b = 1) AND SRC2 *is a register*
    THEN
        SET_RM(EVEX.RC);
    ELSE
        SET_RM(MXCSR.RM);
FI;
FOR j ← 0 TO KL-1
    i ← j * 32
    IF k1[j] OR *no writemask*
        THEN
            IF (EVEX.b = 1) AND (SRC2 *is memory*)
                THEN
                    DEST[i+31:i] ← SRC1[i+31:i] * SRC2[31:0]
                ELSE
                    DEST[i+31:i] ← SRC1[i+31:i] * SRC2[i+31:i]
            FI;
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE                                ; zeroing-masking
                    DEST[i+31:i] ← 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0
```

VMULPS (VEX.256 encoded version)

```
DEST[31:0] ←SRC1[31:0] * SRC2[31:0]
DEST[63:32] ←SRC1[63:32] * SRC2[63:32]
DEST[95:64] ←SRC1[95:64] * SRC2[95:64]
DEST[127:96] ←SRC1[127:96] * SRC2[127:96]
DEST[159:128] ←SRC1[159:128] * SRC2[159:128]
DEST[191:160]←SRC1[191:160] * SRC2[191:160]
DEST[223:192] ←SRC1[223:192] * SRC2[223:192]
DEST[255:224] ←SRC1[255:224] * SRC2[255:224].
DEST[MAXVL-1:256] ←0;
```

VMULPS (VEX.128 encoded version)

```
DEST[31:0] ←SRC1[31:0] * SRC2[31:0]
DEST[63:32] ←SRC1[63:32] * SRC2[63:32]
DEST[95:64] ←SRC1[95:64] * SRC2[95:64]
DEST[127:96] ←SRC1[127:96] * SRC2[127:96]
DEST[MAXVL-1:128] ←0
```

MULPS (128-bit Legacy SSE version)

```
DEST[31:0] ←SRC1[31:0] * SRC2[31:0]
DEST[63:32] ←SRC1[63:32] * SRC2[63:32]
DEST[95:64] ←SRC1[95:64] * SRC2[95:64]
DEST[127:96] ←SRC1[127:96] * SRC2[127:96]
DEST[MAXVL-1:128] (Unmodified)
```

Intel C/C++ Compiler Intrinsic Equivalent

VMULPS __m512 _mm512_mul_ps(__m512 a, __m512 b);
VMULPS __m512 _mm512_mask_mul_ps(__m512 s, __mmask16 k, __m512 a, __m512 b);
VMULPS __m512 _mm512_maskz_mul_ps(__mmask16 k, __m512 a, __m512 b);
VMULPS __m512 _mm512_mul_round_ps(__m512 a, __m512 b, int);
VMULPS __m512 _mm512_mask_mul_round_ps(__m512 s, __mmask16 k, __m512 a, __m512 b, int);
VMULPS __m512 _mm512_maskz_mul_round_ps(__mmask16 k, __m512 a, __m512 b, int);
VMULPS __m256 _mm256_mask_mul_ps(__m256 s, __mmask8 k, __m256 a, __m256 b);
VMULPS __m256 _mm256_maskz_mul_ps(__mmask8 k, __m256 a, __m256 b);
VMULPS __m128 _mm_mask_mul_ps(__m128 s, __mmask8 k, __m128 a, __m128 b);
VMULPS __m128 _mm_maskz_mul_ps(__mmask8 k, __m128 a, __m128 b);
VMULPS __m256 _mm256_mul_ps (__m256 a, __m256 b);
MULPS __m128 _mm_mul_ps (__m128 a, __m128 b);

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal

Other Exceptions

Non-EVEX-encoded instruction, see Exceptions Type 2.
EVEX-encoded instruction, see Exceptions Type E2.

MULSD—Multiply Scalar Double-Precision Floating-Point Value

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
F2 0F 59 /r MULSD xmm1,xmm2/m64	A	V/V	SSE2	Multiply the low double-precision floating-point value in xmm2/m64 by low double-precision floating-point value in xmm1.
VEX.NDS.LIG.F2.0F.WIG 59 /r VMULSD xmm1,xmm2, xmm3/m64	B	V/V	AVX	Multiply the low double-precision floating-point value in xmm3/m64 by low double-precision floating-point value in xmm2.
EVEX.NDS.LIG.F2.0F.W1 59 /r VMULSD xmm1 {k1}{z}, xmm2, xmm3/m64 {er}	C	V/V	AVX512F	Multiply the low double-precision floating-point value in xmm3/m64 by low double-precision floating-point value in xmm2.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	NA	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
B	NA	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA
C	Tuple1 Scalar	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA

Description

Multiplies the low double-precision floating-point value in the second source operand by the low double-precision floating-point value in the first source operand, and stores the double-precision floating-point result in the destination operand. The second source operand can be an XMM register or a 64-bit memory location. The first source operand and the destination operands are XMM registers.

128-bit Legacy SSE version: The first source operand and the destination operand are the same. Bits (MAXVL-1:64) of the corresponding destination register remain unchanged.

VEX.128 and EVEX encoded version: The quadword at bits 127:64 of the destination operand is copied from the same bits of the first source operand. Bits (MAXVL-1:128) of the destination register are zeroed.

EVEX encoded version: The low quadword element of the destination operand is updated according to the writemask.

Software should ensure VMULSD is encoded with VEX.L=0. Encoding VMULSD with VEX.L=1 may encounter unpredictable behavior across different processor generations.

Operation

VMULSD (EVEX encoded version)

```
IF (EVEX.b = 1) AND SRC2 *is a register*
    THEN
        SET_RM(EVEX.RC);
    ELSE
        SET_RM(MXCSR.RM);
FI;
IF k1[0] or *no writemask*
    THEN    DEST[63:0] ← SRC1[63:0] * SRC2[63:0]
    ELSE
        IF *merging-masking*                    ; merging-masking
            THEN *DEST[63:0] remains unchanged*
            ELSE                                ; zeroing-masking
                THEN DEST[63:0] ← 0
            FI
    FI;
ENDFOR
DEST[127:64] ← SRC1[127:64]
DEST[MAXVL-1:128] ← 0
```

VMULSD (VEX.128 encoded version)

```
DEST[63:0] ←SRC1[63:0] * SRC2[63:0]
DEST[127:64] ←SRC1[127:64]
DEST[MAXVL-1:128] ←0
```

MULSD (128-bit Legacy SSE version)

```
DEST[63:0] ←DEST[63:0] * SRC[63:0]
DEST[MAXVL-1:64] (Unmodified)
```

Intel C/C++ Compiler Intrinsic Equivalent

VMULSD __m128d _mm_mask_mul_sd(__m128d s, __mmask8 k, __m128d a, __m128d b);
VMULSD __m128d _mm_maskz_mul_sd(__mmask8 k, __m128d a, __m128d b);
VMULSD __m128d _mm_mul_round_sd(__m128d a, __m128d b, int);
VMULSD __m128d _mm_mask_mul_round_sd(__m128d s, __mmask8 k, __m128d a, __m128d b, int);
VMULSD __m128d _mm_maskz_mul_round_sd(__mmask8 k, __m128d a, __m128d b, int);
MULSD __m128d _mm_mul_sd (__m128d a, __m128d b)

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal

Other Exceptions

Non-EVEX-encoded instruction, see Exceptions Type 3.
EVEX-encoded instruction, see Exceptions Type E3.

MULSS—Multiply Scalar Single-Precision Floating-Point Values

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
F3 0F 59 /r MULSS xmm1,xmm2/m32	A	V/V	SSE	Multiply the low single-precision floating-point value in xmm2/m32 by the low single-precision floating-point value in xmm1.
VEX.NDS.LIG.F3.0F.WIG 59 /r VMULSS xmm1,xmm2, xmm3/m32	B	V/V	AVX	Multiply the low single-precision floating-point value in xmm3/m32 by the low single-precision floating-point value in xmm2.
EVEX.NDS.LIG.F3.0F.W0 59 /r VMULSS xmm1 {k1}{z}, xmm2, xmm3/m32 {er}	C	V/V	AVX512F	Multiply the low single-precision floating-point value in xmm3/m32 by the low single-precision floating-point value in xmm2.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	NA	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
B	NA	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA
C	Tuple1 Scalar	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA

Description

Multiplies the low single-precision floating-point value from the second source operand by the low single-precision floating-point value in the first source operand, and stores the single-precision floating-point result in the destination operand. The second source operand can be an XMM register or a 32-bit memory location. The first source operand and the destination operands are XMM registers.

128-bit Legacy SSE version: The first source operand and the destination operand are the same. Bits (MAXVL-1:32) of the corresponding YMM destination register remain unchanged.

VEX.128 and EVEX encoded version: The first source operand is an xmm register encoded by VEX.vvvv. The three high-order doublewords of the destination operand are copied from the first source operand. Bits (MAXVL-1:128) of the destination register are zeroed.

EVEX encoded version: The low doubleword element of the destination operand is updated according to the writemask.

Software should ensure VMULSS is encoded with VEX.L=0. Encoding VMULSS with VEX.L=1 may encounter unpredictable behavior across different processor generations.

Operation

VMULSS (EVEX encoded version)

```
IF (EVEX.b = 1) AND SRC2 *is a register*
    THEN
        SET_RM(EVEX.RC);
    ELSE
        SET_RM(MXCSR.RM);
FI;
IF k1[0] or *no writemask*
    THEN    DEST[31:0] ← SRC1[31:0] * SRC2[31:0]
    ELSE
        IF *merging-masking*                    ; merging-masking
            THEN *DEST[31:0] remains unchanged*
            ELSE                                ; zeroing-masking
                THEN DEST[31:0] ← 0
            FI
    FI;
ENDFOR
DEST[127:32] ← SRC1[127:32]
DEST[MAXVL-1:128] ← 0
```

VMULSS (VEX.128 encoded version)

```
DEST[31:0] ←SRC1[31:0] * SRC2[31:0]
DEST[127:32] ←SRC1[127:32]
DEST[MAXVL-1:128] ←0
```

MULSS (128-bit Legacy SSE version)

```
DEST[31:0] ←DEST[31:0] * SRC[31:0]
DEST[MAXVL-1:32] (Unmodified)
```

Intel C/C++ Compiler Intrinsic Equivalent

VMULSS __m128 _mm_mask_mul_ss(__m128 s, __mmask8 k, __m128 a, __m128 b);
VMULSS __m128 _mm_maskz_mul_ss(__mmask8 k, __m128 a, __m128 b);
VMULSS __m128 _mm_mul_round_ss(__m128 a, __m128 b, int);
VMULSS __m128 _mm_mask_mul_round_ss(__m128 s, __mmask8 k, __m128 a, __m128 b, int);
VMULSS __m128 _mm_maskz_mul_round_ss(__mmask8 k, __m128 a, __m128 b, int);
MULSS __m128 _mm_mul_ss(__m128 a, __m128 b)

SIMD Floating-Point Exceptions

Underflow, Overflow, Invalid, Precision, Denormal

Other Exceptions

Non-EVEX-encoded instruction, see Exceptions Type 3.
EVEX-encoded instruction, see Exceptions Type E3.

MULX — Unsigned Multiply Without Affecting Flags

Opcode/ Instruction	Op/ En	64/32 -bit Mode	CPUID Feature Flag	Description
VEX.NDD.LZ.F2.0F38.W0 F6 /r MULX *r32a, r32b, r/m32*	RVM	V/V	BMI2	Unsigned multiply of *r/m32* with EDX without affecting arithmetic flags.
VEX.NDD.LZ.F2.0F38.W1 F6 /r MULX *r64a, r64b, r/m64*	RVM	V/N.E.	BMI2	Unsigned multiply of *r/m64* with RDX without affecting arithmetic flags.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RVM	ModRM:reg (w)	VEX.vvvv (w)	ModRM:r/m (r)	RDX/EDX is implied 64/32 bits source

Description

Performs an unsigned multiplication of the implicit source operand (EDX/RDX) and the specified source operand (the third operand) and stores the low half of the result in the second destination (second operand), the high half of the result in the first destination operand (first operand), without reading or writing the arithmetic flags. This enables efficient programming where the software can interleave add with carry operations and multiplications.

If the first and second operand are identical, it will contain the high half of the multiplication result.

This instruction is not supported in real mode and virtual-8086 mode. The operand size is always 32 bits if not in 64-bit mode. In 64-bit mode operand size 64 requires VEX.W1. VEX.W1 is ignored in non-64-bit modes. An attempt to execute this instruction with VEX.L not equal to 0 will cause #UD.

Operation

```
// DEST1: ModRM:reg
// DEST2: VEX.vvvv
IF (OperandSize = 32)
    SRC1 ← EDX;
    DEST2 ← (SRC1*SRC2)[31:0];
    DEST1 ← (SRC1*SRC2)[63:32];
ELSE IF (OperandSize = 64)
    SRC1 ← RDX;
        DEST2 ← (SRC1*SRC2)[63:0];
        DEST1 ← (SRC1*SRC2)[127:64];
FI
```

Flags Affected

None

Intel C/C++ Compiler Intrinsic Equivalent

Auto-generated from high-level language when possible.

unsigned int mulx_u32(unsigned int a, unsigned int b, unsigned int * hi);

unsigned __int64 mulx_u64(unsigned __int64 a, unsigned __int64 b, unsigned __int64 * hi);

SIMD Floating-Point Exceptions

None

Other Exceptions

See Section 2.5.1, "Exception Conditions for VEX-Encoded GPR Instructions", Table 2-29; additionally

#UD	If VEX.W = 1.

MWAIT—Monitor Wait

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
0F 01 C9	MWAIT	ZO	Valid	Valid	A hint that allow the processor to stop instruction execution and enter an implementation-dependent optimized state until occurrence of a class of events.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
ZO	NA	NA	NA	NA

Description

MWAIT instruction provides hints to allow the processor to enter an implementation-dependent optimized state. There are two principal targeted usages: address-range monitor and advanced power management. Both usages of MWAIT require the use of the MONITOR instruction.

CPUID.01H:ECX.MONITOR[bit 3] indicates the availability of MONITOR and MWAIT in the processor. When set, MWAIT may be executed only at privilege level 0 (use at any other privilege level results in an invalid-opcode exception). The operating system or system BIOS may disable this instruction by using the IA32_MISC_ENABLE MSR; disabling MWAIT clears the CPUID feature flag and causes execution to generate an invalid-opcode exception.

This instruction's operation is the same in non-64-bit modes and 64-bit mode.

ECX specifies optional extensions for the MWAIT instruction. EAX may contain hints such as the preferred optimized state the processor should enter. The first processors to implement MWAIT supported only the zero value for EAX and ECX. Later processors allowed setting ECX[0] to enable masked interrupts as break events for MWAIT (see below). Software can use the CPUID instruction to determine the extensions and hints supported by the processor.

MWAIT for Address Range Monitoring

For address-range monitoring, the MWAIT instruction operates with the MONITOR instruction. The two instructions allow the definition of an address at which to wait (MONITOR) and a implementation-dependent-optimized operation to commence at the wait address (MWAIT). The execution of MWAIT is a hint to the processor that it can enter an implementation-dependent-optimized state while waiting for an event or a store operation to the address range armed by MONITOR.

The following cause the processor to exit the implementation-dependent-optimized state: a store to the address range armed by the MONITOR instruction, an NMI or SMI, a debug exception, a machine check exception, the BINIT# signal, the INIT# signal, and the RESET# signal. Other implementation-dependent events may also cause the processor to exit the implementation-dependent-optimized state.

In addition, an external interrupt causes the processor to exit the implementation-dependent-optimized state either (1) if the interrupt would be delivered to software (e.g., as it would be if HLT had been executed instead of MWAIT); or (2) if ECX[0] = 1. Software can execute MWAIT with ECX[0] = 1 only if CPUID.05H:ECX[bit 1] = 1. (Implementation-specific conditions may result in an interrupt causing the processor to exit the implementation-dependent-optimized state even if interrupts are masked and ECX[0] = 0.)

Following exit from the implementation-dependent-optimized state, control passes to the instruction following the MWAIT instruction. A pending interrupt that is not masked (including an NMI or an SMI) may be delivered before execution of that instruction. Unlike the HLT instruction, the MWAIT instruction does not support a restart at the MWAIT instruction following the handling of an SMI.

If the preceding MONITOR instruction did not successfully arm an address range or if the MONITOR instruction has not been executed prior to executing MWAIT, then the processor will not enter the implementation-dependent-optimized state. Execution will resume at the instruction following the MWAIT.

MWAIT for Power Management

MWAIT accepts a hint and optional extension to the processor that it can enter a specified target C state while waiting for an event or a store operation to the address range armed by MONITOR. Support for MWAIT extensions for power management is indicated by CPUID.05H:ECX[bit 0] reporting 1.

EAX and ECX are used to communicate the additional information to the MWAIT instruction, such as the kind of optimized state the processor should enter. ECX specifies optional extensions for the MWAIT instruction. EAX may contain hints such as the preferred optimized state the processor should enter. Implementation-specific conditions may cause a processor to ignore the hint and enter a different optimized state. Future processor implementations may implement several optimized "waiting" states and will select among those states based on the hint argument.

Table 4-10 describes the meaning of ECX and EAX registers for MWAIT extensions.

Table 4-10. MWAIT Extension Register (ECX)

Bits	Description
0	Treat interrupts as break events even if masked (e.g., even if EFLAGS.IF=0). May be set only if CPUID.05H:ECX[bit 1] = 1.
31: 1	Reserved

Table 4-11. MWAIT Hints Register (EAX)

Bits	Description
3 : 0	Sub C-state within a C-state, indicated by bits [7:4]
7 : 4	Target C-state* Value of 0 means C1; 1 means C2 and so on Value of 01111B means C0 Note: Target C states for MWAIT extensions are processor-specific C-states, not ACPI C-states
31: 8	Reserved

Note that if MWAIT is used to enter any of the C-states that are numerically higher than C1, a store to the address range armed by the MONITOR instruction will cause the processor to exit MWAIT only if the store was originated by other processor agents. A store from non-processor agent might not cause the processor to exit MWAIT in such cases.

For additional details of MWAIT extensions, see Chapter 14, "Power and Thermal Management," of *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*.

Operation

```
(* MWAIT takes the argument in EAX as a hint extension and is architected to take the argument in ECX as an instruction extension
MWAIT EAX, ECX *)
{
WHILE ( ("Monitor Hardware is in armed state")) {
    implementation_dependent_optimized_state(EAX, ECX); }
Set the state of Monitor Hardware as triggered;
}
```

Intel C/C++ Compiler Intrinsic Equivalent

MWAIT: void _mm_mwait(unsigned extensions, unsigned hints)

Example

MONITOR/MWAIT instruction pair must be coded in the same loop because execution of the MWAIT instruction will trigger the monitor hardware. It is not a proper usage to execute MONITOR once and then execute MWAIT in a loop. Setting up MONITOR without executing MWAIT has no adverse effects.

Typically the MONITOR/MWAIT pair is used in a sequence, such as:

```
EAX = Logical Address(Trigger)
ECX = 0 (*Hints *)
EDX = 0 (* Hints *)

IF ( !trigger_store_happened) {
    MONITOR EAX, ECX, EDX
    IF ( !trigger_store_happened ) {
        MWAIT EAX, ECX
    }
}
```

The above code sequence makes sure that a triggering store does not happen between the first check of the trigger and the execution of the monitor instruction. Without the second check that triggering store would go un-noticed. Typical usage of MONITOR and MWAIT would have the above code sequence within a loop.

Numeric Exceptions

None

Protected Mode Exceptions

#GP(0)	If ECX[31:1] ≠ 0.
	If ECX[0] = 1 and CPUID.05H:ECX[bit 1] = 0.
#UD	If CPUID.01H:ECX.MONITOR[bit 3] = 0.
	If current privilege level is not 0.

Real Address Mode Exceptions

#GP	If ECX[31:1] ≠ 0.
	If ECX[0] = 1 and CPUID.05H:ECX[bit 1] = 0.
#UD	If CPUID.01H:ECX.MONITOR[bit 3] = 0.

Virtual 8086 Mode Exceptions

#UD	The MWAIT instruction is not recognized in virtual-8086 mode (even if CPUID.01H:ECX.MONITOR[bit 3] = 1).

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#GP(0)	If RCX[63:1] ≠ 0.
	If RCX[0] = 1 and CPUID.05H:ECX[bit 1] = 0.
#UD	If the current privilege level is not 0.
	If CPUID.01H:ECX.MONITOR[bit 3] = 0.

NEG—Two's Complement Negation

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
F6 /3	NEG *r/m8*	M	Valid	Valid	Two's complement negate *r/m8.*
REX + F6 /3	NEG *r/m8**	M	Valid	N.E.	Two's complement negate *r/m8.*
F7 /3	NEG *r/m16*	M	Valid	Valid	Two's complement negate *r/m16.*
F7 /3	NEG *r/m32*	M	Valid	Valid	Two's complement negate *r/m32.*
REX.W + F7 /3	NEG *r/m64*	M	Valid	N.E.	Two's complement negate *r/m64.*

NOTES:

* In 64-bit mode, r/m8 can not be encoded to access the following byte registers if a REX prefix is used: AH, BH, CH, DH.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
M	ModRM:r/m (r, w)	NA	NA	NA

Description

Replaces the value of operand (the destination operand) with its two's complement. (This operation is equivalent to subtracting the operand from 0.) The destination operand is located in a general-purpose register or a memory location.

This instruction can be used with a LOCK prefix to allow the instruction to be executed atomically.

In 64-bit mode, the instruction's default operation size is 32 bits. Using a REX prefix in the form of REX.R permits access to additional registers (R8-R15). Using a REX prefix in the form of REX.W promotes operation to 64 bits. See the summary chart at the beginning of this section for encoding data and limits.

Operation

```
IF DEST = 0
    THEN CF ← 0;
    ELSE CF ← 1;
FI;
DEST ← [– (DEST)]
```

Flags Affected

The CF flag set to 0 if the source operand is 0; otherwise it is set to 1. The OF, SF, ZF, AF, and PF flags are set according to the result.

Protected Mode Exceptions

#GP(0)	If the destination is located in a non-writable segment.
	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
	If the DS, ES, FS, or GS register contains a NULL segment selector.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If the LOCK prefix is used but the destination is not a memory operand.

Real-Address Mode Exceptions

#GP	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS	If a memory operand effective address is outside the SS segment limit.
#UD	If the LOCK prefix is used but the destination is not a memory operand.

Virtual-8086 Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.
#UD	If the LOCK prefix is used but the destination is not a memory operand.

Compatibility Mode Exceptions

Same as for protected mode exceptions.

64-Bit Mode Exceptions

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form.
#PF(fault-code)	For a page fault.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If the LOCK prefix is used but the destination is not a memory operand.

NOP—No Operation

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
NP 90	NOP	ZO	Valid	Valid	One byte no-operation instruction.
NP 0F 1F /0	NOP r/m16	M	Valid	Valid	Multi-byte no-operation instruction.
NP 0F 1F /0	NOP r/m32	M	Valid	Valid	Multi-byte no-operation instruction.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
ZO	NA	NA	NA	NA
M	ModRM:r/m (r)	NA	NA	NA

Description

This instruction performs no operation. It is a one-byte or multi-byte NOP that takes up space in the instruction stream but does not impact machine context, except for the EIP register.

The multi-byte form of NOP is available on processors with model encoding:

- CPUID.01H.EAX[Bytes 11:8] = 0110B or 1111B

The multi-byte NOP instruction does not alter the content of a register and will not issue a memory operation. The instruction's operation is the same in non-64-bit modes and 64-bit mode.

Operation

The one-byte NOP instruction is an alias mnemonic for the XCHG (E)AX, (E)AX instruction.

The multi-byte NOP instruction performs no operation on supported processors and generates undefined opcode exception on processors that do not support the multi-byte NOP instruction.

The memory operand form of the instruction allows software to create a byte sequence of "no operation" as one instruction. For situations where multiple-byte NOPs are needed, the recommended operations (32-bit mode and 64-bit mode) are:

Table 4-12. Recommended Multi-Byte Sequence of NOP Instruction

Length	Assembly	Byte Sequence
2 bytes	66 NOP	66 90H
3 bytes	NOP DWORD ptr [EAX]	0F 1F 00H
4 bytes	NOP DWORD ptr [EAX + 00H]	0F 1F 40 00H
5 bytes	NOP DWORD ptr [EAX + EAX*1 + 00H]	0F 1F 44 00 00H
6 bytes	66 NOP DWORD ptr [EAX + EAX*1 + 00H]	66 0F 1F 44 00 00H
7 bytes	NOP DWORD ptr [EAX + 00000000H]	0F 1F 80 00 00 00 00H
8 bytes	NOP DWORD ptr [EAX + EAX*1 + 00000000H]	0F 1F 84 00 00 00 00 00H
9 bytes	66 NOP DWORD ptr [EAX + EAX*1 + 00000000H]	66 0F 1F 84 00 00 00 00 00H

Flags Affected

None

Exceptions (All Operating Modes)

#UD	If the LOCK prefix is used.

NOT—One's Complement Negation

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
F6 /2	NOT *r/m8*	M	Valid	Valid	Reverse each bit of *r/m8.*
REX + F6 /2	NOT *r/m8**	M	Valid	N.E.	Reverse each bit of *r/m8.*
F7 /2	NOT *r/m16*	M	Valid	Valid	Reverse each bit of *r/m16.*
F7 /2	NOT *r/m32*	M	Valid	Valid	Reverse each bit of *r/m32.*
REX.W + F7 /2	NOT *r/m64*	M	Valid	N.E.	Reverse each bit of *r/m64.*

NOTES:

* In 64-bit mode, r/m8 can not be encoded to access the following byte registers if a REX prefix is used: AH, BH, CH, DH.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
M	ModRM:r/m (r, w)	NA	NA	NA

Description

Performs a bitwise NOT operation (each 1 is set to 0, and each 0 is set to 1) on the destination operand and stores the result in the destination operand location. The destination operand can be a register or a memory location.

This instruction can be used with a LOCK prefix to allow the instruction to be executed atomically.

In 64-bit mode, the instruction's default operation size is 32 bits. Using a REX prefix in the form of REX.R permits access to additional registers (R8-R15). Using a REX prefix in the form of REX.W promotes operation to 64 bits. See the summary chart at the beginning of this section for encoding data and limits.

Operation

DEST ← NOT DEST;

Flags Affected

None

Protected Mode Exceptions

#GP(0)	If the destination operand points to a non-writable segment. If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit. If the DS, ES, FS, or GS register contains a NULL segment selector.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If the LOCK prefix is used but the destination is not a memory operand.

Real-Address Mode Exceptions

#GP	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS	If a memory operand effective address is outside the SS segment limit.
#UD	If the LOCK prefix is used but the destination is not a memory operand.

Virtual-8086 Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.
#UD	If the LOCK prefix is used but the destination is not a memory operand.

Compatibility Mode Exceptions

Same as for protected mode exceptions.

64-Bit Mode Exceptions

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If the LOCK prefix is used but the destination is not a memory operand.

OR—Logical Inclusive OR

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
0C *ib*	OR AL, i*mm8*	I	Valid	Valid	AL OR *imm8.*
0D *iw*	OR AX, i*mm16*	I	Valid	Valid	AX OR *imm16.*
0D *id*	OR EAX, i*mm32*	I	Valid	Valid	EAX OR *imm32.*
REX.W + 0D *id*	OR RAX, i*mm32*	I	Valid	N.E.	RAX OR *imm32 (sign-extended).*
80 /1 *ib*	OR *r/m8, imm8*	MI	Valid	Valid	*r/m8* OR *imm8.*
REX + 80 /1 *ib*	OR *r/m8*, imm8*	MI	Valid	N.E.	*r/m8* OR *imm8.*
81 /1 *iw*	OR *r/m16, imm16*	MI	Valid	Valid	*r/m16* OR *imm16.*
81 /1 *id*	OR *r/m32, imm32*	MI	Valid	Valid	*r/m32* OR *imm32.*
REX.W + 81 /1 *id*	OR *r/m64, imm32*	MI	Valid	N.E.	*r/m64* OR *imm32 (sign-extended).*
83 /1 *ib*	OR *r/m16, imm8*	MI	Valid	Valid	*r/m16* OR *imm8 (sign-extended).*
83 /1 *ib*	OR *r/m32, imm8*	MI	Valid	Valid	*r/m32* OR *imm8 (sign-extended).*
REX.W + 83 /1 *ib*	OR *r/m64, imm8*	MI	Valid	N.E.	*r/m64* OR *imm8 (sign-extended).*
08 /*r*	OR *r/m8, r8*	MR	Valid	Valid	*r/m8* OR *r8.*
REX + 08 /*r*	OR *r/m8*, r8**	MR	Valid	N.E.	*r/m8* OR *r8.*
09 /*r*	OR *r/m16, r16*	MR	Valid	Valid	*r/m16* OR *r16.*
09 /*r*	OR *r/m32, r32*	MR	Valid	Valid	*r/m32* OR *r32.*
REX.W + 09 /*r*	OR *r/m64, r64*	MR	Valid	N.E.	*r/m64* OR *r64.*
0A /*r*	OR *r8, r/m8*	RM	Valid	Valid	*r8* OR *r/m8.*
REX + 0A /*r*	OR *r8*, r/m8**	RM	Valid	N.E.	*r8* OR *r/m8.*
0B /*r*	OR *r16, r/m16*	RM	Valid	Valid	*r16* OR *r/m16.*
0B /*r*	OR *r32, r/m32*	RM	Valid	Valid	*r32* OR *r/m32.*
REX.W + 0B /*r*	OR *r64, r/m64*	RM	Valid	N.E.	*r64* OR *r/m64.*

NOTES:

* In 64-bit mode, r/m8 can not be encoded to access the following byte registers if a REX prefix is used: AH, BH, CH, DH.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
I	AL/AX/EAX/RAX	imm8/16/32	NA	NA
MI	ModRM:r/m (r, w)	imm8/16/32	NA	NA
MR	ModRM:r/m (r, w)	ModRM:reg (r)	NA	NA
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA

Description

Performs a bitwise inclusive OR operation between the destination (first) and source (second) operands and stores the result in the destination operand location. The source operand can be an immediate, a register, or a memory location; the destination operand can be a register or a memory location. (However, two memory operands cannot be used in one instruction.) Each bit of the result of the OR instruction is set to 0 if both corresponding bits of the first and second operands are 0; otherwise, each bit is set to 1.

This instruction can be used with a LOCK prefix to allow the instruction to be executed atomically.

In 64-bit mode, the instruction's default operation size is 32 bits. Using a REX prefix in the form of REX.R permits access to additional registers (R8-R15). Using a REX prefix in the form of REX.W promotes operation to 64 bits. See the summary chart at the beginning of this section for encoding data and limits.

Operation

```
DEST ← DEST OR SRC;
```

Flags Affected

The OF and CF flags are cleared; the SF, ZF, and PF flags are set according to the result. The state of the AF flag is undefined.

Protected Mode Exceptions

#GP(0)	If the destination operand points to a non-writable segment.
	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
	If the DS, ES, FS, or GS register contains a NULL segment selector.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If the LOCK prefix is used but the destination is not a memory operand.

Real-Address Mode Exceptions

#GP	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS	If a memory operand effective address is outside the SS segment limit.
#UD	If the LOCK prefix is used but the destination is not a memory operand.

Virtual-8086 Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.
#UD	If the LOCK prefix is used but the destination is not a memory operand.

Compatibility Mode Exceptions

Same as for protected mode exceptions.

64-Bit Mode Exceptions

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If the LOCK prefix is used but the destination is not a memory operand.

ORPD—Bitwise Logical OR of Packed Double Precision Floating-Point Values

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 56/r ORPD xmm1, xmm2/m128	A	V/V	SSE2	Return the bitwise logical OR of packed double-precision floating-point values in xmm1 and xmm2/mem.
VEX.NDS.128.66.0F 56 /r VORPD xmm1,xmm2, xmm3/m128	B	V/V	AVX	Return the bitwise logical OR of packed double-precision floating-point values in xmm2 and xmm3/mem.
VEX.NDS.256.66.0F 56 /r VORPD ymm1, ymm2, ymm3/m256	B	V/V	AVX	Return the bitwise logical OR of packed double-precision floating-point values in ymm2 and ymm3/mem.
EVEX.NDS.128.66.0F.W1 56 /r VORPD xmm1 {k1}{z}, xmm2, xmm3/m128/m64bcst	C	V/V	AVX512VL AVX512DQ	Return the bitwise logical OR of packed double-precision floating-point values in xmm2 and xmm3/m128/m64bcst subject to writemask k1.
EVEX.NDS.256.66.0F.W1 56 /r VORPD ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst	C	V/V	AVX512VL AVX512DQ	Return the bitwise logical OR of packed double-precision floating-point values in ymm2 and ymm3/m256/m64bcst subject to writemask k1.
EVEX.NDS.512.66.0F.W1 56 /r VORPD zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst	C	V/V	AVX512DQ	Return the bitwise logical OR of packed double-precision floating-point values in zmm2 and zmm3/m512/m64bcst subject to writemask k1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	NA	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
B	NA	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA
C	Full Vector	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA

Description

Performs a bitwise logical OR of the two, four or eight packed double-precision floating-point values from the first source operand and the second source operand, and stores the result in the destination operand.

EVEX encoded versions: The first source operand is a ZMM/YMM/XMM register. The second source operand can be a ZMM/YMM/XMM register, a 512/256/128-bit memory location, or a 512/256/128-bit vector broadcasted from a 32-bit memory location. The destination operand is a ZMM/YMM/XMM register conditionally updated with writemask k1.

VEX.256 encoded version: The first source operand is a YMM register. The second source operand is a YMM register or a 256-bit memory location. The destination operand is a YMM register. The upper bits (MAXVL-1:256) of the corresponding ZMM register destination are zeroed.

VEX.128 encoded version: The first source operand is an XMM register. The second source operand is an XMM register or 128-bit memory location. The destination operand is an XMM register. The upper bits (MAXVL-1:128) of the corresponding ZMM register destination are zeroed.

128-bit Legacy SSE version: The second source can be an XMM register or an 128-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (MAXVL-1:128) of the corresponding register destination are unmodified.

Operation

VORPD (EVEX encoded versions)

(KL, VL) = (2, 128), (4, 256), (8, 512)

```
FOR j ← 0 TO KL-1
    i ← j * 64
    IF k1[j] OR *no writemask*
        THEN
            IF (EVEX.b == 1) AND (SRC2 *is memory*)
                THEN
                    DEST[i+63:i] ← SRC1[i+63:i] BITWISE OR SRC2[63:0]
                ELSE
                    DEST[i+63:i] ← SRC1[i+63:i] BITWISE OR SRC2[i+63:i]
            FI;
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE *zeroing-masking*                  ; zeroing-masking
                    DEST[i+63:i] ← 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0
```

VORPD (VEX.256 encoded version)

```
DEST[63:0] ← SRC1[63:0] BITWISE OR SRC2[63:0]
DEST[127:64] ← SRC1[127:64] BITWISE OR SRC2[127:64]
DEST[191:128] ← SRC1[191:128] BITWISE OR SRC2[191:128]
DEST[255:192] ← SRC1[255:192] BITWISE OR SRC2[255:192]
DEST[MAXVL-1:256] ← 0
```

VORPD (VEX.128 encoded version)

```
DEST[63:0] ← SRC1[63:0] BITWISE OR SRC2[63:0]
DEST[127:64] ← SRC1[127:64] BITWISE OR SRC2[127:64]
DEST[MAXVL-1:128] ← 0
```

ORPD (128-bit Legacy SSE version)

```
DEST[63:0] ← DEST[63:0] BITWISE OR SRC[63:0]
DEST[127:64] ← DEST[127:64] BITWISE OR SRC[127:64]
DEST[MAXVL-1:128] (Unmodified)
```

Intel C/C++ Compiler Intrinsic Equivalent

VORPD __m512d _mm512_or_pd (__m512d a, __m512d b);
VORPD __m512d _mm512_mask_or_pd (__m512d s, __mmask8 k, __m512d a, __m512d b);
VORPD __m512d _mm512_maskz_or_pd (__mmask8 k, __m512d a, __m512d b);
VORPD __m256d _mm256_mask_or_pd (__m256d s, ___mmask8 k, __m256d a, __m256d b);
VORPD __m256d _mm256_maskz_or_pd (__mmask8 k, __m256d a, __m256d b);
VORPD __m128d _mm_mask_or_pd (__m128d s, __mmask8 k, __m128d a, __m128d b);
VORPD __m128d _mm_maskz_or_pd (__mmask8 k, __m128d a, __m128d b);
VORPD __m256d _mm256_or_pd (__m256d a, __m256d b);
ORPD __m128d _mm_or_pd (__m128d a, __m128d b);

SIMD Floating-Point Exceptions

None

Other Exceptions

Non-EVEX-encoded instruction, see Exceptions Type 4.
EVEX-encoded instruction, see Exceptions Type E4.

ORPS—Bitwise Logical OR of Packed Single Precision Floating-Point Values

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F 56 /r ORPS xmm1, xmm2/m128	A	V/V	SSE	Return the bitwise logical OR of packed single-precision floating-point values in xmm1 and xmm2/mem.
VEX.NDS.128.0F 56 /r VORPS xmm1,xmm2, xmm3/m128	B	V/V	AVX	Return the bitwise logical OR of packed single-precision floating-point values in xmm2 and xmm3/mem.
VEX.NDS.256.0F 56 /r VORPS ymm1, ymm2, ymm3/m256	B	V/V	AVX	Return the bitwise logical OR of packed single-precision floating-point values in ymm2 and ymm3/mem.
EVEX.NDS.128.0F.W0 56 /r VORPS xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst	C	V/V	AVX512VL AVX512DQ	Return the bitwise logical OR of packed single-precision floating-point values in xmm2 and xmm3/m128/m32bcst subject to writemask k1.
EVEX.NDS.256.0F.W0 56 /r VORPS ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst	C	V/V	AVX512VL AVX512DQ	Return the bitwise logical OR of packed single-precision floating-point values in ymm2 and ymm3/m256/m32bcst subject to writemask k1.
EVEX.NDS.512.0F.W0 56 /r VORPS zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst	C	V/V	AVX512DQ	Return the bitwise logical OR of packed single-precision floating-point values in zmm2 and zmm3/m512/m32bcst subject to writemask k1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	NA	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
B	NA	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA
C	Full Vector	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA

Description

Performs a bitwise logical OR of the four, eight or sixteen packed single-precision floating-point values from the first source operand and the second source operand, and stores the result in the destination operand

EVEX encoded versions: The first source operand is a ZMM/YMM/XMM register. The second source operand can be a ZMM/YMM/XMM register, a 512/256/128-bit memory location, or a 512/256/128-bit vector broadcasted from a 32-bit memory location. The destination operand is a ZMM/YMM/XMM register conditionally updated with writemask k1.

VEX.256 encoded version: The first source operand is a YMM register. The second source operand is a YMM register or a 256-bit memory location. The destination operand is a YMM register. The upper bits (MAXVL-1:256) of the corresponding ZMM register destination are zeroed.

VEX.128 encoded version: The first source operand is an XMM register. The second source operand is an XMM register or 128-bit memory location. The destination operand is an XMM register. The upper bits (MAXVL-1:128) of the corresponding ZMM register destination are zeroed.

128-bit Legacy SSE version: The second source can be an XMM register or an 128-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (MAXVL-1:128) of the corresponding register destination are unmodified.

Operation

VORPS (EVEX encoded versions)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j ← 0 TO KL-1
    i ← j * 32
    IF k1[j] OR *no writemask*
        THEN
            IF (EVEX.b == 1) AND (SRC2 *is memory*)
                THEN
                    DEST[i+31:i] ← SRC1[i+31:i] BITWISE OR SRC2[31:0]
                ELSE
                    DEST[i+31:i] ← SRC1[i+31:i] BITWISE OR SRC2[i+31:i]
            FI;
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE *zeroing-masking*                  ; zeroing-masking
                    DEST[i+31:i] ← 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0
```

VORPS (VEX.256 encoded version)

```
DEST[31:0] ← SRC1[31:0] BITWISE OR SRC2[31:0]
DEST[63:32] ← SRC1[63:32] BITWISE OR SRC2[63:32]
DEST[95:64] ← SRC1[95:64] BITWISE OR SRC2[95:64]
DEST[127:96] ← SRC1[127:96] BITWISE OR SRC2[127:96]
DEST[159:128] ← SRC1[159:128] BITWISE OR SRC2[159:128]
DEST[191:160] ← SRC1[191:160] BITWISE OR SRC2[191:160]
DEST[223:192] ← SRC1[223:192] BITWISE OR SRC2[223:192]
DEST[255:224] ← SRC1[255:224] BITWISE OR SRC2[255:224].
DEST[MAXVL-1:256] ← 0
```

VORPS (VEX.128 encoded version)

```
DEST[31:0] ← SRC1[31:0] BITWISE OR SRC2[31:0]
DEST[63:32] ← SRC1[63:32] BITWISE OR SRC2[63:32]
DEST[95:64] ← SRC1[95:64] BITWISE OR SRC2[95:64]
DEST[127:96] ← SRC1[127:96] BITWISE OR SRC2[127:96]
DEST[MAXVL-1:128] ← 0
```

ORPS (128-bit Legacy SSE version)

```
DEST[31:0] ← SRC1[31:0] BITWISE OR SRC2[31:0]
DEST[63:32] ← SRC1[63:32] BITWISE OR SRC2[63:32]
DEST[95:64] ← SRC1[95:64] BITWISE OR SRC2[95:64]
DEST[127:96] ← SRC1[127:96] BITWISE OR SRC2[127:96]
DEST[MAXVL-1:128] (Unmodified)
```

Intel C/C++ Compiler Intrinsic Equivalent

VORPS __m512 _mm512_or_ps (__m512 a, __m512 b);
VORPS __m512 _mm512_mask_or_ps (__m512 s, __mmask16 k, __m512 a, __m512 b);
VORPS __m512 _mm512_maskz_or_ps (__mmask16 k, __m512 a, __m512 b);
VORPS __m256 _mm256_mask_or_ps (__m256 s, ___mmask8 k, __m256 a, __m256 b);
VORPS __m256 _mm256_maskz_or_ps (__mmask8 k, __m256 a, __m256 b);
VORPS __m128 _mm_mask_or_ps (__m128 s, __mmask8 k, __m128 a, __m128 b);
VORPS __m128 _mm_maskz_or_ps (__mmask8 k, __m128 a, __m128 b);
VORPS __m256 _mm256_or_ps (__m256 a, __m256 b);
ORPS __m128 _mm_or_ps (__m128 a, __m128 b);

SIMD Floating-Point Exceptions

None

Other Exceptions

Non-EVEX-encoded instruction, see Exceptions Type 4.

EVEX-encoded instruction, see Exceptions Type E4.

OUT—Output to Port

Opcode*	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
E6 *ib*	OUT *imm8*, AL	I	Valid	Valid	Output byte in AL to I/O port address *imm8*.
E7 *ib*	OUT *imm8*, AX	I	Valid	Valid	Output word in AX to I/O port address *imm8*.
E7 *ib*	OUT *imm8*, EAX	I	Valid	Valid	Output doubleword in EAX to I/O port address *imm8*.
EE	OUT DX, AL	ZO	Valid	Valid	Output byte in AL to I/O port address in DX.
EF	OUT DX, AX	ZO	Valid	Valid	Output word in AX to I/O port address in DX.
EF	OUT DX, EAX	ZO	Valid	Valid	Output doubleword in EAX to I/O port address in DX.

NOTES:

* See IA-32 Architecture Compatibility section below.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
I	imm8	NA	NA	NA
ZO	NA	NA	NA	NA

Description

Copies the value from the second operand (source operand) to the I/O port specified with the destination operand (first operand). The source operand can be register AL, AX, or EAX, depending on the size of the port being accessed (8, 16, or 32 bits, respectively); the destination operand can be a byte-immediate or the DX register. Using a byte immediate allows I/O port addresses 0 to 255 to be accessed; using the DX register as a source operand allows I/O ports from 0 to 65,535 to be accessed.

The size of the I/O port being accessed is determined by the opcode for an 8-bit I/O port or by the operand-size attribute of the instruction for a 16- or 32-bit I/O port.

At the machine code level, I/O instructions are shorter when accessing 8-bit I/O ports. Here, the upper eight bits of the port address will be 0.

This instruction is only useful for accessing I/O ports located in the processor's I/O address space. See Chapter 18, "Input/Output," in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*, for more information on accessing I/O ports in the I/O address space.

This instruction's operation is the same in non-64-bit modes and 64-bit mode.

IA-32 Architecture Compatibility

After executing an OUT instruction, the Pentium® processor ensures that the EWBE# pin has been sampled active before it begins to execute the next instruction. (Note that the instruction can be prefetched if EWBE# is not active, but it will not be executed until the EWBE# pin is sampled active.) Only the Pentium processor family has the EWBE# pin.

Operation

```
IF ((PE = 1) and ((CPL > IOPL) or (VM = 1)))
    THEN (* Protected mode with CPL > IOPL or virtual-8086 mode *)
        IF (Any I/O Permission Bit for I/O port being accessed = 1)
            THEN (* I/O operation is not allowed *)
                #GP(0);
            ELSE ( * I/O operation is allowed *)
                DEST ← SRC; (* Writes to selected I/O port *)
        FI;
    ELSE (Real Mode or Protected Mode with CPL ≤ IOPL *)
        DEST ← SRC; (* Writes to selected I/O port *)
FI;
```

Flags Affected

None

Protected Mode Exceptions

#GP(0)	If the CPL is greater than (has less privilege) the I/O privilege level (IOPL) and any of the corresponding I/O permission bits in TSS for the I/O port being accessed is 1.
#UD	If the LOCK prefix is used.

Real-Address Mode Exceptions

#UD	If the LOCK prefix is used.

Virtual-8086 Mode Exceptions

#GP(0)	If any of the I/O permission bits in the TSS for the I/O port being accessed is 1.
#PF(fault-code)	If a page fault occurs.
#UD	If the LOCK prefix is used.

Compatibility Mode Exceptions

Same as protected mode exceptions.

64-Bit Mode Exceptions

Same as protected mode exceptions.

OUTS/OUTSB/OUTSW/OUTSD—Output String to Port

Opcode*	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
6E	OUTS DX, *m8*	ZO	Valid	Valid	Output byte from memory location specified in DS:(E)SI or RSI to I/O port specified in DX**.
6F	OUTS DX, *m16*	ZO	Valid	Valid	Output word from memory location specified in DS:(E)SI or RSI to I/O port specified in DX**.
6F	OUTS DX, *m32*	ZO	Valid	Valid	Output doubleword from memory location specified in DS:(E)SI or RSI to I/O port specified in DX**.
6E	OUTSB	ZO	Valid	Valid	Output byte from memory location specified in DS:(E)SI or RSI to I/O port specified in DX**.
6F	OUTSW	ZO	Valid	Valid	Output word from memory location specified in DS:(E)SI or RSI to I/O port specified in DX**.
6F	OUTSD	ZO	Valid	Valid	Output doubleword from memory location specified in DS:(E)SI or RSI to I/O port specified in DX**.

NOTES:

* See IA-32 Architecture Compatibility section below.

** In 64-bit mode, only 64-bit (RSI) and 32-bit (ESI) address sizes are supported. In non-64-bit mode, only 32-bit (ESI) and 16-bit (SI) address sizes are supported.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
ZO	NA	NA	NA	NA

Description

Copies data from the source operand (second operand) to the I/O port specified with the destination operand (first operand). The source operand is a memory location, the address of which is read from either the DS:SI, DS:ESI or the RSI registers (depending on the address-size attribute of the instruction, 16, 32 or 64, respectively). (The DS segment may be overridden with a segment override prefix.) The destination operand is an I/O port address (from 0 to 65,535) that is read from the DX register. The size of the I/O port being accessed (that is, the size of the source and destination operands) is determined by the opcode for an 8-bit I/O port or by the operand-size attribute of the instruction for a 16- or 32-bit I/O port.

At the assembly-code level, two forms of this instruction are allowed: the "explicit-operands" form and the "no-operands" form. The explicit-operands form (specified with the OUTS mnemonic) allows the source and destination operands to be specified explicitly. Here, the source operand should be a symbol that indicates the size of the I/O port and the source address, and the destination operand must be DX. This explicit-operands form is provided to allow documentation; however, note that the documentation provided by this form can be misleading. That is, the source operand symbol must specify the correct **type** (size) of the operand (byte, word, or doubleword), but it does not have to specify the correct **location**. The location is always specified by the DS:(E)SI or RSI registers, which must be loaded correctly before the OUTS instruction is executed.

The no-operands form provides "short forms" of the byte, word, and doubleword versions of the OUTS instructions. Here also DS:(E)SI is assumed to be the source operand and DX is assumed to be the destination operand. The size of the I/O port is specified with the choice of mnemonic: OUTSB (byte), OUTSW (word), or OUTSD (doubleword).

After the byte, word, or doubleword is transferred from the memory location to the I/O port, the SI/ESI/RSI register is incremented or decremented automatically according to the setting of the DF flag in the EFLAGS register. (If the DF flag is 0, the (E)SI register is incremented; if the DF flag is 1, the SI/ESI/RSI register is decremented.) The SI/ESI/RSI register is incremented or decremented by 1 for byte operations, by 2 for word operations, and by 4 for doubleword operations.

The OUTS, OUTSB, OUTSW, and OUTSD instructions can be preceded by the REP prefix for block input of ECX bytes, words, or doublewords. See "REP/REPE/REPZ /REPNE/REPNZ—Repeat String Operation Prefix" in this chapter for a description of the REP prefix. This instruction is only useful for accessing I/O ports located in the processor's I/O address space. See Chapter 18, "Input/Output," in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*, for more information on accessing I/O ports in the I/O address space.

In 64-bit mode, the default operand size is 32 bits; operand size is not promoted by the use of REX.W. In 64-bit mode, the default address size is 64 bits, and 64-bit address is specified using RSI by default. 32-bit address using ESI is support using the prefix 67H, but 16-bit address is not supported in 64-bit mode.

IA-32 Architecture Compatibility

After executing an OUTS, OUTSB, OUTSW, or OUTSD instruction, the Pentium processor ensures that the EWBE# pin has been sampled active before it begins to execute the next instruction. (Note that the instruction can be prefetched if EWBE# is not active, but it will not be executed until the EWBE# pin is sampled active.) Only the Pentium processor family has the EWBE# pin.

For the Pentium 4, Intel® Xeon®, and P6 processor family, upon execution of an OUTS, OUTSB, OUTSW, or OUTSD instruction, the processor will not execute the next instruction until the data phase of the transaction is complete.

Operation

```
IF ((PE = 1) and ((CPL > IOPL) or (VM = 1)))
    THEN (* Protected mode with CPL > IOPL or virtual-8086 mode *)
        IF (Any I/O Permission Bit for I/O port being accessed = 1)
            THEN (* I/O operation is not allowed *)
                #GP(0);
            ELSE (* I/O operation is allowed *)
                DEST ← SRC; (* Writes to I/O port *)
        FI;
    ELSE (Real Mode or Protected Mode or 64-Bit Mode with CPL ≤ IOPL *)
        DEST ← SRC; (* Writes to I/O port *)
FI;

Byte transfer:
    IF 64-bit mode
        Then
            IF 64-Bit Address Size
                THEN
                    IF DF = 0
                        THEN RSI ← RSI RSI + 1;
                        ELSE RSI ← RSI or – 1;
                    FI;
                ELSE (* 32-Bit Address Size *)
                    IF DF = 0
                        THEN    ESI ← ESI + 1;
                        ELSE    ESI ← ESI – 1;
                    FI;
            FI;
        ELSE
            IF DF = 0
                THEN    (E)SI ← (E)SI + 1;
                ELSE (E)SI ← (E)SI – 1;
            FI;
    FI;
Word transfer:
    IF 64-bit mode
```

```
        Then
            IF 64-Bit Address Size
                THEN
                    IF DF = 0
                        THEN RSI ← RSI RSI + 2;
                        ELSE RSI ← RSI or – 2;
                    FI;
                ELSE (* 32-Bit Address Size *)
                    IF DF = 0
                        THEN    ESI ← ESI + 2;
                        ELSE    ESI ← ESI – 2;
                    FI;
            FI;
        ELSE
            IF DF = 0
                THEN    (E)SI ← (E)SI + 2;
                ELSE (E)SI ← (E)SI – 2;
            FI;
    FI;
Doubleword transfer:
    IF 64-bit mode
        Then
            IF 64-Bit Address Size
                THEN
                    IF DF = 0
                        THEN RSI ← RSI RSI + 4;
                        ELSE RSI ← RSI or – 4;
                    FI;
                ELSE (* 32-Bit Address Size *)
                    IF DF = 0
                        THEN    ESI ← ESI + 4;
                        ELSE    ESI ← ESI – 4;
                    FI;
            FI;
        ELSE
            IF DF = 0
                THEN    (E)SI ← (E)SI + 4;
                ELSE (E)SI ← (E)SI – 4;
            FI;
    FI;
```

Flags Affected

None

Protected Mode Exceptions

#GP(0)	If the CPL is greater than (has less privilege) the I/O privilege level (IOPL) and any of the corresponding I/O permission bits in TSS for the I/O port being accessed is 1.
	If a memory operand effective address is outside the limit of the CS, DS, ES, FS, or GS segment.
	If the segment register contains a NULL segment selector.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If the LOCK prefix is used.

Real-Address Mode Exceptions

#GP	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS	If a memory operand effective address is outside the SS segment limit.
#UD	If the LOCK prefix is used.

Virtual-8086 Mode Exceptions

#GP(0)	If any of the I/O permission bits in the TSS for the I/O port being accessed is 1.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.
#UD	If the LOCK prefix is used.

Compatibility Mode Exceptions

Same as for protected mode exceptions.

64-Bit Mode Exceptions

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the CPL is greater than (has less privilege) the I/O privilege level (IOPL) and any of the corresponding I/O permission bits in TSS for the I/O port being accessed is 1.
	If the memory address is in a non-canonical form.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If the LOCK prefix is used.

PABSB/PABSW/PABSD/PABSQ — Packed Absolute Value

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F 38 1C /r[1] PABSB *mm1, mm2/m64*	A	V/V	SSSE3	Compute the absolute value of bytes in *mm2/m64* and store UNSIGNED result in *mm1*.
66 0F 38 1C /r PABSB *xmm1, xmm2/m128*	A	V/V	SSSE3	Compute the absolute value of bytes in *xmm2/m128* and store UNSIGNED result in *xmm1*.
NP 0F 38 1D /r[1] PABSW *mm1, mm2/m64*	A	V/V	SSSE3	Compute the absolute value of 16-bit integers in *mm2/m64* and store UNSIGNED result in *mm1*.
66 0F 38 1D /r PABSW *xmm1, xmm2/m128*	A	V/V	SSSE3	Compute the absolute value of 16-bit integers in *xmm2/m128* and store UNSIGNED result in *xmm1*.
NP 0F 38 1E /r[1] PABSD *mm1, mm2/m64*	A	V/V	SSSE3	Compute the absolute value of 32-bit integers in *mm2/m64* and store UNSIGNED result in *mm1*.
66 0F 38 1E /r PABSD *xmm1, xmm2/m128*	A	V/V	SSSE3	Compute the absolute value of 32-bit integers in *xmm2/m128* and store UNSIGNED result in *xmm1*.
VEX.128.66.0F38.WIG 1C /r VPABSB *xmm1, xmm2/m128*	A	V/V	AVX	Compute the absolute value of bytes in *xmm2/m128* and store UNSIGNED result in *xmm1*.
VEX.128.66.0F38.WIG 1D /r VPABSW *xmm1, xmm2/m128*	A	V/V	AVX	Compute the absolute value of 16- bit integers in *xmm2/m128* and store UNSIGNED result in *xmm1*.
VEX.128.66.0F38.WIG 1E /r VPABSD *xmm1, xmm2/m128*	A	V/V	AVX	Compute the absolute value of 32- bit integers in *xmm2/m128* and store UNSIGNED result in *xmm1*.
VEX.256.66.0F38.WIG 1C /r VPABSB *ymm1, ymm2/m256*	A	V/V	AVX2	Compute the absolute value of bytes in *ymm2/m256* and store UNSIGNED result in *ymm1*.
VEX.256.66.0F38.WIG 1D /r VPABSW *ymm1, ymm2/m256*	A	V/V	AVX2	Compute the absolute value of 16-bit integers in *ymm2/m256* and store UNSIGNED result in *ymm1*.
VEX.256.66.0F38.WIG 1E /r VPABSD *ymm1, ymm2/m256*	A	V/V	AVX2	Compute the absolute value of 32-bit integers in *ymm2/m256* and store UNSIGNED result in *ymm1*.
EVEX.128.66.0F38.WIG 1C /r VPABSB xmm1 {k1}{z}, xmm2/m128	B	V/V	AVX512VL AVX512BW	Compute the absolute value of bytes in xmm2/m128 and store UNSIGNED result in xmm1 using writemask k1.
EVEX.256.66.0F38.WIG 1C /r VPABSB ymm1 {k1}{z}, ymm2/m256	B	V/V	AVX512VL AVX512BW	Compute the absolute value of bytes in ymm2/m256 and store UNSIGNED result in ymm1 using writemask k1.
EVEX.512.66.0F38.WIG 1C /r VPABSB zmm1 {k1}{z}, zmm2/m512	B	V/V	AVX512BW	Compute the absolute value of bytes in zmm2/m512 and store UNSIGNED result in zmm1 using writemask k1.
EVEX.128.66.0F38.WIG 1D /r VPABSW xmm1 {k1}{z}, xmm2/m128	B	V/V	AVX512VL AVX512BW	Compute the absolute value of 16-bit integers in xmm2/m128 and store UNSIGNED result in xmm1 using writemask k1.

EVEX.256.66.0F38.WIG 1D /r VPABSW ymm1 {k1}{z}, ymm2/m256	B	V/V	AVX512VL AVX512BW	Compute the absolute value of 16-bit integers in ymm2/m256 and store UNSIGNED result in ymm1 using writemask k1.
EVEX.512.66.0F38.WIG 1D /r VPABSW zmm1 {k1}{z}, zmm2/m512	B	V/V	AVX512BW	Compute the absolute value of 16-bit integers in zmm2/m512 and store UNSIGNED result in zmm1 using writemask k1.
EVEX.128.66.0F38.W0 1E /r VPABSD xmm1 {k1}{z}, xmm2/m128/m32bcst	C	V/V	AVX512VL AVX512F	Compute the absolute value of 32-bit integers in xmm2/m128/m32bcst and store UNSIGNED result in xmm1 using writemask k1.
EVEX.256.66.0F38.W0 1E /r VPABSD ymm1 {k1}{z}, ymm2/m256/m32bcst	C	V/V	AVX512VL AVX512F	Compute the absolute value of 32-bit integers in ymm2/m256/m32bcst and store UNSIGNED result in ymm1 using writemask k1.
VPABSD zmm1 {k1}{z}, zmm2/m512/m32bcst	C	V/V	AVX512F	Compute the absolute value of 32-bit integers in zmm2/m512/m32bcst and store UNSIGNED result in zmm1 using writemask k1.
EVEX.128.66.0F38.W1 1F /r VPABSQ xmm1 {k1}{z}, xmm2/m128/m64bcst	C	V/V	AVX512VL AVX512F	Compute the absolute value of 64-bit integers in xmm2/m128/m64bcst and store UNSIGNED result in xmm1 using writemask k1.
EVEX.256.66.0F38.W1 1F /r VPABSQ ymm1 {k1}{z}, ymm2/m256/m64bcst	C	V/V	AVX512VL AVX512F	Compute the absolute value of 64-bit integers in ymm2/m256/m64bcst and store UNSIGNED result in ymm1 using writemask k1.
EVEX.512.66.0F38.W1 1F /r VPABSQ zmm1 {k1}{z}, zmm2/m512/m64bcst	C	V/V	AVX512F	Compute the absolute value of 64-bit integers in zmm2/m512/m64bcst and store UNSIGNED result in zmm1 using writemask k1.

NOTES:

1. See note in Section 2.4, "AVX and SSE Instruction Exception Specification" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A* and Section 22.25.3, "Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	NA	ModRM:reg (w)	ModRM:r/m (r)	NA	NA
B	Full Vector Mem	ModRM:reg (w)	ModRM:r/m (r)	NA	NA
C	Full Vector	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

Description

PABSB/W/D computes the absolute value of each data element of the source operand (the second operand) and stores the UNSIGNED results in the destination operand (the first operand). PABSB operates on signed bytes, PABSW operates on signed 16-bit words, and PABSD operates on signed 32-bit integers.

EVEX encoded VPABSD/Q: The source operand is a ZMM/YMM/XMM register, a 512/256/128-bit memory location, or a 512/256/128-bit vector broadcasted from a 32/64-bit memory location. The destination operand is a ZMM/YMM/XMM register updated according to the writemask.

EVEX encoded VPABSB/W: The source operand is a ZMM/YMM/XMM register, or a 512/256/128-bit memory location. The destination operand is a ZMM/YMM/XMM register updated according to the writemask.

VEX.256 encoded versions: The source operand is a YMM register or a 256-bit memory location. The destination operand is a YMM register. The upper bits (MAXVL-1:256) of the corresponding register destination are zeroed.

VEX.128 encoded versions: The source operand is an XMM register or 128-bit memory location. The destination operand is an XMM register. The upper bits (MAXVL-1:128) of the corresponding register destination are zeroed.

128-bit Legacy SSE version: The source operand can be an XMM register or an 128-bit memory location. The destination is an XMM register. The upper bits (VL_MAX-1:128) of the corresponding register destination are unmodified.

VEX.vvvv and EVEX.vvvv are reserved and must be 1111b otherwise instructions will #UD.

Operation

PABSB with 128 bit operands:

```
Unsigned DEST[7:0] ←ABS(SRC[7: 0])
Repeat operation for 2nd through 15th bytes
Unsigned DEST[127:120] ←ABS(SRC[127:120])
```

VPABSB with 128 bit operands:

```
Unsigned DEST[7:0] ←ABS(SRC[7: 0])
Repeat operation for 2nd through 15th bytes
Unsigned DEST[127:120]←ABS(SRC[127:120])
```

VPABSB with 256 bit operands:

```
Unsigned DEST[7:0]←ABS(SRC[7: 0])
Repeat operation for 2nd through 31st bytes
Unsigned DEST[255:248]←ABS(SRC[255:248])
```

VPABSB (EVEX encoded versions)

```
(KL, VL) = (16, 128), (32, 256), (64, 512)

FOR j ← 0 TO KL-1
    i ← j * 8
    IF k1[j] OR *no writemask*
        THEN
            Unsigned DEST[i+7:i] ← ABS(SRC[i+7:i])
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+7:i] remains unchanged*
                ELSE *zeroing-masking*              ; zeroing-masking
                    DEST[i+7:i] ← 0
            FI
    FI;
ENDFOR;
DEST[MAXVL-1:VL] ← 0
```

PABSW with 128 bit operands:

```
Unsigned DEST[15:0]←ABS(SRC[15:0])
Repeat operation for 2nd through 7th 16-bit words
Unsigned DEST[127:112]←ABS(SRC[127:112])
```

VPABSW with 128 bit operands:

```
Unsigned DEST[15:0] ←ABS(SRC[15:0])
Repeat operation for 2nd through 7th 16-bit words
Unsigned DEST[127:112]←ABS(SRC[127:112])
```

VPABSW with 256 bit operands:

```
Unsigned DEST[15:0]←ABS(SRC[15:0])
Repeat operation for 2nd through 15th 16-bit words
Unsigned DEST[255:240] ←ABS(SRC[255:240])
```

VPABSW (EVEX encoded versions)

```
(KL, VL) = (8, 128), (16, 256), (32, 512)

FOR j ← 0 TO KL-1
    i ← j * 16
    IF k1[j] OR *no writemask*
        THEN
            Unsigned DEST[i+15:i] ← ABS(SRC[i+15:i])
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+15:i] remains unchanged*
                ELSE *zeroing-masking*              ; zeroing-masking
                    DEST[i+15:i] ← 0
            FI
    FI;
ENDFOR;
DEST[MAXVL-1:VL] ← 0
```

PABSD with 128 bit operands:

```
Unsigned DEST[31:0]←ABS(SRC[31:0])
Repeat operation for 2nd through 3rd 32-bit double words
Unsigned DEST[127:96]←ABS(SRC[127:96])
```

VPABSD with 128 bit operands:

```
Unsigned DEST[31:0]←ABS(SRC[31:0])
Repeat operation for 2nd through 3rd 32-bit double words
Unsigned DEST[127:96]←ABS(SRC[127:96])
```

VPABSD with 256 bit operands:

```
Unsigned DEST[31:0] ←ABS(SRC[31:0])
Repeat operation for 2nd through 7th 32-bit double words
Unsigned DEST[255:224] ←ABS(SRC[255:224])
```

VPABSD (EVEX encoded versions)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j ← 0 TO KL-1
    i ← j * 32
    IF k1[j] OR *no writemask*
        THEN
            IF (EVEX.b = 1) AND (SRC *is memory*)
                THEN
                    Unsigned DEST[i+31:i] ← ABS(SRC[31:0])
                ELSE
                    Unsigned DEST[i+31:i] ← ABS(SRC[i+31:i])
            FI;
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE *zeroing-masking*              ; zeroing-masking
                    DEST[i+31:i] ← 0
            FI
    FI;
ENDFOR;
DEST[MAXVL-1:VL] ← 0
```

VPABSQ (EVEX encoded versions)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j ← 0 TO KL-1
    i ← j * 64
    IF k1[j] OR *no writemask*
        THEN
            IF (EVEX.b = 1) AND (SRC *is memory*)
                THEN
                    Unsigned DEST[i+63:i] ← ABS(SRC[63:0])
                ELSE
                    Unsigned DEST[i+63:i] ← ABS(SRC[i+63:i])
            FI;
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE *zeroing-masking*                  ; zeroing-masking
                    DEST[i+63:i] ← 0
            FI
    FI;
ENDFOR;
DEST[MAXVL-1:VL] ← 0
```

Intel C/C++ Compiler Intrinsic Equivalents

VPABSB__m512i _mm512_abs_epi8 (__m512i a)
VPABSW__m512i _mm512_abs_epi16 (__m512i a)
VPABSB__m512i _mm512_mask_abs_epi8 (__m512i s, __mmask64 m, __m512i a)
VPABSW__m512i _mm512_mask_abs_epi16 (__m512i s, __mmask32 m, __m512i a)
VPABSB__m512i _mm512_maskz_abs_epi8 (__mmask64 m, __m512i a)
VPABSW__m512i _mm512_maskz_abs_epi16 (__mmask32 m, __m512i a)
VPABSB__m256i _mm256_mask_abs_epi8 (__m256i s, __mmask32 m, __m256i a)
VPABSW__m256i _mm256_mask_abs_epi16 (__m256i s, __mmask16 m, __m256i a)
VPABSB__m256i _mm256_maskz_abs_epi8 (__mmask32 m, __m256i a)
VPABSW__m256i _mm256_maskz_abs_epi16 (__mmask16 m, __m256i a)
VPABSB__m128i _mm_mask_abs_epi8 (__m128i s, __mmask16 m, __m128i a)
VPABSW__m128i _mm_mask_abs_epi16 (__m128i s, __mmask8 m, __m128i a)
VPABSB__m128i _mm_maskz_abs_epi8 (__mmask16 m, __m128i a)
VPABSW__m128i _mm_maskz_abs_epi16 (__mmask8 m, __m128i a)
VPABSD __m256i _mm256_mask_abs_epi32(__m256i s, __mmask8 k, __m256i a);
VPABSD __m256i _mm256_maskz_abs_epi32(__mmask8 k, __m256i a);
VPABSD __m128i _mm_mask_abs_epi32(__m128i s, __mmask8 k, __m128i a);
VPABSD __m128i _mm_maskz_abs_epi32(__mmask8 k, __m128i a);
VPABSD __m512i _mm512_abs_epi32(__m512i a);
VPABSD __m512i _mm512_mask_abs_epi32(__m512i s, __mmask16 k, __m512i a);
VPABSD __m512i _mm512_maskz_abs_epi32(__mmask16 k, __m512i a);
VPABSQ __m512i _mm512_abs_epi64(__m512i a);
VPABSQ __m512i _mm512_mask_abs_epi64(__m512i s, __mmask8 k, __m512i a);
VPABSQ __m512i _mm512_maskz_abs_epi64(__mmask8 k, __m512i a);
VPABSQ __m256i _mm256_mask_abs_epi64(__m256i s, __mmask8 k, __m256i a);
VPABSQ __m256i _mm256_maskz_abs_epi64(__mmask8 k, __m256i a);
VPABSQ __m128i _mm_mask_abs_epi64(__m128i s, __mmask8 k, __m128i a);
VPABSQ __m128i _mm_maskz_abs_epi64(__mmask8 k, __m128i a);
PABSB __m128i _mm_abs_epi8 (__m128i a)
VPABSB __m128i _mm_abs_epi8 (__m128i a)

VPABSB __m256i _mm256_abs_epi8 (__m256i a)
PABSW __m128i _mm_abs_epi16 (__m128i a)
VPABSW __m128i _mm_abs_epi16 (__m128i a)
VPABSW __m256i _mm256_abs_epi16 (__m256i a)
PABSD __m128i _mm_abs_epi32 (__m128i a)
VPABSD __m128i _mm_abs_epi32 (__m128i a)
VPABSD __m256i _mm256_abs_epi32 (__m256i a)

SIMD Floating-Point Exceptions

None

Other Exceptions

Non-EVEX-encoded instruction, see Exceptions Type 4.
EVEX-encoded VPABSD/Q, see Exceptions Type E4.
EVEX-encoded VPABSB/W, see Exceptions Type E4.nb.

PACKSSWB/PACKSSDW—Pack with Signed Saturation

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F 63 /r[1] PACKSSWB *mm1, mm2/m64*	A	V/V	MMX	Converts 4 packed signed word integers from *mm1* and from *mm2/m64* into 8 packed signed byte integers in *mm1* using signed saturation.
66 0F 63 /r PACKSSWB *xmm1, xmm2/m128*	A	V/V	SSE2	Converts 8 packed signed word integers from *xmm1* and from *xxm2/m128* into 16 packed signed byte integers in *xxm1* using signed saturation.
NP 0F 6B /r[1] PACKSSDW *mm1, mm2/m64*	A	V/V	MMX	Converts 2 packed signed doubleword integers from *mm1* and from *mm2/m64* into 4 packed signed word integers in *mm1* using signed saturation.
66 0F 6B /r PACKSSDW *xmm1, xmm2/m128*	A	V/V	SSE2	Converts 4 packed signed doubleword integers from *xmm1* and from *xxm2/m128* into 8 packed signed word integers in *xxm1* using signed saturation.
VEX.NDS.128.66.0F.WIG 63 /r VPACKSSWB *xmm1,xmm2, xmm3/m128*	B	V/V	AVX	Converts 8 packed signed word integers from *xmm2* and from *xmm3/m128* into 16 packed signed byte integers in *xmm1* using signed saturation.
VEX.NDS.128.66.0F.WIG 6B /r VPACKSSDW *xmm1,xmm2, xmm3/m128*	B	V/V	AVX	Converts 4 packed signed doubleword integers from *xmm2* and from *xmm3/m128* into 8 packed signed word integers in *xmm1* using signed saturation.
VEX.NDS.256.66.0F.WIG 63 /r VPACKSSWB *ymm1, ymm2, ymm3/m256*	B	V/V	AVX2	Converts 16 packed signed word integers from *ymm2* and from *ymm3/m256* into 32 packed signed byte integers in *ymm1* using signed saturation.
VEX.NDS.256.66.0F.WIG 6B /r VPACKSSDW *ymm1, ymm2, ymm3/m256*	B	V/V	AVX2	Converts 8 packed signed doubleword integers from *ymm2* and from *ymm3/m256* into 16 packed signed word integers in *ymm1*using signed saturation.
EVEX.NDS.128.66.0F.WIG 63 /r VPACKSSWB *xmm1 {k1}{z}, xmm2, xmm3/m128*	C	V/V	AVX512VL AVX512BW	Converts packed signed word integers from *xmm2* and from *xmm3/m128* into packed signed byte integers in *xmm1* using signed saturation under writemask k1.
EVEX.NDS.256.66.0F.WIG 63 /r VPACKSSWB *ymm1 {k1}{z}, ymm2, ymm3/m256*	C	V/V	AVX512VL AVX512BW	Converts packed signed word integers from *ymm2* and from *ymm3/m256* into packed signed byte integers in *ymm1* using signed saturation under writemask k1.
EVEX.NDS.512.66.0F.WIG 63 /r VPACKSSWB *zmm1 {k1}{z}, zmm2, zmm3/m512*	C	V/V	AVX512BW	Converts packed signed word integers from *zmm2* and from *zmm3/m512* into packed signed byte integers in *zmm1* using signed saturation under writemask k1.
EVEX.NDS.128.66.0F.W0 6B /r VPACKSSDW xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst	D	V/V	AVX512VL AVX512BW	Converts packed signed doubleword integers from *xmm2* and from xmm3/m128/m32bcst into packed signed word integers in xmm1 using signed saturation under writemask k1.

EVEX.NDS.256.66.0F.W0 6B /r VPACKSSDW ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst	D	V/V	AVX512VL AVX512BW	Converts packed signed doubleword integers from *ymm2* and from ymm3/m256/m32bcst into packed signed word integers in ymm1 using signed saturation under writemask k1.
EVEX.NDS.512.66.0F.W0 6B /r VPACKSSDW zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst	D	V/V	AVX512BW	Converts packed signed doubleword integers from zmm2 and from zmm3/m512/m32bcst into packed signed word integers in zmm1 using signed saturation under writemask k1.

NOTES:

1. See note in Section 2.4, "AVX and SSE Instruction Exception Specification" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A* and Section 22.25.3, "Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	NA	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
B	NA	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA
C	Full Vector Mem	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA
D	Full Vector	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA

Description

Converts packed signed word integers into packed signed byte integers (PACKSSWB) or converts packed signed doubleword integers into packed signed word integers (PACKSSDW), using saturation to handle overflow conditions. See Figure 4-6 for an example of the packing operation.

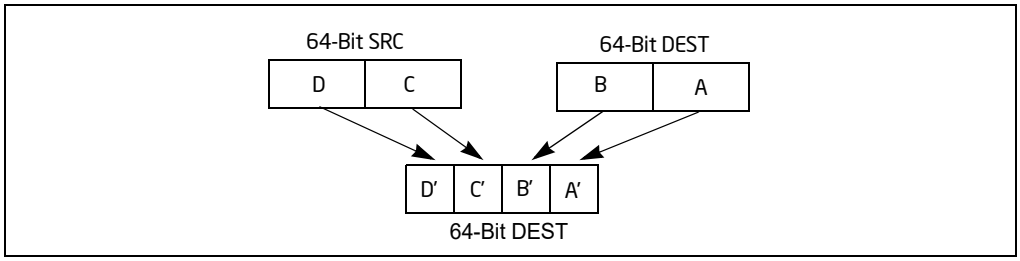


Figure 4-6. Operation of the PACKSSDW Instruction Using 64-bit Operands

PACKSSWB converts packed signed word integers in the first and second source operands into packed signed byte integers using signed saturation to handle overflow conditions beyond the range of signed byte integers. If the signed doubleword value is beyond the range of an unsigned word (i.e. greater than 7FH or less than 80H), the saturated signed byte integer value of 7FH or 80H, respectively, is stored in the destination. PACKSSDW converts packed signed doubleword integers in the first and second source operands into packed signed word integers using signed saturation to handle overflow conditions beyond 7FFFH and 8000H.

EVEX encoded PACKSSWB: The first source operand is a ZMM/YMM/XMM register. The second source operand is a ZMM/YMM/XMM register or a 512/256/128-bit memory location. The destination operand is a ZMM/YMM/XMM register, updated conditional under the writemask k1.

EVEX encoded PACKSSDW: The first source operand is a ZMM/YMM/XMM register. The second source operand is a ZMM/YMM/XMM register, a 512/256/128-bit memory location, or a 512/256/128-bit vector broadcasted from a 32-bit memory location. The destination operand is a ZMM/YMM/XMM register, updated conditional under the writemask k1.

VEX.256 encoded version: The first source operand is a YMM register. The second source operand is a YMM register or a 256-bit memory location. The destination operand is a YMM register. The upper bits (MAXVL-1:256) of the corresponding ZMM register destination are zeroed.

VEX.128 encoded version: The first source operand is an XMM register. The second source operand is an XMM register or 128-bit memory location. The destination operand is an XMM register. The upper bits (MAXVL-1:128) of the corresponding ZMM register destination are zeroed.

128-bit Legacy SSE version: The first source operand is an XMM register. The second operand can be an XMM register or an 128-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (MAXVL-1:128) of the corresponding ZMM destination register destination are unmodified.

Operation

PACKSSWB instruction (128-bit Legacy SSE version)

```
DEST[7:0] ← SaturateSignedWordToSignedByte (DEST[15:0]);
DEST[15:8] ← SaturateSignedWordToSignedByte (DEST[31:16]);
DEST[23:16] ← SaturateSignedWordToSignedByte (DEST[47:32]);
DEST[31:24] ← SaturateSignedWordToSignedByte (DEST[63:48]);
DEST[39:32] ← SaturateSignedWordToSignedByte (DEST[79:64]);
DEST[47:40] ← SaturateSignedWordToSignedByte (DEST[95:80]);
DEST[55:48] ← SaturateSignedWordToSignedByte (DEST[111:96]);
DEST[63:56] ← SaturateSignedWordToSignedByte (DEST[127:112]);
DEST[71:64] ← SaturateSignedWordToSignedByte (SRC[15:0]);
DEST[79:72] ← SaturateSignedWordToSignedByte (SRC[31:16]);
DEST[87:80] ← SaturateSignedWordToSignedByte (SRC[47:32]);
DEST[95:88] ← SaturateSignedWordToSignedByte (SRC[63:48]);
DEST[103:96] ← SaturateSignedWordToSignedByte (SRC[79:64]);
DEST[111:104] ← SaturateSignedWordToSignedByte (SRC[95:80]);
DEST[119:112] ← SaturateSignedWordToSignedByte (SRC[111:96]);
DEST[127:120] ← SaturateSignedWordToSignedByte (SRC[127:112]);
DEST[MAXVL-1:128] (Unmodified)
```

PACKSSDW instruction (128-bit Legacy SSE version)

```
DEST[15:0] ← SaturateSignedDwordToSignedWord (DEST[31:0]);
DEST[31:16] ← SaturateSignedDwordToSignedWord (DEST[63:32]);
DEST[47:32] ← SaturateSignedDwordToSignedWord (DEST[95:64]);
DEST[63:48] ← SaturateSignedDwordToSignedWord (DEST[127:96]);
DEST[79:64] ← SaturateSignedDwordToSignedWord (SRC[31:0]);
DEST[95:80] ← SaturateSignedDwordToSignedWord (SRC[63:32]);
DEST[111:96] ← SaturateSignedDwordToSignedWord (SRC[95:64]);
DEST[127:112] ← SaturateSignedDwordToSignedWord (SRC[127:96]);
DEST[MAXVL-1:128] (Unmodified)
```

VPACKSSWB instruction (VEX.128 encoded version)

```
DEST[7:0] ← SaturateSignedWordToSignedByte (SRC1[15:0]);
DEST[15:8] ← SaturateSignedWordToSignedByte (SRC1[31:16]);
DEST[23:16] ← SaturateSignedWordToSignedByte (SRC1[47:32]);
DEST[31:24] ← SaturateSignedWordToSignedByte (SRC1[63:48]);
DEST[39:32] ← SaturateSignedWordToSignedByte (SRC1[79:64]);
DEST[47:40] ← SaturateSignedWordToSignedByte (SRC1[95:80]);
DEST[55:48] ← SaturateSignedWordToSignedByte (SRC1[111:96]);
DEST[63:56] ← SaturateSignedWordToSignedByte (SRC1[127:112]);
DEST[71:64] ← SaturateSignedWordToSignedByte (SRC2[15:0]);
DEST[79:72] ← SaturateSignedWordToSignedByte (SRC2[31:16]);
DEST[87:80] ← SaturateSignedWordToSignedByte (SRC2[47:32]);
DEST[95:88] ← SaturateSignedWordToSignedByte (SRC2[63:48]);
DEST[103:96] ← SaturateSignedWordToSignedByte (SRC2[79:64]);
DEST[111:104] ← SaturateSignedWordToSignedByte (SRC2[95:80]);
DEST[119:112] ← SaturateSignedWordToSignedByte (SRC2[111:96]);
DEST[127:120] ← SaturateSignedWordToSignedByte (SRC2[127:112]);
DEST[MAXVL-1:128] ← 0;
```

VPACKSSDW instruction (VEX.128 encoded version)

```
DEST[15:0] ← SaturateSignedDwordToSignedWord (SRC1[31:0]);
DEST[31:16] ← SaturateSignedDwordToSignedWord (SRC1[63:32]);
DEST[47:32] ← SaturateSignedDwordToSignedWord (SRC1[95:64]);
DEST[63:48] ← SaturateSignedDwordToSignedWord (SRC1[127:96]);
DEST[79:64] ← SaturateSignedDwordToSignedWord (SRC2[31:0]);
DEST[95:80] ← SaturateSignedDwordToSignedWord (SRC2[63:32]);
DEST[111:96] ← SaturateSignedDwordToSignedWord (SRC2[95:64]);
DEST[127:112] ← SaturateSignedDwordToSignedWord (SRC2[127:96]);
DEST[MAXVL-1:128] ← 0;
```

VPACKSSWB instruction (VEX.256 encoded version)

```
DEST[7:0] ← SaturateSignedWordToSignedByte (SRC1[15:0]);
DEST[15:8] ← SaturateSignedWordToSignedByte (SRC1[31:16]);
DEST[23:16] ← SaturateSignedWordToSignedByte (SRC1[47:32]);
DEST[31:24] ← SaturateSignedWordToSignedByte (SRC1[63:48]);
DEST[39:32] ← SaturateSignedWordToSignedByte (SRC1[79:64]);
DEST[47:40] ← SaturateSignedWordToSignedByte (SRC1[95:80]);
DEST[55:48] ← SaturateSignedWordToSignedByte (SRC1[111:96]);
DEST[63:56] ← SaturateSignedWordToSignedByte (SRC1[127:112]);
DEST[71:64] ← SaturateSignedWordToSignedByte (SRC2[15:0]);
DEST[79:72] ← SaturateSignedWordToSignedByte (SRC2[31:16]);
DEST[87:80] ← SaturateSignedWordToSignedByte (SRC2[47:32]);
DEST[95:88] ← SaturateSignedWordToSignedByte (SRC2[63:48]);
DEST[103:96] ← SaturateSignedWordToSignedByte (SRC2[79:64]);
DEST[111:104] ← SaturateSignedWordToSignedByte (SRC2[95:80]);
DEST[119:112] ← SaturateSignedWordToSignedByte (SRC2[111:96]);
DEST[127:120] ← SaturateSignedWordToSignedByte (SRC2[127:112]);
DEST[135:128] ← SaturateSignedWordToSignedByte (SRC1[143:128]);
DEST[143:136] ← SaturateSignedWordToSignedByte (SRC1[159:144]);
DEST[151:144] ← SaturateSignedWordToSignedByte (SRC1[175:160]);
DEST[159:152] ← SaturateSignedWordToSignedByte (SRC1[191:176]);
DEST[167:160] ← SaturateSignedWordToSignedByte (SRC1[207:192]);
DEST[175:168] ← SaturateSignedWordToSignedByte (SRC1[223:208]);
DEST[183:176] ← SaturateSignedWordToSignedByte (SRC1[239:224]);
```

```
    DEST[191:184] ← SaturateSignedWordToSignedByte (SRC1[255:240]);
    DEST[199:192] ← SaturateSignedWordToSignedByte (SRC2[143:128]);
    DEST[207:200] ← SaturateSignedWordToSignedByte (SRC2[159:144]);
    DEST[215:208] ← SaturateSignedWordToSignedByte (SRC2[175:160]);
    DEST[223:216] ← SaturateSignedWordToSignedByte (SRC2[191:176]);
    DEST[231:224] ← SaturateSignedWordToSignedByte (SRC2[207:192]);
    DEST[239:232] ← SaturateSignedWordToSignedByte (SRC2[223:208]);
    DEST[247:240] ← SaturateSignedWordToSignedByte (SRC2[239:224]);
    DEST[255:248] ← SaturateSignedWordToSignedByte (SRC2[255:240]);
    DEST[MAXVL-1:256] ← 0;
```

VPACKSSDW instruction (VEX.256 encoded version)

```
    DEST[15:0] ← SaturateSignedDwordToSignedWord (SRC1[31:0]);
    DEST[31:16] ← SaturateSignedDwordToSignedWord (SRC1[63:32]);
    DEST[47:32] ← SaturateSignedDwordToSignedWord (SRC1[95:64]);
    DEST[63:48] ← SaturateSignedDwordToSignedWord (SRC1[127:96]);
    DEST[79:64] ← SaturateSignedDwordToSignedWord (SRC2[31:0]);
    DEST[95:80] ← SaturateSignedDwordToSignedWord (SRC2[63:32]);
    DEST[111:96] ← SaturateSignedDwordToSignedWord (SRC2[95:64]);
    DEST[127:112] ← SaturateSignedDwordToSignedWord (SRC2[127:96]);
    DEST[143:128] ← SaturateSignedDwordToSignedWord (SRC1[159:128]);
    DEST[159:144] ← SaturateSignedDwordToSignedWord (SRC1[191:160]);
    DEST[175:160] ← SaturateSignedDwordToSignedWord (SRC1[223:192]);
    DEST[191:176] ← SaturateSignedDwordToSignedWord (SRC1[255:224]);
    DEST[207:192] ← SaturateSignedDwordToSignedWord (SRC2[159:128]);
    DEST[223:208] ← SaturateSignedDwordToSignedWord (SRC2[191:160]);
    DEST[239:224] ← SaturateSignedDwordToSignedWord (SRC2[223:192]);
    DEST[255:240] ← SaturateSignedDwordToSignedWord (SRC2[255:224]);
    DEST[MAXVL-1:256] ← 0;
```

VPACKSSWB (EVEX encoded versions)

```
(KL, VL) = (16, 128), (32, 256), (64, 512)
TMP_DEST[7:0] ← SaturateSignedWordToSignedByte (SRC1[15:0]);
TMP_DEST[15:8] ← SaturateSignedWordToSignedByte (SRC1[31:16]);
TMP_DEST[23:16] ← SaturateSignedWordToSignedByte (SRC1[47:32]);
TMP_DEST[31:24] ← SaturateSignedWordToSignedByte (SRC1[63:48]);
TMP_DEST[39:32] ← SaturateSignedWordToSignedByte (SRC1[79:64]);
TMP_DEST[47:40] ← SaturateSignedWordToSignedByte (SRC1[95:80]);
TMP_DEST[55:48] ← SaturateSignedWordToSignedByte (SRC1[111:96]);
TMP_DEST[63:56] ← SaturateSignedWordToSignedByte (SRC1[127:112]);
TMP_DEST[71:64] ← SaturateSignedWordToSignedByte (SRC2[15:0]);
TMP_DEST[79:72] ← SaturateSignedWordToSignedByte (SRC2[31:16]);
TMP_DEST[87:80] ← SaturateSignedWordToSignedByte (SRC2[47:32]);
TMP_DEST[95:88] ← SaturateSignedWordToSignedByte (SRC2[63:48]);
TMP_DEST[103:96] ← SaturateSignedWordToSignedByte (SRC2[79:64]);
TMP_DEST[111:104] ← SaturateSignedWordToSignedByte (SRC2[95:80]);
TMP_DEST[119:112] ← SaturateSignedWordToSignedByte (SRC2[111:96]);
TMP_DEST[127:120] ← SaturateSignedWordToSignedByte (SRC2[127:112]);
IF VL >= 256
    TMP_DEST[135:128]← SaturateSignedWordToSignedByte (SRC1[143:128]);
    TMP_DEST[143:136] ← SaturateSignedWordToSignedByte (SRC1[159:144]);
    TMP_DEST[151:144] ← SaturateSignedWordToSignedByte (SRC1[175:160]);
    TMP_DEST[159:152] ← SaturateSignedWordToSignedByte (SRC1[191:176]);
    TMP_DEST[167:160] ← SaturateSignedWordToSignedByte (SRC1[207:192]);
```

```
    TMP_DEST[175:168] ← SaturateSignedWordToSignedByte (SRC1[223:208]);
    TMP_DEST[183:176] ← SaturateSignedWordToSignedByte (SRC1[239:224]);
    TMP_DEST[191:184] ← SaturateSignedWordToSignedByte (SRC1[255:240]);
    TMP_DEST[199:192] ← SaturateSignedWordToSignedByte (SRC2[143:128]);
    TMP_DEST[207:200] ← SaturateSignedWordToSignedByte (SRC2[159:144]);
    TMP_DEST[215:208] ← SaturateSignedWordToSignedByte (SRC2[175:160]);
    TMP_DEST[223:216] ← SaturateSignedWordToSignedByte (SRC2[191:176]);
    TMP_DEST[231:224] ← SaturateSignedWordToSignedByte (SRC2[207:192]);
    TMP_DEST[239:232] ← SaturateSignedWordToSignedByte (SRC2[223:208]);
    TMP_DEST[247:240] ← SaturateSignedWordToSignedByte (SRC2[239:224]);
    TMP_DEST[255:248] ← SaturateSignedWordToSignedByte (SRC2[255:240]);
FI;
IF VL >= 512
    TMP_DEST[263:256] ← SaturateSignedWordToSignedByte (SRC1[271:256]);
    TMP_DEST[271:264] ← SaturateSignedWordToSignedByte (SRC1[287:272]);
    TMP_DEST[279:272] ← SaturateSignedWordToSignedByte (SRC1[303:288]);
    TMP_DEST[287:280] ← SaturateSignedWordToSignedByte (SRC1[319:304]);
    TMP_DEST[295:288] ← SaturateSignedWordToSignedByte (SRC1[335:320]);
    TMP_DEST[303:296] ← SaturateSignedWordToSignedByte (SRC1[351:336]);
    TMP_DEST[311:304] ← SaturateSignedWordToSignedByte (SRC1[367:352]);
    TMP_DEST[319:312] ← SaturateSignedWordToSignedByte (SRC1[383:368]);

    TMP_DEST[327:320] ← SaturateSignedWordToSignedByte (SRC2[271:256]);
    TMP_DEST[335:328] ← SaturateSignedWordToSignedByte (SRC2[287:272]);
    TMP_DEST[343:336] ← SaturateSignedWordToSignedByte (SRC2[303:288]);
    TMP_DEST[351:344] ← SaturateSignedWordToSignedByte (SRC2[319:304]);
    TMP_DEST[359:352] ← SaturateSignedWordToSignedByte (SRC2[335:320]);
    TMP_DEST[367:360] ← SaturateSignedWordToSignedByte (SRC2[351:336]);
    TMP_DEST[375:368] ← SaturateSignedWordToSignedByte (SRC2[367:352]);
    TMP_DEST[383:376] ← SaturateSignedWordToSignedByte (SRC2[383:368]);

    TMP_DEST[391:384] ← SaturateSignedWordToSignedByte (SRC1[399:384]);
    TMP_DEST[399:392] ← SaturateSignedWordToSignedByte (SRC1[415:400]);
    TMP_DEST[407:400] ← SaturateSignedWordToSignedByte (SRC1[431:416]);
    TMP_DEST[415:408] ← SaturateSignedWordToSignedByte (SRC1[447:432]);
    TMP_DEST[423:416] ← SaturateSignedWordToSignedByte (SRC1[463:448]);
    TMP_DEST[431:424] ← SaturateSignedWordToSignedByte (SRC1[479:464]);
    TMP_DEST[439:432] ← SaturateSignedWordToSignedByte (SRC1[495:480]);
    TMP_DEST[447:440] ← SaturateSignedWordToSignedByte (SRC1[511:496]);

    TMP_DEST[455:448] ← SaturateSignedWordToSignedByte (SRC2[399:384]);
    TMP_DEST[463:456] ← SaturateSignedWordToSignedByte (SRC2[415:400]);
    TMP_DEST[471:464] ← SaturateSignedWordToSignedByte (SRC2[431:416]);
    TMP_DEST[479:472] ← SaturateSignedWordToSignedByte (SRC2[447:432]);
    TMP_DEST[487:480] ← SaturateSignedWordToSignedByte (SRC2[463:448]);
    TMP_DEST[495:488] ← SaturateSignedWordToSignedByte (SRC2[479:464]);
    TMP_DEST[503:496] ← SaturateSignedWordToSignedByte (SRC2[495:480]);
    TMP_DEST[511:504] ← SaturateSignedWordToSignedByte (SRC2[511:496]);
FI;
FOR j ← 0 TO KL-1
    i ← j * 8
    IF k1[j] OR *no writemask*
        THEN
            DEST[i+7:i] ← TMP_DEST[i+7:i]
```

```
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+7:i] remains unchanged*
                ELSE *zeroing-masking*              ; zeroing-masking
                    DEST[i+7:i] ← 0
            FI
    FI;
ENDFOR;
DEST[MAXVL-1:VL] ← 0
```

VPACKSSDW (EVEX encoded versions)

```
(KL, VL) = (8, 128), (16, 256), (32, 512)
FOR j ← 0 TO ((KL/2) - 1)
    i ← j * 32

    IF (EVEX.b == 1) AND (SRC2 *is memory*)
        THEN
            TMP_SRC2[i+31:i] ← SRC2[31:0]
        ELSE
            TMP_SRC2[i+31:i] ← SRC2[i+31:i]
    FI;
ENDFOR;

TMP_DEST[15:0] ← SaturateSignedDwordToSignedWord (SRC1[31:0]);
TMP_DEST[31:16] ← SaturateSignedDwordToSignedWord (SRC1[63:32]);
TMP_DEST[47:32] ← SaturateSignedDwordToSignedWord (SRC1[95:64]);
TMP_DEST[63:48] ← SaturateSignedDwordToSignedWord (SRC1[127:96]);
TMP_DEST[79:64] ← SaturateSignedDwordToSignedWord (TMP_SRC2[31:0]);
TMP_DEST[95:80] ← SaturateSignedDwordToSignedWord (TMP_SRC2[63:32]);
TMP_DEST[111:96] ← SaturateSignedDwordToSignedWord (TMP_SRC2[95:64]);
TMP_DEST[127:112] ← SaturateSignedDwordToSignedWord (TMP_SRC2[127:96]);
IF VL >= 256
    TMP_DEST[143:128] ← SaturateSignedDwordToSignedWord (SRC1[159:128]);
    TMP_DEST[159:144] ← SaturateSignedDwordToSignedWord (SRC1[191:160]);
    TMP_DEST[175:160] ← SaturateSignedDwordToSignedWord (SRC1[223:192]);
    TMP_DEST[191:176] ← SaturateSignedDwordToSignedWord (SRC1[255:224]);
    TMP_DEST[207:192] ← SaturateSignedDwordToSignedWord (TMP_SRC2[159:128]);
    TMP_DEST[223:208] ← SaturateSignedDwordToSignedWord (TMP_SRC2[191:160]);
    TMP_DEST[239:224] ← SaturateSignedDwordToSignedWord (TMP_SRC2[223:192]);
    TMP_DEST[255:240] ← SaturateSignedDwordToSignedWord (TMP_SRC2[255:224]);
FI;
IF VL >= 512
    TMP_DEST[271:256] ← SaturateSignedDwordToSignedWord (SRC1[287:256]);
    TMP_DEST[287:272] ← SaturateSignedDwordToSignedWord (SRC1[319:288]);
    TMP_DEST[303:288] ← SaturateSignedDwordToSignedWord (SRC1[351:320]);
    TMP_DEST[319:304] ← SaturateSignedDwordToSignedWord (SRC1[383:352]);
    TMP_DEST[335:320] ← SaturateSignedDwordToSignedWord (TMP_SRC2[287:256]);
    TMP_DEST[351:336] ← SaturateSignedDwordToSignedWord (TMP_SRC2[319:288]);
    TMP_DEST[367:352] ← SaturateSignedDwordToSignedWord (TMP_SRC2[351:320]);
    TMP_DEST[383:368] ← SaturateSignedDwordToSignedWord (TMP_SRC2[383:352]);

    TMP_DEST[399:384] ← SaturateSignedDwordToSignedWord (SRC1[415:384]);
    TMP_DEST[415:400] ← SaturateSignedDwordToSignedWord (SRC1[447:416]);
    TMP_DEST[431:416] ← SaturateSignedDwordToSignedWord (SRC1[479:448]);
```

```
    TMP_DEST[447:432] ← SaturateSignedDwordToSignedWord (SRC1[511:480]);
    TMP_DEST[463:448] ← SaturateSignedDwordToSignedWord (TMP_SRC2[415:384]);
    TMP_DEST[479:464] ← SaturateSignedDwordToSignedWord (TMP_SRC2[447:416]);
    TMP_DEST[495:480] ← SaturateSignedDwordToSignedWord (TMP_SRC2[479:448]);
    TMP_DEST[511:496] ← SaturateSignedDwordToSignedWord (TMP_SRC2[511:480]);
FI;
FOR j ← 0 TO KL-1
    i ← j * 16
    IF k1[j] OR *no writemask*
        THEN DEST[i+15:i] ← TMP_DEST[i+15:i]
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+15:i] remains unchanged*
                ELSE *zeroing-masking*                  ; zeroing-masking
                    DEST[i+15:i] ← 0
            FI
    FI;
ENDFOR;
DEST[MAXVL-1:VL] ← 0
```

Intel C/C++ Compiler Intrinsic Equivalents

VPACKSSDW__m512i _mm512_packs_epi32(__m512i m1, __m512i m2);
VPACKSSDW__m512i _mm512_mask_packs_epi32(__m512i s, __mmask32 k, __m512i m1, __m512i m2);
VPACKSSDW__m512i _mm512_maskz_packs_epi32(__mmask32 k, __m512i m1, __m512i m2);
VPACKSSDW__m256i _mm256_mask_packs_epi32(__m256i s, __mmask16 k, __m256i m1, __m256i m2);
VPACKSSDW__m256i _mm256_maskz_packs_epi32(__mmask16 k, __m256i m1, __m256i m2);
VPACKSSDW__m128i _mm_mask_packs_epi32(__m128i s, __mmask8 k, __m128i m1, __m128i m2);
VPACKSSDW__m128i _mm_maskz_packs_epi32(__mmask8 k, __m128i m1, __m128i m2);
VPACKSSWB__m512i _mm512_packs_epi16(__m512i m1, __m512i m2);
VPACKSSWB__m512i _mm512_mask_packs_epi16(__m512i s, __mmask32 k, __m512i m1, __m512i m2);
VPACKSSWB__m512i _mm512_maskz_packs_epi16(__mmask32 k, __m512i m1, __m512i m2);
VPACKSSWB__m256i _mm256_mask_packs_epi16(__m256i s, __mmask16 k, __m256i m1, __m256i m2);
VPACKSSWB__m256i _mm256_maskz_packs_epi16(__mmask16 k, __m256i m1, __m256i m2);
VPACKSSWB__m128i _mm_mask_packs_epi16(__m128i s, __mmask8 k, __m128i m1, __m128i m2);
VPACKSSWB__m128i _mm_maskz_packs_epi16(__mmask8 k, __m128i m1, __m128i m2);
PACKSSWB __m128i _mm_packs_epi16(__m128i m1, __m128i m2)
PACKSSDW __m128i _mm_packs_epi32(__m128i m1, __m128i m2)
VPACKSSWB __m256i _mm256_packs_epi16(__m256i m1, __m256i m2)
VPACKSSDW __m256i _mm256_packs_epi32(__m256i m1, __m256i m2)

SIMD Floating-Point Exceptions

None

Other Exceptions

Non-EVEX-encoded instruction, see Exceptions Type 4.

EVEX-encoded VPACKSSDW, see Exceptions Type E4NF.

EVEX-encoded VPACKSSWB, see Exceptions Type E4NF.nb.

PACKUSDW—Pack with Unsigned Saturation

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 38 2B /r PACKUSDW *xmm1, xmm2/m128*	A	V/V	SSE4_1	Convert 4 packed signed doubleword integers from *xmm1* and 4 packed signed doubleword integers from *xmm2/m128* into 8 packed unsigned word integers in *xmm1* using unsigned saturation.
VEX.NDS.128.66.0F38 2B /r VPACKUSDW *xmm1,xmm2, xmm3/m128*	B	V/V	AVX	Convert 4 packed signed doubleword integers from *xmm2* and 4 packed signed doubleword integers from *xmm3/m128* into 8 packed unsigned word integers in *xmm1* using unsigned saturation.
VEX.NDS.256.66.0F38 2B /r VPACKUSDW *ymm1, ymm2, ymm3/m256*	B	V/V	AVX2	Convert 8 packed signed doubleword integers from *ymm2* and 8 packed signed doubleword integers from *ymm3/m256* into 16 packed unsigned word integers in *ymm1* using unsigned saturation.
EVEX.NDS.128.66.0F38.W0 2B /r VPACKUSDW *xmm1{k1}{z}, xmm2, xmm3/m128/m32bcst*	C	V/V	AVX512VL AVX512BW	Convert packed signed doubleword integers from xmm2 and packed signed doubleword integers from xmm3/m128/m32bcst into packed unsigned word integers in xmm1 using unsigned saturation under writemask k1.
EVEX.NDS.256.66.0F38.W0 2B /r	C	V/V	AVX512VL AVX512BW	Convert packed signed doubleword integers from ymm2 and packed signed doubleword integers from ymm3/m256/m32bcst into packed unsigned word integers in ymm1 using unsigned saturation under writemask k1.
EVEX.NDS.512.66.0F38.W0 2B /r VPACKUSDW *zmm1{k1}{z}, zmm2, zmm3/m512/m32bcst*	C	V/V	AVX512BW	Convert packed signed doubleword integers from *zmm2* and packed signed doubleword integers from *zmm3/m512/m32bcst* into packed unsigned word integers in *zmm1* using unsigned saturation under writemask k1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	NA	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
B	NA	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA
C	Full Vector	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA

Description

Converts packed signed doubleword integers in the first and second source operands into packed unsigned word integers using unsigned saturation to handle overflow conditions. If the signed doubleword value is beyond the range of an unsigned word (that is, greater than FFFFH or less than 0000H), the saturated unsigned word integer value of FFFFH or 0000H, respectively, is stored in the destination.

EVEX encoded versions: The first source operand is a ZMM/YMM/XMM register. The second source operand is a ZMM/YMM/XMM register, a 512/256/128-bit memory location, or a 512/256/128-bit vector broadcasted from a 32-bit memory location. The destination operand is a ZMM register, updated conditionally under the writemask k1.

VEX.256 encoded version: The first source operand is a YMM register. The second source operand is a YMM register or a 256-bit memory location. The destination operand is a YMM register. The upper bits (MAXVL-1:256) of the corresponding ZMM register destination are zeroed.

VEX.128 encoded version: The first source operand is an XMM register. The second source operand is an XMM register or 128-bit memory location. The destination operand is an XMM register. The upper bits (MAXVL-1:128) of the corresponding ZMM register destination are zeroed.

128-bit Legacy SSE version: The first source operand is an XMM register. The second operand can be an XMM register or an 128-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (MAXVL-1:128) of the corresponding destination register destination are unmodified.

Operation

PACKUSDW (Legacy SSE instruction)

```
TMP[15:0] ← (DEST[31:0] < 0) ? 0 : DEST[15:0];
DEST[15:0] ← (DEST[31:0] > FFFFH) ? FFFFH : TMP[15:0] ;
TMP[31:16] ← (DEST[63:32] < 0) ? 0 : DEST[47:32];
DEST[31:16] ← (DEST[63:32] > FFFFH) ? FFFFH : TMP[31:16] ;
TMP[47:32] ← (DEST[95:64] < 0) ? 0 : DEST[79:64];
DEST[47:32] ← (DEST[95:64] > FFFFH) ? FFFFH : TMP[47:32] ;
TMP[63:48] ← (DEST[127:96] < 0) ? 0 : DEST[111:96];
DEST[63:48] ← (DEST[127:96] > FFFFH) ? FFFFH : TMP[63:48] ;
TMP[79:64] ← (SRC[31:0] < 0) ? 0 : SRC[15:0];
DEST[79:64] ← (SRC[31:0] > FFFFH) ? FFFFH : TMP[79:64] ;
TMP[95:80] ← (SRC[63:32] < 0) ? 0 : SRC[47:32];
DEST[95:80] ← (SRC[63:32] > FFFFH) ? FFFFH : TMP[95:80] ;
TMP[111:96] ← (SRC[95:64] < 0) ? 0 : SRC[79:64];
DEST[111:96] ← (SRC[95:64] > FFFFH) ? FFFFH : TMP[111:96] ;
TMP[127:112] ← (SRC[127:96] < 0) ? 0 : SRC[111:96];
DEST[127:112] ← (SRC[127:96] > FFFFH) ? FFFFH : TMP[127:112] ;
DEST[MAXVL-1:128] (Unmodified)
```

PACKUSDW (VEX.128 encoded version)

```
TMP[15:0] ← (SRC1[31:0] < 0) ? 0 : SRC1[15:0];
DEST[15:0] ← (SRC1[31:0] > FFFFH) ? FFFFH : TMP[15:0] ;
TMP[31:16] ← (SRC1[63:32] < 0) ? 0 : SRC1[47:32];
DEST[31:16] ← (SRC1[63:32] > FFFFH) ? FFFFH : TMP[31:16] ;
TMP[47:32] ← (SRC1[95:64] < 0) ? 0 : SRC1[79:64];
DEST[47:32] ← (SRC1[95:64] > FFFFH) ? FFFFH : TMP[47:32] ;
TMP[63:48] ← (SRC1[127:96] < 0) ? 0 : SRC1[111:96];
DEST[63:48] ← (SRC1[127:96] > FFFFH) ? FFFFH : TMP[63:48] ;
TMP[79:64] ← (SRC2[31:0] < 0) ? 0 : SRC2[15:0];
DEST[79:64] ← (SRC2[31:0] > FFFFH) ? FFFFH : TMP[79:64] ;
TMP[95:80] ← (SRC2[63:32] < 0) ? 0 : SRC2[47:32];
DEST[95:80] ← (SRC2[63:32] > FFFFH) ? FFFFH : TMP[95:80] ;
TMP[111:96] ← (SRC2[95:64] < 0) ? 0 : SRC2[79:64];
DEST[111:96] ← (SRC2[95:64] > FFFFH) ? FFFFH : TMP[111:96] ;
TMP[127:112] ← (SRC2[127:96] < 0) ? 0 : SRC2[111:96];
DEST[127:112] ← (SRC2[127:96] > FFFFH) ? FFFFH : TMP[127:112];
DEST[MAXVL-1:128] ← 0;
```

VPACKUSDW (VEX.256 encoded version)

```
TMP[15:0] ← (SRC1[31:0] < 0) ? 0 : SRC1[15:0];
DEST[15:0] ← (SRC1[31:0] > FFFFH) ? FFFFH : TMP[15:0] ;
TMP[31:16] ← (SRC1[63:32] < 0) ? 0 : SRC1[47:32];
DEST[31:16] ← (SRC1[63:32] > FFFFH) ? FFFFH : TMP[31:16] ;
TMP[47:32] ← (SRC1[95:64] < 0) ? 0 : SRC1[79:64];
DEST[47:32] ← (SRC1[95:64] > FFFFH) ? FFFFH : TMP[47:32] ;
TMP[63:48] ← (SRC1[127:96] < 0) ? 0 : SRC1[111:96];
DEST[63:48] ← (SRC1[127:96] > FFFFH) ? FFFFH : TMP[63:48] ;
TMP[79:64] ← (SRC2[31:0] < 0) ? 0 : SRC2[15:0];
DEST[79:64] ← (SRC2[31:0] > FFFFH) ? FFFFH : TMP[79:64] ;
TMP[95:80] ← (SRC2[63:32] < 0) ? 0 : SRC2[47:32];
DEST[95:80] ← (SRC2[63:32] > FFFFH) ? FFFFH : TMP[95:80] ;
TMP[111:96] ← (SRC2[95:64] < 0) ? 0 : SRC2[79:64];
DEST[111:96] ← (SRC2[95:64] > FFFFH) ? FFFFH : TMP[111:96] ;
```

```
TMP[127:112] ← (SRC2[127:96] < 0) ? 0 : SRC2[111:96];
DEST[127:112] ← (SRC2[127:96] > FFFFH) ? FFFFH : TMP[127:112] ;
TMP[143:128] ← (SRC1[159:128] < 0) ? 0 : SRC1[143:128];
DEST[143:128] ← (SRC1[159:128] > FFFFH) ? FFFFH : TMP[143:128] ;
TMP[159:144] ← (SRC1[191:160] < 0) ? 0 : SRC1[175:160];
DEST[159:144] ← (SRC1[191:160] > FFFFH) ? FFFFH : TMP[159:144] ;
TMP[175:160] ← (SRC1[223:192] < 0) ? 0 : SRC1[207:192];
DEST[175:160] ← (SRC1[223:192] > FFFFH) ? FFFFH : TMP[175:160] ;
TMP[191:176] ← (SRC1[255:224] < 0) ? 0 : SRC1[239:224];
DEST[191:176] ← (SRC1[255:224] > FFFFH) ? FFFFH : TMP[191:176] ;
TMP[207:192] ← (SRC2[159:128] < 0) ? 0 : SRC2[143:128];
DEST[207:192] ← (SRC2[159:128] > FFFFH) ? FFFFH : TMP[207:192] ;
TMP[223:208] ← (SRC2[191:160] < 0) ? 0 : SRC2[175:160];
DEST[223:208] ← (SRC2[191:160] > FFFFH) ? FFFFH : TMP[223:208] ;
TMP[239:224] ← (SRC2[223:192] < 0) ? 0 : SRC2[207:192];
DEST[239:224] ← (SRC2[223:192] > FFFFH) ? FFFFH : TMP[239:224] ;
TMP[255:240] ← (SRC2[255:224] < 0) ? 0 : SRC2[239:224];
DEST[255:240] ← (SRC2[255:224] > FFFFH) ? FFFFH : TMP[255:240] ;
DEST[MAXVL-1:256] ← 0;
```

VPACKUSDW (EVEX encoded versions)

```
(KL, VL) = (8, 128), (16, 256), (32, 512)
FOR j ← 0 TO ((KL/2) - 1)
    i ← j * 32

    IF (EVEX.b == 1) AND (SRC2 *is memory*)
        THEN
            TMP_SRC2[i+31:i] ← SRC2[31:0]
        ELSE
            TMP_SRC2[i+31:i] ← SRC2[i+31:i]
    FI;
ENDFOR;

TMP[15:0] ← (SRC1[31:0] < 0) ? 0 : SRC1[15:0];
DEST[15:0] ← (SRC1[31:0] > FFFFH) ? FFFFH : TMP[15:0] ;
TMP[31:16] ← (SRC1[63:32] < 0) ? 0 : SRC1[47:32];
DEST[31:16] ← (SRC1[63:32] > FFFFH) ? FFFFH : TMP[31:16] ;
TMP[47:32] ← (SRC1[95:64] < 0) ? 0 : SRC1[79:64];
DEST[47:32] ← (SRC1[95:64] > FFFFH) ? FFFFH : TMP[47:32] ;
TMP[63:48] ← (SRC1[127:96] < 0) ? 0 : SRC1[111:96];
DEST[63:48] ← (SRC1[127:96] > FFFFH) ? FFFFH : TMP[63:48] ;
TMP[79:64] ← (TMP_SRC2[31:0] < 0) ? 0 : TMP_SRC2[15:0];
DEST[79:64] ← (TMP_SRC2[31:0] > FFFFH) ? FFFFH : TMP[79:64] ;
TMP[95:80] ← (TMP_SRC2[63:32] < 0) ? 0 : TMP_SRC2[47:32];
DEST[95:80] ← (TMP_SRC2[63:32] > FFFFH) ? FFFFH : TMP[95:80] ;
TMP[111:96] ← (TMP_SRC2[95:64] < 0) ? 0 : TMP_SRC2[79:64];
DEST[111:96] ← (TMP_SRC2[95:64] > FFFFH) ? FFFFH : TMP[111:96] ;
TMP[127:112] ← (TMP_SRC2[127:96] < 0) ? 0 : TMP_SRC2[111:96];
DEST[127:112] ← (TMP_SRC2[127:96] > FFFFH) ? FFFFH : TMP[127:112] ;
IF VL >= 256
    TMP[143:128] ← (SRC1[159:128] < 0) ? 0 : SRC1[143:128];
    DEST[143:128] ← (SRC1[159:128] > FFFFH) ? FFFFH : TMP[143:128] ;
    TMP[159:144] ← (SRC1[191:160] < 0) ? 0 : SRC1[175:160];
    DEST[159:144] ← (SRC1[191:160] > FFFFH) ? FFFFH : TMP[159:144] ;
```

```
    TMP[175:160] ← (SRC1[223:192] < 0) ? 0 : SRC1[207:192];
    DEST[175:160] ← (SRC1[223:192] > FFFFH) ? FFFFH : TMP[175:160] ;
    TMP[191:176] ← (SRC1[255:224] < 0) ? 0 : SRC1[239:224];
    DEST[191:176] ← (SRC1[255:224] > FFFFH) ? FFFFH : TMP[191:176] ;
    TMP[207:192] ← (TMP_SRC2[159:128] < 0) ? 0 : TMP_SRC2[143:128];
    DEST[207:192] ← (TMP_SRC2[159:128] > FFFFH) ? FFFFH : TMP[207:192] ;
    TMP[223:208] ← (TMP_SRC2[191:160] < 0) ? 0 : TMP_SRC2[175:160];
    DEST[223:208] ← (TMP_SRC2[191:160] > FFFFH) ? FFFFH : TMP[223:208] ;
    TMP[239:224] ← (TMP_SRC2[223:192] < 0) ? 0 : TMP_SRC2[207:192];
    DEST[239:224] ← (TMP_SRC2[223:192] > FFFFH) ? FFFFH : TMP[239:224] ;
    TMP[255:240] ← (TMP_SRC2[255:224] < 0) ? 0 : TMP_SRC2[239:224];
    DEST[255:240] ← (TMP_SRC2[255:224] > FFFFH) ? FFFFH : TMP[255:240] ;
FI;
IF VL >= 512
    TMP[271:256] ← (SRC1[287:256] < 0) ? 0 : SRC1[271:256];
    DEST[271:256] ← (SRC1[287:256] > FFFFH) ? FFFFH : TMP[271:256] ;
    TMP[287:272] ← (SRC1[319:288] < 0) ? 0 : SRC1[303:288];
    DEST[287:272] ← (SRC1[319:288] > FFFFH) ? FFFFH : TMP[287:272] ;
    TMP[303:288] ← (SRC1[351:320] < 0) ? 0 : SRC1[335:320];
    DEST[303:288] ← (SRC1[351:320] > FFFFH) ? FFFFH : TMP[303:288] ;
    TMP[319:304] ← (SRC1[383:352] < 0) ? 0 : SRC1[367:352];
    DEST[319:304] ← (SRC1[383:352] > FFFFH) ? FFFFH : TMP[319:304] ;
    TMP[335:320] ← (TMP_SRC2[287:256] < 0) ? 0 : TMP_SRC2[271:256];
    DEST[335:304] ← (TMP_SRC2[287:256] > FFFFH) ? FFFFH : TMP[79:64] ;
    TMP[351:336] ← (TMP_SRC2[319:288] < 0) ? 0 : TMP_SRC2[303:288];
    DEST[351:336] ← (TMP_SRC2[319:288] > FFFFH) ? FFFFH : TMP[351:336] ;
    TMP[367:352] ← (TMP_SRC2[351:320] < 0) ? 0 : TMP_SRC2[315:320];
    DEST[367:352] ← (TMP_SRC2[351:320] > FFFFH) ? FFFFH : TMP[367:352] ;
    TMP[383:368] ← (TMP_SRC2[383:352] < 0) ? 0 : TMP_SRC2[367:352];
    DEST[383:368] ← (TMP_SRC2[383:352] > FFFFH) ? FFFFH : TMP[383:368] ;
    TMP[399:384] ← (SRC1[415:384] < 0) ? 0 : SRC1[399:384];
    DEST[399:384] ← (SRC1[415:384] > FFFFH) ? FFFFH : TMP[399:384] ;
    TMP[415:400] ← (SRC1[447:416] < 0) ? 0 : SRC1[431:416];
    DEST[415:400] ← (SRC1[447:416] > FFFFH) ? FFFFH : TMP[415:400] ;
    TMP[431:416] ← (SRC1[479:448] < 0) ? 0 : SRC1[463:448];
    DEST[431:416] ← (SRC1[479:448] > FFFFH) ? FFFFH : TMP[431:416] ;
    TMP[447:432] ← (SRC1[511:480] < 0) ? 0 : SRC1[495:480];
    DEST[447:432] ← (SRC1[511:480] > FFFFH) ? FFFFH : TMP[447:432] ;
    TMP[463:448] ← (TMP_SRC2[415:384] < 0) ? 0 : TMP_SRC2[399:384];
    DEST[463:448] ← (TMP_SRC2[415:384] > FFFFH) ? FFFFH : TMP[463:448] ;
    TMP[475:464] ← (TMP_SRC2[447:416] < 0) ? 0 : TMP_SRC2[431:416];
    DEST[475:464] ← (TMP_SRC2[447:416] > FFFFH) ? FFFFH : TMP[475:464] ;
    TMP[491:476] ← (TMP_SRC2[479:448] < 0) ? 0 : TMP_SRC2[463:448];
    DEST[491:476] ← (TMP_SRC2[479:448] > FFFFH) ? FFFFH : TMP[491:476] ;
    TMP[511:492] ← (TMP_SRC2[511:480] < 0) ? 0 : TMP_SRC2[495:480];
    DEST[511:492] ← (TMP_SRC2[511:480] > FFFFH) ? FFFFH : TMP[511:492] ;
FI;
FOR j ← 0 TO KL-1
    i ← j * 16
    IF k1[j] OR *no writemask*
        THEN
            DEST[i+15:i] ← TMP_DEST[i+15:i]
        ELSE
            IF *merging-masking*                    ; merging-masking
```

```
                    THEN *DEST[i+15:i] remains unchanged*
                    ELSE *zeroing-masking*                  ; zeroing-masking
                        DEST[i+15:i] ← 0
                FI
    FI;
ENDFOR;
DEST[MAXVL-1:VL] ← 0
```

Intel C/C++ Compiler Intrinsic Equivalents

VPACKUSDW__m512i _mm512_packus_epi32(__m512i m1, __m512i m2);
VPACKUSDW__m512i _mm512_mask_packus_epi32(__m512i s, __mmask32 k, __m512i m1, __m512i m2);
VPACKUSDW__m512i _mm512_maskz_packus_epi32(__mmask32 k, __m512i m1, __m512i m2);
VPACKUSDW__m256i _mm256_mask_packus_epi32(__m256i s, __mmask16 k, __m256i m1, __m256i m2);
VPACKUSDW__m256i _mm256_maskz_packus_epi32(__mmask16 k, __m256i m1, __m256i m2);
VPACKUSDW__m128i _mm_mask_packus_epi32(__m128i s, __mmask8 k, __m128i m1, __m128i m2);
VPACKUSDW__m128i _mm_maskz_packus_epi32(__mmask8 k, __m128i m1, __m128i m2);
PACKUSDW__m128i _mm_packus_epi32(__m128i m1, __m128i m2);
VPACKUSDW__m256i _mm256_packus_epi32(__m256i m1, __m256i m2);

SIMD Floating-Point Exceptions

None

Other Exceptions

Non-EVEX-encoded instruction, see Exceptions Type 4.

EVEX-encoded instruction, see Exceptions Type E4NF.

PACKUSWB—Pack with Unsigned Saturation

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F 67 /r[1] PACKUSWB *mm, mm/m64*	A	V/V	MMX	Converts 4 signed word integers from *mm* and 4 signed word integers from *mm/m64* into 8 unsigned byte integers in *mm* using unsigned saturation.
66 0F 67 /r PACKUSWB *xmm1, xmm2/m128*	A	V/V	SSE2	Converts 8 signed word integers from *xmm1* and 8 signed word integers from *xmm2/m128* into 16 unsigned byte integers in *xmm1* using unsigned saturation.
VEX.NDS.128.66.0F.WIG 67 /r VPACKUSWB *xmm1, xmm2, xmm3/m128*	B	V/V	AVX	Converts 8 signed word integers from *xmm2* and 8 signed word integers from *xmm3/m128* into 16 unsigned byte integers in *xmm1* using unsigned saturation.
VEX.NDS.256.66.0F.WIG 67 /r VPACKUSWB *ymm1, ymm2, ymm3/m256*	B	V/V	AVX2	Converts 16 signed word integers from *ymm2* and 16signed word integers from *ymm3/m256* into 32 unsigned byte integers in *ymm1* using unsigned saturation.
EVEX.NDS.128.66.0F.WIG 67 /r VPACKUSWB *xmm1{k1}{z}, xmm2, xmm3/m128*	C	V/V	AVX512VL AVX512BW	Converts signed word integers from *xmm2* and signed word integers from *xmm3/m128* into unsigned byte integers in *xmm1* using unsigned saturation under writemask k1.
EVEX.NDS.256.66.0F.WIG 67 /r VPACKUSWB *ymm1{k1}{z}, ymm2, ymm3/m256*	C	V/V	AVX512VL AVX512BW	Converts signed word integers from *ymm2* and signed word integers from *ymm3/m256* into unsigned byte integers in *ymm1* using unsigned saturation under writemask k1.
EVEX.NDS.512.66.0F.WIG 67 /r VPACKUSWB *zmm1{k1}{z}, zmm2, zmm3/m512*	C	V/V	AVX512BW	Converts signed word integers from *zmm2* and signed word integers from *zmm3/m512* into unsigned byte integers in *zmm1* using unsigned saturation under writemask k1.

NOTES:

1. See note in Section 2.4, "AVX and SSE Instruction Exception Specification" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A* and Section 22.25.3, "Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	NA	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
B	NA	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA
C	Full Vector Mem	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA

Description

Converts 4, 8, 16 or 32 signed word integers from the destination operand (first operand) and 4, 8, 16 or 32 signed word integers from the source operand (second operand) into 8, 16, 32 or 64 unsigned byte integers and stores the result in the destination operand. (See Figure 4-6 for an example of the packing operation.) If a signed word integer value is beyond the range of an unsigned byte integer (that is, greater than FFH or less than 00H), the saturated unsigned byte integer value of FFH or 00H, respectively, is stored in the destination.

EVEX.512 encoded version: The first source operand is a ZMM register. The second source operand is a ZMM register or a 512-bit memory location. The destination operand is a ZMM register.

VEX.256 and EVEX.256 encoded versions: The first source operand is a YMM register. The second source operand is a YMM register or a 256-bit memory location. The destination operand is a YMM register. The upper bits (MAXVL-1:256) of the corresponding ZMM register destination are zeroed.

VEX.128 and EVEX.128 encoded versions: The first source operand is an XMM register. The second source operand is an XMM register or 128-bit memory location. The destination operand is an XMM register. The upper bits (MAXVL-1:128) of the corresponding register destination are zeroed.

128-bit Legacy SSE version: The first source operand is an XMM register. The second operand can be an XMM register or an 128-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (MAXVL-1:128) of the corresponding register destination are unmodified.

Operation

PACKUSWB (with 64-bit operands)

```
DEST[7:0] ← SaturateSignedWordToUnsignedByte DEST[15:0];
DEST[15:8] ← SaturateSignedWordToUnsignedByte DEST[31:16];
DEST[23:16] ← SaturateSignedWordToUnsignedByte DEST[47:32];
DEST[31:24] ← SaturateSignedWordToUnsignedByte DEST[63:48];
DEST[39:32] ← SaturateSignedWordToUnsignedByte SRC[15:0];
DEST[47:40] ← SaturateSignedWordToUnsignedByte SRC[31:16];
DEST[55:48] ← SaturateSignedWordToUnsignedByte SRC[47:32];
DEST[63:56] ← SaturateSignedWordToUnsignedByte SRC[63:48];
```

PACKUSWB (Legacy SSE instruction)

```
DEST[7:0]←SaturateSignedWordToUnsignedByte (DEST[15:0]);
DEST[15:8] ←SaturateSignedWordToUnsignedByte (DEST[31:16]);
DEST[23:16] ←SaturateSignedWordToUnsignedByte (DEST[47:32]);
DEST[31:24] ← SaturateSignedWordToUnsignedByte (DEST[63:48]);
DEST[39:32] ← SaturateSignedWordToUnsignedByte (DEST[79:64]);
DEST[47:40] ← SaturateSignedWordToUnsignedByte (DEST[95:80]);
DEST[55:48] ← SaturateSignedWordToUnsignedByte (DEST[111:96]);
DEST[63:56] ← SaturateSignedWordToUnsignedByte (DEST[127:112]);
DEST[71:64] ← SaturateSignedWordToUnsignedByte (SRC[15:0]);
DEST[79:72] ← SaturateSignedWordToUnsignedByte (SRC[31:16]);
DEST[87:80] ← SaturateSignedWordToUnsignedByte (SRC[47:32]);
DEST[95:88] ← SaturateSignedWordToUnsignedByte (SRC[63:48]);
DEST[103:96] ← SaturateSignedWordToUnsignedByte (SRC[79:64]);
DEST[111:104] ← SaturateSignedWordToUnsignedByte (SRC[95:80]);
DEST[119:112] ← SaturateSignedWordToUnsignedByte (SRC[111:96]);
DEST[127:120] ← SaturateSignedWordToUnsignedByte (SRC[127:112]);
```

PACKUSWB (VEX.128 encoded version)

```
DEST[7:0]← SaturateSignedWordToUnsignedByte (SRC1[15:0]);
DEST[15:8] ←SaturateSignedWordToUnsignedByte (SRC1[31:16]);
DEST[23:16] ←SaturateSignedWordToUnsignedByte (SRC1[47:32]);
DEST[31:24] ← SaturateSignedWordToUnsignedByte (SRC1[63:48]);
DEST[39:32] ← SaturateSignedWordToUnsignedByte (SRC1[79:64]);
DEST[47:40] ← SaturateSignedWordToUnsignedByte (SRC1[95:80]);
DEST[55:48] ← SaturateSignedWordToUnsignedByte (SRC1[111:96]);
DEST[63:56] ← SaturateSignedWordToUnsignedByte (SRC1[127:112]);
DEST[71:64] ← SaturateSignedWordToUnsignedByte (SRC2[15:0]);
DEST[79:72] ← SaturateSignedWordToUnsignedByte (SRC2[31:16]);
DEST[87:80] ← SaturateSignedWordToUnsignedByte (SRC2[47:32]);
DEST[95:88] ← SaturateSignedWordToUnsignedByte (SRC2[63:48]);
DEST[103:96] ← SaturateSignedWordToUnsignedByte (SRC2[79:64]);
DEST[111:104] ← SaturateSignedWordToUnsignedByte (SRC2[95:80]);
```

DEST[119:112] ← SaturateSignedWordToUnsignedByte (SRC2[111:96]);
DEST[127:120] ← SaturateSignedWordToUnsignedByte (SRC2[127:112]);
DEST[MAXVL-1:128] ← 0;

VPACKUSWB (VEX.256 encoded version)

DEST[7:0]← SaturateSignedWordToUnsignedByte (SRC1[15:0]);
DEST[15:8] ←SaturateSignedWordToUnsignedByte (SRC1[31:16]);
DEST[23:16] ←SaturateSignedWordToUnsignedByte (SRC1[47:32]);
DEST[31:24] ← SaturateSignedWordToUnsignedByte (SRC1[63:48]);
DEST[39:32] ←SaturateSignedWordToUnsignedByte (SRC1[79:64]);
DEST[47:40] ← SaturateSignedWordToUnsignedByte (SRC1[95:80]);
DEST[55:48] ← SaturateSignedWordToUnsignedByte (SRC1[111:96]);
DEST[63:56] ← SaturateSignedWordToUnsignedByte (SRC1[127:112]);
DEST[71:64] ←SaturateSignedWordToUnsignedByte (SRC2[15:0]);
DEST[79:72] ← SaturateSignedWordToUnsignedByte (SRC2[31:16]);
DEST[87:80] ← SaturateSignedWordToUnsignedByte (SRC2[47:32]);
DEST[95:88] ← SaturateSignedWordToUnsignedByte (SRC2[63:48]);
DEST[103:96] ← SaturateSignedWordToUnsignedByte (SRC2[79:64]);
DEST[111:104] ← SaturateSignedWordToUnsignedByte (SRC2[95:80]);
DEST[119:112] ← SaturateSignedWordToUnsignedByte (SRC2[111:96]);
DEST[127:120] ← SaturateSignedWordToUnsignedByte (SRC2[127:112]);
DEST[135:128]← SaturateSignedWordToUnsignedByte (SRC1[143:128]);
DEST[143:136] ←SaturateSignedWordToUnsignedByte (SRC1[159:144]);
DEST[151:144] ←SaturateSignedWordToUnsignedByte (SRC1[175:160]);
DEST[159:152] ←SaturateSignedWordToUnsignedByte (SRC1[191:176]);
DEST[167:160] ← SaturateSignedWordToUnsignedByte (SRC1[207:192]);
DEST[175:168] ← SaturateSignedWordToUnsignedByte (SRC1[223:208]);
DEST[183:176] ← SaturateSignedWordToUnsignedByte (SRC1[239:224]);
DEST[191:184] ← SaturateSignedWordToUnsignedByte (SRC1[255:240]);
DEST[199:192] ← SaturateSignedWordToUnsignedByte (SRC2[143:128]);
DEST[207:200] ← SaturateSignedWordToUnsignedByte (SRC2[159:144]);
DEST[215:208] ← SaturateSignedWordToUnsignedByte (SRC2[175:160]);
DEST[223:216] ← SaturateSignedWordToUnsignedByte (SRC2[191:176]);
DEST[231:224] ← SaturateSignedWordToUnsignedByte (SRC2[207:192]);
DEST[239:232] ← SaturateSignedWordToUnsignedByte (SRC2[223:208]);
DEST[247:240] ← SaturateSignedWordToUnsignedByte (SRC2[239:224]);
DEST[255:248] ← SaturateSignedWordToUnsignedByte (SRC2[255:240]);

VPACKUSWB (EVEX encoded versions)

(KL, VL) = (16, 128), (32, 256), (64, 512)
TMP_DEST[7:0] ← SaturateSignedWordToUnsignedByte (SRC1[15:0]);
TMP_DEST[15:8] ← SaturateSignedWordToUnsignedByte (SRC1[31:16]);
TMP_DEST[23:16] ← SaturateSignedWordToUnsignedByte (SRC1[47:32]);
TMP_DEST[31:24] ← SaturateSignedWordToUnsignedByte (SRC1[63:48]);
TMP_DEST[39:32] ← SaturateSignedWordToUnsignedByte (SRC1[79:64]);
TMP_DEST[47:40] ← SaturateSignedWordToUnsignedByte (SRC1[95:80]);
TMP_DEST[55:48] ← SaturateSignedWordToUnsignedByte (SRC1[111:96]);
TMP_DEST[63:56] ← SaturateSignedWordToUnsignedByte (SRC1[127:112]);
TMP_DEST[71:64] ← SaturateSignedWordToUnsignedByte (SRC2[15:0]);
TMP_DEST[79:72] ← SaturateSignedWordToUnsignedByte (SRC2[31:16]);
TMP_DEST[87:80] ← SaturateSignedWordToUnsignedByte (SRC2[47:32]);
TMP_DEST[95:88] ← SaturateSignedWordToUnsignedByte (SRC2[63:48]);
TMP_DEST[103:96] ← SaturateSignedWordToUnsignedByte (SRC2[79:64]);
TMP_DEST[111:104] ← SaturateSignedWordToUnsignedByte (SRC2[95:80]);

```
TMP_DEST[119:112] ← SaturateSignedWordToUnsignedByte (SRC2[111:96]);
TMP_DEST[127:120] ← SaturateSignedWordToUnsignedByte (SRC2[127:112]);
IF VL >= 256
    TMP_DEST[135:128]← SaturateSignedWordToUnsignedByte (SRC1[143:128]);
    TMP_DEST[143:136] ← SaturateSignedWordToUnsignedByte (SRC1[159:144]);
    TMP_DEST[151:144] ← SaturateSignedWordToUnsignedByte (SRC1[175:160]);
    TMP_DEST[159:152] ← SaturateSignedWordToUnsignedByte (SRC1[191:176]);
    TMP_DEST[167:160] ← SaturateSignedWordToUnsignedByte (SRC1[207:192]);
    TMP_DEST[175:168] ← SaturateSignedWordToUnsignedByte (SRC1[223:208]);
    TMP_DEST[183:176] ← SaturateSignedWordToUnsignedByte (SRC1[239:224]);
    TMP_DEST[191:184] ← SaturateSignedWordToUnsignedByte (SRC1[255:240]);
    TMP_DEST[199:192] ← SaturateSignedWordToUnsignedByte (SRC2[143:128]);
    TMP_DEST[207:200] ← SaturateSignedWordToUnsignedByte (SRC2[159:144]);
    TMP_DEST[215:208] ← SaturateSignedWordToUnsignedByte (SRC2[175:160]);
    TMP_DEST[223:216] ← SaturateSignedWordToUnsignedByte (SRC2[191:176]);
    TMP_DEST[231:224] ← SaturateSignedWordToUnsignedByte (SRC2[207:192]);
    TMP_DEST[239:232] ← SaturateSignedWordToUnsignedByte (SRC2[223:208]);
    TMP_DEST[247:240] ← SaturateSignedWordToUnsignedByte (SRC2[239:224]);
    TMP_DEST[255:248] ← SaturateSignedWordToUnsignedByte (SRC2[255:240]);
FI;
IF VL >= 512
    TMP_DEST[263:256] ← SaturateSignedWordToUnsignedByte (SRC1[271:256]);
    TMP_DEST[271:264] ← SaturateSignedWordToUnsignedByte (SRC1[287:272]);
    TMP_DEST[279:272] ← SaturateSignedWordToUnsignedByte (SRC1[303:288]);
    TMP_DEST[287:280] ← SaturateSignedWordToUnsignedByte (SRC1[319:304]);
    TMP_DEST[295:288] ← SaturateSignedWordToUnsignedByte (SRC1[335:320]);
    TMP_DEST[303:296] ← SaturateSignedWordToUnsignedByte (SRC1[351:336]);
    TMP_DEST[311:304] ← SaturateSignedWordToUnsignedByte (SRC1[367:352]);
    TMP_DEST[319:312] ← SaturateSignedWordToUnsignedByte (SRC1[383:368]);

    TMP_DEST[327:320] ← SaturateSignedWordToUnsignedByte (SRC2[271:256]);
    TMP_DEST[335:328] ← SaturateSignedWordToUnsignedByte (SRC2[287:272]);
    TMP_DEST[343:336] ← SaturateSignedWordToUnsignedByte (SRC2[303:288]);
    TMP_DEST[351:344] ← SaturateSignedWordToUnsignedByte (SRC2[319:304]);
    TMP_DEST[359:352] ← SaturateSignedWordToUnsignedByte (SRC2[335:320]);
    TMP_DEST[367:360] ← SaturateSignedWordToUnsignedByte (SRC2[351:336]);
    TMP_DEST[375:368] ← SaturateSignedWordToUnsignedByte (SRC2[367:352]);
    TMP_DEST[383:376] ← SaturateSignedWordToUnsignedByte (SRC2[383:368]);

    TMP_DEST[391:384] ← SaturateSignedWordToUnsignedByte (SRC1[399:384]);
    TMP_DEST[399:392] ← SaturateSignedWordToUnsignedByte (SRC1[415:400]);
    TMP_DEST[407:400] ← SaturateSignedWordToUnsignedByte (SRC1[431:416]);
    TMP_DEST[415:408] ← SaturateSignedWordToUnsignedByte (SRC1[447:432]);
    TMP_DEST[423:416] ← SaturateSignedWordToUnsignedByte (SRC1[463:448]);
    TMP_DEST[431:424] ← SaturateSignedWordToUnsignedByte (SRC1[479:464]);
    TMP_DEST[439:432] ← SaturateSignedWordToUnsignedByte (SRC1[495:480]);
    TMP_DEST[447:440] ← SaturateSignedWordToUnsignedByte (SRC1[511:496]);

    TMP_DEST[455:448] ← SaturateSignedWordToUnsignedByte (SRC2[399:384]);
    TMP_DEST[463:456] ← SaturateSignedWordToUnsignedByte (SRC2[415:400]);
    TMP_DEST[471:464] ← SaturateSignedWordToUnsignedByte (SRC2[431:416]);
    TMP_DEST[479:472] ← SaturateSignedWordToUnsignedByte (SRC2[447:432]);
    TMP_DEST[487:480] ← SaturateSignedWordToUnsignedByte (SRC2[463:448]);
    TMP_DEST[495:488] ← SaturateSignedWordToUnsignedByte (SRC2[479:464]);
```

```
    TMP_DEST[503:496] ← SaturateSignedWordToUnsignedByte (SRC2[495:480]);
    TMP_DEST[511:504] ← SaturateSignedWordToUnsignedByte (SRC2[511:496]);
FI;
FOR j ← 0 TO KL-1
    i ← j * 8
    IF k1[j] OR *no writemask*
        THEN
            DEST[i+7:i] ← TMP_DEST[i+7:i]
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+7:i] remains unchanged*
                ELSE *zeroing-masking*                  ; zeroing-masking
                    DEST[i+7:i] ← 0
            FI
    FI;
ENDFOR;
DEST[MAXVL-1:VL] ← 0
```

Intel C/C++ Compiler Intrinsic Equivalents

VPACKUSWB__m512i _mm512_packus_epi16(__m512i m1, __m512i m2);
VPACKUSWB__m512i _mm512_mask_packus_epi16(__m512i s, __mmask64 k, __m512i m1, __m512i m2);
VPACKUSWB__m512i _mm512_maskz_packus_epi16(__mmask64 k, __m512i m1, __m512i m2);
VPACKUSWB__m256i _mm256_mask_packus_epi16(__m256i s, __mmask32 k, __m256i m1, __m256i m2);
VPACKUSWB__m256i _mm256_maskz_packus_epi16(__mmask32 k, __m256i m1, __m256i m2);
VPACKUSWB__m128i _mm_mask_packus_epi16(__m128i s, __mmask16 k, __m128i m1, __m128i m2);
VPACKUSWB__m128i _mm_maskz_packus_epi16(__mmask16 k, __m128i m1, __m128i m2);

PACKUSWB: __m64 _mm_packs_pu16(__m64 m1, __m64 m2)

(V)PACKUSWB: __m128i _mm_packus_epi16(__m128i m1, __m128i m2)

VPACKUSWB: __m256i _mm256_packus_epi16(__m256i m1, __m256i m2);

Flags Affected

None

SIMD Floating-Point Exceptions

None

Other Exceptions

Non-EVEX-encoded instruction, see Exceptions Type 4.

EVEX-encoded instruction, see Exceptions Type E4NF.nb.

PADDB/PADDW/PADDD/PADDQ—Add Packed Integers

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F FC /r[1] PADDB *mm, mm/m64*	A	V/V	MMX	Add packed byte integers from *mm/m64* and *mm*.
NP 0F FD /r[1] PADDW *mm, mm/m64*	A	V/V	MMX	Add packed word integers from *mm/m64* and *mm*.
NP 0F FE /r[1] PADDD *mm, mm/m64*	A	V/V	MMX	Add packed doubleword integers from *mm/m64* and *mm*.
NP 0F D4 /r[1] PADDQ *mm, mm/m64*	A	V/V	MMX	Add packed quadword integers from *mm/m64* and *mm*.
66 0F FC /r PADDB *xmm1, xmm2/m128*	A	V/V	SSE2	Add packed byte integers from *xmm2/m128* and *xmm1*.
66 0F FD /r PADDW *xmm1, xmm2/m128*	A	V/V	SSE2	Add packed word integers from *xmm2/m128* and *xmm1*.
66 0F FE /r PADDD *xmm1, xmm2/m128*	A	V/V	SSE2	Add packed doubleword integers from *xmm2/m128* and *xmm1*.
66 0F D4 /r PADDQ *xmm1, xmm2/m128*	A	V/V	SSE2	Add packed quadword integers from *xmm2/m128* and *xmm1*.
VEX.NDS.128.66.0F.WIG FC /r VPADDB *xmm1, xmm2, xmm3/m128*	B	V/V	AVX	Add packed byte integers from *xmm2, and xmm3/m128* and store in *xmm1*.
VEX.NDS.128.66.0F.WIG FD /r VPADDW *xmm1, xmm2, xmm3/m128*	B	V/V	AVX	Add packed word integers from *xmm2, xmm3/m128* and *store in xmm1*.
VEX.NDS.128.66.0F.WIG FE /r VPADDD *xmm1, xmm2, xmm3/m128*	B	V/V	AVX	Add packed doubleword integers from *xmm2, xmm3/m128* and store in *xmm1*.
VEX.NDS.128.66.0F.WIG D4 /r VPADDQ *xmm1, xmm2, xmm3/m128*	B	V/V	AVX	Add packed quadword integers from *xmm2, xmm3/m128* and store in *xmm1*.
VEX.NDS.256.66.0F.WIG FC /r VPADDB *ymm1, ymm2, ymm3/m256*	B	V/V	AVX2	Add packed byte integers from *ymm2, and ymm3/m256* and store in *ymm1*.
VEX.NDS.256.66.0F.WIG FD /r VPADDW *ymm1, ymm2, ymm3/m256*	B	V/V	AVX2	Add packed word integers from *ymm2, ymm3/m256* and *store in ymm1*.
VEX.NDS.256.66.0F.WIG FE /r VPADDD *ymm1, ymm2, ymm3/m256*	B	V/V	AVX2	Add packed doubleword integers from *ymm2, ymm3/m256* and *store in ymm1*.
VEX.NDS.256.66.0F.WIG D4 /r VPADDQ *ymm1, ymm2, ymm3/m256*	B	V/V	AVX2	Add packed quadword integers from *ymm2, ymm3/m256* and store in *ymm1*.
EVEX.NDS.128.66.0F.WIG FC /r VPADDB *xmm1 {k1}{z}, xmm2, xmm3/m128*	C	V/V	AVX512VL AVX512BW	Add packed byte integers from *xmm2, and xmm3/m128* and store in *xmm1* using writemask k1.
EVEX.NDS.128.66.0F.WIG FD /r VPADDW *xmm1 {k1}{z}, xmm2, xmm3/m128*	C	V/V	AVX512VL AVX512BW	Add packed word integers from *xmm2, and xmm3/m128* and store in *xmm1* using writemask k1.
EVEX.NDS.128.66.0F.W0 FE /r VPADDD *xmm1 {k1}{z}, xmm2,* xmm3/m128/m32bcst	D	V/V	AVX512VL AVX512F	Add packed doubleword integers from *xmm2, and xmm3/m128/m32bcst* and store in *xmm1* using writemask k1.
EVEX.NDS.128.66.0F.W1 D4 /r VPADDQ *xmm1 {k1}{z}, xmm2,* xmm3/m128/m64bcst	D	V/V	AVX512VL AVX512F	Add packed quadword integers from *xmm2, and xmm3/m128/m64bcst* and store in *xmm1* using writemask k1.
EVEX.NDS.256.66.0F.WIG FC /r VPADDB *ymm1 {k1}{z}, ymm2, ymm3/m256*	C	V/V	AVX512VL AVX512BW	Add packed byte integers from *ymm2, and ymm3/m256* and store in *ymm1* using writemask k1.
EVEX.NDS.256.66.0F.WIG FD /r VPADDW *ymm1 {k1}{z}, ymm2, ymm3/m256*	C	V/V	AVX512VL AVX512BW	Add packed word integers from *ymm2, and ymm3/m256* and store in *ymm1* using writemask k1.

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.NDS.256.66.0F.W0 FE /r VPADDD *ymm1 {k1}{z}, ymm2,* ymm3/m256/m32bcst	D	V/V	AVX512VL AVX512F	Add packed doubleword integers from *ymm2, ymm3/m256/m32bcst* and store in *ymm1* using writemask k1.
EVEX.NDS.256.66.0F.W1 D4 /r VPADDQ *ymm1 {k1}{z}, ymm2,* ymm3/m256/m64bcst	D	V/V	AVX512VL AVX512F	Add packed quadword integers from *ymm2, ymm3/m256/m64bcst* and store in *ymm1* using writemask k1.
EVEX.NDS.512.66.0F.WIG FC /r VPADDB *zmm1 {k1}{z}, zmm2, zmm3/m512*	C	V/V	AVX512BW	Add packed byte integers from *zmm2, and zmm3/m512* and store in *zmm1* using writemask k1.
EVEX.NDS.512.66.0F.WIG FD /r VPADDW *zmm1 {k1}{z}, zmm2, zmm3/m512*	C	V/V	AVX512BW	Add packed word integers from *zmm2, and zmm3/m512* and store in *zmm1* using writemask k1.
EVEX.NDS.512.66.0F.W0 FE /r VPADDD *zmm1 {k1}{z}, zmm2,* zmm3/m512/m32bcst	D	V/V	AVX512F	Add packed doubleword integers from *zmm2, zmm3/m512/m32bcst* and store in *zmm1* using writemask k1.
EVEX.NDS.512.66.0F.W1 D4 /r VPADDQ *zmm1 {k1}{z}, zmm2,* zmm3/m512/m64bcst	D	V/V	AVX512F	Add packed quadword integers from *zmm2, zmm3/m512/m64bcst* and store in *zmm1* using writemask k1.

NOTES:

1. See note in Section 2.4, "AVX and SSE Instruction Exception Specification" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A* and Section 22.25.3, "Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	NA	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
B	NA	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA
C	Full Vector Mem	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA
D	Full Vector	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA

Description

Performs a SIMD add of the packed integers from the source operand (second operand) and the destination operand (first operand), and stores the packed integer results in the destination operand. See Figure 9-4 in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*, for an illustration of a SIMD operation. Overflow is handled with wraparound, as described in the following paragraphs.

The PADDB and VPADDB instructions add packed byte integers from the first source operand and second source operand and store the packed integer results in the destination operand. When an individual result is too large to be represented in 8 bits (overflow), the result is wrapped around and the low 8 bits are written to the destination operand (that is, the carry is ignored).

The PADDW and VPADDW instructions add packed word integers from the first source operand and second source operand and store the packed integer results in the destination operand. When an individual result is too large to be represented in 16 bits (overflow), the result is wrapped around and the low 16 bits are written to the destination operand (that is, the carry is ignored).

The PADDD and VPADDD instructions add packed doubleword integers from the first source operand and second source operand and store the packed integer results in the destination operand. When an individual result is too large to be represented in 32 bits (overflow), the result is wrapped around and the low 32 bits are written to the destination operand (that is, the carry is ignored).

The PADDQ and VPADDQ instructions add packed quadword integers from the first source operand and second source operand and store the packed integer results in the destination operand. When a quadword result is too

large to be represented in 64 bits (overflow), the result is wrapped around and the low 64 bits are written to the destination operand (that is, the carry is ignored).

Note that the (V)PADDB, (V)PADDW, (V)PADDD and (V)PADDQ instructions can operate on either unsigned or signed (two's complement notation) packed integers; however, it does not set bits in the EFLAGS register to indicate overflow and/or a carry. To prevent undetected overflow conditions, software must control the ranges of values operated on.

EVEX encoded VPADDD/Q: The first source operand is a ZMM/YMM/XMM register. The second source operand is a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 32/64-bit memory location. The destination operand is a ZMM/YMM/XMM register updated according to the writemask.

EVEX encoded VPADDB/W: The first source operand is a ZMM/YMM/XMM register. The second source operand is a ZMM/YMM/XMM register, a 512/256/128-bit memory location. The destination operand is a ZMM/YMM/XMM register updated according to the writemask.

VEX.256 encoded version: The first source operand is a YMM register. The second source operand is a YMM register or a 256-bit memory location. The destination operand is a YMM register. the upper bits (MAXVL-1:256) of the destination are cleared.

VEX.128 encoded version: The first source operand is an XMM register. The second source operand is an XMM register or 128-bit memory location. The destination operand is an XMM register. The upper bits (MAXVL-1:128) of the corresponding ZMM register destination are zeroed.

128-bit Legacy SSE version: The first source operand is an XMM register. The second operand can be an XMM register or an 128-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (MAXVL-1:128) of the corresponding ZMM register destination are unmodified.

Operation

PADDB (with 64-bit operands)

```
DEST[7:0] ← DEST[7:0] + SRC[7:0];
(* Repeat add operation for 2nd through 7th byte *)
DEST[63:56] ← DEST[63:56] + SRC[63:56];
```

PADDW (with 64-bit operands)

```
DEST[15:0] ← DEST[15:0] + SRC[15:0];
(* Repeat add operation for 2nd and 3th word *)
DEST[63:48] ← DEST[63:48] + SRC[63:48];
```

PADDD (with 64-bit operands)

```
DEST[31:0] ← DEST[31:0] + SRC[31:0];
DEST[63:32] ← DEST[63:32] + SRC[63:32];
```

PADDQ (with 64-Bit operands)

```
DEST[63:0] ← DEST[63:0] + SRC[63:0];
```

PADDB (Legacy SSE instruction)

```
DEST[7:0]← DEST[7:0] + SRC[7:0];
(* Repeat add operation for 2nd through 15th byte *)
DEST[127:120]← DEST[127:120] + SRC[127:120];
DEST[MAXVL-1:128] (Unmodified)
```

PADDW (Legacy SSE instruction)

```
DEST[15:0] ← DEST[15:0] + SRC[15:0];
(* Repeat add operation for 2nd through 7th word *)
DEST[127:112]← DEST[127:112] + SRC[127:112];
DEST[MAXVL-1:128] (Unmodified)
```

PADDD (Legacy SSE instruction)

```
DEST[31:0]← DEST[31:0]  + SRC[31:0];
(* Repeat add operation for 2nd and 3th doubleword *)
DEST[127:96]← DEST[127:96] + SRC[127:96];
DEST[MAXVL-1:128] (Unmodified)
```

PADDQ (Legacy SSE instruction)

```
DEST[63:0]← DEST[63:0]  + SRC[63:0];
DEST[127:64]← DEST[127:64] + SRC[127:64];
DEST[MAXVL-1:128] (Unmodified)
```

VPADDB (VEX.128 encoded instruction)

```
DEST[7:0]← SRC1[7:0] + SRC2[7:0];
(* Repeat add operation for 2nd through 15th byte *)
DEST[127:120]← SRC1[127:120] + SRC2[127:120];
DEST[MAXVL-1:128] ← 0;
```

VPADDW (VEX.128 encoded instruction)

```
DEST[15:0] ← SRC1[15:0] + SRC2[15:0];
(* Repeat add operation for 2nd through 7th word *)
DEST[127:112]← SRC1[127:112] + SRC2[127:112];
DEST[MAXVL-1:128] ← 0;
```

VPADDD (VEX.128 encoded instruction)

```
DEST[31:0]← SRC1[31:0]  + SRC2[31:0];
(* Repeat add operation for 2nd and 3th doubleword *)
DEST[127:96] ← SRC1[127:96] + SRC2[127:96];
DEST[MAXVL-1:128] ← 0;
```

VPADDQ (VEX.128 encoded instruction)

```
DEST[63:0]← SRC1[63:0]  + SRC2[63:0];
DEST[127:64] ← SRC1[127:64] + SRC2[127:64];
DEST[MAXVL-1:128] ← 0;
```

VPADDB (VEX.256 encoded instruction)

```
DEST[7:0]← SRC1[7:0] + SRC2[7:0];
(* Repeat add operation for 2nd through 31th byte *)
DEST[255:248]← SRC1[255:248] + SRC2[255:248];
```

VPADDW (VEX.256 encoded instruction)

```
DEST[15:0] ← SRC1[15:0] + SRC2[15:0];
(* Repeat add operation for 2nd through 15th word *)
DEST[255:240]← SRC1[255:240] + SRC2[255:240];
```

VPADDD (VEX.256 encoded instruction)

```
DEST[31:0]← SRC1[31:0]  + SRC2[31:0];
(* Repeat add operation for 2nd and 7th doubleword *)
DEST[255:224] ← SRC1[255:224] + SRC2[255:224];
```

VPADDQ (VEX.256 encoded instruction)

```
DEST[63:0]← SRC1[63:0]  + SRC2[63:0];
DEST[127:64] ← SRC1[127:64] + SRC2[127:64];
DEST[191:128]← SRC1[191:128]  + SRC2[191:128];
DEST[255:192] ← SRC1[255:192] + SRC2[255:192];
```

VPADDB (EVEX encoded versions)
(KL, VL) = (16, 128), (32, 256), (64, 512)

```
FOR j ← 0 TO KL-1
    i ← j * 8
    IF k1[j] OR *no writemask*
        THEN DEST[i+7:i] ← SRC1[i+7:i] + SRC2[i+7:i]
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+7:i] remains unchanged*
                ELSE *zeroing-masking*              ; zeroing-masking
                    DEST[i+7:i] = 0
            FI
    FI;
ENDFOR;
DEST[MAXVL-1:VL] ← 0
```

VPADDW (EVEX encoded versions)
(KL, VL) = (8, 128), (16, 256), (32, 512)

```
FOR j ← 0 TO KL-1
    i ← j * 16
    IF k1[j] OR *no writemask*
        THEN DEST[i+15:i] ← SRC1[i+15:i] + SRC2[i+15:i]
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+15:i] remains unchanged*
                ELSE *zeroing-masking*              ; zeroing-masking
                    DEST[i+15:i] = 0
            FI
    FI;
ENDFOR;
DEST[MAXVL-1:VL] ← 0
```

VPADDD (EVEX encoded versions)
(KL, VL) = (4, 128), (8, 256), (16, 512)

```
FOR j ← 0 TO KL-1
    i ← j * 32
    IF k1[j] OR *no writemask*
        THEN
            IF (EVEX.b = 1) AND (SRC2 *is memory*)
                THEN DEST[i+31:i] ← SRC1[i+31:i] + SRC2[31:0]
                ELSE DEST[i+31:i] ← SRC1[i+31:i] + SRC2[i+31:i]
            FI;
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE *zeroing-masking*              ; zeroing-masking
                    DEST[i+31:i] ← 0
            FI
    FI;
ENDFOR;
DEST[MAXVL-1:VL] ← 0
```

VPADDQ (EVEX encoded versions)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j ← 0 TO KL-1
    i ← j * 64
    IF k1[j] OR *no writemask*
        THEN
            IF (EVEX.b = 1) AND (SRC2 *is memory*)
                THEN DEST[i+63:i] ← SRC1[i+63:i] + SRC2[63:0]
                ELSE DEST[i+63:i] ← SRC1[i+63:i] + SRC2[i+63:i]
            FI;
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE *zeroing-masking*                  ; zeroing-masking
                    DEST[i+63:i] ← 0
            FI
    FI;
ENDFOR;
DEST[MAXVL-1:VL] ← 0
```

Intel C/C++ Compiler Intrinsic Equivalents

VPADDB__m512i _mm512_add_epi8 (__m512i a, __m512i b)
VPADDW__m512i _mm512_add_epi16 (__m512i a, __m512i b)
VPADDB__m512i _mm512_mask_add_epi8 (__m512i s, __mmask64 m, __m512i a, __m512i b)
VPADDW__m512i _mm512_mask_add_epi16 (__m512i s, __mmask32 m, __m512i a, __m512i b)
VPADDB__m512i _mm512_maskz_add_epi8 (__mmask64 m, __m512i a, __m512i b)
VPADDW__m512i _mm512_maskz_add_epi16 (__mmask32 m, __m512i a, __m512i b)
VPADDB__m256i _mm256_mask_add_epi8 (__m256i s, __mmask32 m, __m256i a, __m256i b)
VPADDW__m256i _mm256_mask_add_epi16 (__m256i s, __mmask16 m, __m256i a, __m256i b)
VPADDB__m256i _mm256_maskz_add_epi8 (__mmask32 m, __m256i a, __m256i b)
VPADDW__m256i _mm256_maskz_add_epi16 (__mmask16 m, __m256i a, __m256i b)
VPADDB__m128i _mm_mask_add_epi8 (__m128i s, __mmask16 m, __m128i a, __m128i b)
VPADDW__m128i _mm_mask_add_epi16 (__m128i s, __mmask8 m, __m128i a, __m128i b)
VPADDB__m128i _mm_maskz_add_epi8 (__mmask16 m, __m128i a, __m128i b)
VPADDW__m128i _mm_maskz_add_epi16 (__mmask8 m, __m128i a, __m128i b)
VPADDD __m512i _mm512_add_epi32(__m512i a, __m512i b);
VPADDD __m512i _mm512_mask_add_epi32(__m512i s, __mmask16 k, __m512i a, __m512i b);
VPADDD __m512i _mm512_maskz_add_epi32(__mmask16 k, __m512i a, __m512i b);
VPADDD __m256i _mm256_mask_add_epi32(__m256i s, __mmask8 k, __m256i a, __m256i b);
VPADDD __m256i _mm256_maskz_add_epi32(__mmask8 k, __m256i a, __m256i b);
VPADDD __m128i _mm_mask_add_epi32(__m128i s, __mmask8 k, __m128i a, __m128i b);
VPADDD __m128i _mm_maskz_add_epi32(__mmask8 k, __m128i a, __m128i b);
VPADDQ __m512i _mm512_add_epi64(__m512i a, __m512i b);
VPADDQ __m512i _mm512_mask_add_epi64(__m512i s, __mmask8 k, __m512i a, __m512i b);
VPADDQ __m512i _mm512_maskz_add_epi64(__mmask8 k, __m512i a, __m512i b);
VPADDQ __m256i _mm256_mask_add_epi64(__m256i s, __mmask8 k, __m256i a, __m256i b);
VPADDQ __m256i _mm256_maskz_add_epi64(__mmask8 k, __m256i a, __m256i b);
VPADDQ __m128i _mm_mask_add_epi64(__m128i s, __mmask8 k, __m128i a, __m128i b);
VPADDQ __m128i _mm_maskz_add_epi64(__mmask8 k, __m128i a, __m128i b);
PADDB __m128i _mm_add_epi8 (__m128i a,__m128i b);
PADDW __m128i _mm_add_epi16 (__m128i a, __m128i b);
PADDD __m128i _mm_add_epi32 (__m128i a, __m128i b);
PADDQ __m128i _mm_add_epi64 (__m128i a, __m128i b);

VPADDB __m256i _mm256_add_epi8 (__m256ia,__m256i b);
VPADDW __m256i _mm256_add_epi16 (__m256i a, __m256i b);
VPADDD __m256i _mm256_add_epi32 (__m256i a, __m256i b);
VPADDQ __m256i _mm256_add_epi64 (__m256i a, __m256i b);
PADDB __m64 _mm_add_pi8(__m64 m1, __m64 m2)
PADDW __m64 _mm_add_pi16(__m64 m1, __m64 m2)
PADDD __m64 _mm_add_pi32(__m64 m1, __m64 m2)
PADDQ __m64 _mm_add_pi64(__m64 m1, __m64 m2)

SIMD Floating-Point Exceptions

None

Other Exceptions

Non-EVEX-encoded instruction, see Exceptions Type 4.

EVEX-encoded VPADDD/Q, see Exceptions Type E4.

EVEX-encoded VPADDB/W, see Exceptions Type E4.nb.

PADDSB/PADDSW—Add Packed Signed Integers with Signed Saturation

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F EC /r[1] PADDSB *mm, mm/m64*	A	V/V	MMX	Add packed signed byte integers from *mm/m64 and mm* and saturate the results.
66 0F EC /r PADDSB *xmm1, xmm2/m128*	A	V/V	SSE2	Add packed signed byte integers from *xmm2/m128* and *xmm1* saturate the results.
NP 0F ED /r[1] PADDSW *mm, mm/m64*	A	V/V	MMX	Add packed signed word integers from *mm/m64 and mm* and saturate the results.
66 0F ED /r PADDSW *xmm1, xmm2/m128*	A	V/V	SSE2	Add packed signed word integers from *xmm2/m128* and *xmm1* and saturate the results.
VEX.NDS.128.66.0F.WIG EC /r VPADDSB *xmm1, xmm2, xmm3/m128*	B	V/V	AVX	Add packed signed byte integers from *xmm3/m128* and *xmm2* saturate the results.
VEX.NDS.128.66.0F.WIG ED /r VPADDSW *xmm1, xmm2, xmm3/m128*	B	V/V	AVX	Add packed signed word integers from *xmm3/m128* and *xmm2* and saturate the results.
VEX.NDS.256.66.0F.WIG EC /r VPADDSB *ymm1, ymm2, ymm3/m256*	B	V/V	AVX2	Add packed signed byte integers from ymm2, and *ymm3/m256* and store the saturated results in *ymm1*.
VEX.NDS.256.66.0F.WIG ED /r VPADDSW *ymm1, ymm2, ymm3/m256*	B	V/V	AVX2	Add packed signed word integers from ymm2, and *ymm3/m256* and store the saturated results in *ymm1*.
EVEX.NDS.128.66.0F.WIG EC /r VPADDSB *xmm1 {k1}{z}, xmm2, xmm3/m128*	C	V/V	AVX512VL AVX512BW	Add packed signed byte integers from xmm2, and *xmm3/m128* and store the saturated results in xmm1 under writemask k1.
EVEX.NDS.256.66.0F.WIG EC /r VPADDSB *ymm1 {k1}{z}, ymm2, ymm3/m256*	C	V/V	AVX512VL AVX512BW	Add packed signed byte integers from ymm2, and *ymm3/m256* and store the saturated results in ymm1 under writemask k1.
EVEX.NDS.512.66.0F.WIG EC /r VPADDSB *zmm1 {k1}{z}, zmm2, zmm3/m512*	C	V/V	AVX512BW	Add packed signed byte integers from zmm2, and *zmm3/m512* and store the saturated results in zmm1 under writemask k1.
EVEX.NDS.128.66.0F.WIG ED /r VPADDSW *xmm1 {k1}{z}, xmm2, xmm3/m128*	C	V/V	AVX512VL AVX512BW	Add packed signed word integers from xmm2, and *xmm3/m128* and store the saturated results in xmm1 under writemask k1.
EVEX.NDS.256.66.0F.WIG ED /r VPADDSW *ymm1 {k1}{z}, ymm2, ymm3/m256*	C	V/V	AVX512VL AVX512BW	Add packed signed word integers from ymm2, and *ymm3/m256* and store the saturated results in ymm1 under writemask k1.
EVEX.NDS.512.66.0F.WIG ED /r VPADDSW *zmm1 {k1}{z}, zmm2, zmm3/m512*	C	V/V	AVX512BW	Add packed signed word integers from zmm2, and *zmm3/m512* and store the saturated results in zmm1 under writemask k1.

NOTES:

1. See note in Section 2.4, "AVX and SSE Instruction Exception Specification" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A* and Section 22.25.3, "Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	NA	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
B	NA	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA
C	Full Vector Mem	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA

Description

Performs a SIMD add of the packed signed integers from the source operand (second operand) and the destination operand (first operand), and stores the packed integer results in the destination operand. See Figure 9-4 in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*, for an illustration of a SIMD operation. Overflow is handled with signed saturation, as described in the following paragraphs.

(V)PADDSB performs a SIMD add of the packed signed integers with saturation from the first source operand and second source operand and stores the packed integer results in the destination operand. When an individual byte result is beyond the range of a signed byte integer (that is, greater than 7FH or less than 80H), the saturated value of 7FH or 80H, respectively, is written to the destination operand.

(V)PADDSW performs a SIMD add of the packed signed word integers with saturation from the first source operand and second source operand and stores the packed integer results in the destination operand. When an individual word result is beyond the range of a signed word integer (that is, greater than 7FFFH or less than 8000H), the saturated value of 7FFFH or 8000H, respectively, is written to the destination operand.

EVEX encoded versions: The first source operand is an ZMM/YMM/XMM register. The second source operand is an ZMM/YMM/XMM register or a memory location. The destination operand is an ZMM/YMM/XMM register.

VEX.256 encoded version: The first source operand is a YMM register. The second source operand is a YMM register or a 256-bit memory location. The destination operand is a YMM register.

VEX.128 encoded version: The first source operand is an XMM register. The second source operand is an XMM register or 128-bit memory location. The destination operand is an XMM register. The upper bits (MAXVL-1:128) of the corresponding register destination are zeroed.

128-bit Legacy SSE version: The first source operand is an XMM register. The second operand can be an XMM register or an 128-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (MAXVL-1:128) of the corresponding register destination are unmodified.

Operation

PADDSB (with 64-bit operands)

```
DEST[7:0] ← SaturateToSignedByte(DEST[7:0] + SRC (7:0]);
(* Repeat add operation for 2nd through 7th bytes *)
DEST[63:56] ← SaturateToSignedByte(DEST[63:56] + SRC[63:56] );
```

PADDSB (with 128-bit operands)

```
DEST[7:0] ←SaturateToSignedByte (DEST[7:0] + SRC[7:0]);
(* Repeat add operation for 2nd through 14th bytes *)
DEST[127:120] ← SaturateToSignedByte (DEST[111:120] + SRC[127:120]);
```

VPADDSB (VEX.128 encoded version)

```
DEST[7:0] ← SaturateToSignedByte (SRC1[7:0] + SRC2[7:0]);
(* Repeat subtract operation for 2nd through 14th bytes *)
DEST[127:120] ← SaturateToSignedByte (SRC1[111:120] + SRC2[127:120]);
DEST[MAXVL-1:128] ← 0
```

VPADDSB (VEX.256 encoded version)

```
DEST[7:0] ← SaturateToSignedByte (SRC1[7:0] + SRC2[7:0]);
(* Repeat add operation for 2nd through 31st bytes *)
DEST[255:248]← SaturateToSignedByte (SRC1[255:248] + SRC2[255:248]);
```

VPADDSB (EVEX encoded versions)
(KL, VL) = (16, 128), (32, 256), (64, 512)

```
FOR j ← 0 TO KL-1
    i ← j * 8
    IF k1[j] OR *no writemask*
        THEN DEST[i+7:i] ← SaturateToSignedByte (SRC1[i+7:i] + SRC2[i+7:i])
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+7:i] remains unchanged*
                ELSE *zeroing-masking*              ; zeroing-masking
                    DEST[i+7:i] = 0
            FI
    FI;
ENDFOR;
DEST[MAXVL-1:VL] ← 0
```

PADDSW (with 64-bit operands)

```
    DEST[15:0] ← SaturateToSignedWord(DEST[15:0] + SRC[15:0] );
    (* Repeat add operation for 2nd and 7th words *)
    DEST[63:48] ← SaturateToSignedWord(DEST[63:48] + SRC[63:48] );
```

PADDSW (with 128-bit operands)

```
    DEST[15:0]  ← SaturateToSignedWord (DEST[15:0] + SRC[15:0]);
    (* Repeat add operation for 2nd through 7th words *)
    DEST[127:112] ← SaturateToSignedWord (DEST[127:112] + SRC[127:112]);
```

VPADDSW (VEX.128 encoded version)

```
    DEST[15:0] ← SaturateToSignedWord (SRC1[15:0] + SRC2[15:0]);
    (* Repeat subtract operation for 2nd through 7th words *)
    DEST[127:112] ← SaturateToSignedWord (SRC1[127:112] + SRC2[127:112]);
    DEST[MAXVL-1:128] ← 0
```

VPADDSW (VEX.256 encoded version)

```
    DEST[15:0] ← SaturateToSignedWord (SRC1[15:0] + SRC2[15:0]);
    (* Repeat add operation for 2nd through 15th words *)
    DEST[255:240] ← SaturateToSignedWord (SRC1[255:240] + SRC2[255:240])
```

VPADDSW (EVEX encoded versions)
(KL, VL) = (8, 128), (16, 256), (32, 512)

```
FOR j ← 0 TO KL-1
    i ← j * 16
    IF k1[j] OR *no writemask*
        THEN DEST[i+15:i] ← SaturateToSignedWord (SRC1[i+15:i] + SRC2[i+15:i])
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+15:i] remains unchanged*
                ELSE *zeroing-masking*              ; zeroing-masking
                    DEST[i+15:i] = 0
            FI
    FI;
ENDFOR;
DEST[MAXVL-1:VL] ← 0
```

Intel C/C++ Compiler Intrinsic Equivalents

PADDSB: __m64 _mm_adds_pi8(__m64 m1, __m64 m2)

(V)PADDSB: __m128i _mm_adds_epi8 (__m128i a, __m128i b)

VPADDSB: __m256i _mm256_adds_epi8 (__m256i a, __m256i b)

PADDSW: __m64 _mm_adds_pi16(__m64 m1, __m64 m2)

(V)PADDSW: __m128i _mm_adds_epi16 (__m128i a, __m128i b)

VPADDSW: __m256i _mm256_adds_epi16 (__m256i a, __m256i b)
VPADDSB__m512i _mm512_adds_epi8 (__m512i a, __m512i b)
VPADDSW__m512i _mm512_adds_epi16 (__m512i a, __m512i b)
VPADDSB__m512i _mm512_mask_adds_epi8 (__m512i s, __mmask64 m, __m512i a, __m512i b)
VPADDSW__m512i _mm512_mask_adds_epi16 (__m512i s, __mmask32 m, __m512i a, __m512i b)
VPADDSB__m512i _mm512_maskz_adds_epi8 (__mmask64 m, __m512i a, __m512i b)
VPADDSW__m512i _mm512_maskz_adds_epi16 (__mmask32 m, __m512i a, __m512i b)
VPADDSB__m256i _mm256_mask_adds_epi8 (__m256i s, __mmask32 m, __m256i a, __m256i b)
VPADDSW__m256i _mm256_mask_adds_epi16 (__m256i s, __mmask16 m, __m256i a, __m256i b)
VPADDSB__m256i _mm256_maskz_adds_epi8 (__mmask32 m, __m256i a, __m256i b)
VPADDSW__m256i _mm256_maskz_adds_epi16 (__mmask16 m, __m256i a, __m256i b)
VPADDSB__m128i _mm_mask_adds_epi8 (__m128i s, __mmask16 m, __m128i a, __m128i b)
VPADDSW__m128i _mm_mask_adds_epi16 (__m128i s, __mmask8 m, __m128i a, __m128i b)
VPADDSB__m128i _mm_maskz_adds_epi8 (__mmask16 m, __m128i a, __m128i b)
VPADDSW__m128i _mm_maskz_adds_epi16 (__mmask8 m, __m128i a, __m128i b)

Flags Affected

None.

SIMD Floating-Point Exceptions

None.

Other Exceptions

Non-EVEX-encoded instruction, see Exceptions Type 4.

EVEX-encoded instruction, see Exceptions Type E4.nb.

PADDUSB/PADDUSW—Add Packed Unsigned Integers with Unsigned Saturation

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F DC /r[1] PADDUSB *mm, mm/m64*	A	V/V	MMX	Add packed unsigned byte integers from *mm/m64 and mm* and saturate the results.
66 0F DC /r PADDUSB *xmm1, xmm2/m128*	A	V/V	SSE2	Add packed unsigned byte integers from *xmm2/m128* and *xmm1* saturate the results.
NP 0F DD /r[1] PADDUSW *mm, mm/m64*	A	V/V	MMX	Add packed unsigned word integers from *mm/m64 and mm* and saturate the results.
66 0F DD /r PADDUSW *xmm1, xmm2/m128*	A	V/V	SSE2	Add packed unsigned word integers from *xmm2/m128* to *xmm1* and saturate the results.
VEX.NDS.128.66.0F.WIG DC /r VPADDUSB *xmm1, xmm2, xmm3/m128*	B	V/V	AVX	Add packed unsigned byte integers from *xmm3/m128* to *xmm2* and saturate the results.
VEX.NDS.128.66.0F.WIG DD /r VPADDUSW *xmm1, xmm2, xmm3/m128*	B	V/V	AVX	Add packed unsigned word integers from *xmm3/m128* to *xmm2* and saturate the results.
VEX.NDS.256.66.0F.WIG DC /r VPADDUSB *ymm1, ymm2, ymm3/m256*	B	V/V	AVX2	Add packed unsigned byte integers from *ymm2*, and *ymm3/m256* and store the saturated results in *ymm1*.
VEX.NDS.256.66.0F.WIG DD /r VPADDUSW *ymm1, ymm2, ymm3/m256*	B	V/V	AVX2	Add packed unsigned word integers from *ymm2*, and *ymm3/m256* and store the saturated results in *ymm1*.
EVEX.NDS.128.66.0F.WIG DC /r VPADDUSB *xmm1 {k1}{z}, xmm2, xmm3/m128*	C	V/V	AVX512VL AVX512BW	Add packed unsigned byte integers from xmm2, and *xmm3/m128* and store the saturated results in xmm1 under writemask k1.
EVEX.NDS.256.66.0F.WIG DC /r VPADDUSB *ymm1 {k1}{z}, ymm2, ymm3/m256*	C	V/V	AVX512VL AVX512BW	Add packed unsigned byte integers from ymm2, and *ymm3/m256* and store the saturated results in ymm1 under writemask k1.
EVEX.NDS.512.66.0F.WIG DC /r VPADDUSB *zmm1 {k1}{z}, zmm2, zmm3/m512*	C	V/V	AVX512BW	Add packed unsigned byte integers from zmm2, and *zmm3/m512* and store the saturated results in zmm1 under writemask k1.
EVEX.NDS.128.66.0F.WIG DD /r VPADDUSW *xmm1 {k1}{z}, xmm2, xmm3/m128*	C	V/V	AVX512VL AVX512BW	Add packed unsigned word integers from xmm2, and *xmm3/m128* and store the saturated results in xmm1 under writemask k1.
EVEX.NDS.256.66.0F.WIG DD /r VPADDUSW *ymm1 {k1}{z}, ymm2, ymm3/m256*	C	V/V	AVX512VL AVX512BW	Add packed unsigned word integers from ymm2, and *ymm3/m256* and store the saturated results in ymm1 under writemask k1.

EVEX.NDS.512.66.0F.WIG DD /r VPADDUSW *zmm1 {k1}{z}, zmm2, zmm3/m512*	C	V/V	AVX512BW	Add packed unsigned word integers from zmm2, and *zmm3/m512* and store the saturated results in zmm1 under writemask k1.

NOTES:

1. See note in Section 2.4, "AVX and SSE Instruction Exception Specification" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A* and Section 22.25.3, "Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	NA	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
B	NA	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA
C	Full Vector Mem	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA

Description

Performs a SIMD add of the packed unsigned integers from the source operand (second operand) and the destination operand (first operand), and stores the packed integer results in the destination operand. See Figure 9-4 in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*, for an illustration of a SIMD operation. Overflow is handled with unsigned saturation, as described in the following paragraphs.

(V)PADDUSB performs a SIMD add of the packed unsigned integers with saturation from the first source operand and second source operand and stores the packed integer results in the destination operand. When an individual byte result is beyond the range of an unsigned byte integer (that is, greater than FFH), the saturated value of FFH is written to the destination operand.

(V)PADDUSW performs a SIMD add of the packed unsigned word integers with saturation from the first source operand and second source operand and stores the packed integer results in the destination operand. When an individual word result is beyond the range of an unsigned word integer (that is, greater than FFFFH), the saturated value of FFFFH is written to the destination operand.

EVEX encoded versions: The first source operand is an ZMM/YMM/XMM register. The second source operand is an ZMM/YMM/XMM register or a 512/256/128-bit memory location. The destination is an ZMM/YMM/XMM register.

VEX.256 encoded version: The first source operand is a YMM register. The second source operand is a YMM register or a 256-bit memory location. The destination operand is a YMM register.

VEX.128 encoded version: The first source operand is an XMM register. The second source operand is an XMM register or 128-bit memory location. The destination operand is an XMM register. The upper bits (MAXVL-1:128) of the corresponding destination register destination are zeroed.

128-bit Legacy SSE version: The first source operand is an XMM register. The second operand can be an XMM register or an 128-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (MAXVL-1:128) of the corresponding register destination are unmodified.

Operation

PADDUSB (with 64-bit operands)

```
DEST[7:0] ← SaturateToUnsignedByte(DEST[7:0] + SRC (7:0] );
(* Repeat add operation for 2nd through 7th bytes *)
DEST[63:56] ← SaturateToUnsignedByte(DEST[63:56] + SRC[63:56]
```

PADDUSB (with 128-bit operands)

```
DEST[7:0] ← SaturateToUnsignedByte (DEST[7:0] + SRC[7:0]);
(* Repeat add operation for 2nd through 14th bytes *)
DEST[127:120] ← SaturateToUnSignedByte (DEST[127:120] + SRC[127:120]);
```

VPADDUSB (VEX.128 encoded version)

```
DEST[7:0] ← SaturateToUnsignedByte (SRC1[7:0] + SRC2[7:0]);
(* Repeat subtract operation for 2nd through 14th bytes *)
DEST[127:120] ← SaturateToUnsignedByte (SRC1[111:120] + SRC2[127:120]);
DEST[MAXVL-1:128] ← 0
```

VPADDUSB (VEX.256 encoded version)

```
DEST[7:0] ← SaturateToUnsignedByte (SRC1[7:0] + SRC2[7:0]);
(* Repeat add operation for 2nd through 31st bytes *)
DEST[255:248]← SaturateToUnsignedByte (SRC1[255:248] + SRC2[255:248]);
```

PADDUSW (with 64-bit operands)

```
DEST[15:0] ← SaturateToUnsignedWord(DEST[15:0] + SRC[15:0] );
(* Repeat add operation for 2nd and 3rd words *)
DEST[63:48] ← SaturateToUnsignedWord(DEST[63:48] + SRC[63:48] );
```

PADDUSW (with 128-bit operands)

```
DEST[15:0] ← SaturateToUnsignedWord (DEST[15:0] + SRC[15:0]);
(* Repeat add operation for 2nd through 7th words *)
DEST[127:112] ← SaturateToUnSignedWord (DEST[127:112] + SRC[127:112]);
```

VPADDUSW (VEX.128 encoded version)

```
DEST[15:0] ← SaturateToUnsignedWord (SRC1[15:0] + SRC2[15:0]);
(* Repeat subtract operation for 2nd through 7th words *)
DEST[127:112] ← SaturateToUnsignedWord (SRC1[127:112] + SRC2[127:112]);
DEST[MAXVL-1:128] ← 0
```

VPADDUSW (VEX.256 encoded version)

```
DEST[15:0] ← SaturateToUnsignedWord (SRC1[15:0] + SRC2[15:0]);
(* Repeat add operation for 2nd through 15th words *)
DEST[255:240] ← SaturateToUnsignedWord (SRC1[255:240] + SRC2[255:240])
```

VPADDUSB (EVEX encoded versions)

```
(KL, VL) = (16, 128), (32, 256), (64, 512)

FOR j ← 0 TO KL-1
    i ← j * 8
    IF k1[j] OR *no writemask*
        THEN DEST[i+7:i] ← SaturateToUnsignedByte (SRC1[i+7:i] + SRC2[i+7:i])
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+7:i] remains unchanged*
                ELSE *zeroing-masking*              ; zeroing-masking
                    DEST[i+7:i] = 0
            FI
    FI;
ENDFOR;
DEST[MAXVL-1:VL] ← 0
```

VPADDUSW (EVEX encoded versions)
(KL, VL) = (8, 128), (16, 256), (32, 512)

```
FOR j ← 0 TO KL-1
    i ← j * 16
    IF k1[j] OR *no writemask*
        THEN DEST[i+15:i] ← SaturateToUnsignedWord (SRC1[i+15:i] + SRC2[i+15:i])
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+15:i] remains unchanged*
                ELSE *zeroing-masking*              ; zeroing-masking
                    DEST[i+15:i] = 0
            FI
    FI;
ENDFOR;
DEST[MAXVL-1:VL] ← 0
```

Intel C/C++ Compiler Intrinsic Equivalents

PADDUSB: __m64 _mm_adds_pu8(__m64 m1, __m64 m2)

PADDUSW: __m64 _mm_adds_pu16(__m64 m1, __m64 m2)

(V)PADDUSB: __m128i _mm_adds_epu8 (__m128i a, __m128i b)

(V)PADDUSW: __m128i _mm_adds_epu16 (__m128i a, __m128i b)

VPADDUSB: __m256i _mm256_adds_epu8 (__m256i a, __m256i b)

VPADDUSW: __m256i _mm256_adds_epu16 (__m256i a, __m256i b)
VPADDUSB__m512i _mm512_adds_epu8 (__m512i a, __m512i b)
VPADDUSW__m512i _mm512_adds_epu16 (__m512i a, __m512i b)
VPADDUSB__m512i _mm512_mask_adds_epu8 (__m512i s, __mmask64 m, __m512i a, __m512i b)
VPADDUSW__m512i _mm512_mask_adds_epu16 (__m512i s, __mmask32 m, __m512i a, __m512i b)
VPADDUSB__m512i _mm512_maskz_adds_epu8 (__mmask64 m, __m512i a, __m512i b)
VPADDUSW__m512i _mm512_maskz_adds_epu16 (__mmask32 m, __m512i a, __m512i b)
VPADDUSB__m256i _mm256_mask_adds_epu8 (__m256i s, __mmask32 m, __m256i a, __m256i b)
VPADDUSW__m256i _mm256_mask_adds_epu16 (__m256i s, __mmask16 m, __m256i a, __m256i b)
VPADDUSB__m256i _mm256_maskz_adds_epu8 (__mmask32 m, __m256i a, __m256i b)
VPADDUSW__m256i _mm256_maskz_adds_epu16 (__mmask16 m, __m256i a, __m256i b)
VPADDUSB__m128i _mm_mask_adds_epu8 (__m128i s, __mmask16 m, __m128i a, __m128i b)
VPADDUSW__m128i _mm_mask_adds_epu16 (__m128i s, __mmask8 m, __m128i a, __m128i b)
VPADDUSB__m128i _mm_maskz_adds_epu8 (__mmask16 m, __m128i a, __m128i b)
VPADDUSW__m128i _mm_maskz_adds_epu16 (__mmask8 m, __m128i a, __m128i b)

Flags Affected

None.

Numeric Exceptions

None.

Other Exceptions

Non-EVEX-encoded instruction, see Exceptions Type 4.

EVEX-encoded instruction, see Exceptions Type E4.nb.

PALIGNR — Packed Align Right

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F 3A 0F /r ib[1] PALIGNR *mm1, mm2/m64, imm8*	RMI	V/V	SSSE3	Concatenate destination and source operands, extract byte-aligned result shifted to the right by constant value in *imm8* into *mm1*.
66 0F 3A 0F /r ib PALIGNR *xmm1, xmm2/m128, imm8*	RMI	V/V	SSSE3	Concatenate destination and source operands, extract byte-aligned result shifted to the right by constant value in *imm8* into *xmm1.*
VEX.NDS.128.66.0F3A.WIG 0F /r ib VPALIGNR *xmm1, xmm2, xmm3/m128, imm8*	RVMI	V/V	AVX	Concatenate *xmm2* and *xmm3/m128*, extract byte aligned result shifted to the right by constant value in *imm8* and result is stored in *xmm1*.
VEX.NDS.256.66.0F3A.WIG 0F /r ib VPALIGNR *ymm1, ymm2, ymm3/m256, imm8*	RVMI	V/V	AVX2	Concatenate pairs of 16 bytes in ymm2 and *ymm3/m256* into 32-byte intermediate result, extract byte-aligned, 16-byte result shifted to the right by constant values in *imm8* from each intermediate result, and two 16-byte results are stored in *ymm1*.
EVEX.NDS.128.66.0F3A.WIG 0F /r ib VPALIGNR xmm1 {k1}{z}, xmm2, xmm3/m128, imm8	FVMI	V/V	AVX512VL AVX512BW	Concatenate xmm2 and xmm3/m128 into a 32-byte intermediate result, extract byte aligned result shifted to the right by constant value in imm8 and result is stored in xmm1.
EVEX.NDS.256.66.0F3A.WIG 0F /r ib VPALIGNR ymm1 {k1}{z}, ymm2, ymm3/m256, imm8	FVMI	V/V	AVX512VL AVX512BW	Concatenate pairs of 16 bytes in ymm2 and ymm3/m256 into 32-byte intermediate result, extract byte-aligned, 16-byte result shifted to the right by constant values in imm8 from each intermediate result, and two 16-byte results are stored in ymm1.
EVEX.NDS.512.66.0F3A.WIG 0F /r ib VPALIGNR zmm1 {k1}{z}, zmm2, zmm3/m512, imm8	FVMI	V/V	AVX512BW	Concatenate pairs of 16 bytes in zmm2 and zmm3/m512 into 32-byte intermediate result, extract byte-aligned, 16-byte result shifted to the right by constant values in imm8 from each intermediate result, and four 16-byte results are stored in zmm1.

NOTES:

1. See note in Section 2.4, "AVX and SSE Instruction Exception Specification" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A* and Section 22.25.3, "Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RMI	ModRM:reg (r, w)	ModRM:r/m (r)	imm8	NA
RVMI	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	imm8
FVMI	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	imm8

Description

(V)PALIGNR concatenates the destination operand (the first operand) and the source operand (the second operand) into an intermediate composite, shifts the composite at byte granularity to the right by a constant immediate, and extracts the right-aligned result into the destination. The first and the second operands can be an MMX,

XMM or a YMM register. The immediate value is considered unsigned. Immediate shift counts larger than the 2L (i.e. 32 for 128-bit operands, or 16 for 64-bit operands) produce a zero result. Both operands can be MMX registers, XMM registers or YMM registers. When the source operand is a 128-bit memory operand, the operand must be aligned on a 16-byte boundary or a general-protection exception (#GP) will be generated.

In 64-bit mode and not encoded by VEX/EVEX prefix, use the REX prefix to access additional registers.

128-bit Legacy SSE version: Bits (MAXVL-1:128) of the corresponding YMM destination register remain unchanged.

EVEX.512 encoded version: The first source operand is a ZMM register and contains four 16-byte blocks. The second source operand is a ZMM register or a 512-bit memory location containing four 16-byte block. The destination operand is a ZMM register and contain four 16-byte results. The imm8[7:0] is the common shift count

used for each of the four successive 16-byte block sources. The low 16-byte block of the two source operands produce the low 16-byte result of the destination operand, the high 16-byte block of the two source operands produce the high 16-byte result of the destination operand and so on for the blocks in the middle.

VEX.256 and EVEX.256 encoded versions: The first source operand is a YMM register and contains two 16-byte blocks. The second source operand is a YMM register or a 256-bit memory location containing two 16-byte block. The destination operand is a YMM register and contain two 16-byte results. The imm8[7:0] is the common shift count used for the two lower 16-byte block sources and the two upper 16-byte block sources. The low 16-byte block of the two source operands produce the low 16-byte result of the destination operand, the high 16-byte block of the two source operands produce the high 16-byte result of the destination operand. The upper bits (MAXVL-1:256) of the corresponding ZMM register destination are zeroed.

VEX.128 and EVEX.128 encoded versions: The first source operand is an XMM register. The second source operand is an XMM register or 128-bit memory location. The destination operand is an XMM register. The upper bits (MAXVL-1:128) of the corresponding ZMM register destination are zeroed.

Concatenation is done with 128-bit data in the first and second source operand for both 128-bit and 256-bit instructions. The high 128-bits of the intermediate composite 256-bit result came from the 128-bit data from the first source operand; the low 128-bits of the intermediate result came from the 128-bit data of the second source operand.

Note: VEX.L must be 0, otherwise the instruction will #UD.

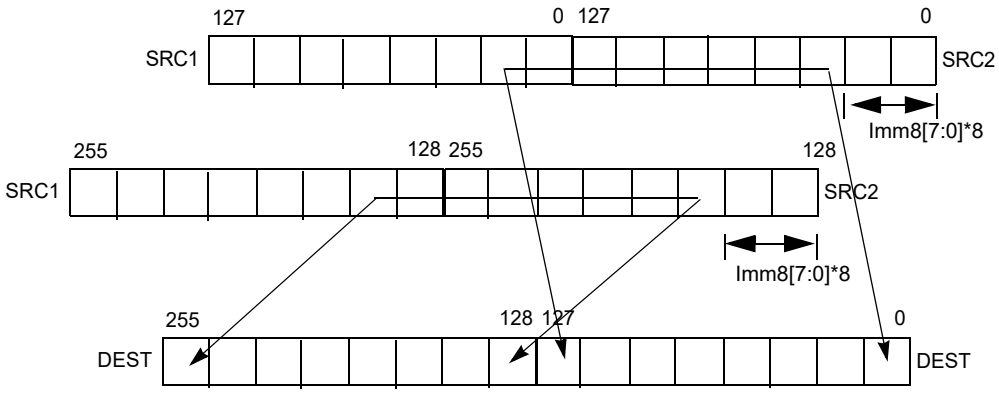


Figure 4-7. 256-bit VPALIGN Instruction Operation

Operation

PALIGNR (with 64-bit operands)

```
temp1[127:0] = CONCATENATE(DEST,SRC)>>(imm8*8)
DEST[63:0] = temp1[63:0]
```

PALIGNR (with 128-bit operands)

```
temp1[255:0] ← ((DEST[127:0] << 128) OR SRC[127:0])>>(imm8*8);
DEST[127:0] ← temp1[127:0]
DEST[MAXVL-1:128] (Unmodified)
```

VPALIGNR (VEX.128 encoded version)

```
temp1[255:0] ← ((SRC1[127:0] << 128) OR SRC2[127:0])>>(imm8*8);
DEST[127:0] ← temp1[127:0]
DEST[MAXVL-1:128] ← 0
```

VPALIGNR (VEX.256 encoded version)

```
temp1[255:0] ← ((SRC1[127:0] << 128) OR SRC2[127:0])>>(imm8[7:0]*8);
DEST[127:0] ← temp1[127:0]
temp1[255:0] ← ((SRC1[255:128] << 128) OR SRC2[255:128])>>(imm8[7:0]*8);
DEST[MAXVL-1:128] ← temp1[127:0]
```

VPALIGNR (EVEX encoded versions)

```
(KL, VL) = (16, 128), (32, 256), (64, 512)

FOR l ← 0 TO VL-1 with increments of 128
    temp1[255:0] ← ((SRC1[l+127:l] << 128) OR SRC2[l+127:l])>>(imm8[7:0]*8);
    TMP_DEST[l+127:l] ← temp1[127:0]
ENDFOR;

FOR j ← 0 TO KL-1
    i ← j * 8
    IF k1[j] OR *no writemask*
        THEN DEST[i+7:i] ← TMP_DEST[i+7:i]
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+7:i] remains unchanged*
                ELSE *zeroing-masking*              ; zeroing-masking
                    DEST[i+7:i] = 0
            FI
    FI;
ENDFOR;
DEST[MAXVL-1:VL] ← 0
```

Intel C/C++ Compiler Intrinsic Equivalents

PALIGNR: __m64 _mm_alignr_pi8 (__m64 a, __m64 b, int n)

(V)PALIGNR: __m128i _mm_alignr_epi8 (__m128i a, __m128i b, int n)

VPALIGNR: __m256i _mm256_alignr_epi8 (__m256i a, __m256i b, const int n)

VPALIGNR __m512i _mm512_alignr_epi8 (__m512i a, __m512i b, const int n)
VPALIGNR __m512i _mm512_mask_alignr_epi8 (__m512i s, __mmask64 m, __m512i a, __m512i b, const int n)
VPALIGNR __m512i _mm512_maskz_alignr_epi8 (__mmask64 m, __m512i a, __m512i b, const int n)
VPALIGNR __m256i _mm256_mask_alignr_epi8 (__m256i s, __mmask32 m, __m256i a, __m256i b, const int n)
VPALIGNR __m256i _mm256_maskz_alignr_epi8 (__mmask32 m, __m256i a, __m256i b, const int n)
VPALIGNR __m128i _mm_mask_alignr_epi8 (__m128i s, __mmask16 m, __m128i a, __m128i b, const int n)
VPALIGNR __m128i _mm_maskz_alignr_epi8 (__mmask16 m, __m128i a, __m128i b, const int n)

SIMD Floating-Point Exceptions

None.

Other Exceptions

Non-EVEX-encoded instruction, see Exceptions Type 4.

EVEX-encoded instruction, see Exceptions Type E4NF.nb.

PAND—Logical AND

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F DB /r[1] PAND *mm, mm/m64*	A	V/V	MMX	Bitwise AND *mm/m64* and *mm*.
66 0F DB /r PAND *xmm1, xmm2/m128*	A	V/V	SSE2	Bitwise AND of *xmm2/m128* and *xmm1*.
VEX.NDS.128.66.0F.WIG DB /r VPAND *xmm1, xmm2, xmm3/m128*	B	V/V	AVX	Bitwise AND of *xmm3/m128* and *xmm*.
VEX.NDS.256.66.0F.WIG DB /r VPAND *ymm1, ymm2, ymm3/.m256*	B	V/V	AVX2	Bitwise AND of *ymm2*, and *ymm3/m256* and store result in *ymm1*.
EVEX.NDS.128.66.0F.W0 DB /r VPANDD xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst	C	V/V	AVX512VL AVX512F	Bitwise AND of packed doubleword integers in xmm2 and xmm3/m128/m32bcst and store result in xmm1 using writemask k1.
EVEX.NDS.256.66.0F.W0 DB /r VPANDD ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst	C	V/V	AVX512VL AVX512F	Bitwise AND of packed doubleword integers in ymm2 and ymm3/m256/m32bcst and store result in ymm1 using writemask k1.
EVEX.NDS.512.66.0F.W0 DB /r VPANDD zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst	C	V/V	AVX512F	Bitwise AND of packed doubleword integers in zmm2 and zmm3/m512/m32bcst and store result in zmm1 using writemask k1.
EVEX.NDS.128.66.0F.W1 DB /r VPANDQ xmm1 {k1}{z}, xmm2, xmm3/m128/m64bcst	C	V/V	AVX512VL AVX512F	Bitwise AND of packed quadword integers in xmm2 and xmm3/m128/m64bcst and store result in xmm1 using writemask k1.
EVEX.NDS.256.66.0F.W1 DB /r VPANDQ ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst	C	V/V	AVX512VL AVX512F	Bitwise AND of packed quadword integers in ymm2 and ymm3/m256/m64bcst and store result in ymm1 using writemask k1.
EVEX.NDS.512.66.0F.W1 DB /r VPANDQ zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst	C	V/V	AVX512F	Bitwise AND of packed quadword integers in zmm2 and zmm3/m512/m64bcst and store result in zmm1 using writemask k1.

NOTES:

1. See note in Section 2.4, "AVX and SSE Instruction Exception Specification" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A* and Section 22.25.3, "Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	NA	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
B	NA	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA
C	Full Vector	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA

Description

Performs a bitwise logical AND operation on the first source operand and second source operand and stores the result in the destination operand. Each bit of the result is set to 1 if the corresponding bits of the first and second operands are 1, otherwise it is set to 0.

In 64-bit mode and not encoded with VEX/EVEX, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

Legacy SSE instructions: The source operand can be an MMX technology register or a 64-bit memory location. The destination operand can be an MMX technology register.

128-bit Legacy SSE version: The first source operand is an XMM register. The second operand can be an XMM register or an 128-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (MAXVL-1:128) of the corresponding ZMM register destination are unmodified.

EVEX encoded versions: The first source operand is a ZMM/YMM/XMM register. The second source operand can be a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 32/64-bit memory location. The destination operand is a ZMM/YMM/XMM register conditionally updated with writemask k1 at 32/64-bit granularity.

VEX.256 encoded versions: The first source operand is a YMM register. The second source operand is a YMM register or a 256-bit memory location. The destination operand is a YMM register. The upper bits (MAXVL-1:256) of the corresponding ZMM register destination are zeroed.

VEX.128 encoded versions: The first source operand is an XMM register. The second source operand is an XMM register or 128-bit memory location. The destination operand is an XMM register. The upper bits (MAXVL-1:128) of the corresponding ZMM register destination are zeroed.

Operation

PAND (64-bit operand)

```
DEST ← DEST AND SRC
```

PAND (128-bit Legacy SSE version)

```
DEST ← DEST AND SRC
DEST[MAXVL-1:128] (Unmodified)
```

VPAND (VEX.128 encoded version)

```
DEST ← SRC1 AND SRC2
DEST[MAXVL-1:128] ← 0
```

VPAND (VEX.256 encoded instruction)

```
DEST[255:0] ← (SRC1[255:0] AND SRC2[255:0])
DEST[MAXVL-1:256] ← 0
```

VPANDD (EVEX encoded versions)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j ← 0 TO KL-1
    i ← j * 32
    IF k1[j] OR *no writemask*
        THEN
            IF (EVEX.b = 1) AND (SRC2 *is memory*)
                THEN DEST[i+31:i] ← SRC1[i+31:i] BITWISE AND SRC2[31:0]
                ELSE DEST[i+31:i] ← SRC1[i+31:i] BITWISE AND SRC2[i+31:i]
            FI;
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE                                ; zeroing-masking
                    DEST[i+31:i] ← 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0
```

VPANDQ (EVEX encoded versions)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j ← 0 TO KL-1
    i ← j * 64
    IF k1[j] OR *no writemask*
        THEN
            IF (EVEX.b = 1) AND (SRC2 *is memory*)
                THEN DEST[i+63:i] ← SRC1[i+63:i] BITWISE AND SRC2[63:0]
                ELSE DEST[i+63:i] ← SRC1[i+63:i] BITWISE AND SRC2[i+63:i]
            FI;
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE                                ; zeroing-masking
                    DEST[i+63:i] ← 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0
```

Intel C/C++ Compiler Intrinsic Equivalents

VPANDD __m512i _mm512_and_epi32(__m512i a, __m512i b);
VPANDD __m512i _mm512_mask_and_epi32(__m512i s, __mmask16 k, __m512i a, __m512i b);
VPANDD __m512i _mm512_maskz_and_epi32(__mmask16 k, __m512i a, __m512i b);
VPANDQ __m512i _mm512_and_epi64(__m512i a, __m512i b);
VPANDQ __m512i _mm512_mask_and_epi64(__m512i s, __mmask8 k, __m512i a, __m512i b);
VPANDQ __m512i _mm512_maskz_and_epi64(__mmask8 k, __m512i a, __m512i b);
VPANDND __m256i _mm256_mask_and_epi32(__m256i s, __mmask8 k, __m256i a, __m256i b);
VPANDND __m256i _mm256_maskz_and_epi32(__mmask8 k, __m256i a, __m256i b);
VPANDND __m128i _mm_mask_and_epi32(__m128i s, __mmask8 k, __m128i a, __m128i b);
VPANDND __m128i _mm_maskz_and_epi32(__mmask8 k, __m128i a, __m128i b);
VPANDNQ __m256i _mm256_mask_and_epi64(__m256i s, __mmask8 k, __m256i a, __m256i b);
VPANDNQ __m256i _mm256_maskz_and_epi64(__mmask8 k, __m256i a, __m256i b);
VPANDNQ __m128i _mm_mask_and_epi64(__m128i s, __mmask8 k, __m128i a, __m128i b);
VPANDNQ __m128i _mm_maskz_and_epi64(__mmask8 k, __m128i a, __m128i b);
PAND: __m64 _mm_and_si64 (__m64 m1, __m64 m2)
(V)PAND:__m128i _mm_and_si128 (__m128i a, __m128i b)
VPAND: __m256i _mm256_and_si256 (__m256i a, __m256i b)

Flags Affected

None.

Numeric Exceptions

None.

Other Exceptions

Non-EVEX-encoded instruction, see Exceptions Type 4.

EVEX-encoded instruction, see Exceptions Type E4.

PANDN—Logical AND NOT

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F DF /r[1] PANDN *mm, mm/m64*	A	V/V	MMX	Bitwise AND NOT of *mm/m64* and *mm*.
66 0F DF /r PANDN *xmm1, xmm2/m128*	A	V/V	SSE2	Bitwise AND NOT of *xmm2/m128* and *xmm1*.
VEX.NDS.128.66.0F.WIG DF /r VPANDN *xmm1, xmm2, xmm3/m128*	B	V/V	AVX	Bitwise AND NOT of *xmm3/m128* and *xmm2*.
VEX.NDS.256.66.0F.WIG DF /r VPANDN *ymm1, ymm2, ymm3/m256*	B	V/V	AVX2	Bitwise AND NOT of *ymm2*, and *ymm3/m256* and store result in *ymm1*.
EVEX.NDS.128.66.0F.W0 DF /r VPANDND xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst	C	V/V	AVX512VL AVX512F	Bitwise AND NOT of packed doubleword integers in xmm2 and xmm3/m128/m32bcst and store result in xmm1 using writemask k1.
EVEX.NDS.256.66.0F.W0 DF /r VPANDND ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst	C	V/V	AVX512VL AVX512F	Bitwise AND NOT of packed doubleword integers in ymm2 and ymm3/m256/m32bcst and store result in ymm1 using writemask k1.
EVEX.NDS.512.66.0F.W0 DF /r VPANDND zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst	C	V/V	AVX512F	Bitwise AND NOT of packed doubleword integers in zmm2 and zmm3/m512/m32bcst and store result in zmm1 using writemask k1.
EVEX.NDS.128.66.0F.W1 DF /r VPANDNQ xmm1 {k1}{z}, xmm2, xmm3/m128/m64bcst	C	V/V	AVX512VL AVX512F	Bitwise AND NOT of packed quadword integers in xmm2 and xmm3/m128/m64bcst and store result in xmm1 using writemask k1.
EVEX.NDS.256.66.0F.W1 DF /r VPANDNQ ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst	C	V/V	AVX512VL AVX512F	Bitwise AND NOT of packed quadword integers in ymm2 and ymm3/m256/m64bcst and store result in ymm1 using writemask k1.
EVEX.NDS.512.66.0F.W1 DF /r VPANDNQ zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst	C	V/V	AVX512F	Bitwise AND NOT of packed quadword integers in zmm2 and zmm3/m512/m64bcst and store result in zmm1 using writemask k1.

NOTES:

1. See note in Section 2.4, "AVX and SSE Instruction Exception Specification" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A* and Section 22.25.3, "Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	NA	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
B	NA	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA
C	Full Vector	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA

Description

Performs a bitwise logical NOT operation on the first source operand, then performs bitwise AND with second source operand and stores the result in the destination operand. Each bit of the result is set to 1 if the corresponding bit in the first operand is 0 and the corresponding bit in the second operand is 1, otherwise it is set to 0.

In 64-bit mode and not encoded with VEX/EVEX, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

Legacy SSE instructions: The source operand can be an MMX technology register or a 64-bit memory location. The destination operand can be an MMX technology register.

128-bit Legacy SSE version: The first source operand is an XMM register. The second operand can be an XMM register or an 128-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (MAXVL-1:128) of the corresponding ZMM register destination are unmodified.

EVEX encoded versions: The first source operand is a ZMM/YMM/XMM register. The second source operand can be a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 32/64-bit memory location. The destination operand is a ZMM/YMM/XMM register conditionally updated with writemask k1 at 32/64-bit granularity.

VEX.256 encoded versions: The first source operand is a YMM register. The second source operand is a YMM register or a 256-bit memory location. The destination operand is a YMM register. The upper bits (MAXVL-1:256) of the corresponding ZMM register destination are zeroed.

VEX.128 encoded versions: The first source operand is an XMM register. The second source operand is an XMM register or 128-bit memory location. The destination operand is an XMM register. The upper bits (MAXVL-1:128) of the corresponding ZMM register destination are zeroed.

Operation

PANDN (64-bit operand)

```
DEST ← NOT(DEST) AND SRC
```

PANDN (128-bit Legacy SSE version)

```
DEST ← NOT(DEST) AND SRC
DEST[MAXVL-1:128] (Unmodified)
```

VPANDN (VEX.128 encoded version)

```
DEST ← NOT(SRC1) AND SRC2
DEST[MAXVL-1:128] ← 0
```

VPANDN (VEX.256 encoded instruction)

```
DEST[255:0] ← ((NOT SRC1[255:0]) AND SRC2[255:0])
DEST[MAXVL-1:256] ← 0
```

VPANDND (EVEX encoded versions)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j ← 0 TO KL-1
    i ← j * 32
    IF k1[j] OR *no writemask*
        THEN
            IF (EVEX.b = 1) AND (SRC2 *is memory*)
                THEN DEST[i+31:i] ← ((NOT SRC1[i+31:i]) AND SRC2[31:0])
                ELSE DEST[i+31:i] ← ((NOT SRC1[i+31:i]) AND SRC2[i+31:i])
            FI;
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE                                ; zeroing-masking
                    DEST[i+31:i] ← 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0
```

VPANDNQ (EVEX encoded versions)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j ← 0 TO KL-1
    i ← j * 64
    IF k1[j] OR *no writemask*
        THEN
            IF (EVEX.b = 1) AND (SRC2 *is memory*)
                THEN DEST[i+63:i] ← ((NOT SRC1[i+63:i]) AND SRC2[63:0])
                ELSE DEST[i+63:i] ← ((NOT SRC1[i+63:i]) AND SRC2[i+63:i])
            FI;
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE                                ; zeroing-masking
                    DEST[i+63:i] ← 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0
```

Intel C/C++ Compiler Intrinsic Equivalents

VPANDND __m512i _mm512_andnot_epi32(__m512i a, __m512i b);
VPANDND __m512i _mm512_mask_andnot_epi32(__m512i s, __mmask16 k, __m512i a, __m512i b);
VPANDND __m512i _mm512_maskz_andnot_epi32(__mmask16 k, __m512i a, __m512i b);
VPANDND __m256i _mm256_mask_andnot_epi32(__m256i s, __mmask8 k, __m256i a, __m256i b);
VPANDND __m256i _mm256_maskz_andnot_epi32(__mmask8 k, __m256i a, __m256i b);
VPANDND __m128i _mm_mask_andnot_epi32(__m128i s, __mmask8 k, __m128i a, __m128i b);
VPANDND __m128i _mm_maskz_andnot_epi32(__mmask8 k, __m128i a, __m128i b);
VPANDNQ __m512i _mm512_andnot_epi64(__m512i a, __m512i b);
VPANDNQ __m512i _mm512_mask_andnot_epi64(__m512i s, __mmask8 k, __m512i a, __m512i b);
VPANDNQ __m512i _mm512_maskz_andnot_epi64(__mmask8 k, __m512i a, __m512i b);
VPANDNQ __m256i _mm256_mask_andnot_epi64(__m256i s, __mmask8 k, __m256i a, __m256i b);
VPANDNQ __m256i _mm256_maskz_andnot_epi64(__mmask8 k, __m256i a, __m256i b);
VPANDNQ __m128i _mm_mask_andnot_epi64(__m128i s, __mmask8 k, __m128i a, __m128i b);
VPANDNQ __m128i _mm_maskz_andnot_epi64(__mmask8 k, __m128i a, __m128i b);
PANDN: __m64 _mm_andnot_si64 (__m64 m1, __m64 m2)
(V)PANDN:__m128i _mm_andnot_si128 (__m128i a, __m128i b)
VPANDN: __m256i _mm256_andnot_si256 (__m256i a, __m256i b)

Flags Affected

None.

Numeric Exceptions

None.

Other Exceptions

Non-EVEX-encoded instruction, see Exceptions Type 4.

EVEX-encoded instruction, see Exceptions Type E4.

PAUSE—Spin Loop Hint

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
F3 90	PAUSE	ZO	Valid	Valid	Gives hint to processor that improves performance of spin-wait loops.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
ZO	NA	NA	NA	NA

Description

Improves the performance of spin-wait loops. When executing a "spin-wait loop," processors will suffer a severe performance penalty when exiting the loop because it detects a possible memory order violation. The PAUSE instruction provides a hint to the processor that the code sequence is a spin-wait loop. The processor uses this hint to avoid the memory order violation in most situations, which greatly improves processor performance. For this reason, it is recommended that a PAUSE instruction be placed in all spin-wait loops.

An additional function of the PAUSE instruction is to reduce the power consumed by a processor while executing a spin loop. A processor can execute a spin-wait loop extremely quickly, causing the processor to consume a lot of power while it waits for the resource it is spinning on to become available. Inserting a pause instruction in a spin-wait loop greatly reduces the processor's power consumption.

This instruction was introduced in the Pentium 4 processors, but is backward compatible with all IA-32 processors. In earlier IA-32 processors, the PAUSE instruction operates like a NOP instruction. The Pentium 4 and Intel Xeon processors implement the PAUSE instruction as a delay. The delay is finite and can be zero for some processors. This instruction does not change the architectural state of the processor (that is, it performs essentially a delaying no-op operation).

This instruction's operation is the same in non-64-bit modes and 64-bit mode.

Operation

```
Execute_Next_Instruction(DELAY);
```

Numeric Exceptions

None.

Exceptions (All Operating Modes)

#UD	If the LOCK prefix is used.

PAVGB/PAVGW—Average Packed Integers

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F E0 /r[1] PAVGB *mm1, mm2/m64*	A	V/V	SSE	Average packed unsigned byte integers from *mm2/m64* and *mm1* with rounding.
66 0F E0, /r PAVGB *xmm1, xmm2/m128*	A	V/V	SSE2	Average packed unsigned byte integers from *xmm2/m128* and *xmm1* with rounding.
NP 0F E3 /r[1] PAVGW *mm1, mm2/m64*	A	V/V	SSE	Average packed unsigned word integers from mm2/m64 and *mm1* with rounding.
66 0F E3 /r PAVGW *xmm1, xmm2/m128*	A	V/V	SSE2	Average packed unsigned word integers from *xmm2/m128* and *xmm1* with rounding.
VEX.NDS.128.66.0F.WIG E0 /r VPAVGB *xmm1, xmm2, xmm3/m128*	B	V/V	AVX	Average packed unsigned byte integers from *xmm3/m128* and *xmm2* with rounding.
VEX.NDS.128.66.0F.WIG E3 /r VPAVGW *xmm1, xmm2, xmm3/m128*	B	V/V	AVX	Average packed unsigned word integers from *xmm3/m128* and *xmm2* with rounding.
VEX.NDS.256.66.0F.WIG E0 /r VPAVGB *ymm1, ymm2, ymm3/m256*	B	V/V	AVX2	Average packed unsigned byte integers from ymm2, and *ymm3/m256* with rounding and store to *ymm1*.
VEX.NDS.256.66.0F.WIG E3 /r VPAVGW *ymm1, ymm2, ymm3/m256*	B	V/V	AVX2	Average packed unsigned word integers from *ymm2, ymm3/m256* with rounding to *ymm1*.
EVEX.NDS.128.66.0F.WIG E0 /r VPAVGB *xmm1* {k1}{z}, *xmm2, xmm3/m128*	C	V/V	AVX512VL AVX512BW	Average packed unsigned byte integers from xmm2, and *xmm3/m128* with rounding and store to xmm1 under writemask k1.
EVEX.NDS.256.66.0F.WIG E0 /r VPAVGB *ymm1* {k1}{z}, *ymm2, ymm3/m256*	C	V/V	AVX512VL AVX512BW	Average packed unsigned byte integers from ymm2, and *ymm3/m256* with rounding and store to ymm1 under writemask k1.
EVEX.NDS.512.66.0F.WIG E0 /r VPAVGB *zmm1* {k1}{z}, *zmm2, zmm3/m512*	C	V/V	AVX512BW	Average packed unsigned byte integers from zmm2, and *zmm3/m512* with rounding and store to zmm1 under writemask k1.
EVEX.NDS.128.66.0F.WIG E3 /r VPAVGW *xmm1* {k1}{z}, *xmm2, xmm3/m128*	C	V/V	AVX512VL AVX512BW	Average packed unsigned word integers from xmm2, *xmm3/m128* with rounding to xmm1 under writemask k1.
EVEX.NDS.256.66.0F.WIG E3 /r VPAVGW *ymm1* {k1}{z}, *ymm2, ymm3/m256*	C	V/V	AVX512VL AVX512BW	Average packed unsigned word integers from ymm2, *ymm3/m256* with rounding to ymm1 under writemask k1.
EVEX.NDS.512.66.0F.WIG E3 /r VPAVGW *zmm1* {k1}{z}, *zmm2, zmm3/m512*	C	V/V	AVX512BW	Average packed unsigned word integers from zmm2, *zmm3/m512* with rounding to zmm1 under writemask k1.

NOTES:

1. See note in Section 2.4, "AVX and SSE Instruction Exception Specification" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A* and Section 22.25.3, "Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	NA	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
B	NA	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA
C	Full Vector Mem	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA

Description

Performs a SIMD average of the packed unsigned integers from the source operand (second operand) and the destination operand (first operand), and stores the results in the destination operand. For each corresponding pair of data elements in the first and second operands, the elements are added together, a 1 is added to the temporary sum, and that result is shifted right one bit position.

The (V)PAVGB instruction operates on packed unsigned bytes and the (V)PAVGW instruction operates on packed unsigned words.

In 64-bit mode and not encoded with VEX/EVEX, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

Legacy SSE instructions: The source operand can be an MMX technology register or a 64-bit memory location. The destination operand can be an MMX technology register.

128-bit Legacy SSE version: The first source operand is an XMM register. The second operand can be an XMM register or an 128-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (MAXVL-1:128) of the corresponding register destination are unmodified.

EVEX.512 encoded version: The first source operand is a ZMM register. The second source operand is a ZMM register or a 512-bit memory location. The destination operand is a ZMM register.

VEX.256 and EVEX.256 encoded versions: The first source operand is a YMM register. The second source operand is a YMM register or a 256-bit memory location. The destination operand is a YMM register.

VEX.128 and EVEX.128 encoded versions: The first source operand is an XMM register. The second source operand is an XMM register or 128-bit memory location. The destination operand is an XMM register. The upper bits (MAXVL-1:128) of the corresponding register destination are zeroed.

Operation

PAVGB (with 64-bit operands)

```
DEST[7:0] ← (SRC[7:0] + DEST[7:0] + 1) >> 1; (* Temp sum before shifting is 9 bits *)
(* Repeat operation performed for bytes 2 through 6 *)
DEST[63:56] ← (SRC[63:56] + DEST[63:56] + 1) >> 1;
```

PAVGW (with 64-bit operands)

```
DEST[15:0] ← (SRC[15:0] + DEST[15:0] + 1) >> 1; (* Temp sum before shifting is 17 bits *)
(* Repeat operation performed for words 2 and 3 *)
DEST[63:48] ← (SRC[63:48] + DEST[63:48] + 1) >> 1;
```

PAVGB (with 128-bit operands)

```
DEST[7:0] ← (SRC[7:0] + DEST[7:0] + 1) >> 1; (* Temp sum before shifting is 9 bits *)
(* Repeat operation performed for bytes 2 through 14 *)
DEST[127:120] ← (SRC[127:120] + DEST[127:120] + 1) >> 1;
```

PAVGW (with 128-bit operands)

```
DEST[15:0] ← (SRC[15:0] + DEST[15:0] + 1) >> 1; (* Temp sum before shifting is 17 bits *)
(* Repeat operation performed for words 2 through 6 *)
DEST[127:112] ← (SRC[127:112] + DEST[127:112] + 1) >> 1;
```

VPAVGB (VEX.128 encoded version)

```
    DEST[7:0] ← (SRC1[7:0] + SRC2[7:0] + 1) >> 1;
    (* Repeat operation performed for bytes 2 through 15 *)
    DEST[127:120] ← (SRC1[127:120] + SRC2[127:120] + 1) >> 1
    DEST[MAXVL-1:128] ← 0
```

VPAVGW (VEX.128 encoded version)

```
    DEST[15:0] ← (SRC1[15:0] + SRC2[15:0] + 1) >> 1;
    (* Repeat operation performed for 16-bit words 2 through 7 *)
    DEST[127:112] ← (SRC1[127:112] + SRC2[127:112] + 1) >> 1
    DEST[MAXVL-1:128] ← 0
```

VPAVGB (VEX.256 encoded instruction)

```
    DEST[7:0] ← (SRC1[7:0] + SRC2[7:0] + 1) >> 1; (* Temp sum before shifting is 9 bits *)
    (* Repeat operation performed for bytes 2 through 31)
    DEST[255:248] ← (SRC1[255:248] + SRC2[255:248] + 1) >> 1;
```

VPAVGW (VEX.256 encoded instruction)

```
    DEST[15:0] ← (SRC1[15:0] + SRC2[15:0] + 1) >> 1; (* Temp sum before shifting is 17 bits *)
    (* Repeat operation performed for words 2 through 15)
    DEST[255:14]) ← (SRC1[255:240] + SRC2[255:240] + 1) >> 1;
VPAVGB (EVEX encoded versions)
(KL, VL) = (16, 128), (32, 256), (64, 512)
FOR j ← 0 TO KL-1
    i ← j * 8
    IF k1[j] OR *no writemask*
        THEN DEST[i+7:i] ← (SRC1[i+7:i] + SRC2[i+7:i] + 1) >> 1; (* Temp sum before shifting is 9 bits *)
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+7:i] remains unchanged*
                ELSE *zeroing-masking*              ; zeroing-masking
                    DEST[i+7:i] = 0
            FI
    FI;
ENDFOR;
DEST[MAXVL-1:VL] ← 0
```

VPAVGW (EVEX encoded versions)

```
(KL, VL) = (8, 128), (16, 256), (32, 512)
FOR j ← 0 TO KL-1
    i ← j * 16
    IF k1[j] OR *no writemask*
        THEN DEST[i+15:i] ← (SRC1[i+15:i] + SRC2[i+15:i] + 1) >> 1
                                    ; (* Temp sum before shifting is 17 bits *)
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+15:i] remains unchanged*
                ELSE *zeroing-masking*              ; zeroing-masking
                    DEST[i+15:i] = 0
            FI
    FI;
ENDFOR;
DEST[MAXVL-1:VL] ← 0
```

Intel C/C++ Compiler Intrinsic Equivalents

VPAVGB __m512i _mm512_avg_epu8(__m512i a, __m512i b);
VPAVGW __m512i _mm512_avg_epu16(__m512i a, __m512i b);
VPAVGB __m512i _mm512_mask_avg_epu8(__m512i s, __mmask64 m, __m512i a, __m512i b);
VPAVGW __m512i _mm512_mask_avg_epu16(__m512i s, __mmask32 m, __m512i a, __m512i b);
VPAVGB __m512i _mm512_maskz_avg_epu8(__mmask64 m, __m512i a, __m512i b);
VPAVGW __m512i _mm512_maskz_avg_epu16(__mmask32 m, __m512i a, __m512i b);
VPAVGB __m256i _mm256_mask_avg_epu8(__m256i s, __mmask32 m, __m256i a, __m256i b);
VPAVGW __m256i _mm256_mask_avg_epu16(__m256i s, __mmask16 m, __m256i a, __m256i b);
VPAVGB __m256i _mm256_maskz_avg_epu8(__mmask32 m, __m256i a, __m256i b);
VPAVGW __m256i _mm256_maskz_avg_epu16(__mmask16 m, __m256i a, __m256i b);
VPAVGB __m128i _mm_mask_avg_epu8(__m128i s, __mmask16 m, __m128i a, __m128i b);
VPAVGW __m128i _mm_mask_avg_epu16(__m128i s, __mmask8 m, __m128i a, __m128i b);
VPAVGB __m128i _mm_maskz_avg_epu8(__mmask16 m, __m128i a, __m128i b);
VPAVGW __m128i _mm_maskz_avg_epu16(__mmask8 m, __m128i a, __m128i b);
PAVGB: __m64 _mm_avg_pu8 (__m64 a, __m64 b)
PAVGW: __m64 _mm_avg_pu16 (__m64 a, __m64 b)
(V)PAVGB: __m128i _mm_avg_epu8 (__m128i a, __m128i b)
(V)PAVGW: __m128i _mm_avg_epu16 (__m128i a, __m128i b)
VPAVGB: __m256i _mm256_avg_epu8 (__m256i a, __m256i b)
VPAVGW: __m256i _mm256_avg_epu16 (__m256i a, __m256i b)

Flags Affected

None.

Numeric Exceptions

None.

Other Exceptions

Non-EVEX-encoded instruction, see Exceptions Type 4.
EVEX-encoded instruction, see Exceptions Type E4.nb.

PBLENDVB — Variable Blend Packed Bytes

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 38 10 /r PBLENDVB *xmm1, xmm2/m128, <XMM0>*	RM	V/V	SSE4_1	Select byte values from *xmm1* and *xmm2/m128* from mask specified in the high bit of each byte in *XMM0* and store the values into *xmm1.*
VEX.NDS.128.66.0F3A.W0 4C /r /is4 VPBLENDVB *xmm1, xmm2, xmm3/m128, xmm4*	RVMR	V/V	AVX	Select byte values from *xmm2* and *xmm3/m128* using mask bits in the specified mask register, *xmm4*, and store the values into *xmm1*.
VEX.NDS.256.66.0F3A.W0 4C /r /is4 VPBLENDVB *ymm1, ymm2, ymm3/m256, ymm4*	RVMR	V/V	AVX2	Select byte values from *ymm2* and *ymm3/m256* from mask specified in the high bit of each byte in *ymm4* and store the values into *ymm1*.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r, w)	ModRM:r/m (r)	<XMM0>	NA
RVMR	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	imm8[7:4]

Description

Conditionally copies byte elements from the source operand (second operand) to the destination operand (first operand) depending on mask bits defined in the implicit third register argument, XMM0. The mask bits are the most significant bit in each byte element of the XMM0 register.

If a mask bit is "1", then the corresponding byte element in the source operand is copied to the destination, else the byte element in the destination operand is left unchanged.

The register assignment of the implicit third operand is defined to be the architectural register XMM0.

128-bit Legacy SSE version: The first source operand and the destination operand is the same. Bits (MAXVL-1:128) of the corresponding YMM destination register remain unchanged. The mask register operand is implicitly defined to be the architectural register XMM0. An attempt to execute PBLENDVB with a VEX prefix will cause #UD.

VEX.128 encoded version: The first source operand and the destination operand are XMM registers. The second source operand is an XMM register or 128-bit memory location. The mask operand is the third source register, and encoded in bits[7:4] of the immediate byte(imm8). The bits[3:0] of imm8 are ignored. In 32-bit mode, imm8[7] is ignored. The upper bits (MAXVL-1:128) of the corresponding YMM register (destination register) are zeroed. VEX.L must be 0, otherwise the instruction will #UD. VEX.W must be 0, otherwise, the instruction will #UD.

VEX.256 encoded version: The first source operand and the destination operand are YMM registers. The second source operand is an YMM register or 256-bit memory location. The third source register is an YMM register and encoded in bits[7:4] of the immediate byte(imm8). The bits[3:0] of imm8 are ignored. In 32-bit mode, imm8[7] is ignored.

VPBLENDVB permits the mask to be any XMM or YMM register. In contrast, PBLENDVB treats XMM0 implicitly as the mask and do not support non-destructive destination operation. An attempt to execute PBLENDVB encoded with a VEX prefix will cause a #UD exception.

Operation

PBLENDVB (128-bit Legacy SSE version)

```
MASK ← XMM0
IF (MASK[7] = 1) THEN DEST[7:0] ← SRC[7:0];
ELSE DEST[7:0] ← DEST[7:0];
IF (MASK[15] = 1) THEN DEST[15:8] ← SRC[15:8];
```

```
ELSE DEST[15:8] ← DEST[15:8];
IF (MASK[23] = 1) THEN DEST[23:16] ← SRC[23:16]
ELSE DEST[23:16] ← DEST[23:16];
IF (MASK[31] = 1) THEN DEST[31:24] ← SRC[31:24]
ELSE DEST[31:24] ← DEST[31:24];
IF (MASK[39] = 1) THEN DEST[39:32] ← SRC[39:32]
ELSE DEST[39:32] ← DEST[39:32];
IF (MASK[47] = 1) THEN DEST[47:40] ← SRC[47:40]
ELSE DEST[47:40] ← DEST[47:40];
IF (MASK[55] = 1) THEN DEST[55:48] ← SRC[55:48]
ELSE DEST[55:48] ← DEST[55:48];
IF (MASK[63] = 1) THEN DEST[63:56] ← SRC[63:56]
ELSE DEST[63:56] ← DEST[63:56];
IF (MASK[71] = 1) THEN DEST[71:64] ← SRC[71:64]
ELSE DEST[71:64] ← DEST[71:64];
IF (MASK[79] = 1) THEN DEST[79:72] ← SRC[79:72]
ELSE DEST[79:72] ← DEST[79:72];
IF (MASK[87] = 1) THEN DEST[87:80] ← SRC[87:80]
ELSE DEST[87:80] ← DEST[87:80];
IF (MASK[95] = 1) THEN DEST[95:88] ← SRC[95:88]
ELSE DEST[95:88] ←    DEST[95:88];
IF (MASK[103] = 1) THEN DEST[103:96] ← SRC[103:96]
ELSE DEST[103:96] ←    DEST[103:96];
IF (MASK[111] = 1) THEN DEST[111:104] ← SRC[111:104]
ELSE DEST[111:104] ← DEST[111:104];
IF (MASK[119] = 1) THEN DEST[119:112] ← SRC[119:112]
ELSE DEST[119:112] ← DEST[119:112];
IF (MASK[127] = 1) THEN DEST[127:120] ← SRC[127:120]
ELSE DEST[127:120] ← DEST[127:120])
DEST[MAXVL-1:128] (Unmodified)
```

VPBLENDVB (VEX.128 encoded version)

```
MASK ← SRC3
IF (MASK[7] = 1) THEN DEST[7:0] ← SRC2[7:0];
ELSE DEST[7:0] ← SRC1[7:0];
IF (MASK[15] = 1) THEN DEST[15:8] ← SRC2[15:8];
ELSE DEST[15:8] ← SRC1[15:8];
IF (MASK[23] = 1) THEN DEST[23:16] ← SRC2[23:16]
ELSE DEST[23:16] ← SRC1[23:16];
IF (MASK[31] = 1) THEN DEST[31:24] ← SRC2[31:24]
ELSE DEST[31:24] ← SRC1[31:24];
IF (MASK[39] = 1) THEN DEST[39:32] ← SRC2[39:32]
ELSE DEST[39:32] ← SRC1[39:32];
IF (MASK[47] = 1) THEN DEST[47:40] ← SRC2[47:40]
ELSE DEST[47:40] ← SRC1[47:40];
IF (MASK[55] = 1) THEN DEST[55:48] ← SRC2[55:48]
ELSE DEST[55:48] ← SRC1[55:48];
IF (MASK[63] = 1) THEN DEST[63:56] ← SRC2[63:56]
ELSE DEST[63:56] ← SRC1[63:56];
IF (MASK[71] = 1) THEN DEST[71:64] ← SRC2[71:64]
ELSE DEST[71:64] ← SRC1[71:64];
IF (MASK[79] = 1) THEN DEST[79:72] ← SRC2[79:72]
ELSE DEST[79:72] ← SRC1[79:72];
IF (MASK[87] = 1) THEN DEST[87:80] ← SRC2[87:80]
```

```
ELSE DEST[87:80] ← SRC1[87:80];
IF (MASK[95] = 1) THEN DEST[95:88] ← SRC2[95:88]
ELSE DEST[95:88] ←    SRC1[95:88];
IF (MASK[103] = 1) THEN DEST[103:96] ← SRC2[103:96]
ELSE DEST[103:96] ←    SRC1[103:96];
IF (MASK[111] = 1) THEN DEST[111:104] ← SRC2[111:104]
ELSE DEST[111:104] ← SRC1[111:104];
IF (MASK[119] = 1) THEN DEST[119:112] ← SRC2[119:112]
ELSE DEST[119:112] ← SRC1[119:112];
IF (MASK[127] = 1) THEN DEST[127:120] ← SRC2[127:120]
ELSE DEST[127:120] ← SRC1[127:120])
DEST[MAXVL-1:128] ← 0
```

VPBLENDVB (VEX.256 encoded version)

```
MASK ← SRC3
IF (MASK[7] == 1) THEN DEST[7:0] ← SRC2[7:0];
ELSE DEST[7:0] ← SRC1[7:0];
IF (MASK[15] == 1) THEN DEST[15:8] ←SRC2[15:8];
ELSE DEST[15:8] ← SRC1[15:8];
IF (MASK[23] == 1) THEN DEST[23:16] ←SRC2[23:16]
ELSE DEST[23:16] ← SRC1[23:16];
IF (MASK[31] == 1) THEN DEST[31:24] ← SRC2[31:24]
ELSE DEST[31:24] ← SRC1[31:24];
IF (MASK[39] == 1) THEN DEST[39:32] ← SRC2[39:32]
ELSE DEST[39:32] ← SRC1[39:32];
IF (MASK[47] == 1) THEN DEST[47:40] ← SRC2[47:40]
ELSE DEST[47:40] ← SRC1[47:40];
IF (MASK[55] == 1) THEN DEST[55:48] ← SRC2[55:48]
ELSE DEST[55:48] ← SRC1[55:48];
IF (MASK[63] == 1) THEN DEST[63:56] ←SRC2[63:56]
ELSE DEST[63:56] ← SRC1[63:56];
IF (MASK[71] == 1) THEN DEST[71:64] ←SRC2[71:64]
ELSE DEST[71:64] ← SRC1[71:64];
IF (MASK[79] == 1) THEN DEST[79:72] ← SRC2[79:72]
ELSE DEST[79:72] ← SRC1[79:72];
IF (MASK[87] == 1) THEN DEST[87:80] ← SRC2[87:80]
ELSE DEST[87:80] ← SRC1[87:80];
IF (MASK[95] == 1) THEN DEST[95:88] ← SRC2[95:88]
ELSE DEST[95:88] ← SRC1[95:88];
IF (MASK[103] == 1) THEN DEST[103:96] ← SRC2[103:96]
ELSE DEST[103:96] ← SRC1[103:96];
IF (MASK[111] == 1) THEN DEST[111:104] ← SRC2[111:104]
ELSE DEST[111:104] ← SRC1[111:104];
IF (MASK[119] == 1) THEN DEST[119:112] ← SRC2[119:112]
ELSE DEST[119:112] ← SRC1[119:112];
IF (MASK[127] == 1) THEN DEST[127:120] ← SRC2[127:120]
ELSE DEST[127:120] ← SRC1[127:120])
IF (MASK[135] == 1) THEN DEST[135:128] ← SRC2[135:128];
ELSE DEST[135:128] ← SRC1[135:128];
IF (MASK[143] == 1) THEN DEST[143:136] ← SRC2[143:136];
ELSE DEST[[143:136] ← SRC1[143:136];
IF (MASK[151] == 1) THEN DEST[151:144] ← SRC2[151:144]
ELSE DEST[151:144] ← SRC1[151:144];
IF (MASK[159] == 1) THEN DEST[159:152] ← SRC2[159:152]
```

```
ELSE DEST[159:152] ← SRC1[159:152];
IF (MASK[167] == 1) THEN DEST[167:160] ← SRC2[167:160]
ELSE DEST[167:160] ← SRC1[167:160];
IF (MASK[175] == 1) THEN DEST[175:168] ← SRC2[175:168]
ELSE DEST[175:168] ← SRC1[175:168];
IF (MASK[183] == 1) THEN DEST[183:176] ← SRC2[183:176]
ELSE DEST[183:176] ← SRC1[183:176];
IF (MASK[191] == 1) THEN DEST[191:184] ← SRC2[191:184]
ELSE DEST[191:184] ← SRC1[191:184];
IF (MASK[199] == 1) THEN DEST[199:192] ← SRC2[199:192]
ELSE DEST[199:192] ← SRC1[199:192];
IF (MASK[207] == 1) THEN DEST[207:200] ← SRC2[207:200]
ELSE DEST[207:200] ← SRC1[207:200]
IF (MASK[215] == 1) THEN DEST[215:208] ← SRC2[215:208]
ELSE DEST[215:208] ← SRC1[215:208];
IF (MASK[223] == 1) THEN DEST[223:216] ← SRC2[223:216]
ELSE DEST[223:216] ← SRC1[223:216];
IF (MASK[231] == 1) THEN DEST[231:224] ← SRC2[231:224]
ELSE DEST[231:224] ← SRC1[231:224];
IF (MASK[239] == 1) THEN DEST[239:232] ← SRC2[239:232]
ELSE DEST[239:232] ← SRC1[239:232];
IF (MASK[247] == 1) THEN DEST[247:240] ← SRC2[247:240]
ELSE DEST[247:240] ← SRC1[247:240];
IF (MASK[255] == 1) THEN DEST[255:248] ← SRC2[255:248]
ELSE DEST[255:248] ← SRC1[255:248]
```

Intel C/C++ Compiler Intrinsic Equivalent

(V)PBLENDVB: __m128i _mm_blendv_epi8 (__m128i v1, __m128i v2, __m128i mask);

VPBLENDVB: __m256i _mm256_blendv_epi8 (__m256i v1, __m256i v2, __m256i mask);

Flags Affected

None.

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Exceptions Type 4; additionally

#UD	If VEX.W = 1.

PBLENDW — Blend Packed Words

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 3A 0E /r ib PBLENDW *xmm1, xmm2/m128, imm8*	RMI	V/V	SSE4_1	Select words from *xmm1* and *xmm2/m128* from mask specified in *imm8* and store the values into *xmm1*.
VEX.NDS.128.66.0F3A.WIG 0E /r ib VPBLENDW *xmm1, xmm2, xmm3/m128, imm8*	RVMI	V/V	AVX	Select words from *xmm2* and *xmm3/m128* from mask specified in *imm8* and store the values into *xmm1*.
VEX.NDS.256.66.0F3A.WIG 0E /r ib VPBLENDW *ymm1, ymm2, ymm3/m256, imm8*	RVMI	V/V	AVX2	Select words from *ymm2* and *ymm3/m256* from mask specified in *imm8* and store the values into *ymm1*.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RMI	ModRM:reg (r, w)	ModRM:r/m (r)	imm8	NA
RVMI	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	imm8

Description

Words from the source operand (second operand) are conditionally written to the destination operand (first operand) depending on bits in the immediate operand (third operand). The immediate bits (bits 7:0) form a mask that determines whether the corresponding word in the destination is copied from the source. If a bit in the mask, corresponding to a word, is "1", then the word is copied, else the word element in the destination operand is unchanged.

128-bit Legacy SSE version: The second source operand can be an XMM register or a 128-bit memory location. The first source and destination operands are XMM registers. Bits (MAXVL-1:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: The second source operand can be an XMM register or a 128-bit memory location. The first source and destination operands are XMM registers. Bits (MAXVL-1:128) of the corresponding YMM register are zeroed.

VEX.256 encoded version: The first source operand is a YMM register. The second source operand is a YMM register or a 256-bit memory location. The destination operand is a YMM register.

Operation

PBLENDW (128-bit Legacy SSE version)

```
IF (imm8[0] = 1) THEN DEST[15:0] ← SRC[15:0]
ELSE DEST[15:0] ← DEST[15:0]
IF (imm8[1] = 1) THEN DEST[31:16] ← SRC[31:16]
ELSE DEST[31:16] ← DEST[31:16]
IF (imm8[2] = 1) THEN DEST[47:32] ← SRC[47:32]
ELSE DEST[47:32] ← DEST[47:32]
IF (imm8[3] = 1) THEN DEST[63:48] ← SRC[63:48]
ELSE DEST[63:48] ← DEST[63:48]
IF (imm8[4] = 1) THEN DEST[79:64] ← SRC[79:64]
ELSE DEST[79:64] ← DEST[79:64]
IF (imm8[5] = 1) THEN DEST[95:80] ← SRC[95:80]
ELSE DEST[95:80] ← DEST[95:80]
IF (imm8[6] = 1) THEN DEST[111:96] ← SRC[111:96]
ELSE DEST[111:96] ← DEST[111:96]
IF (imm8[7] = 1) THEN DEST[127:112] ← SRC[127:112]
```

ELSE DEST[127:112] ← DEST[127:112]

VPBLENDW (VEX.128 encoded version)

IF (imm8[0] = 1) THEN DEST[15:0] ← SRC2[15:0]
ELSE DEST[15:0] ← SRC1[15:0]
IF (imm8[1] = 1) THEN DEST[31:16] ← SRC2[31:16]
ELSE DEST[31:16] ← SRC1[31:16]
IF (imm8[2] = 1) THEN DEST[47:32] ← SRC2[47:32]
ELSE DEST[47:32] ← SRC1[47:32]
IF (imm8[3] = 1) THEN DEST[63:48] ← SRC2[63:48]
ELSE DEST[63:48] ← SRC1[63:48]
IF (imm8[4] = 1) THEN DEST[79:64] ← SRC2[79:64]
ELSE DEST[79:64] ← SRC1[79:64]
IF (imm8[5] = 1) THEN DEST[95:80] ← SRC2[95:80]
ELSE DEST[95:80] ← SRC1[95:80]
IF (imm8[6] = 1) THEN DEST[111:96] ← SRC2[111:96]
ELSE DEST[111:96] ← SRC1[111:96]
IF (imm8[7] = 1) THEN DEST[127:112] ← SRC2[127:112]
ELSE DEST[127:112] ← SRC1[127:112]
DEST[MAXVL-1:128] ← 0

VPBLENDW (VEX.256 encoded version)

IF (imm8[0] == 1) THEN DEST[15:0] ← SRC2[15:0]
ELSE DEST[15:0] ← SRC1[15:0]
IF (imm8[1] == 1) THEN DEST[31:16] ← SRC2[31:16]
ELSE DEST[31:16] ← SRC1[31:16]
IF (imm8[2] == 1) THEN DEST[47:32] ← SRC2[47:32]
ELSE DEST[47:32] ← SRC1[47:32]
IF (imm8[3] == 1) THEN DEST[63:48] ← SRC2[63:48]
ELSE DEST[63:48] ← SRC1[63:48]
IF (imm8[4] == 1) THEN DEST[79:64] ← SRC2[79:64]
ELSE DEST[79:64] ← SRC1[79:64]
IF (imm8[5] == 1) THEN DEST[95:80] ← SRC2[95:80]
ELSE DEST[95:80] ← SRC1[95:80]
IF (imm8[6] == 1) THEN DEST[111:96] ← SRC2[111:96]
ELSE DEST[111:96] ← SRC1[111:96]
IF (imm8[7] == 1) THEN DEST[127:112] ← SRC2[127:112]
ELSE DEST[127:112] ← SRC1[127:112]
IF (imm8[0] == 1) THEN DEST[143:128] ← SRC2[143:128]
ELSE DEST[143:128] ← SRC1[143:128]
IF (imm8[1] == 1) THEN DEST[159:144] ← SRC2[159:144]
ELSE DEST[159:144] ← SRC1[159:144]
IF (imm8[2] == 1) THEN DEST[175:160] ← SRC2[175:160]
ELSE DEST[175:160] ← SRC1[175:160]
IF (imm8[3] == 1) THEN DEST[191:176] ← SRC2[191:176]
ELSE DEST[191:176] ← SRC1[191:176]
IF (imm8[4] == 1) THEN DEST[207:192] ← SRC2[207:192]
ELSE DEST[207:192] ← SRC1[207:192]
IF (imm8[5] == 1) THEN DEST[223:208] ← SRC2[223:208]
ELSE DEST[223:208] ← SRC1[223:208]
IF (imm8[6] == 1) THEN DEST[239:224] ← SRC2[239:224]
ELSE DEST[239:224] ← SRC1[239:224]
IF (imm8[7] == 1) THEN DEST[255:240] ← SRC2[255:240]
ELSE DEST[255:240] ← SRC1[255:240]

Intel C/C++ Compiler Intrinsic Equivalent

(V)PBLENDW: __m128i _mm_blend_epi16 (__m128i v1, __m128i v2, const int mask);

VPBLENDW: __m256i _mm256_blend_epi16 (__m256i v1, __m256i v2, const int mask)

Flags Affected

None.

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Exceptions Type 4; additionally

#UD	If VEX.L = 1 and AVX2 = 0.

PCLMULQDQ — Carry-Less Multiplication Quadword

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 3A 44 /r ib PCLMULQDQ *xmm1, xmm2/m128, imm8*	RMI	V/V	PCLMUL-QDQ	Carry-less multiplication of one quadword of xmm1 by one quadword of *xmm2/m128*, stores the 128-bit result in *xmm1*. The immediate is used to determine which quadwords of *xmm1* and *xmm2/m128* should be used.
VEX.NDS.128.66.0F3A.WIG 44 /r ib VPCLMULQDQ *xmm1, xmm2, xmm3/m128, imm8*	RVMI	V/V	Both PCL-MULQDQ and AVX flags	Carry-less multiplication of one quadword of *xmm2* by one quadword of *xmm3/m128*, stores the 128-bit result in *xmm1*. The immediate is used to determine which quadwords of *xmm2* and *xmm3/m128* should be used.

Instruction Operand Encoding

Op/En	Operand 1	Operand2	Operand3	Operand4
RMI	ModRM:reg (r, w)	ModRM:r/m (r)	imm8	NA
RVMI	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	imm8

Description

Performs a carry-less multiplication of two quadwords, selected from the first source and second source operand according to the value of the immediate byte. Bits 4 and 0 are used to select which 64-bit half of each operand to use according to Table 4-13, other bits of the immediate byte are ignored.

Table 4-13. PCLMULQDQ Quadword Selection of Immediate Byte

Imm[4]	Imm[0]	PCLMULQDQ Operation
0	0	CL_MUL(SRC2[1][63:0], SRC1[63:0])
0	1	CL_MUL(SRC2[63:0], SRC1[127:64])
1	0	CL_MUL(SRC2[127:64], SRC1[63:0])
1	1	CL_MUL(SRC2[127:64], SRC1[127:64])

NOTES:

1. SRC2 denotes the second source operand, which can be a register or memory; SRC1 denotes the first source and destination operand.

The first source operand and the destination operand are the same and must be an XMM register. The second source operand can be an XMM register or a 128-bit memory location. Bits (MAXVL-1:128) of the corresponding YMM destination register remain unchanged.

Compilers and assemblers may implement the following pseudo-op syntax to simply programming and emit the required encoding for Imm8.

Table 4-14. Pseudo-Op and PCLMULQDQ Implementation

Pseudo-Op	Imm8 Encoding
PCLMULLQLQDQ *xmm1, xmm2*	**0000_0000B**
PCLMULHQLQDQ *xmm1, xmm2*	**0000_0001B**
PCLMULLQHQDQ *xmm1, xmm2*	**0001_0000B**
PCLMULHQHQDQ *xmm1, xmm2*	**0001_0001B**

Operation

PCLMULQDQ

```
IF (Imm8[0] = 0 )
    THEN
        TEMP1 ← SRC1 [63:0];
    ELSE
        TEMP1 ← SRC1 [127:64];
FI
IF (Imm8[4] = 0 )
    THEN
        TEMP2 ← SRC2 [63:0];
    ELSE
        TEMP2 ← SRC2 [127:64];
FI
For i = 0 to 63 {
    TmpB [ i ] ← (TEMP1[ 0 ] and TEMP2[ i ]);
    For j = 1 to i {
        TmpB [ i ] ← TmpB [ i ] xor (TEMP1[ j ] and TEMP2[ i - j ])
    }
    DEST[ i ] ← TmpB[ i ];
}
For i = 64 to 126 {
    TmpB [ i ] ← 0;
    For j = i - 63 to 63 {
        TmpB [ i ] ← TmpB [ i ] xor (TEMP1[ j ] and TEMP2[ i - j ])
    }
    DEST[ i ] ← TmpB[ i ];
}
DEST[127] ← 0;
DEST[MAXVL-1:128] (Unmodified)
```

VPCLMULQDQ

```
IF (Imm8[0] = 0 )
    THEN
        TEMP1 ← SRC1 [63:0];
    ELSE
        TEMP1 ← SRC1 [127:64];
FI
IF (Imm8[4] = 0 )
    THEN
        TEMP2 ← SRC2 [63:0];
    ELSE
        TEMP2 ← SRC2 [127:64];
FI
For i = 0 to 63 {
    TmpB [ i ] ← (TEMP1[ 0 ] and TEMP2[ i ]);
    For j = 1 to i {
        TmpB [i] ← TmpB [i] xor (TEMP1[ j ] and TEMP2[ i - j ])
    }
    DEST[i] ← TmpB[i];
}
For i = 64 to 126 {
    TmpB [ i ] ← 0;
    For j = i - 63 to 63 {
```

```
            TmpB [i] ← TmpB [i] xor (TEMP1[ j ] and TEMP2[ i - j ])
        }
        DEST[i] ← TmpB[i];
}
DEST[MAXVL-1:127] ← 0;
```

Intel C/C++ Compiler Intrinsic Equivalent

(V)PCLMULQDQ: __m128i _mm_clmulepi64_si128 (__m128i, __m128i, const int)

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Exceptions Type 4, additionally

#UD	If VEX.L = 1.

PCMPEQB/PCMPEQW/PCMPEQD— Compare Packed Data for Equal

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F 74 /*r*[1] PCMPEQB *mm, mm/m64*	A	V/V	MMX	Compare packed bytes in *mm/m64* and *mm* for equality.
66 0F 74 /*r* PCMPEQB *xmm1, xmm2/m128*	A	V/V	SSE2	Compare packed bytes in *xmm2/m128* and *xmm1* for equality.
NP 0F 75 /*r*[1] PCMPEQW *mm, mm/m64*	A	V/V	MMX	Compare packed words in *mm/m64* and *mm* for equality.
66 0F 75 /*r* PCMPEQW *xmm1, xmm2/m128*	A	V/V	SSE2	Compare packed words in *xmm2/m128* and *xmm1* for equality.
NP 0F 76 /*r*[1] PCMPEQD *mm, mm/m64*	A	V/V	MMX	Compare packed doublewords in *mm/m64* and *mm* for equality.
66 0F 76 /*r* PCMPEQD *xmm1, xmm2/m128*	A	V/V	SSE2	Compare packed doublewords in *xmm2/m128* and *xmm1* for equality.
VEX.NDS.128.66.0F.WIG 74 /r VPCMPEQB *xmm1, xmm2, xmm3/m128*	B	V/V	AVX	Compare packed bytes in *xmm3/m128* and *xmm2* for equality.
VEX.NDS.128.66.0F.WIG 75 /r VPCMPEQW *xmm1, xmm2, xmm3/m128*	B	V/V	AVX	Compare packed words in *xmm3/m128* and *xmm2* for equality.
VEX.NDS.128.66.0F.WIG 76 /r VPCMPEQD *xmm1, xmm2, xmm3/m128*	B	V/V	AVX	Compare packed doublewords in *xmm3/m128* and *xmm2* for equality.
VEX.NDS.256.66.0F.WIG 74 /r VPCMPEQB ymm1, ymm2, ymm3 /m256	B	V/V	AVX2	Compare packed bytes in ymm3/m256 and ymm2 for equality.
VEX.NDS.256.66.0F.WIG 75 /r VPCMPEQW *ymm1, ymm2, ymm3 /m256*	B	V/V	AVX2	Compare packed words in *ymm3/m256* and *ymm2* for equality.
VEX.NDS.256.66.0F.WIG 76 /r VPCMPEQD *ymm1, ymm2, ymm3 /m256*	B	V/V	AVX2	Compare packed doublewords in *ymm3/m256* and *ymm2* for equality.
EVEX.NDS.128.66.0F.W0 76 /r VPCMPEQD k1 {k2}, xmm2, xmm3/m128/m32bcst	C	V/V	AVX512VL AVX512F	Compare Equal between int32 vector xmm2 and int32 vector xmm3/m128/m32bcst, and set vector mask k1 to reflect the zero/nonzero status of each element of the result, under writemask.
EVEX.NDS.256.66.0F.W0 76 /r VPCMPEQD k1 {k2}, ymm2, ymm3/m256/m32bcst	C	V/V	AVX512VL AVX512F	Compare Equal between int32 vector ymm2 and int32 vector ymm3/m256/m32bcst, and set vector mask k1 to reflect the zero/nonzero status of each element of the result, under writemask.
EVEX.NDS.512.66.0F.W0 76 /r VPCMPEQD k1 {k2}, zmm2, zmm3/m512/m32bcst	C	V/V	AVX512F	Compare Equal between int32 vectors in zmm2 and zmm3/m512/m32bcst, and set destination k1 according to the comparison results under writemask k2.
EVEX.NDS.128.66.0F.WIG 74 /r VPCMPEQB k1 {k2}, xmm2, xmm3 /m128	D	V/V	AVX512VL AVX512BW	Compare packed bytes in xmm3/m128 and xmm2 for equality and set vector mask k1 to reflect the zero/nonzero status of each element of the result, under writemask.

EVEX.NDS.256.66.0F.WIG 74 /r VPCMPEQB k1 {k2}, ymm2, ymm3 /m256	D	V/V	AVX512VL AVX512BW	Compare packed bytes in ymm3/m256 and ymm2 for equality and set vector mask k1 to reflect the zero/nonzero status of each element of the result, under writemask.
EVEX.NDS.512.66.0F.WIG 74 /r VPCMPEQB k1 {k2}, zmm2, zmm3 /m512	D	V/V	AVX512BW	Compare packed bytes in zmm3/m512 and zmm2 for equality and set vector mask k1 to reflect the zero/nonzero status of each element of the result, under writemask.
EVEX.NDS.128.66.0F.WIG 75 /r VPCMPEQW k1 {k2}, xmm2, xmm3 /m128	D	V/V	AVX512VL AVX512BW	Compare packed words in xmm3/m128 and xmm2 for equality and set vector mask k1 to reflect the zero/nonzero status of each element of the result, under writemask.
EVEX.NDS.256.66.0F.WIG 75 /r VPCMPEQW k1 {k2}, ymm2, ymm3 /m256	D	V/V	AVX512VL AVX512BW	Compare packed words in ymm3/m256 and ymm2 for equality and set vector mask k1 to reflect the zero/nonzero status of each element of the result, under writemask.
EVEX.NDS.512.66.0F.WIG 75 /r VPCMPEQW k1 {k2}, zmm2, zmm3 /m512	D	V/V	AVX512BW	Compare packed words in zmm3/m512 and zmm2 for equality and set vector mask k1 to reflect the zero/nonzero status of each element of the result, under writemask.

NOTES:

1. See note in Section 2.4, "AVX and SSE Instruction Exception Specification" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A* and Section 22.25.3, "Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	NA	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
B	NA	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA
C	Full Vector	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA
D	Full Vector Mem	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA

Description

Performs a SIMD compare for equality of the packed bytes, words, or doublewords in the destination operand (first operand) and the source operand (second operand). If a pair of data elements is equal, the corresponding data element in the destination operand is set to all 1s; otherwise, it is set to all 0s.

The (V)PCMPEQB instruction compares the corresponding bytes in the destination and source operands; the (V)PCMPEQW instruction compares the corresponding words in the destination and source operands; and the (V)PCMPEQD instruction compares the corresponding doublewords in the destination and source operands.

In 64-bit mode and not encoded with VEX/EVEX, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

Legacy SSE instructions: The source operand can be an MMX technology register or a 64-bit memory location. The destination operand can be an MMX technology register.

128-bit Legacy SSE version: The second source operand can be an XMM register or a 128-bit memory location. The first source and destination operands are XMM registers. Bits (MAXVL-1:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: The second source operand can be an XMM register or a 128-bit memory location. The first source and destination operands are XMM registers. Bits (MAXVL-1:128) of the corresponding YMM register are zeroed.

VEX.256 encoded version: The first source operand is a YMM register. The second source operand is a YMM register or a 256-bit memory location. The destination operand is a YMM register.

EVEX encoded VPCMPEQD: The first source operand (second operand) is a ZMM/YMM/XMM register. The second source operand can be a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 32-bit memory location. The destination operand (first operand) is a mask register updated according to the writemask k2.

EVEX encoded VPCMPEQB/W: The first source operand (second operand) is a ZMM/YMM/XMM register. The second source operand can be a ZMM/YMM/XMM register, a 512/256/128-bit memory location. The destination operand (first operand) is a mask register updated according to the writemask k2.

Operation

PCMPEQB (with 64-bit operands)

```
    IF DEST[7:0] = SRC[7:0]
        THEN DEST[7:0) ← FFH;
        ELSE DEST[7:0] ← 0; FI;
    (* Continue comparison of 2nd through 7th bytes in DEST and SRC *)
    IF DEST[63:56] = SRC[63:56]
        THEN DEST[63:56] ← FFH;
        ELSE DEST[63:56] ← 0; FI;
```

COMPARE_BYTES_EQUAL (SRC1, SRC2)

```
    IF SRC1[7:0] = SRC2[7:0]
    THEN DEST[7:0] ←FFH;
    ELSE DEST[7:0] ←0; FI;
(* Continue comparison of 2nd through 15th bytes in SRC1 and SRC2 *)
    IF SRC1[127:120] = SRC2[127:120]
    THEN DEST[127:120] ←FFH;
    ELSE DEST[127:120] ←0; FI;
```

COMPARE_WORDS_EQUAL (SRC1, SRC2)

```
    IF SRC1[15:0] = SRC2[15:0]
    THEN DEST[15:0] ←FFFFH;
    ELSE DEST[15:0] ←0; FI;
(* Continue comparison of 2nd through 7th 16-bit words in SRC1 and SRC2 *)
    IF SRC1[127:112] = SRC2[127:112]
    THEN DEST[127:112] ←FFFFH;
    ELSE DEST[127:112] ←0; FI;
```

COMPARE_DWORDS_EQUAL (SRC1, SRC2)

```
    IF SRC1[31:0] = SRC2[31:0]
    THEN DEST[31:0] ←FFFFFFFFH;
    ELSE DEST[31:0] ←0; FI;
(* Continue comparison of 2nd through 3rd 32-bit dwords in SRC1 and SRC2 *)
    IF SRC1[127:96] = SRC2[127:96]
    THEN DEST[127:96] ←FFFFFFFFH;
    ELSE DEST[127:96] ←0; FI;
```

PCMPEQB (with 128-bit operands)

```
DEST[127:0] ←COMPARE_BYTES_EQUAL(DEST[127:0],SRC[127:0])
DEST[MAXVL-1:128] (Unmodified)
```

VPCMPEQB (VEX.128 encoded version)
DEST[127:0] ←COMPARE_BYTES_EQUAL(SRC1[127:0],SRC2[127:0])
DEST[MAXVL-1:128] ← 0

VPCMPEQB (VEX.256 encoded version)
DEST[127:0] ←COMPARE_BYTES_EQUAL(SRC1[127:0],SRC2[127:0])
DEST[255:128] ←COMPARE_BYTES_EQUAL(SRC1[255:128],SRC2[255:128])
DEST[MAXVL-1:256] ← 0

VPCMPEQB (EVEX encoded versions)
(KL, VL) = (16, 128), (32, 256), (64, 512)

```
FOR j ← 0 TO KL-1
    i ← j * 8
    IF k2[j] OR *no writemask*
        THEN
            /* signed comparison */
            CMP ← SRC1[i+7:i] == SRC2[i+7:i];
            IF CMP = TRUE
                THEN DEST[j] ← 1;
                ELSE DEST[j] ← 0; FI;
        ELSE    DEST[j] ← 0                    ; zeroing-masking onlyFI;
    FI;
ENDFOR
DEST[MAX_KL-1:KL] ← 0
```

PCMPEQW (with 64-bit operands)

```
    IF DEST[15:0] = SRC[15:0]
        THEN DEST[15:0] ← FFFFH;
        ELSE DEST[15:0] ← 0; FI;
    (* Continue comparison of 2nd and 3rd words in DEST and SRC *)
    IF DEST[63:48] = SRC[63:48]
        THEN DEST[63:48] ← FFFFH;
        ELSE DEST[63:48] ← 0; FI;
```

PCMPEQW (with 128-bit operands)
DEST[127:0] ←COMPARE_WORDS_EQUAL(DEST[127:0],SRC[127:0])
DEST[MAXVL-1:128] (Unmodified)

VPCMPEQW (VEX.128 encoded version)
DEST[127:0] ←COMPARE_WORDS_EQUAL(SRC1[127:0],SRC2[127:0])
DEST[MAXVL-1:128] ← 0

VPCMPEQW (VEX.256 encoded version)
DEST[127:0] ←COMPARE_WORDS_EQUAL(SRC1[127:0],SRC2[127:0])
DEST[255:128] ←COMPARE_WORDS_EQUAL(SRC1[255:128],SRC2[255:128])
DEST[MAXVL-1:256] ← 0

VPCMPEQW (EVEX encoded versions)

```
(KL, VL) = (8, 128), (16, 256), (32, 512)
FOR j ← 0 TO KL-1
    i ← j * 16
    IF k2[j] OR *no writemask*
        THEN
            /* signed comparison */
            CMP ← SRC1[i+15:i] == SRC2[i+15:i];
            IF CMP = TRUE
                THEN DEST[j] ← 1;
                ELSE DEST[j] ← 0; FI;
        ELSE    DEST[j] ← 0                     ; zeroing-masking onlyFI;
    FI;
ENDFOR
DEST[MAX_KL-1:KL] ← 0
```

PCMPEQD (with 64-bit operands)

```
    IF DEST[31:0] = SRC[31:0]
        THEN DEST[31:0] ← FFFFFFFFH;
        ELSE DEST[31:0] ← 0; FI;
    IF DEST[63:32] = SRC[63:32]
        THEN DEST[63:32] ← FFFFFFFFH;
        ELSE DEST[63:32] ← 0; FI;
```

PCMPEQD (with 128-bit operands)

```
DEST[127:0] ←COMPARE_DWORDS_EQUAL(DEST[127:0],SRC[127:0])
DEST[MAXVL-1:128] (Unmodified)
```

VPCMPEQD (VEX.128 encoded version)

```
DEST[127:0] ←COMPARE_DWORDS_EQUAL(SRC1[127:0],SRC2[127:0])
DEST[MAXVL-1:128] ← 0
```

VPCMPEQD (VEX.256 encoded version)

```
DEST[127:0] ←COMPARE_DWORDS_EQUAL(SRC1[127:0],SRC2[127:0])
DEST[255:128] ←COMPARE_DWORDS_EQUAL(SRC1[255:128],SRC2[255:128])
DEST[MAXVL-1:256] ← 0
```

VPCMPEQD (EVEX encoded versions)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j ← 0 TO KL-1
    i ← j * 32
    IF k2[j] OR *no writemask*
        THEN
            /* signed comparison */
            IF (EVEX.b = 1) AND (SRC2 *is memory*)
                THEN CMP ← SRC1[i+31:i] = SRC2[31:0];
                ELSE CMP ← SRC1[i+31:i] = SRC2[i+31:i];
            FI;
            IF CMP = TRUE
                THEN DEST[j] ← 1;
                ELSE DEST[j] ← 0; FI;
        ELSE    DEST[j] ← 0                     ; zeroing-masking only
    FI;
ENDFOR
DEST[MAX_KL-1:KL] ← 0
```

Intel C/C++ Compiler Intrinsic Equivalents

VPCMPEQB __mmask64 _mm512_cmpeq_epi8_mask(__m512i a, __m512i b);
VPCMPEQB __mmask64 _mm512_mask_cmpeq_epi8_mask(__mmask64 k, __m512i a, __m512i b);
VPCMPEQB __mmask32 _mm256_cmpeq_epi8_mask(__m256i a, __m256i b);
VPCMPEQB __mmask32 _mm256_mask_cmpeq_epi8_mask(__mmask32 k, __m256i a, __m256i b);
VPCMPEQB __mmask16 _mm_cmpeq_epi8_mask(__m128i a, __m128i b);
VPCMPEQB __mmask16 _mm_mask_cmpeq_epi8_mask(__mmask16 k, __m128i a, __m128i b);
VPCMPEQW __mmask32 _mm512_cmpeq_epi16_mask(__m512i a, __m512i b);
VPCMPEQW __mmask32 _mm512_mask_cmpeq_epi16_mask(__mmask32 k, __m512i a, __m512i b);
VPCMPEQW __mmask16 _mm256_cmpeq_epi16_mask(__m256i a, __m256i b);
VPCMPEQW __mmask16 _mm256_mask_cmpeq_epi16_mask(__mmask16 k, __m256i a, __m256i b);
VPCMPEQW __mmask8 _mm_cmpeq_epi16_mask(__m128i a, __m128i b);
VPCMPEQW __mmask8 _mm_mask_cmpeq_epi16_mask(__mmask8 k, __m128i a, __m128i b);
VPCMPEQD __mmask16 _mm512_cmpeq_epi32_mask(__m512i a, __m512i b);
VPCMPEQD __mmask16 _mm512_mask_cmpeq_epi32_mask(__mmask16 k, __m512i a, __m512i b);
VPCMPEQD __mmask8 _mm256_cmpeq_epi32_mask(__m256i a, __m256i b);
VPCMPEQD __mmask8 _mm256_mask_cmpeq_epi32_mask(__mmask8 k, __m256i a, __m256i b);
VPCMPEQD __mmask8 _mm_cmpeq_epi32_mask(__m128i a, __m128i b);
VPCMPEQD __mmask8 _mm_mask_cmpeq_epi32_mask(__mmask8 k, __m128i a, __m128i b);
PCMPEQB: __m64 _mm_cmpeq_pi8 (__m64 m1, __m64 m2)
PCMPEQW: __m64 _mm_cmpeq_pi16 (__m64 m1, __m64 m2)
PCMPEQD: __m64 _mm_cmpeq_pi32 (__m64 m1, __m64 m2)
(V)PCMPEQB: __m128i _mm_cmpeq_epi8 (__m128i a, __m128i b)
(V)PCMPEQW: __m128i _mm_cmpeq_epi16 (__m128i a, __m128i b)
(V)PCMPEQD: __m128i _mm_cmpeq_epi32 (__m128i a, __m128i b)
VPCMPEQB: __m256i _mm256_cmpeq_epi8 (__m256i a, __m256i b)
VPCMPEQW: __m256i _mm256_cmpeq_epi16 (__m256i a, __m256i b)
VPCMPEQD: __m256i _mm256_cmpeq_epi32 (__m256i a, __m256i b)

Flags Affected

None.

SIMD Floating-Point Exceptions

None.

Other Exceptions

Non-EVEX-encoded instruction, see Exceptions Type 4.
EVEX-encoded VPCMPEQD, see Exceptions Type E4.
EVEX-encoded VPCMPEQB/W, see Exceptions Type E4.nb.

PCMPEQQ — Compare Packed Qword Data for Equal

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 38 29 /r PCMPEQQ *xmm1, xmm2/m128*	A	V/V	SSE4_1	Compare packed qwords in *xmm2/m128* and *xmm1* for equality.
VEX.NDS.128.66.0F38.WIG 29 /r VPCMPEQQ *xmm1, xmm2, xmm3/m128*	B	V/V	AVX	Compare packed quadwords in *xmm3/m128* and *xmm2* for equality.
VEX.NDS.256.66.0F38.WIG 29 /r VPCMPEQQ *ymm1, ymm2, ymm3 /m256*	B	V/V	AVX2	Compare packed quadwords in *ymm3/m256* and *ymm2* for equality.
EVEX.NDS.128.66.0F38.W1 29 /r VPCMPEQQ k1 {k2}, xmm2, xmm3/m128/m64bcst	C	V/V	AVX512VL AVX512F	Compare Equal between int64 vector xmm2 and int64 vector xmm3/m128/m64bcst, and set vector mask k1 to reflect the zero/nonzero status of each element of the result, under writemask.
EVEX.NDS.256.66.0F38.W1 29 /r VPCMPEQQ k1 {k2}, ymm2, ymm3/m256/m64bcst	C	V/V	AVX512VL AVX512F	Compare Equal between int64 vector ymm2 and int64 vector ymm3/m256/m64bcst, and set vector mask k1 to reflect the zero/nonzero status of each element of the result, under writemask.
EVEX.NDS.512.66.0F38.W1 29 /r VPCMPEQQ k1 {k2}, zmm2, zmm3/m512/m64bcst	C	V/V	AVX512F	Compare Equal between int64 vector zmm2 and int64 vector zmm3/m512/m64bcst, and set vector mask k1 to reflect the zero/nonzero status of each element of the result, under writemask.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	NA	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
B	NA	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA
C	Full Vector	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA

Description

Performs an SIMD compare for equality of the packed quadwords in the destination operand (first operand) and the source operand (second operand). If a pair of data elements is equal, the corresponding data element in the destination is set to all 1s; otherwise, it is set to 0s.

128-bit Legacy SSE version: The second source operand can be an XMM register or a 128-bit memory location. The first source and destination operands are XMM registers. Bits (MAXVL-1:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: The second source operand can be an XMM register or a 128-bit memory location. The first source and destination operands are XMM registers. Bits (MAXVL-1:128) of the corresponding YMM register are zeroed.

VEX.256 encoded version: The first source operand is a YMM register. The second source operand is a YMM register or a 256-bit memory location. The destination operand is a YMM register.

EVEX encoded VPCMPEQQ: The first source operand (second operand) is a ZMM/YMM/XMM register. The second source operand can be a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 64-bit memory location. The destination operand (first operand) is a mask register updated according to the writemask k2.

Operation

PCMPEQQ (with 128-bit operands)

```
IF (DEST[63:0] = SRC[63:0])
    THEN DEST[63:0] ← FFFFFFFFFFFFFFFFH;
    ELSE DEST[63:0] ← 0; FI;
IF (DEST[127:64] = SRC[127:64])
    THEN DEST[127:64] ← FFFFFFFFFFFFFFFFH;
    ELSE DEST[127:64] ← 0; FI;
DEST[MAXVL-1:128] (Unmodified)
```

COMPARE_QWORDS_EQUAL (SRC1, SRC2)

```
    IF SRC1[63:0] = SRC2[63:0]
    THEN DEST[63:0] ←FFFFFFFFFFFFFFFFH;
    ELSE DEST[63:0] ←0; FI;
    IF SRC1[127:64] = SRC2[127:64]
    THEN DEST[127:64] ←FFFFFFFFFFFFFFFFH;
    ELSE DEST[127:64] ←0; FI;
```

VPCMPEQQ (VEX.128 encoded version)

```
DEST[127:0] ←COMPARE_QWORDS_EQUAL(SRC1,SRC2)
DEST[MAXVL-1:128] ← 0
```

VPCMPEQQ (VEX.256 encoded version)

```
DEST[127:0] ←COMPARE_QWORDS_EQUAL(SRC1[127:0],SRC2[127:0])
DEST[255:128] ←COMPARE_QWORDS_EQUAL(SRC1[255:128],SRC2[255:128])
DEST[MAXVL-1:256] ← 0
```

VPCMPEQQ (EVEX encoded versions)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j ← 0 TO KL-1
    i ← j * 64
    IF k2[j] OR *no writemask*
        THEN
            IF (EVEX.b = 1) AND (SRC2 *is memory*)
                THEN CMP ← SRC1[i+63:i] = SRC2[63:0];
                ELSE CMP ← SRC1[i+63:i] = SRC2[i+63:i];
            FI;
            IF CMP = TRUE
                THEN DEST[j] ← 1;
                ELSE DEST[j] ← 0; FI;
        ELSE    DEST[j] ← 0                    ; zeroing-masking only
    FI;
ENDFOR
DEST[MAX_KL-1:KL] ← 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VPCMPEQQ __mmask8 _mm512_cmpeq_epi64_mask(__m512i a, __m512i b);
VPCMPEQQ __mmask8 _mm512_mask_cmpeq_epi64_mask(__mmask8 k, __m512i a, __m512i b);
VPCMPEQQ __mmask8 _mm256_cmpeq_epi64_mask(__m256i a, __m256i b);
VPCMPEQQ __mmask8 _mm256_mask_cmpeq_epi64_mask(__mmask8 k, __m256i a, __m256i b);
VPCMPEQQ __mmask8 _mm_cmpeq_epi64_mask(__m128i a, __m128i b);
VPCMPEQQ __mmask8 _mm_mask_cmpeq_epi64_mask(__mmask8 k, __m128i a, __m128i b);
(V)PCMPEQQ: __m128i _mm_cmpeq_epi64(__m128i a, __m128i b);
VPCMPEQQ: __m256i _mm256_cmpeq_epi64(__m256i a, __m256i b);

Flags Affected

None.

SIMD Floating-Point Exceptions

None.

Other Exceptions

Non-EVEX-encoded instruction, see Exceptions Type 4.
EVEX-encoded VPCMPEQQ, see Exceptions Type E4.

PCMPESTRI — Packed Compare Explicit Length Strings, Return Index

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 3A 61 */r imm8* PCMPESTRI *xmm1, xmm2/m128, imm8*	RMI	V/V	SSE4_2	Perform a packed comparison of string data with explicit lengths, generating an index, and storing the result in ECX.
VEX.128.66.0F3A 61 /r ib VPCMPESTRI *xmm1, xmm2/m128, imm8*	RMI	V/V	AVX	Perform a packed comparison of string data with explicit lengths, generating an index, and storing the result in ECX.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RMI	ModRM:reg (r)	ModRM:r/m (r)	imm8	NA

Description

The instruction compares and processes data from two string fragments based on the encoded value in the Imm8 Control Byte (see Section 4.1, "Imm8 Control Byte Operation for PCMPESTRI / PCMPESTRM / PCMPISTRI / PCMPISTRM"), and generates an index stored to the count register (ECX).

Each string fragment is represented by two values. The first value is an xmm (or possibly m128 for the second operand) which contains the data elements of the string (byte or word data). The second value is stored in an input length register. The input length register is EAX/RAX (for xmm1) or EDX/RDX (for xmm2/m128). The length represents the number of bytes/words which are valid for the respective xmm/m128 data.

The length of each input is interpreted as being the absolute-value of the value in the length register. The absolute-value computation saturates to 16 (for bytes) and 8 (for words), based on the value of imm8[bit3] when the value in the length register is greater than 16 (8) or less than -16 (-8).

The comparison and aggregation operations are performed according to the encoded value of Imm8 bit fields (see Section 4.1). The index of the first (or last, according to imm8[6]) set bit of IntRes2 (see Section 4.1.4) is returned in ECX. If no bits are set in IntRes2, ECX is set to 16 (8).

Note that the Arithmetic Flags are written in a non-standard manner in order to supply the most relevant information:

CFlag – Reset if IntRes2 is equal to zero, set otherwise
ZFlag – Set if absolute-value of EDX is < 16 (8), reset otherwise
SFlag – Set if absolute-value of EAX is < 16 (8), reset otherwise
OFlag – IntRes2[0]
AFlag – Reset
PFlag – Reset

Effective Operand Size

Operating mode/size	Operand 1	Operand 2	Length 1	Length 2	Result
16 bit	xmm	xmm/m128	EAX	EDX	ECX
32 bit	xmm	xmm/m128	EAX	EDX	ECX
64 bit	xmm	xmm/m128	EAX	EDX	ECX
64 bit + REX.W	xmm	xmm/m128	RAX	RDX	ECX

Intel C/C++ Compiler Intrinsic Equivalent For Returning Index

int _mm_cmpestri (__m128i a, int la, __m128i b, int lb, const int mode);

Intel C/C++ Compiler Intrinsics For Reading EFlag Results

int _mm_cmpestra (__m128i a, int la, __m128i b, int lb, const int mode);

int _mm_cmpestrc (__m128i a, int la, __m128i b, int lb, const int mode);

int _mm_cmpestro (__m128i a, int la, __m128i b, int lb, const int mode);

int _mm_cmpestrs (__m128i a, int la, __m128i b, int lb, const int mode);

int _mm_cmpestrz (__m128i a, int la, __m128i b, int lb, const int mode);

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Exceptions Type 4; additionally, this instruction does not cause #GP if the memory operand is not aligned to 16 Byte boundary, and

#UD	If VEX.L = 1.
	If VEX.vvvv ≠ 1111B.

PCMPESTRM — Packed Compare Explicit Length Strings, Return Mask

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 3A 60 /r imm8 PCMPESTRM *xmm1, xmm2/m128, imm8*	RMI	V/V	SSE4_2	Perform a packed comparison of string data with explicit lengths, generating a mask, and storing the result in *XMM0*.
VEX.128.66.0F3A 60 /r ib VPCMPESTRM *xmm1, xmm2/m128, imm8*	RMI	V/V	AVX	Perform a packed comparison of string data with explicit lengths, generating a mask, and storing the result in *XMM0*.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RMI	ModRM:reg (r)	ModRM:r/m (r)	imm8	NA

Description

The instruction compares data from two string fragments based on the encoded value in the imm8 contol byte (see Section 4.1, "Imm8 Control Byte Operation for PCMPESTRI / PCMPESTRM / PCMPISTRI / PCMPISTRM"), and generates a mask stored to XMM0.

Each string fragment is represented by two values. The first value is an xmm (or possibly m128 for the second operand) which contains the data elements of the string (byte or word data). The second value is stored in an input length register. The input length register is EAX/RAX (for xmm1) or EDX/RDX (for xmm2/m128). The length represents the number of bytes/words which are valid for the respective xmm/m128 data.

The length of each input is interpreted as being the absolute-value of the value in the length register. The absolute-value computation saturates to 16 (for bytes) and 8 (for words), based on the value of imm8[bit3] when the value in the length register is greater than 16 (8) or less than -16 (-8).

The comparison and aggregation operations are performed according to the encoded value of Imm8 bit fields (see Section 4.1). As defined by imm8[6], IntRes2 is then either stored to the least significant bits of XMM0 (zero extended to 128 bits) or expanded into a byte/word-mask and then stored to XMM0.

Note that the Arithmetic Flags are written in a non-standard manner in order to supply the most relevant information:

CFlag – Reset if IntRes2 is equal to zero, set otherwise
ZFlag – Set if absolute-value of EDX is < 16 (8), reset otherwise
SFlag – Set if absolute-value of EAX is < 16 (8), reset otherwise
OFlag –IntRes2[0]
AFlag – Reset
PFlag – Reset

Note: In VEX.128 encoded versions, bits (MAXVL-1:128) of XMM0 are zeroed. VEX.vvvv is reserved and must be 1111b, VEX.L must be 0, otherwise the instruction will #UD.

Effective Operand Size

Operating mode/size	Operand1	Operand 2	Length1	Length2	Result
16 bit	xmm	xmm/m128	EAX	EDX	XMM0
32 bit	xmm	xmm/m128	EAX	EDX	XMM0
64 bit	xmm	xmm/m128	EAX	EDX	XMM0
64 bit + REX.W	xmm	xmm/m128	RAX	RDX	XMM0

Intel C/C++ Compiler Intrinsic Equivalent For Returning Mask

__m128i _mm_cmpestrm (__m128i a, int la, __m128i b, int lb, const int mode);

Intel C/C++ Compiler Intrinsics For Reading EFlag Results

int _mm_cmpestra (__m128i a, int la, __m128i b, int lb, const int mode);

int _mm_cmpestrc (__m128i a, int la, __m128i b, int lb, const int mode);

int _mm_cmpestro (__m128i a, int la, __m128i b, int lb, const int mode);

int _mm_cmpestrs (__m128i a, int la, __m128i b, int lb, const int mode);

int _mm_cmpestrz (__m128i a, int la, __m128i b, int lb, const int mode);

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Exceptions Type 4; additionally, this instruction does not cause #GP if the memory operand is not aligned to 16 Byte boundary, and

#UD	If VEX.L = 1.
	If VEX.vvvv ≠ 1111B.

PCMPGTB/PCMPGTW/PCMPGTD—Compare Packed Signed Integers for Greater Than

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F 64 /r[1] PCMPGTB *mm, mm/m64*	A	V/V	MMX	Compare packed signed byte integers in *mm* and *mm/m64* for greater than.
66 0F 64 /r PCMPGTB *xmm1, xmm2/m128*	A	V/V	SSE2	Compare packed signed byte integers in *xmm1* and *xmm2/m128* for greater than.
NP 0F 65 /r[1] PCMPGTW *mm, mm/m64*	A	V/V	MMX	Compare packed signed word integers in *mm* and *mm/m64* for greater than.
66 0F 65 /r PCMPGTW *xmm1, xmm2/m128*	A	V/V	SSE2	Compare packed signed word integers in *xmm1* and *xmm2/m128* for greater than.
NP 0F 66 /r[1] PCMPGTD *mm, mm/m64*	A	V/V	MMX	Compare packed signed doubleword integers in *mm* and *mm/m64* for greater than.
66 0F 66 /r PCMPGTD *xmm1, xmm2/m128*	A	V/V	SSE2	Compare packed signed doubleword integers in *xmm1* and *xmm2/m128* for greater than.
VEX.NDS.128.66.0F.WIG 64 /r VPCMPGTB *xmm1, xmm2, xmm3/m128*	B	V/V	AVX	Compare packed signed byte integers in *xmm2* and *xmm3/m128* for greater than.
VEX.NDS.128.66.0F.WIG 65 /r VPCMPGTW *xmm1, xmm2, xmm3/m128*	B	V/V	AVX	Compare packed signed word integers in *xmm2* and *xmm3/m128* for greater than.
VEX.NDS.128.66.0F.WIG 66 /r VPCMPGTD *xmm1, xmm2, xmm3/m128*	B	V/V	AVX	Compare packed signed doubleword integers in *xmm2* and *xmm3/m128* for greater than.
VEX.NDS.256.66.0F.WIG 64 /r VPCMPGTB *ymm1, ymm2, ymm3/m256*	B	V/V	AVX2	Compare packed signed byte integers in *ymm2* and *ymm3/m256* for greater than.
VEX.NDS.256.66.0F.WIG 65 /r VPCMPGTW *ymm1, ymm2, ymm3/m256*	B	V/V	AVX2	Compare packed signed word integers in *ymm2* and *ymm3/m256* for greater than.
VEX.NDS.256.66.0F.WIG 66 /r VPCMPGTD *ymm1, ymm2, ymm3/m256*	B	V/V	AVX2	Compare packed signed doubleword integers in *ymm2* and *ymm3/m256* for greater than.
EVEX.NDS.128.66.0F.W0 66 /r VPCMPGTD k1 {k2}, xmm2, xmm3/m128/m32bcst	C	V/V	AVX512VL AVX512F	Compare Greater between int32 vector xmm2 and int32 vector xmm3/m128/m32bcst, and set vector mask k1 to reflect the zero/nonzero status of each element of the result, under writemask.
EVEX.NDS.256.66.0F.W0 66 /r VPCMPGTD k1 {k2}, ymm2, ymm3/m256/m32bcst	C	V/V	AVX512VL AVX512F	Compare Greater between int32 vector ymm2 and int32 vector ymm3/m256/m32bcst, and set vector mask k1 to reflect the zero/nonzero status of each element of the result, under writemask.
EVEX.NDS.512.66.0F.W0 66 /r VPCMPGTD k1 {k2}, zmm2, zmm3/m512/m32bcst	C	V/V	AVX512F	Compare Greater between int32 elements in zmm2 and zmm3/m512/m32bcst, and set destination k1 according to the comparison results under writemask. k2.
EVEX.NDS.128.66.0F.WIG 64 /r VPCMPGTB k1 {k2}, xmm2, xmm3/m128	D	V/V	AVX512VL AVX512BW	Compare packed signed byte integers in xmm2 and xmm3/m128 for greater than, and set vector mask k1 to reflect the zero/nonzero status of each element of the result, under writemask.
EVEX.NDS.256.66.0F.WIG 64 /r VPCMPGTB k1 {k2}, ymm2, ymm3/m256	D	V/V	AVX512VL AVX512BW	Compare packed signed byte integers in ymm2 and ymm3/m256 for greater than, and set vector mask k1 to reflect the zero/nonzero status of each element of the result, under writemask.

EVEX.NDS.512.66.0F.WIG 64 /r VPCMPGTB k1 {k2}, zmm2, zmm3/m512	D	V/V	AVX512BW	Compare packed signed byte integers in zmm2 and zmm3/m512 for greater than, and set vector mask k1 to reflect the zero/nonzero status of each element of the result, under writemask.
EVEX.NDS.128.66.0F.WIG 65 /r VPCMPGTW k1 {k2}, xmm2, xmm3/m128	D	V/V	AVX512VL AVX512BW	Compare packed signed word integers in xmm2 and xmm3/m128 for greater than, and set vector mask k1 to reflect the zero/nonzero status of each element of the result, under writemask.
EVEX.NDS.256.66.0F.WIG 65 /r VPCMPGTW k1 {k2}, ymm2, ymm3/m256	D	V/V	AVX512VL AVX512BW	Compare packed signed word integers in ymm2 and ymm3/m256 for greater than, and set vector mask k1 to reflect the zero/nonzero status of each element of the result, under writemask.
EVEX.NDS.512.66.0F.WIG 65 /r VPCMPGTW k1 {k2}, zmm2, zmm3/m512	D	V/V	AVX512BW	Compare packed signed word integers in zmm2 and zmm3/m512 for greater than, and set vector mask k1 to reflect the zero/nonzero status of each element of the result, under writemask.

NOTES:

1. See note in Section 2.4, "AVX and SSE Instruction Exception Specification" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A* and Section 22.25.3, "Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	NA	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
B	NA	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA
C	Full Vector	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA
D	Full Vector Mem	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA

Description

Performs an SIMD signed compare for the greater value of the packed byte, word, or doubleword integers in the destination operand (first operand) and the source operand (second operand). If a data element in the destination operand is greater than the corresponding date element in the source operand, the corresponding data element in the destination operand is set to all 1s; otherwise, it is set to all 0s.

The PCMPGTB instruction compares the corresponding signed byte integers in the destination and source operands; the PCMPGTW instruction compares the corresponding signed word integers in the destination and source operands; and the PCMPGTD instruction compares the corresponding signed doubleword integers in the destination and source operands.

In 64-bit mode and not encoded with VEX/EVEX, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

Legacy SSE instructions: The source operand can be an MMX technology register or a 64-bit memory location. The destination operand can be an MMX technology register.

128-bit Legacy SSE version: The second source operand can be an XMM register or a 128-bit memory location. The first source operand and destination operand are XMM registers. Bits (MAXVL-1:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: The second source operand can be an XMM register or a 128-bit memory location. The first source operand and destination operand are XMM registers. Bits (MAXVL-1:128) of the corresponding YMM register are zeroed.

VEX.256 encoded version: The first source operand is a YMM register. The second source operand is a YMM register or a 256-bit memory location. The destination operand is a YMM register.

EVEX encoded VPCMPGTD: The first source operand (second operand) is a ZMM/YMM/XMM register. The second source operand can be a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 32-bit memory location. The destination operand (first operand) is a mask register updated according to the writemask k2.

EVEX encoded VPCMPGTB/W: The first source operand (second operand) is a ZMM/YMM/XMM register. The second source operand can be a ZMM/YMM/XMM register, a 512/256/128-bit memory location. The destination operand (first operand) is a mask register updated according to the writemask k2.

Operation

PCMPGTB (with 64-bit operands)

```
    IF DEST[7:0] > SRC[7:0]
        THEN DEST[7:0) ← FFH;
        ELSE DEST[7:0] ← 0; FI;
    (* Continue comparison of 2nd through 7th bytes in DEST and SRC *)
    IF DEST[63:56] > SRC[63:56]
        THEN DEST[63:56] ← FFH;
        ELSE DEST[63:56] ← 0; FI;
```

COMPARE_BYTES_GREATER (SRC1, SRC2)

```
    IF SRC1[7:0] > SRC2[7:0]
    THEN DEST[7:0] ←FFH;
    ELSE DEST[7:0] ←0; FI;
(* Continue comparison of 2nd through 15th bytes in SRC1 and SRC2 *)
    IF SRC1[127:120] > SRC2[127:120]
    THEN DEST[127:120] ←FFH;
    ELSE DEST[127:120] ←0; FI;
```

COMPARE_WORDS_GREATER (SRC1, SRC2)

```
    IF SRC1[15:0] > SRC2[15:0]
    THEN DEST[15:0] ←FFFFH;
    ELSE DEST[15:0] ←0; FI;
(* Continue comparison of 2nd through 7th 16-bit words in SRC1 and SRC2 *)
    IF SRC1[127:112] > SRC2[127:112]
    THEN DEST[127:112] ←FFFFH;
    ELSE DEST[127:112] ←0; FI;
```

COMPARE_DWORDS_GREATER (SRC1, SRC2)

```
    IF SRC1[31:0] > SRC2[31:0]
    THEN DEST[31:0] ←FFFFFFFFH;
    ELSE DEST[31:0] ←0; FI;
(* Continue comparison of 2nd through 3rd 32-bit dwords in SRC1 and SRC2 *)
    IF SRC1[127:96] > SRC2[127:96]
    THEN DEST[127:96] ←FFFFFFFFH;
    ELSE DEST[127:96] ←0; FI;
```

PCMPGTB (with 128-bit operands)

```
DEST[127:0] ←COMPARE_BYTES_GREATER(DEST[127:0],SRC[127:0])
DEST[MAXVL-1:128] (Unmodified)
```

VPCMPGTB (VEX.128 encoded version)

```
DEST[127:0] ←COMPARE_BYTES_GREATER(SRC1,SRC2)
DEST[MAXVL-1:128] ← 0
```

VPCMPGTB (VEX.256 encoded version)

```
DEST[127:0] ←COMPARE_BYTES_GREATER(SRC1[127:0],SRC2[127:0])
DEST[255:128] ←COMPARE_BYTES_GREATER(SRC1[255:128],SRC2[255:128])
DEST[MAXVL-1:256] ← 0
```

VPCMPGTB (EVEX encoded versions)

```
(KL, VL) = (16, 128), (32, 256), (64, 512)
FOR j ← 0 TO KL-1
    i ← j * 8
    IF k2[j] OR *no writemask*
        THEN
            /* signed comparison */
            CMP ← SRC1[i+7:i] > SRC2[i+7:i];
            IF CMP = TRUE
                THEN DEST[j] ← 1;
                ELSE DEST[j] ← 0; FI;
        ELSE    DEST[j] ← 0                    ; zeroing-masking onlyFI;
    FI;
ENDFOR
DEST[MAX_KL-1:KL] ← 0
```

PCMPGTW (with 64-bit operands)

```
    IF DEST[15:0] > SRC[15:0]
        THEN DEST[15:0] ← FFFFH;
        ELSE DEST[15:0] ← 0; FI;
    (* Continue comparison of 2nd and 3rd words in DEST and SRC *)
    IF DEST[63:48] > SRC[63:48]
        THEN DEST[63:48] ← FFFFH;
        ELSE DEST[63:48] ← 0; FI;
```

PCMPGTW (with 128-bit operands)

```
DEST[127:0] ←COMPARE_WORDS_GREATER(DEST[127:0],SRC[127:0])
DEST[MAXVL-1:128] (Unmodified)
```

VPCMPGTW (VEX.128 encoded version)

```
DEST[127:0] ←COMPARE_WORDS_GREATER(SRC1,SRC2)
DEST[MAXVL-1:128] ← 0
```

VPCMPGTW (VEX.256 encoded version)

```
DEST[127:0] ←COMPARE_WORDS_GREATER(SRC1[127:0],SRC2[127:0])
DEST[255:128] ←COMPARE_WORDS_GREATER(SRC1[255:128],SRC2[255:128])
DEST[MAXVL-1:256] ← 0
```

VPCMPGTW (EVEX encoded versions)

```
(KL, VL) = (8, 128), (16, 256), (32, 512)
FOR j ← 0 TO KL-1
    i ← j * 16
    IF k2[j] OR *no writemask*
        THEN
            /* signed comparison */
            CMP ← SRC1[i+15:i] > SRC2[i+15:i];
            IF CMP = TRUE
                THEN DEST[j] ← 1;
                ELSE DEST[j] ← 0; FI;
```

```
        ELSE      DEST[j] ← 0                    ; zeroing-masking onlyFI;
    FI;
ENDFOR
DEST[MAX_KL-1:KL] ← 0
```

PCMPGTD (with 64-bit operands)

```
    IF DEST[31:0] > SRC[31:0]
        THEN DEST[31:0] ← FFFFFFFFH;
        ELSE DEST[31:0] ← 0; FI;
    IF DEST[63:32] > SRC[63:32]
        THEN DEST[63:32] ← FFFFFFFFH;
        ELSE DEST[63:32] ← 0; FI;
```

PCMPGTD (with 128-bit operands)

```
DEST[127:0] ←COMPARE_DWORDS_GREATER(DEST[127:0],SRC[127:0])
DEST[MAXVL-1:128] (Unmodified)
```

VPCMPGTD (VEX.128 encoded version)

```
DEST[127:0] ←COMPARE_DWORDS_GREATER(SRC1,SRC2)
DEST[MAXVL-1:128] ← 0
```

VPCMPGTD (VEX.256 encoded version)

```
DEST[127:0] ←COMPARE_DWORDS_GREATER(SRC1[127:0],SRC2[127:0])
DEST[255:128] ←COMPARE_DWORDS_GREATER(SRC1[255:128],SRC2[255:128])
DEST[MAXVL-1:256] ← 0
```

VPCMPGTD (EVEX encoded versions)

```
(KL, VL) = (4, 128), (8, 256), (8, 512)
FOR j ← 0 TO KL-1
    i ← j * 32
    IF k2[j] OR *no writemask*
        THEN
            /* signed comparison */
            IF (EVEX.b = 1) AND (SRC2 *is memory*)
                THEN CMP ← SRC1[i+31:i] > SRC2[31:0];
                ELSE CMP ← SRC1[i+31:i] > SRC2[i+31:i];
            FI;
            IF CMP = TRUE
                THEN DEST[j] ← 1;
                ELSE DEST[j] ← 0; FI;
        ELSE      DEST[j] ← 0                    ; zeroing-masking only
    FI;
ENDFOR
DEST[MAX_KL-1:KL] ← 0
```

Intel C/C++ Compiler Intrinsic Equivalents

VPCMPGTB __mmask64 _mm512_cmpgt_epi8_mask(__m512i a, __m512i b);
VPCMPGTB __mmask64 _mm512_mask_cmpgt_epi8_mask(__mmask64 k, __m512i a, __m512i b);
VPCMPGTB __mmask32 _mm256_cmpgt_epi8_mask(__m256i a, __m256i b);
VPCMPGTB __mmask32 _mm256_mask_cmpgt_epi8_mask(__mmask32 k, __m256i a, __m256i b);
VPCMPGTB __mmask16 _mm_cmpgt_epi8_mask(__m128i a, __m128i b);
VPCMPGTB __mmask16 _mm_mask_cmpgt_epi8_mask(__mmask16 k, __m128i a, __m128i b);
VPCMPGTD __mmask16 _mm512_cmpgt_epi32_mask(__m512i a, __m512i b);
VPCMPGTD __mmask16 _mm512_mask_cmpgt_epi32_mask(__mmask16 k, __m512i a, __m512i b);
VPCMPGTD __mmask8 _mm256_cmpgt_epi32_mask(__m256i a, __m256i b);
VPCMPGTD __mmask8 _mm256_mask_cmpgt_epi32_mask(__mmask8 k, __m256i a, __m256i b);
VPCMPGTD __mmask8 _mm_cmpgt_epi32_mask(__m128i a, __m128i b);
VPCMPGTD __mmask8 _mm_mask_cmpgt_epi32_mask(__mmask8 k, __m128i a, __m128i b);
VPCMPGTW __mmask32 _mm512_cmpgt_epi16_mask(__m512i a, __m512i b);
VPCMPGTW __mmask32 _mm512_mask_cmpgt_epi16_mask(__mmask32 k, __m512i a, __m512i b);
VPCMPGTW __mmask16 _mm256_cmpgt_epi16_mask(__m256i a, __m256i b);
VPCMPGTW __mmask16 _mm256_mask_cmpgt_epi16_mask(__mmask16 k, __m256i a, __m256i b);
VPCMPGTW __mmask8 _mm_cmpgt_epi16_mask(__m128i a, __m128i b);
VPCMPGTW __mmask8 _mm_mask_cmpgt_epi16_mask(__mmask8 k, __m128i a, __m128i b);
PCMPGTB:__m64 _mm_cmpgt_pi8 (__m64 m1, __m64 m2)
PCMPGTW:__m64 _mm_pcmpgt_pi16 (__m64 m1, __m64 m2)
PCMPGTD:__m64 _mm_pcmpgt_pi32 (__m64 m1, __m64 m2)
(V)PCMPGTB:__m128i _mm_cmpgt_epi8 (__m128i a, __m128i b)
(V)PCMPGTW:__m128i _mm_cmpgt_epi16 (__m128i a, __m128i b)
(V)DCMPGTD:__m128i _mm_cmpgt_epi32 (__m128i a, __m128i b)
VPCMPGTB: __m256i _mm256_cmpgt_epi8 (__m256i a, __m256i b)
VPCMPGTW: __m256i _mm256_cmpgt_epi16 (__m256i a, __m256i b)
VPCMPGTD: __m256i _mm256_cmpgt_epi32 (__m256i a, __m256i b)

Flags Affected

None.

Numeric Exceptions

None.

Other Exceptions

Non-EVEX-encoded instruction, see Exceptions Type 4.
EVEX-encoded VPCMPGTD, see Exceptions Type E4.
EVEX-encoded VPCMPGTB/W, see Exceptions Type E4.nb.

PCMPGTQ — Compare Packed Data for Greater Than

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 38 37 /r PCMPGTQ *xmm1,xmm2/m128*	A	V/V	SSE4_2	Compare packed signed qwords in *xmm2/m128* and *xmm1* for greater than.
VEX.NDS.128.66.0F38.WIG 37 /r VPCMPGTQ *xmm1, xmm2, xmm3/m128*	B	V/V	AVX	Compare packed signed qwords in *xmm2* and *xmm3/m128* for greater than.
VEX.NDS.256.66.0F38.WIG 37 /r VPCMPGTQ *ymm1, ymm2, ymm3/m256*	B	V/V	AVX2	Compare packed signed qwords in *ymm2* and *ymm3/m256* for greater than.
EVEX.NDS.128.66.0F38.W1 37 /r VPCMPGTQ k1 {k2}, xmm2, xmm3/m128/m64bcst	C	V/V	AVX512VL AVX512F	Compare Greater between int64 vector xmm2 and int64 vector xmm3/m128/m64bcst, and set vector mask k1 to reflect the zero/nonzero status of each element of the result, under writemask.
EVEX.NDS.256.66.0F38.W1 37 /r VPCMPGTQ k1 {k2}, ymm2, ymm3/m256/m64bcst	C	V/V	AVX512VL AVX512F	Compare Greater between int64 vector ymm2 and int64 vector ymm3/m256/m64bcst, and set vector mask k1 to reflect the zero/nonzero status of each element of the result, under writemask.
EVEX.NDS.512.66.0F38.W1 37 /r VPCMPGTQ k1 {k2}, zmm2, zmm3/m512/m64bcst	C	V/V	AVX512F	Compare Greater between int64 vector zmm2 and int64 vector zmm3/m512/m64bcst, and set vector mask k1 to reflect the zero/nonzero status of each element of the result, under writemask.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	NA	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
B	NA	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA
C	Full Vector	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA

Description

Performs an SIMD signed compare for the packed quadwords in the destination operand (first operand) and the source operand (second operand). If the data element in the first (destination) operand is greater than the corresponding element in the second (source) operand, the corresponding data element in the destination is set to all 1s; otherwise, it is set to 0s.

128-bit Legacy SSE version: The second source operand can be an XMM register or a 128-bit memory location. The first source operand and destination operand are XMM registers. Bits (MAXVL-1:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: The second source operand can be an XMM register or a 128-bit memory location. The first source operand and destination operand are XMM registers. Bits (MAXVL-1:128) of the corresponding YMM register are zeroed.

VEX.256 encoded version: The first source operand is a YMM register. The second source operand is a YMM register or a 256-bit memory location. The destination operand is a YMM register.

EVEX encoded VPCMPGTD/Q: The first source operand (second operand) is a ZMM/YMM/XMM register. The second source operand can be a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 64-bit memory location. The destination operand (first operand) is a mask register updated according to the writemask k2.

Operation

COMPARE_QWORDS_GREATER (SRC1, SRC2)

```
    IF SRC1[63:0] > SRC2[63:0]
    THEN DEST[63:0] ←FFFFFFFFFFFFFFFFH;
    ELSE DEST[63:0] ←0; FI;
    IF SRC1[127:64] > SRC2[127:64]
    THEN DEST[127:64] ←FFFFFFFFFFFFFFFFH;
    ELSE DEST[127:64] ←0; FI;
```

VPCMPGTQ (VEX.128 encoded version)

```
DEST[127:0] ←COMPARE_QWORDS_GREATER(SRC1,SRC2)
DEST[MAXVL-1:128] ← 0
```

VPCMPGTQ (VEX.256 encoded version)

```
DEST[127:0] ←COMPARE_QWORDS_GREATER(SRC1[127:0],SRC2[127:0])
DEST[255:128] ←COMPARE_QWORDS_GREATER(SRC1[255:128],SRC2[255:128])
DEST[MAXVL-1:256] ← 0
```

VPCMPGTQ (EVEX encoded versions)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j ← 0 TO KL-1
    i ← j * 64
    IF k2[j] OR *no writemask*
        THEN
            /* signed comparison */
            IF (EVEX.b = 1) AND (SRC2 *is memory*)
                THEN CMP ← SRC1[i+63:i] > SRC2[63:0];
                ELSE CMP ← SRC1[i+63:i] > SRC2[i+63:i];
            FI;
            IF CMP = TRUE
                THEN DEST[j] ← 1;
                ELSE DEST[j] ← 0; FI;
        ELSE    DEST[j] ← 0                    ; zeroing-masking only
    FI;
ENDFOR
DEST[MAX_KL-1:KL] ← 0
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VPCMPGTQ __mmask8 _mm512_cmpgt_epi64_mask( __m512i a, __m512i b);
VPCMPGTQ __mmask8 _mm512_mask_cmpgt_epi64_mask(__mmask8 k, __m512i a, __m512i b);
VPCMPGTQ __mmask8 _mm256_cmpgt_epi64_mask( __m256i a, __m256i b);
VPCMPGTQ __mmask8 _mm256_mask_cmpgt_epi64_mask(__mmask8 k, __m256i a, __m256i b);
VPCMPGTQ __mmask8 _mm_cmpgt_epi64_mask( __m128i a, __m128i b);
VPCMPGTQ __mmask8 _mm_mask_cmpgt_epi64_mask(__mmask8 k, __m128i a, __m128i b);
(V)PCMPGTQ:     __m128i _mm_cmpgt_epi64(__m128i a, __m128i b)
VPCMPGTQ:       __m256i _mm256_cmpgt_epi64( __m256i a, __m256i b);
```

Flags Affected

None.

SIMD Floating-Point Exceptions

None.

Other Exceptions

Non-EVEX-encoded instruction, see Exceptions Type 4.
EVEX-encoded VPCMPGTQ, see Exceptions Type E4.

PCMPISTRI — Packed Compare Implicit Length Strings, Return Index

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 3A 63 */r imm8* PCMPISTRI *xmm1, xmm2/m128, imm8*	RM	V/V	SSE4_2	Perform a packed comparison of string data with implicit lengths, generating an index, and storing the result in ECX.
VEX.128.66.0F3A.WIG 63 /r ib VPCMPISTRI *xmm1, xmm2/m128, imm8*	RM	V/V	AVX	Perform a packed comparison of string data with implicit lengths, generating an index, and storing the result in ECX.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r)	ModRM:r/m (r)	imm8	NA

Description

The instruction compares data from two strings based on the encoded value in the Imm8 Control Byte (see Section 4.1, "Imm8 Control Byte Operation for PCMPESTRI / PCMPESTRM / PCMPISTRI / PCMPISTRM"), and generates an index stored to ECX.

Each string is represented by a single value. The value is an xmm (or possibly m128 for the second operand) which contains the data elements of the string (byte or word data). Each input byte/word is augmented with a valid/invalid tag. A byte/word is considered valid only if it has a lower index than the least significant null byte/word. (The least significant null byte/word is also considered invalid.)

The comparison and aggregation operations are performed according to the encoded value of Imm8 bit fields (see Section 4.1). The index of the first (or last, according to imm8[6]) set bit of IntRes2 is returned in ECX. If no bits are set in IntRes2, ECX is set to 16 (8).

Note that the Arithmetic Flags are written in a non-standard manner in order to supply the most relevant information:

CFlag – Reset if IntRes2 is equal to zero, set otherwise
ZFlag – Set if any byte/word of xmm2/mem128 is null, reset otherwise
SFlag – Set if any byte/word of xmm1 is null, reset otherwise
OFlag –IntRes2[0]
AFlag – Reset
PFlag – Reset

Note: In VEX.128 encoded version, VEX.vvvv is reserved and must be 1111b, VEX.L must be 0, otherwise the instruction will #UD.

Effective Operand Size

Operating mode/size	Operand1	Operand 2	Result
16 bit	xmm	xmm/m128	ECX
32 bit	xmm	xmm/m128	ECX
64 bit	xmm	xmm/m128	ECX

Intel C/C++ Compiler Intrinsic Equivalent For Returning Index

int _mm_cmpistri (__m128i a, __m128i b, const int mode);

Intel C/C++ Compiler Intrinsics For Reading EFlag Results

int _mm_cmpistra (__m128i a, __m128i b, const int mode);

int _mm_cmpistrc (__m128i a, __m128i b, const int mode);

int _mm_cmpistro (__m128i a, __m128i b, const int mode);

int _mm_cmpistrs (__m128i a, __m128i b, const int mode);

int _mm_cmpistrz (__m128i a, __m128i b, const int mode);

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Exceptions Type 4; additionally, this instruction does not cause #GP if the memory operand is not aligned to 16 Byte boundary, and

#UD	If VEX.L = 1.
	If VEX.vvvv ≠ 1111B.

PCMPISTRM — Packed Compare Implicit Length Strings, Return Mask

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 3A 62 */r imm8* PCMPISTRM *xmm1, xmm2/m128, imm8*	RM	V/V	SSE4_2	Perform a packed comparison of string data with implicit lengths, generating a mask, and storing the result in *XMM0*.
VEX.128.66.0F3A.WIG 62 /r ib VPCMPISTRM *xmm1, xmm2/m128, imm8*	RM	V/V	AVX	Perform a packed comparison of string data with implicit lengths, generating a Mask, and storing the result in *XMM0*.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r)	ModRM:r/m (r)	imm8	NA

Description

The instruction compares data from two strings based on the encoded value in the imm8 byte (see Section 4.1, "Imm8 Control Byte Operation for PCMPESTRI / PCMPESTRM / PCMPISTRI / PCMPISTRM") generating a mask stored to XMM0.

Each string is represented by a single value. The value is an xmm (or possibly m128 for the second operand) which contains the data elements of the string (byte or word data). Each input byte/word is augmented with a valid/invalid tag. A byte/word is considered valid only if it has a lower index than the least significant null byte/word. (The least significant null byte/word is also considered invalid.)

The comparison and aggregation operation are performed according to the encoded value of Imm8 bit fields (see Section 4.1). As defined by imm8[6], IntRes2 is then either stored to the least significant bits of XMM0 (zero extended to 128 bits) or expanded into a byte/word-mask and then stored to XMM0.

Note that the Arithmetic Flags are written in a non-standard manner in order to supply the most relevant information:

CFlag – Reset if IntRes2 is equal to zero, set otherwise
ZFlag – Set if any byte/word of xmm2/mem128 is null, reset otherwise
SFlag – Set if any byte/word of xmm1 is null, reset otherwise
OFlag – IntRes2[0]
AFlag – Reset
PFlag – Reset

Note: In VEX.128 encoded versions, bits (MAXVL-1:128) of XMM0 are zeroed. VEX.vvvv is reserved and must be 1111b, VEX.L must be 0, otherwise the instruction will #UD.

Effective Operand Size

Operating mode/size	Operand1	Operand 2	Result
16 bit	xmm	xmm/m128	XMM0
32 bit	xmm	xmm/m128	XMM0
64 bit	xmm	xmm/m128	XMM0

Intel C/C++ Compiler Intrinsic Equivalent For Returning Mask

__m128i _mm_cmpistrm (__m128i a, __m128i b, const int mode);

Intel C/C++ Compiler Intrinsics For Reading EFlag Results

int _mm_cmpistra (__m128i a, __m128i b, const int mode);

int _mm_cmpistrc (__m128i a, __m128i b, const int mode);

int _mm_cmpistro (__m128i a, __m128i b, const int mode);

int _mm_cmpistrs (__m128i a, __m128i b, const int mode);

int _mm_cmpistrz (__m128i a, __m128i b, const int mode);

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Exceptions Type 4; additionally, this instruction does not cause #GP if the memory operand is not aligned to 16 Byte boundary, and

#UD	If VEX.L = 1.
	If VEX.vvvv ≠ 1111B.

PDEP — Parallel Bits Deposit

Opcode/ Instruction	Op/ En	64/32 -bit Mode	CPUID Feature Flag	Description
VEX.NDS.LZ.F2.0F38.W0 F5 /r PDEP *r32a, r32b, r/m32*	RVM	V/V	BMI2	Parallel deposit of bits from *r32b* using mask in *r/m32*, result is written to *r32a*.
VEX.NDS.LZ.F2.0F38.W1 F5 /r PDEP *r64a, r64b, r/m64*	RVM	V/N.E.	BMI2	Parallel deposit of bits from *r64b* using mask in *r/m64*, result is written to *r64a*.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RVM	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA

Description

PDEP uses a mask in the second source operand (the third operand) to transfer/scatter contiguous low order bits in the first source operand (the second operand) into the destination (the first operand). PDEP takes the low bits from the first source operand and deposit them in the destination operand at the corresponding bit locations that are set in the second source operand (mask). All other bits (bits not set in mask) in destination are set to zero.

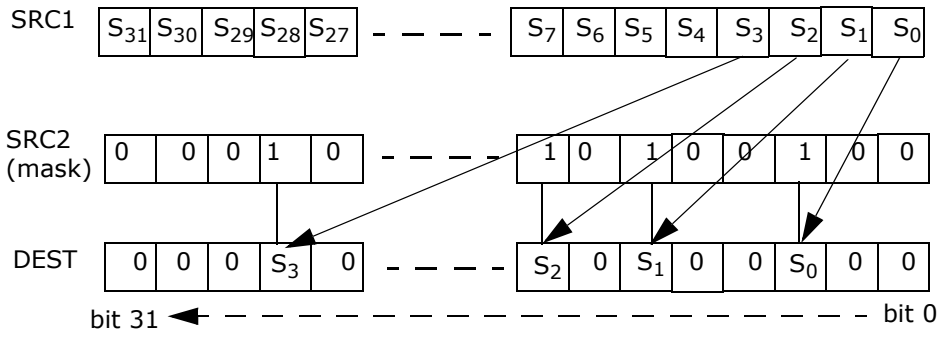


Figure 4-8. PDEP Example

This instruction is not supported in real mode and virtual-8086 mode. The operand size is always 32 bits if not in 64-bit mode. In 64-bit mode operand size 64 requires VEX.W1. VEX.W1 is ignored in non-64-bit modes. An attempt to execute this instruction with VEX.L not equal to 0 will cause #UD.

Operation

```
TEMP ← SRC1;
MASK ← SRC2;
DEST ← 0 ;
m← 0, k← 0;
DO WHILE m< OperandSize

        IF MASK[ m] = 1 THEN
            DEST[ m] ← TEMP[ k];
            k ← k+ 1;
        FI
        m ← m+ 1;
OD
```

Flags Affected

None.

Intel C/C++ Compiler Intrinsic Equivalent

PDEP: unsigned __int32 _pdep_u32(unsigned __int32 src, unsigned __int32 mask);

PDEP: unsigned __int64 _pdep_u64(unsigned __int64 src, unsigned __int32 mask);

SIMD Floating-Point Exceptions

None

Other Exceptions

See Section 2.5.1, "Exception Conditions for VEX-Encoded GPR Instructions", Table 2-29; additionally

#UD	If VEX.W = 1.

PEXT — Parallel Bits Extract

Opcode/ Instruction	Op/ En	64/32 -bit Mode	CPUID Feature Flag	Description
VEX.NDS.LZ.F3.0F38.W0 F5 /r PEXT *r32a, r32b, r/m32*	RVM	V/V	BMI2	Parallel extract of bits from *r32b* using mask in *r/m32*, result is written to *r32a*.
VEX.NDS.LZ.F3.0F38.W1 F5 /r PEXT *r64a, r64b, r/m64*	RVM	V/N.E.	BMI2	Parallel extract of bits from *r64b* using mask in *r/m64*, result is written to *r64a*.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RVM	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA

Description

PEXT uses a mask in the second source operand (the third operand) to transfer either contiguous or non-contiguous bits in the first source operand (the second operand) to contiguous low order bit positions in the destination (the first operand). For each bit set in the MASK, PEXT extracts the corresponding bits from the first source operand and writes them into contiguous lower bits of destination operand. The remaining upper bits of destination are zeroed.

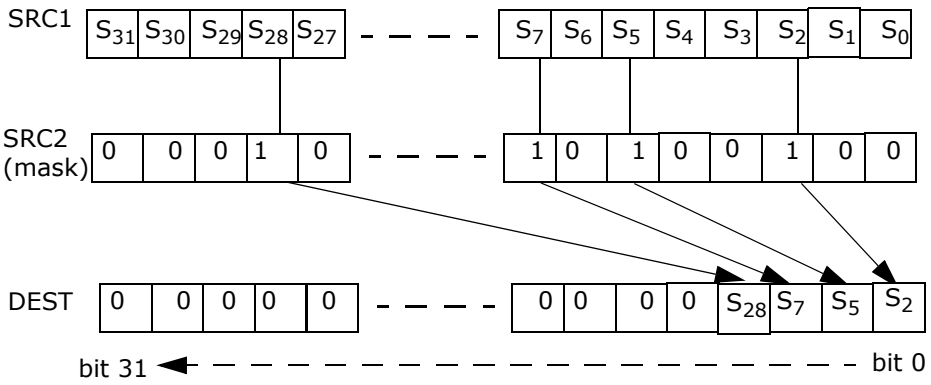


Figure 4-9. PEXT Example

This instruction is not supported in real mode and virtual-8086 mode. The operand size is always 32 bits if not in 64-bit mode. In 64-bit mode operand size 64 requires VEX.W1. VEX.W1 is ignored in non-64-bit modes. An attempt to execute this instruction with VEX.L not equal to 0 will cause #UD.

Operation

```
TEMP ← SRC1;
MASK ← SRC2;
DEST ← 0 ;
m← 0, k← 0;
DO WHILE m< OperandSize

        IF MASK[ m] = 1 THEN
              DEST[ k] ← TEMP[ m];
              k ← k+ 1;
        FI
```

```
        m ← m+ 1;

OD
```

Flags Affected

None.

Intel C/C++ Compiler Intrinsic Equivalent

PEXT: unsigned __int32 _pext_u32(unsigned __int32 src, unsigned __int32 mask);

PEXT: unsigned __int64 _pext_u64(unsigned __int64 src, unsigned __int32 mask);

SIMD Floating-Point Exceptions

None

Other Exceptions

See Section 2.5.1, "Exception Conditions for VEX-Encoded GPR Instructions", Table 2-29; additionally

#UD	If VEX.W = 1.

PEXTRB/PEXTRD/PEXTRQ — Extract Byte/Dword/Qword

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 3A 14 /r ib PEXTRB *reg/m8, xmm2, imm8*	A	V/V	SSE4_1	Extract a byte integer value from *xmm2* at the source byte offset specified by *imm8* into *reg or m8*. The upper bits of r32 or r64 are zeroed.
66 0F 3A 16 /r ib PEXTRD *r/m32, xmm2, imm8*	A	V/V	SSE4_1	Extract a dword integer value from *xmm2* at the source dword offset specified by *imm8* into *r/m32*.
66 REX.W 0F 3A 16 /r ib PEXTRQ *r/m64, xmm2, imm8*	A	V/N.E.	SSE4_1	Extract a qword integer value from *xmm2* at the source qword offset specified by *imm8* into *r/m64*.
VEX.128.66.0F3A.W0 14 /r ib VPEXTRB *reg/m8, xmm2, imm8*	A	V[1]/V	AVX	Extract a byte integer value from *xmm2* at the source byte offset specified by *imm8* into *reg* or *m8*. The upper bits of r64/r32 is filled with zeros.
VEX.128.66.0F3A.W0 16 /r ib VPEXTRD *r32/m32, xmm2, imm8*	A	V/V	AVX	Extract a dword integer value from *xmm2* at the source dword offset specified by *imm8* into *r32/m32*.
VEX.128.66.0F3A.W1 16 /r ib VPEXTRQ *r64/m64, xmm2, imm8*	A	V/I[2]	AVX	Extract a qword integer value from *xmm2* at the source dword offset specified by *imm8* into *r64/m64*.
EVEX.128.66.0F3A.WIG 14 /r ib VPEXTRB reg/m8, xmm2, imm8	B	V/V	AVX512BW	Extract a byte integer value from xmm2 at the source byte offset specified by imm8 into reg or m8. The upper bits of r64/r32 is filled with zeros.
EVEX.128.66.0F3A.W0 16 /r ib VPEXTRD r32/m32, xmm2, imm8	B	V/V	AVX512DQ	Extract a dword integer value from xmm2 at the source dword offset specified by imm8 into r32/m32.
EVEX.128.66.0F3A.W1 16 /r ib VPEXTRQ r64/m64, xmm2, imm8	B	V/N.E.[2]	AVX512DQ	Extract a qword integer value from xmm2 at the source dword offset specified by imm8 into r64/m64.

NOTES:

1. In 64-bit mode, VEX.W1 is ignored for VPEXTRB (similar to legacy REX.W=1 prefix in PEXTRB).
2. VEX.W/EVEX.W in non-64 bit is ignored; the instructions behaves as if the W0 version is used.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	NA	ModRM:r/m (w)	ModRM:reg (r)	imm8	NA
B	Tuple1 Scalar	ModRM:r/m (w)	ModRM:reg (r)	imm8	NA

Description

Extract a byte/dword/qword integer value from the source XMM register at a byte/dword/qword offset determined from imm8[3:0]. The destination can be a register or byte/dword/qword memory location. If the destination is a register, the upper bits of the register are zero extended.

In legacy non-VEX encoded version and if the destination operand is a register, the default operand size in 64-bit mode for PEXTRB/PEXTRD is 64 bits, the bits above the least significant byte/dword data are filled with zeros. PEXTRQ is not encodable in non-64-bit modes and requires REX.W in 64-bit mode.

Note: In VEX.128 encoded versions, VEX.vvvv is reserved and must be 1111b, VEX.L must be 0, otherwise the instruction will #UD. In EVEX.128 encoded versions, EVEX.vvvv is reserved and must be 1111b, EVEX.L"L must be

0, otherwise the instruction will #UD. If the destination operand is a register, the default operand size in 64-bit mode for VPEXTRB/VPEXTRD is 64 bits, the bits above the least significant byte/word/dword data are filled with zeros.

Operation

```
CASE of
    PEXTRB: SEL ← COUNT[3:0];
            TEMP ← (Src >> SEL*8) AND FFH;
            IF (DEST = Mem8)
                THEN
                Mem8 ← TEMP[7:0];
            ELSE IF (64-Bit Mode and 64-bit register selected)
                THEN
                    R64[7:0] ← TEMP[7:0];
                    r64[63:8] ← ZERO_FILL; };
            ELSE
                    R32[7:0] ← TEMP[7:0];
                    r32[31:8] ← ZERO_FILL; };
            FI;
    PEXTRD:SEL ← COUNT[1:0];
            TEMP ← (Src >> SEL*32) AND FFFF_FFFFH;
            DEST ← TEMP;
    PEXTRQ: SEL ← COUNT[0];
            TEMP ← (Src >> SEL*64);
            DEST ← TEMP;
EASC:
```

VPEXTRTD/VPEXTRQ

```
IF (64-Bit Mode and 64-bit dest operand)
THEN
    Src_Offset ← Imm8[0]
    r64/m64 ←(Src >> Src_Offset * 64)
ELSE
    Src_Offset ← Imm8[1:0]
    r32/m32 ← ((Src >> Src_Offset *32) AND 0FFFFFFFFh);
FI
```

VPEXTRB (dest=m8)

```
SRC_Offset ← Imm8[3:0]
Mem8 ← (Src >> Src_Offset*8)
```

VPEXTRB (dest=reg)

```
IF (64-Bit Mode )
THEN
    SRC_Offset ← Imm8[3:0]
    DEST[7:0] ← ((Src >> Src_Offset*8) AND 0FFh)
    DEST[63:8] ←    ZERO_FILL;
ELSE
    SRC_Offset ←. Imm8[3:0];
    DEST[7:0] ← ((Src >> Src_Offset*8) AND 0FFh);
    DEST[31:8] ←    ZERO_FILL;
FI
```

Intel C/C++ Compiler Intrinsic Equivalent

PEXTRB: int _mm_extract_epi8 (__m128i src, const int ndx);

PEXTRD: int _mm_extract_epi32 (__m128i src, const int ndx);

PEXTRQ: __int64 _mm_extract_epi64 (__m128i src, const int ndx);

Flags Affected

None.

SIMD Floating-Point Exceptions

None.

Other Exceptions

Non-EVEX-encoded instruction, see Exceptions Type 5;
EVEX-encoded instruction, see Exceptions Type E9NF.

#UD	If VEX.L = 1 or EVEX.L'L > 0.
	If VEX.vvvv != 1111B or EVEX.vvvv != 1111B.

PEXTRW—Extract Word

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F C5 /*r* ib[1] PEXTRW *reg, mm, imm8*	A	V/V	SSE	Extract the word specified by *imm8* from *mm* and move it to *reg*, bits 15-0. The upper bits of r32 or r64 is zeroed.
66 0F C5 /*r* ib PEXTRW *reg, xmm, imm8*	A	V/V	SSE2	Extract the word specified by *imm8* from *xmm* and move it to *reg*, bits 15-0. The upper bits of r32 or r64 is zeroed.
66 0F 3A 15 /r ib PEXTRW *reg/m16, xmm, imm8*	B	V/V	SSE4_1	Extract the word specified by *imm8* from *xmm* and copy it to lowest 16 bits of *reg or m16*. Zero-extend the result in the destination, r32 or r64.
VEX.128.66.0F.W0 C5 /r ib VPEXTRW *reg, xmm1, imm8*	A	V[2]/V	AVX	Extract the word specified by *imm8* from *xmm1* and move it to reg, bits 15:0. Zero-extend the result. The upper bits of r64/r32 is filled with zeros.
VEX.128.66.0F3A.W0 15 /r ib VPEXTRW *reg/m16, xmm2, imm8*	B	V/V	AVX	Extract a word integer value from *xmm2* at the source word offset specified by *imm8* into *reg* or *m16*. The upper bits of r64/r32 is filled with zeros.
EVEX.128.66.0F.WIG C5 /r ib VPEXTRW reg, xmm1, imm8	A	V/V	AVX512BW	Extract the word specified by imm8 from xmm1 and move it to reg, bits 15:0. Zero-extend the result. The upper bits of r64/r32 is filled with zeros.
EVEX.128.66.0F3A.WIG 15 /r ib VPEXTRW reg/m16, xmm2, imm8	C	V/V	AVX512BW	Extract a word integer value from xmm2 at the source word offset specified by imm8 into reg or m16. The upper bits of r64/r32 is filled with zeros.

NOTES:

1. See note in Section 2.4, "AVX and SSE Instruction Exception Specification" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A* and Section 22.25.3, "Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*.

2. In 64-bit mode, VEX.W1 is ignored for VPEXTRW (similar to legacy REX.W=1 prefix in PEXTRW).

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	NA	ModRM:reg (w)	ModRM:r/m (r)	imm8	NA
B	NA	ModRM:r/m (w)	ModRM:reg (r)	imm8	NA
C	Tuple1 Scalar	ModRM:r/m (w)	ModRM:reg (r)	imm8	NA

Description

Copies the word in the source operand (second operand) specified by the count operand (third operand) to the destination operand (first operand). The source operand can be an MMX technology register or an XMM register. The destination operand can be the low word of a general-purpose register or a 16-bit memory address. The count operand is an 8-bit immediate. When specifying a word location in an MMX technology register, the 2 least-significant bits of the count operand specify the location; for an XMM register, the 3 least-significant bits specify the location. The content of the destination register above bit 16 is cleared (set to all 0s).

In 64-bit mode, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15, R8-15). If the destination operand is a general-purpose register, the default operand size is 64-bits in 64-bit mode.

Note: In VEX.128 encoded versions, VEX.vvvv is reserved and must be 1111b, VEX.L must be 0, otherwise the instruction will #UD. In EVEX.128 encoded versions, EVEX.vvvv is reserved and must be 1111b, EVEX.L must be 0, otherwise the instruction will #UD. If the destination operand is a register, the default operand size in 64-bit mode for VPEXTRW is 64 bits, the bits above the least significant byte/word/dword data are filled with zeros.

Operation

```
IF (DEST = Mem16)
THEN
    SEL ← COUNT[2:0];
    TEMP ← (Src >> SEL*16) AND FFFFH;
    Mem16 ← TEMP[15:0];
ELSE IF (64-Bit Mode and destination is a general-purpose register)
    THEN
        FOR (PEXTRW instruction with 64-bit source operand)
          {  SEL ← COUNT[1:0];
             TEMP ← (SRC >> (SEL * 16)) AND FFFFH;
             r64[15:0] ← TEMP[15:0];
             r64[63:16] ← ZERO_FILL; };
        FOR (PEXTRW instruction with 128-bit source operand)
          {  SEL ← COUNT[2:0];
             TEMP ← (SRC >> (SEL * 16)) AND FFFFH;
             r64[15:0] ← TEMP[15:0];
             r64[63:16] ← ZERO_FILL; }
    ELSE
        FOR (PEXTRW instruction with 64-bit source operand)
          {  SEL ← COUNT[1:0];
             TEMP ← (SRC >> (SEL * 16)) AND FFFFH;
             r32[15:0] ← TEMP[15:0];
             r32[31:16] ← ZERO_FILL; };
        FOR (PEXTRW instruction with 128-bit source operand)
          {  SEL ← COUNT[2:0];
             TEMP ← (SRC >> (SEL * 16)) AND FFFFH;
             r32[15:0] ← TEMP[15:0];
             r32[31:16] ← ZERO_FILL; };
    FI;
FI;
```

VPEXTRW (dest=m16)

```
SRC_Offset ← Imm8[2:0]
Mem16 ← (Src >> Src_Offset*16)
```

VPEXTRW (dest=reg)

```
IF (64-Bit Mode )
THEN
    SRC_Offset ← Imm8[2:0]
    DEST[15:0] ← ((Src >> Src_Offset*16) AND 0FFFFh)
    DEST[63:16] ←   ZERO_FILL;
ELSE
    SRC_Offset ← Imm8[2:0]
    DEST[15:0] ← ((Src >> Src_Offset*16) AND 0FFFFh)
    DEST[31:16] ←   ZERO_FILL;
FI
```

Intel C/C++ Compiler Intrinsic Equivalent

PEXTRW: int _mm_extract_pi16 (__m64 a, int n)

PEXTRW: int _mm_extract_epi16 (__m128i a, int imm)

Flags Affected

None.

Numeric Exceptions

None.

Other Exceptions

Non-EVEX-encoded instruction, see Exceptions Type 5;

EVEX-encoded instruction, see Exceptions Type E9NF.

#UD	If VEX.L = 1 or EVEX.L'L > 0.
	If VEX.vvvv != 1111B or EVEX.vvvv != 1111B.

PHADDW/PHADDD — Packed Horizontal Add

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F 38 01 /r[1] PHADDW *mm1, mm2/m64*	RM	V/V	SSSE3	Add 16-bit integers horizontally, pack to *mm1*.
66 0F 38 01 /r PHADDW *xmm1, xmm2/m128*	RM	V/V	SSSE3	Add 16-bit integers horizontally, pack to *xmm1*.
NP 0F 38 02 /r PHADDD *mm1, mm2/m64*	RM	V/V	SSSE3	Add 32-bit integers horizontally, pack to *mm1*.
66 0F 38 02 /r PHADDD *xmm1, xmm2/m128*	RM	V/V	SSSE3	Add 32-bit integers horizontally, pack to *xmm1*.
VEX.NDS.128.66.0F38.WIG 01 /r VPHADDW *xmm1, xmm2, xmm3/m128*	RVM	V/V	AVX	Add 16-bit integers horizontally, pack to *xmm1*.
VEX.NDS.128.66.0F38.WIG 02 /r VPHADDD *xmm1, xmm2, xmm3/m128*	RVM	V/V	AVX	Add 32-bit integers horizontally, pack to *xmm1*.
VEX.NDS.256.66.0F38.WIG 01 /r VPHADDW *ymm1, ymm2, ymm3/m256*	RVM	V/V	AVX2	Add 16-bit signed integers horizontally, pack to *ymm1*.
VEX.NDS.256.66.0F38.WIG 02 /r VPHADDD *ymm1, ymm2, ymm3/m256*	RVM	V/V	AVX2	Add 32-bit signed integers horizontally, pack to *ymm1*.

NOTES:

1. See note in Section 2.4, "AVX and SSE Instruction Exception Specification" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A* and Section 22.25.3, "Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
RVM	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA

Description

(V)PHADDW adds two adjacent 16-bit signed integers horizontally from the source and destination operands and packs the 16-bit signed results to the destination operand (first operand). (V)PHADDD adds two adjacent 32-bit signed integers horizontally from the source and destination operands and packs the 32-bit signed results to the destination operand (first operand). When the source operand is a 128-bit memory operand, the operand must be aligned on a 16-byte boundary or a general-protection exception (#GP) will be generated.

Note that these instructions can operate on either unsigned or signed (two's complement notation) integers; however, it does not set bits in the EFLAGS register to indicate overflow and/or a carry. To prevent undetected overflow conditions, software must control the ranges of the values operated on.

Legacy SSE instructions: Both operands can be MMX registers. The second source operand can be an MMX register or a 64-bit memory location.

128-bit Legacy SSE version: The first source and destination operands are XMM registers. The second source operand can be an XMM register or a 128-bit memory location. Bits (MAXVL-1:128) of the corresponding YMM destination register remain unchanged.

In 64-bit mode, use the REX prefix to access additional registers.

VEX.128 encoded version: The first source and destination operands are XMM registers. The second source operand can be an XMM register or a 128-bit memory location. Bits (MAXVL-1:128) of the corresponding YMM register are zeroed.

VEX.256 encoded version: Horizontal addition of two adjacent data elements of the low 16-bytes of the first and second source operands are packed into the low 16-bytes of the destination operand. Horizontal addition of two adjacent data elements of the high 16-bytes of the first and second source operands are packed into the high 16-bytes of the destination operand. The first source and destination operands are YMM registers. The second source operand can be an YMM register or a 256-bit memory location.

Note: VEX.L must be 0, otherwise the instruction will #UD.

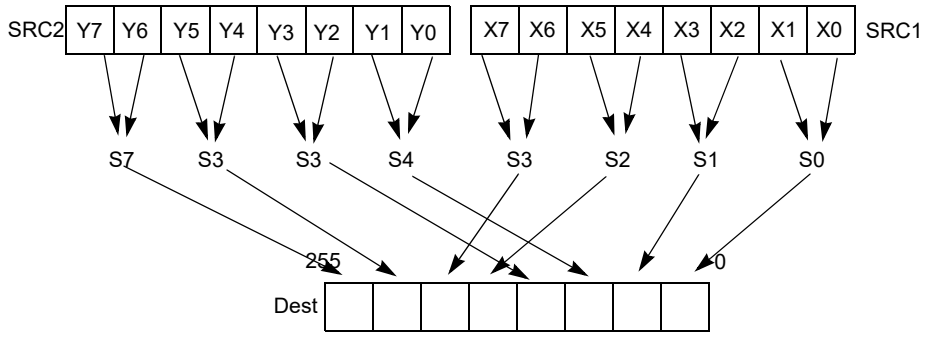


Figure 4-10. 256-bit VPHADDD Instruction Operation

Operation

PHADDW (with 64-bit operands)

```
mm1[15-0]  = mm1[31-16] + mm1[15-0];
mm1[31-16] = mm1[63-48] + mm1[47-32];
mm1[47-32] = mm2/m64[31-16] + mm2/m64[15-0];
mm1[63-48] = mm2/m64[63-48] + mm2/m64[47-32];
```

PHADDW (with 128-bit operands)

```
xmm1[15-0] = xmm1[31-16] + xmm1[15-0];
xmm1[31-16] = xmm1[63-48] + xmm1[47-32];
xmm1[47-32] = xmm1[95-80] + xmm1[79-64];
xmm1[63-48] = xmm1[127-112] + xmm1[111-96];
xmm1[79-64] = xmm2/m128[31-16] + xmm2/m128[15-0];
xmm1[95-80] = xmm2/m128[63-48] + xmm2/m128[47-32];
xmm1[111-96] = xmm2/m128[95-80] + xmm2/m128[79-64];
xmm1[127-112] = xmm2/m128[127-112] + xmm2/m128[111-96];
```

VPHADDW (VEX.128 encoded version)

```
DEST[15:0] ← SRC1[31:16] + SRC1[15:0]
DEST[31:16] ← SRC1[63:48] + SRC1[47:32]
DEST[47:32] ← SRC1[95:80] + SRC1[79:64]
DEST[63:48] ← SRC1[127:112] + SRC1[111:96]
DEST[79:64] ← SRC2[31:16] + SRC2[15:0]
DEST[95:80] ← SRC2[63:48] + SRC2[47:32]
DEST[111:96] ← SRC2[95:80] + SRC2[79:64]
DEST[127:112] ← SRC2[127:112] + SRC2[111:96]
DEST[MAXVL-1:128] ← 0
```

VPHADDW (VEX.256 encoded version)
DEST[15:0] ← SRC1[31:16] + SRC1[15:0]
DEST[31:16] ← SRC1[63:48] + SRC1[47:32]
DEST[47:32] ← SRC1[95:80] + SRC1[79:64]
DEST[63:48] ← SRC1[127:112] + SRC1[111:96]
DEST[79:64] ← SRC2[31:16] + SRC2[15:0]
DEST[95:80] ← SRC2[63:48] + SRC2[47:32]
DEST[111:96] ← SRC2[95:80] + SRC2[79:64]
DEST[127:112] ← SRC2[127:112] + SRC2[111:96]
DEST[143:128] ← SRC1[159:144] + SRC1[143:128]
DEST[159:144] ← SRC1[191:176] + SRC1[175:160]
DEST[175:160] ← SRC1[223:208] + SRC1[207:192]
DEST[191:176] ← SRC1[255:240] + SRC1[239:224]
DEST[207:192] ← SRC2[127:112] + SRC2[143:128]
DEST[223:208] ← SRC2[159:144] + SRC2[175:160]
DEST[239:224] ← SRC2[191:176] + SRC2[207:192]
DEST[255:240] ← SRC2[223:208] + SRC2[239:224]

PHADDD (with 64-bit operands)
mm1[31-0] = mm1[63-32] + mm1[31-0];
mm1[63-32] = mm2/m64[63-32] + mm2/m64[31-0];

PHADDD (with 128-bit operands)
xmm1[31-0] = xmm1[63-32] + xmm1[31-0];
xmm1[63-32] = xmm1[127-96] + xmm1[95-64];
xmm1[95-64] = xmm2/m128[63-32] + xmm2/m128[31-0];
xmm1[127-96] = xmm2/m128[127-96] + xmm2/m128[95-64];

VPHADDD (VEX.128 encoded version)
DEST[31-0] ← SRC1[63-32] + SRC1[31-0]
DEST[63-32] ← SRC1[127-96] + SRC1[95-64]
DEST[95-64] ← SRC2[63-32] + SRC2[31-0]
DEST[127-96] ← SRC2[127-96] + SRC2[95-64]
DEST[MAXVL-1:128] ← 0

VPHADDD (VEX.256 encoded version)
DEST[31-0] ← SRC1[63-32] + SRC1[31-0]
DEST[63-32] ← SRC1[127-96] + SRC1[95-64]
DEST[95-64] ← SRC2[63-32] + SRC2[31-0]
DEST[127-96] ← SRC2[127-96] + SRC2[95-64]
DEST[159-128] ← SRC1[191-160] + SRC1[159-128]
DEST[191-160] ← SRC1[255-224] + SRC1[223-192]
DEST[223-192] ← SRC2[191-160] + SRC2[159-128]
DEST[255-224] ← SRC2[255-224] + SRC2[223-192]

Intel C/C++ Compiler Intrinsic Equivalents

PHADDW: __m64 _mm_hadd_pi16 (__m64 a, __m64 b)
PHADDD: __m64 _mm_hadd_pi32 (__m64 a, __m64 b)
(V)PHADDW: __m128i _mm_hadd_epi16 (__m128i a, __m128i b)
(V)PHADDD: __m128i _mm_hadd_epi32 (__m128i a, __m128i b)
VPHADDW: __m256i _mm256_hadd_epi16 (__m256i a, __m256i b)
VPHADDD: __m256i _mm256_hadd_epi32 (__m256i a, __m256i b)

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Exceptions Type 4; additionally

#UD	If VEX.L = 1.

PHADDSW — Packed Horizontal Add and Saturate

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F 38 03 /r[1] PHADDSW *mm1, mm2/m64*	RM	V/V	SSSE3	Add 16-bit signed integers horizontally, pack saturated integers to *mm1*.
66 0F 38 03 /r PHADDSW *xmm1, xmm2/m128*	RM	V/V	SSSE3	Add 16-bit signed integers horizontally, pack saturated integers to *xmm1*.
VEX.NDS.128.66.0F38.WIG 03 /r VPHADDSW *xmm1, xmm2, xmm3/m128*	RVM	V/V	AVX	Add 16-bit signed integers horizontally, pack saturated integers to *xmm1*.
VEX.NDS.256.66.0F38.WIG 03 /r VPHADDSW *ymm1, ymm2, ymm3/m256*	RVM	V/V	AVX2	Add 16-bit signed integers horizontally, pack saturated integers to *ymm1*.

NOTES:

1. See note in Section 2.4, "AVX and SSE Instruction Exception Specification" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A* and Section 22.25.3, "Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
RVM	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA

Description

(V)PHADDSW adds two adjacent signed 16-bit integers horizontally from the source and destination operands and saturates the signed results; packs the signed, saturated 16-bit results to the destination operand (first operand) When the source operand is a 128-bit memory operand, the operand must be aligned on a 16-byte boundary or a general-protection exception (#GP) will be generated.

Legacy SSE version: Both operands can be MMX registers. The second source operand can be an MMX register or a 64-bit memory location.

128-bit Legacy SSE version: The first source and destination operands are XMM registers. The second source operand is an XMM register or a 128-bit memory location. Bits (MAXVL-1:128) of the corresponding YMM destination register remain unchanged.

In 64-bit mode, use the REX prefix to access additional registers.

VEX.128 encoded version: The first source and destination operands are XMM registers. The second source operand is an XMM register or a 128-bit memory location. Bits (MAXVL-1:128) of the destination YMM register are zeroed.

VEX.256 encoded version: The first source and destination operands are YMM registers. The second source operand can be an YMM register or a 256-bit memory location.

Note: VEX.L must be 0, otherwise the instruction will #UD.

Operation

PHADDSW (with 64-bit operands)

```
mm1[15-0]  = SaturateToSignedWord((mm1[31-16] + mm1[15-0]);
mm1[31-16] = SaturateToSignedWord(mm1[63-48] + mm1[47-32]);
mm1[47-32] = SaturateToSignedWord(mm2/m64[31-16] + mm2/m64[15-0]);
mm1[63-48] = SaturateToSignedWord(mm2/m64[63-48] + mm2/m64[47-32]);
```

PHADDSW (with 128-bit operands)

```
xmm1[15-0]= SaturateToSignedWord(xmm1[31-16] + xmm1[15-0]);
xmm1[31-16] = SaturateToSignedWord(xmm1[63-48] + xmm1[47-32]);
xmm1[47-32] = SaturateToSignedWord(xmm1[95-80] + xmm1[79-64]);
xmm1[63-48] = SaturateToSignedWord(xmm1[127-112] + xmm1[111-96]);
xmm1[79-64] = SaturateToSignedWord(xmm2/m128[31-16] + xmm2/m128[15-0]);
xmm1[95-80] = SaturateToSignedWord(xmm2/m128[63-48] + xmm2/m128[47-32]);
xmm1[111-96] = SaturateToSignedWord(xmm2/m128[95-80] + xmm2/m128[79-64]);
xmm1[127-112] = SaturateToSignedWord(xmm2/m128[127-112] + xmm2/m128[111-96]);
```

VPHADDSW (VEX.128 encoded version)

```
DEST[15:0]= SaturateToSignedWord(SRC1[31:16] + SRC1[15:0])
DEST[31:16] = SaturateToSignedWord(SRC1[63:48] + SRC1[47:32])
DEST[47:32] = SaturateToSignedWord(SRC1[95:80] + SRC1[79:64])
DEST[63:48] = SaturateToSignedWord(SRC1[127:112] + SRC1[111:96])
DEST[79:64] = SaturateToSignedWord(SRC2[31:16] + SRC2[15:0])
DEST[95:80] = SaturateToSignedWord(SRC2[63:48] + SRC2[47:32])
DEST[111:96] = SaturateToSignedWord(SRC2[95:80] + SRC2[79:64])
DEST[127:112] = SaturateToSignedWord(SRC2[127:112] + SRC2[111:96])
DEST[MAXVL-1:128] ← 0
```

VPHADDSW (VEX.256 encoded version)

```
DEST[15:0]= SaturateToSignedWord(SRC1[31:16] + SRC1[15:0])
DEST[31:16] = SaturateToSignedWord(SRC1[63:48] + SRC1[47:32])
DEST[47:32] = SaturateToSignedWord(SRC1[95:80] + SRC1[79:64])
DEST[63:48] = SaturateToSignedWord(SRC1[127:112] + SRC1[111:96])
DEST[79:64] = SaturateToSignedWord(SRC2[31:16] + SRC2[15:0])
DEST[95:80] = SaturateToSignedWord(SRC2[63:48] + SRC2[47:32])
DEST[111:96] = SaturateToSignedWord(SRC2[95:80] + SRC2[79:64])
DEST[127:112] = SaturateToSignedWord(SRC2[127:112] + SRC2[111:96])
DEST[143:128]= SaturateToSignedWord(SRC1[159:144] + SRC1[143:128])
DEST[159:144] = SaturateToSignedWord(SRC1[191:176] + SRC1[175:160])
DEST[175:160] = SaturateToSignedWord( SRC1[223:208] + SRC1[207:192])
DEST[191:176] = SaturateToSignedWord(SRC1[255:240] + SRC1[239:224])
DEST[207:192] = SaturateToSignedWord(SRC2[127:112] + SRC2[143:128])
DEST[223:208] = SaturateToSignedWord(SRC2[159:144] + SRC2[175:160])
DEST[239:224] = SaturateToSignedWord(SRC2[191-160] + SRC2[159-128])
DEST[255:240] = SaturateToSignedWord(SRC2[255:240] + SRC2[239:224])
```

Intel C/C++ Compiler Intrinsic Equivalent

PHADDSW: __m64 _mm_hadds_pi16 (__m64 a, __m64 b)

(V)PHADDSW: __m128i _mm_hadds_epi16 (__m128i a, __m128i b)

VPHADDSW: __m256i _mm256_hadds_epi16 (__m256i a, __m256i b)

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Exceptions Type 4; additionally

#UD	If VEX.L = 1.

PHMINPOSUW — Packed Horizontal Word Minimum

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 38 41 /r PHMINPOSUW *xmm1, xmm2/m128*	RM	V/V	SSE4_1	Find the minimum unsigned word in *xmm2/m128* and place its value in the low word of *xmm1* and its index in the second-lowest word of *xmm1*.
VEX.128.66.0F38.WIG 41 /r VPHMINPOSUW *xmm1, xmm2/m128*	RM	V/V	AVX	Find the minimum unsigned word in *xmm2/m128* and place its value in the low word of xmm1 and its index in the second-lowest word of *xmm1*.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

Description

Determine the minimum unsigned word value in the source operand (second operand) and place the unsigned word in the low word (bits 0-15) of the destination operand (first operand). The word index of the minimum value is stored in bits 16-18 of the destination operand. The remaining upper bits of the destination are set to zero.

128-bit Legacy SSE version: Bits (MAXVL-1:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: Bits (MAXVL-1:128) of the destination YMM register are zeroed. VEX.vvvv is reserved and must be 1111b, VEX.L must be 0, otherwise the instruction will #UD.

Operation

PHMINPOSUW (128-bit Legacy SSE version)

```
INDEX ← 0;
MIN ← SRC[15:0]
IF (SRC[31:16] < MIN)
    THEN INDEX ← 1; MIN ← SRC[31:16]; FI;
IF (SRC[47:32] < MIN)
    THEN INDEX ← 2; MIN ← SRC[47:32]; FI;
* Repeat operation for words 3 through 6
IF (SRC[127:112] < MIN)
    THEN INDEX ← 7; MIN ← SRC[127:112]; FI;
DEST[15:0] ← MIN;
DEST[18:16] ← INDEX;
DEST[127:19] ← 00000000000000000000000000000H;
```

VPHMINPOSUW (VEX.128 encoded version)

```
INDEX ← 0
MIN ← SRC[15:0]
IF (SRC[31:16] < MIN) THEN INDEX ← 1; MIN ← SRC[31:16]
IF (SRC[47:32] < MIN) THEN INDEX ← 2; MIN ← SRC[47:32]
* Repeat operation for words 3 through 6
IF (SRC[127:112] < MIN) THEN INDEX ← 7; MIN ← SRC[127:112]
DEST[15:0] ← MIN
DEST[18:16] ← INDEX
DEST[127:19] ← 00000000000000000000000000000H
DEST[MAXVL-1:128] ← 0
```

Intel C/C++ Compiler Intrinsic Equivalent

PHMINPOSUW: __m128i _mm_minpos_epu16(__m128i packed_words);

Flags Affected

None.

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Exceptions Type 4; additionally

#UD	If VEX.L = 1.
	If VEX.vvvv ≠ 1111B.

PHSUBW/PHSUBD — Packed Horizontal Subtract

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F 38 05 /r[1] PHSUBW *mm1, mm2/m64*	RM	V/V	SSSE3	Subtract 16-bit signed integers horizontally, pack to *mm1*.
66 0F 38 05 /r PHSUBW *xmm1, xmm2/m128*	RM	V/V	SSSE3	Subtract 16-bit signed integers horizontally, pack to *xmm1*.
NP 0F 38 06 /r PHSUBD *mm1, mm2/m64*	RM	V/V	SSSE3	Subtract 32-bit signed integers horizontally, pack to *mm1*.
66 0F 38 06 /r PHSUBD *xmm1, xmm2/m128*	RM	V/V	SSSE3	Subtract 32-bit signed integers horizontally, pack to *xmm1*.
VEX.NDS.128.66.0F38.WIG 05 /r VPHSUBW *xmm1, xmm2, xmm3/m128*	RVM	V/V	AVX	Subtract 16-bit signed integers horizontally, pack to *xmm1*.
VEX.NDS.128.66.0F38.WIG 06 /r VPHSUBD *xmm1, xmm2, xmm3/m128*	RVM	V/V	AVX	Subtract 32-bit signed integers horizontally, pack to *xmm1*.
VEX.NDS.256.66.0F38.WIG 05 /r VPHSUBW *ymm1, ymm2, ymm3/m256*	RVM	V/V	AVX2	Subtract 16-bit signed integers horizontally, pack to *ymm1*.
VEX.NDS.256.66.0F38.WIG 06 /r VPHSUBD *ymm1, ymm2, ymm3/m256*	RVM	V/V	AVX2	Subtract 32-bit signed integers horizontally, pack to *ymm1*.

NOTES:

1. See note in Section 2.4, "AVX and SSE Instruction Exception Specification" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A* and Section 22.25.3, "Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
RVM	ModRM:reg (r, w)	VEX.vvvv (r)	ModRM:r/m (r)	NA

Description

(V)PHSUBW performs horizontal subtraction on each adjacent pair of 16-bit signed integers by subtracting the most significant word from the least significant word of each pair in the source and destination operands, and packs the signed 16-bit results to the destination operand (first operand). (V)PHSUBD performs horizontal subtraction on each adjacent pair of 32-bit signed integers by subtracting the most significant doubleword from the least significant doubleword of each pair, and packs the signed 32-bit result to the destination operand. When the source operand is a 128-bit memory operand, the operand must be aligned on a 16-byte boundary or a general-protection exception (#GP) will be generated.

Legacy SSE version: Both operands can be MMX registers. The second source operand can be an MMX register or a 64-bit memory location.

128-bit Legacy SSE version: The first source and destination operands are XMM registers. The second source operand is an XMM register or a 128-bit memory location. Bits (MAXVL-1:128) of the corresponding YMM destination register remain unchanged.

In 64-bit mode, use the REX prefix to access additional registers.

VEX.128 encoded version: The first source and destination operands are XMM registers. The second source operand is an XMM register or a 128-bit memory location. Bits (MAXVL-1:128) of the destination YMM register are zeroed.

VEX.256 encoded version: The first source and destination operands are YMM registers. The second source operand can be an YMM register or a 256-bit memory location.

Note: VEX.L must be 0, otherwise the instruction will #UD.

Operation

PHSUBW (with 64-bit operands)

```
mm1[15-0] = mm1[15-0] - mm1[31-16];
mm1[31-16] = mm1[47-32] - mm1[63-48];
mm1[47-32] = mm2/m64[15-0] - mm2/m64[31-16];
mm1[63-48] = mm2/m64[47-32] - mm2/m64[63-48];
```

PHSUBW (with 128-bit operands)

```
xmm1[15-0] = xmm1[15-0] - xmm1[31-16];
xmm1[31-16] = xmm1[47-32] - xmm1[63-48];
xmm1[47-32] = xmm1[79-64] - xmm1[95-80];
xmm1[63-48] = xmm1[111-96] - xmm1[127-112];
xmm1[79-64] = xmm2/m128[15-0] - xmm2/m128[31-16];
xmm1[95-80] = xmm2/m128[47-32] - xmm2/m128[63-48];
xmm1[111-96] = xmm2/m128[79-64] - xmm2/m128[95-80];
xmm1[127-112] = xmm2/m128[111-96] - xmm2/m128[127-112];
```

VPHSUBW (VEX.128 encoded version)

```
DEST[15:0] ← SRC1[15:0] - SRC1[31:16]
DEST[31:16] ← SRC1[47:32] - SRC1[63:48]
DEST[47:32] ← SRC1[79:64] - SRC1[95:80]
DEST[63:48] ← SRC1[111:96] - SRC1[127:112]
DEST[79:64] ← SRC2[15:0] - SRC2[31:16]
DEST[95:80] ← SRC2[47:32] - SRC2[63:48]
DEST[111:96] ← SRC2[79:64] - SRC2[95:80]
DEST[127:112] ← SRC2[111:96] - SRC2[127:112]
DEST[MAXVL-1:128] ← 0
```

VPHSUBW (VEX.256 encoded version)

```
DEST[15:0] ← SRC1[15:0] - SRC1[31:16]
DEST[31:16] ← SRC1[47:32] - SRC1[63:48]
DEST[47:32] ← SRC1[79:64] - SRC1[95:80]
DEST[63:48] ← SRC1[111:96] - SRC1[127:112]
DEST[79:64] ← SRC2[15:0] - SRC2[31:16]
DEST[95:80] ← SRC2[47:32] - SRC2[63:48]
DEST[111:96] ← SRC2[79:64] - SRC2[95:80]
DEST[127:112] ← SRC2[111:96] - SRC2[127:112]
DEST[143:128] ← SRC1[143:128] - SRC1[159:144]
DEST[159:144] ← SRC1[175:160] - SRC1[191:176]
DEST[175:160] ← SRC1[207:192] - SRC1[223:208]
DEST[191:176] ← SRC1[239:224] - SRC1[255:240]
DEST[207:192] ← SRC2[143:128] - SRC2[159:144]
DEST[223:208] ← SRC2[175:160] - SRC2[191:176]
DEST[239:224] ← SRC2[207:192] - SRC2[223:208]
DEST[255:240] ← SRC2[239:224] - SRC2[255:240]
```

PHSUBD (with 64-bit operands)

```
mm1[31-0] = mm1[31-0] - mm1[63-32];
mm1[63-32] = mm2/m64[31-0] - mm2/m64[63-32];
```

PHSUBD (with 128-bit operands)

```
xmm1[31-0] = xmm1[31-0] - xmm1[63-32];
xmm1[63-32] = xmm1[95-64] - xmm1[127-96];
xmm1[95-64] = xmm2/m128[31-0] - xmm2/m128[63-32];
xmm1[127-96] = xmm2/m128[95-64] - xmm2/m128[127-96];
```

VPHSUBD (VEX.128 encoded version)

```
DEST[31-0] ← SRC1[31-0] - SRC1[63-32]
DEST[63-32] ← SRC1[95-64] - SRC1[127-96]
DEST[95-64] ← SRC2[31-0] - SRC2[63-32]
DEST[127-96] ← SRC2[95-64] - SRC2[127-96]
DEST[MAXVL-1:128] ← 0
```

VPHSUBD (VEX.256 encoded version)

```
DEST[31:0] ← SRC1[31:0] - SRC1[63:32]
DEST[63:32] ← SRC1[95:64] - SRC1[127:96]
DEST[95:64] ← SRC2[31:0] - SRC2[63:32]
DEST[127:96] ← SRC2[95:64] - SRC2[127:96]
DEST[159:128] ← SRC1[159:128] - SRC1[191:160]
DEST[191:160] ← SRC1[223:192] - SRC1[255:224]
DEST[223:192] ← SRC2[159:128] - SRC2[191:160]
DEST[255:224] ← SRC2[223:192] - SRC2[255:224]
```

Intel C/C++ Compiler Intrinsic Equivalents

PHSUBW: __m64 _mm_hsub_pi16 (__m64 a, __m64 b)

PHSUBD: __m64 _mm_hsub_pi32 (__m64 a, __m64 b)

(V)PHSUBW: __m128i _mm_hsub_epi16 (__m128i a, __m128i b)

(V)PHSUBD: __m128i _mm_hsub_epi32 (__m128i a, __m128i b)

VPHSUBW: __m256i _mm256_hsub_epi16 (__m256i a, __m256i b)

VPHSUBD: __m256i _mm256_hsub_epi32 (__m256i a, __m256i b)

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Exceptions Type 4; additionally

#UD	If VEX.L = 1.

PHSUBSW — Packed Horizontal Subtract and Saturate

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F 38 07 /r[1] PHSUBSW *mm1, mm2/m64*	RM	V/V	SSSE3	Subtract 16-bit signed integer horizontally, pack saturated integers to *mm1.*
66 0F 38 07 /r PHSUBSW *xmm1, xmm2/m128*	RM	V/V	SSSE3	Subtract 16-bit signed integer horizontally, pack saturated integers to *xmm1.*
VEX.NDS.128.66.0F38.WIG 07 /r VPHSUBSW *xmm1, xmm2, xmm3/m128*	RVM	V/V	AVX	Subtract 16-bit signed integer horizontally, pack saturated integers to *xmm1.*
VEX.NDS.256.66.0F38.WIG 07 /r VPHSUBSW *ymm1, ymm2, ymm3/m256*	RVM	V/V	AVX2	Subtract 16-bit signed integer horizontally, pack saturated integers to *ymm1.*

NOTES:

1. See note in Section 2.4, "AVX and SSE Instruction Exception Specification" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A* and Section 22.25.3, "Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
RVM	ModRM:reg (r, w)	VEX.vvvv (r)	ModRM:r/m (r)	NA

Description

(V)PHSUBSW performs horizontal subtraction on each adjacent pair of 16-bit signed integers by subtracting the most significant word from the least significant word of each pair in the source and destination operands. The signed, saturated 16-bit results are packed to the destination operand (first operand). When the source operand is a 128-bit memory operand, the operand must be aligned on a 16-byte boundary or a general-protection exception (#GP) will be generated.

Legacy SSE version: Both operands can be MMX registers. The second source operand can be an MMX register or a 64-bit memory location.

128-bit Legacy SSE version: The first source and destination operands are XMM registers. The second source operand is an XMM register or a 128-bit memory location. Bits (MAXVL-1:128) of the corresponding YMM destination register remain unchanged.

In 64-bit mode, use the REX prefix to access additional registers.

VEX.128 encoded version: The first source and destination operands are XMM registers. The second source operand is an XMM register or a 128-bit memory location. Bits (MAXVL-1:128) of the destination YMM register are zeroed.

VEX.256 encoded version: The first source and destination operands are YMM registers. The second source operand can be an YMM register or a 256-bit memory location.

Note: VEX.L must be 0, otherwise the instruction will #UD.

Operation

PHSUBSW (with 64-bit operands)

```
mm1[15-0] = SaturateToSignedWord(mm1[15-0] - mm1[31-16]);
mm1[31-16] = SaturateToSignedWord(mm1[47-32] - mm1[63-48]);
mm1[47-32] = SaturateToSignedWord(mm2/m64[15-0] - mm2/m64[31-16]);
mm1[63-48] = SaturateToSignedWord(mm2/m64[47-32] - mm2/m64[63-48]);
```

PHSUBSW (with 128-bit operands)

```
xmm1[15-0] = SaturateToSignedWord(xmm1[15-0] - xmm1[31-16]);
xmm1[31-16] = SaturateToSignedWord(xmm1[47-32] - xmm1[63-48]);
xmm1[47-32] = SaturateToSignedWord(xmm1[79-64] - xmm1[95-80]);
xmm1[63-48] = SaturateToSignedWord(xmm1[111-96] - xmm1[127-112]);
xmm1[79-64] = SaturateToSignedWord(xmm2/m128[15-0] - xmm2/m128[31-16]);
xmm1[95-80] =SaturateToSignedWord(xmm2/m128[47-32] - xmm2/m128[63-48]);
xmm1[111-96] =SaturateToSignedWord(xmm2/m128[79-64] - xmm2/m128[95-80]);
xmm1[127-112]= SaturateToSignedWord(xmm2/m128[111-96] - xmm2/m128[127-112]);
```

VPHSUBSW (VEX.128 encoded version)

```
DEST[15:0]= SaturateToSignedWord(SRC1[15:0] - SRC1[31:16])
DEST[31:16] = SaturateToSignedWord(SRC1[47:32] - SRC1[63:48])
DEST[47:32] = SaturateToSignedWord(SRC1[79:64] - SRC1[95:80])
DEST[63:48] = SaturateToSignedWord(SRC1[111:96] - SRC1[127:112])
DEST[79:64] = SaturateToSignedWord(SRC2[15:0] - SRC2[31:16])
DEST[95:80] = SaturateToSignedWord(SRC2[47:32] - SRC2[63:48])
DEST[111:96] = SaturateToSignedWord(SRC2[79:64] - SRC2[95:80])
DEST[127:112] = SaturateToSignedWord(SRC2[111:96] - SRC2[127:112])
DEST[MAXVL-1:128] ← 0
```

VPHSUBSW (VEX.256 encoded version)

```
DEST[15:0]= SaturateToSignedWord(SRC1[15:0] - SRC1[31:16])
DEST[31:16] = SaturateToSignedWord(SRC1[47:32] - SRC1[63:48])
DEST[47:32] = SaturateToSignedWord(SRC1[79:64] - SRC1[95:80])
DEST[63:48] = SaturateToSignedWord(SRC1[111:96] - SRC1[127:112])
DEST[79:64] = SaturateToSignedWord(SRC2[15:0] - SRC2[31:16])
DEST[95:80] = SaturateToSignedWord(SRC2[47:32] - SRC2[63:48])
DEST[111:96] = SaturateToSignedWord(SRC2[79:64] - SRC2[95:80])
DEST[127:112] = SaturateToSignedWord(SRC2[111:96] - SRC2[127:112])
DEST[143:128]= SaturateToSignedWord(SRC1[143:128] - SRC1[159:144])
DEST[159:144] = SaturateToSignedWord(SRC1[175:160] - SRC1[191:176])
DEST[175:160] = SaturateToSignedWord(SRC1[207:192] - SRC1[223:208])
DEST[191:176] = SaturateToSignedWord(SRC1[239:224] - SRC1[255:240])
DEST[207:192] = SaturateToSignedWord(SRC2[143:128] - SRC2[159:144])
DEST[223:208] = SaturateToSignedWord(SRC2[175:160] - SRC2[191:176])
DEST[239:224] = SaturateToSignedWord(SRC2[207:192] - SRC2[223:208])
DEST[255:240] = SaturateToSignedWord(SRC2[239:224] - SRC2[255:240])
```

Intel C/C++ Compiler Intrinsic Equivalent

PHSUBSW: __m64 _mm_hsubs_pi16 (__m64 a, __m64 b)

(V)PHSUBSW: __m128i _mm_hsubs_epi16 (__m128i a, __m128i b)

VPHSUBSW: __m256i _mm256_hsubs_epi16 (__m256i a, __m256i b)

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Exceptions Type 4; additionally

#UD	If VEX.L = 1.

PINSRB/PINSRD/PINSRQ — Insert Byte/Dword/Qword

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 3A 20 /r ib PINSRB *xmm1, r32/m8, imm8*	A	V/V	SSE4_1	Insert a byte integer value from *r32/m8* into *xmm1* at the destination element in *xmm1* specified by *imm8.*
66 0F 3A 22 /r ib PINSRD *xmm1, r/m32, imm8*	A	V/V	SSE4_1	Insert a dword integer value from *r/m32* into the *xmm1* at the destination element specified by *imm8.*
66 REX.W 0F 3A 22 /r ib PINSRQ *xmm1, r/m64, imm8*	A	V/N. E.	SSE4_1	Insert a qword integer value from *r/m64 i*nto the *xmm1* at the destination element specified by *imm8.*
VEX.NDS.128.66.0F3A.W0 20 /r ib VPINSRB *xmm1, xmm2, r32/m8, imm8*	B	V[1]/V	AVX	Merge a byte integer value from *r32/m8* and rest from *xmm2* into *xmm1* at the byte offset in *imm8*.
VEX.NDS.128.66.0F3A.W0 22 /r ib VPINSRD *xmm1, xmm2, r/m32, imm8*	B	V/V	AVX	Insert a dword integer value from *r32/m32* and rest from *xmm2* into *xmm1* at the dword offset in *imm8*.
VEX.NDS.128.66.0F3A.W1 22 /r ib VPINSRQ *xmm1, xmm2, r/m64, imm8*	B	V/I[2]	AVX	Insert a qword integer value from *r64/m64* and rest from *xmm2* into *xmm1* at the qword offset in *imm8*.
EVEX.NDS.128.66.0F3A.WIG 20 /r ib VPINSRB xmm1, xmm2, r32/m8, imm8	C	V/V	AVX512BW	Merge a byte integer value from r32/m8 and rest from xmm2 into xmm1 at the byte offset in imm8.
EVEX.NDS.128.66.0F3A.W0 22 /r ib VPINSRD xmm1, xmm2, r32/m32, imm8	C	V/V	AVX512DQ	Insert a dword integer value from r32/m32 and rest from xmm2 into xmm1 at the dword offset in imm8.
EVEX.NDS.128.66.0F3A.W1 22 /r ib VPINSRQ xmm1, xmm2, r64/m64, imm8	C	V/N.E.[2]	AVX512DQ	Insert a qword integer value from r64/m64 and rest from xmm2 into xmm1 at the qword offset in imm8.

NOTES:

1. In 64-bit mode, VEX.W1 is ignored for VPINSRB (similar to legacy REX.W=1 prefix with PINSRB).
2. VEX.W/EVEX.W in non-64 bit is ignored; the instructions behaves as if the W0 version is used.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	NA	ModRM:reg (w)	ModRM:r/m (r)	imm8	NA
B	NA	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	imm8
C	Tuple1 Scalar	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	Imm8

Description

Copies a byte/dword/qword from the source operand (second operand) and inserts it in the destination operand (first operand) at the location specified with the count operand (third operand). (The other elements in the destination register are left untouched.) The source operand can be a general-purpose register or a memory location. (When the source operand is a general-purpose register, PINSRB copies the low byte of the register.) The destination operand is an XMM register. The count operand is an 8-bit immediate. When specifying a qword[dword, byte] location in an XMM register, the [2, 4] least-significant bit(s) of the count operand specify the location.

In 64-bit mode and not encoded with VEX/EVEX, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15, R8-15). Use of REX.W permits the use of 64 bit general purpose registers.

128-bit Legacy SSE version: Bits (MAXVL-1:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: Bits (MAXVL-1:128) of the destination register are zeroed. VEX.L must be 0, otherwise the instruction will #UD. Attempt to execute VPINSRQ in non-64-bit mode will cause #UD.

EVEX.128 encoded version: Bits (MAXVL-1:128) of the destination register are zeroed. EVEX.L'L must be 0, otherwise the instruction will #UD.

Operation

```
CASE OF
    PINSRB:  SEL ← COUNT[3:0];
             MASK ← (0FFH << (SEL * 8));
             TEMP ← (((SRC[7:0] << (SEL *8)) AND MASK);
    PINSRD:  SEL ← COUNT[1:0];
             MASK ← (0FFFFFFFFH << (SEL * 32));
             TEMP ← (((SRC << (SEL *32)) AND MASK)  ;
    PINSRQ:  SEL ← COUNT[0]
             MASK ← (0FFFFFFFFFFFFFFFFH << (SEL * 64));
             TEMP ← (((SRC << (SEL *64)) AND MASK)  ;
ESAC;
        DEST ← ((DEST AND NOT MASK) OR TEMP);
```

VPINSRB (VEX/EVEX encoded version)

```
SEL ← imm8[3:0]
DEST[127:0] ← write_b_element(SEL, SRC2, SRC1)
DEST[MAXVL-1:128] ← 0
```

VPINSRD (VEX/EVEX encoded version)

```
SEL ← imm8[1:0]
DEST[127:0] ← write_d_element(SEL, SRC2, SRC1)
DEST[MAXVL-1:128] ← 0
```

VPINSRQ (VEX/EVEX encoded version)

```
SEL ← imm8[0]
DEST[127:0] ← write_q_element(SEL, SRC2, SRC1)
DEST[MAXVL-1:128] ← 0
```

Intel C/C++ Compiler Intrinsic Equivalent

PINSRB: __m128i _mm_insert_epi8 (__m128i s1, int s2, const int ndx);

PINSRD: __m128i _mm_insert_epi32 (__m128i s2, int s, const int ndx);

PINSRQ: __m128i _mm_insert_epi64(__m128i s2, __int64 s, const int ndx);

Flags Affected

None.

SIMD Floating-Point Exceptions

None.

Other Exceptions

EVEX-encoded instruction, see Exceptions Type 5;

EVEX-encoded instruction, see Exceptions Type E9NF.

#UD	If VEX.L = 1 or EVEX.L'L > 0.

PINSRW—Insert Word

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F C4 /*r* ib[1] PINSRW *mm, r32/m16, imm8*	A	V/V	SSE	Insert the low word from *r32* or from *m16* into *mm* at the word position specified by *imm8*.
66 0F C4 /*r* ib PINSRW *xmm, r32/m16, imm8*	A	V/V	SSE2	Move the low word of *r32* or from *m16* into *xmm* at the word position specified by *imm8*.
VEX.NDS.128.66.0F.W0 C4 /r ib VPINSRW *xmm1, xmm2, r32/m16, imm8*	B	V[2]/V	AVX	Insert a word integer value from *r32/m16* and rest from *xmm2* into *xmm1* at the word offset in *imm8*.
EVEX.NDS.128.66.0F.WIG C4 /r ib VPINSRW xmm1, xmm2, r32/m16, imm8	C	V/V	AVX512BW	Insert a word integer value from r32/m16 and rest from xmm2 into xmm1 at the word offset in imm8.

NOTES:

1. See note in Section 2.4, "AVX and SSE Instruction Exception Specification" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A* and Section 22.25.3, "Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*.

2. In 64-bit mode, VEX.W1 is ignored for VPINSRW (similar to legacy REX.W=1 prefix in PINSRW).

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	NA	ModRM:reg (w)	ModRM:r/m (r)	imm8	NA
B	NA	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	imm8
C	Tuple1 Scalar	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	Imm8

Description

Copies a word from the source operand (second operand) and inserts it in the destination operand (first operand) at the location specified with the count operand (third operand). (The other words in the destination register are left untouched.) The source operand can be a general-purpose register or a 16-bit memory location. (When the source operand is a general-purpose register, the low word of the register is copied.) The destination operand can be an MMX technology register or an XMM register. The count operand is an 8-bit immediate. When specifying a word location in an MMX technology register, the 2 least-significant bits of the count operand specify the location; for an XMM register, the 3 least-significant bits specify the location.

In 64-bit mode and not encoded with VEX/EVEX, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15, R8-15).

128-bit Legacy SSE version: Bits (MAXVL-1:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: Bits (MAXVL-1:128) of the destination YMM register are zeroed. VEX.L must be 0, otherwise the instruction will #UD.

EVEX.128 encoded version: Bits (MAXVL-1:128) of the destination register are zeroed. EVEX.L'L must be 0, otherwise the instruction will #UD.

Operation

PINSRW (with 64-bit source operand)

```
SEL ← COUNT AND 3H;
    CASE (Determine word position) OF
        SEL ← 0:    MASK ← 000000000000FFFFH;
        SEL ← 1:    MASK ← 00000000FFFF0000H;
        SEL ← 2:    MASK ← 0000FFFF00000000H;
        SEL ← 3:    MASK ← FFFF000000000000H;
DEST ← (DEST AND NOT MASK) OR (((SRC << (SEL * 16)) AND MASK);
```

PINSRW (with 128-bit source operand)

```
SEL ← COUNT AND 7H;
    CASE (Determine word position) OF
        SEL ← 0:    MASK ← 00000000000000000000000000000FFFFH;
        SEL ← 1:    MASK ← 00000000000000000000000FFFF0000H;
        SEL ← 2:    MASK ← 000000000000000000000FFFF00000000H;
        SEL ← 3:    MASK ← 0000000000000000FFFF000000000000H;
        SEL ← 4:    MASK ← 00000000000FFFF0000000000000000H;
        SEL ← 5:    MASK ← 000000000FFFF00000000000000000000H;
        SEL ← 6:    MASK ← 0000FFFF0000000000000000000000000H;
        SEL ← 7:    MASK ← FFFF00000000000000000000000000000H;
DEST ← (DEST AND NOT MASK) OR (((SRC << (SEL * 16)) AND MASK);
```

VPINSRW (VEX/EVEX encoded version)

```
SEL ← imm8[2:0]
DEST[127:0] ← write_w_element(SEL, SRC2, SRC1)
DEST[MAXVL-1:128] ← 0
```

Intel C/C++ Compiler Intrinsic Equivalent

PINSRW: __m64 _mm_insert_pi16 (__m64 a, int d, int n)

PINSRW: __m128i _mm_insert_epi16 (__m128i a, int b, int imm)

Flags Affected

None.

Numeric Exceptions

None.

Other Exceptions

EVEX-encoded instruction, see Exceptions Type 5;

EVEX-encoded instruction, see Exceptions Type E9NF.

#UD	If VEX.L = 1 or EVEX.L'L > 0.

PMADDUBSW — Multiply and Add Packed Signed and Unsigned Bytes

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F 38 04 /r[1] PMADDUBSW *mm1, mm2/m64*	A	V/V	SSSE3	Multiply signed and unsigned bytes, add horizontal pair of signed words, pack saturated signed-words to *mm1*.
66 0F 38 04 /r PMADDUBSW *xmm1, xmm2/m128*	A	V/V	SSSE3	Multiply signed and unsigned bytes, add horizontal pair of signed words, pack saturated signed-words to *xmm1*.
VEX.NDS.128.66.0F38.WIG 04 /r VPMADDUBSW *xmm1, xmm2, xmm3/m128*	B	V/V	AVX	Multiply signed and unsigned bytes, add horizontal pair of signed words, pack saturated signed-words to *xmm1*.
VEX.NDS.256.66.0F38.WIG 04 /r VPMADDUBSW *ymm1, ymm2, ymm3/m256*	B	V/V	AVX2	Multiply signed and unsigned bytes, add horizontal pair of signed words, pack saturated signed-words to *ymm1*.
EVEX.NDS.128.66.0F38.WIG 04 /r VPMADDUBSW xmm1 {k1}{z}, xmm2, xmm3/m128	C	V/V	AVX512VL AVX512BW	Multiply signed and unsigned bytes, add horizontal pair of signed words, pack saturated signed-words to xmm1 under writemask k1.
EVEX.NDS.256.66.0F38.WIG 04 /r VPMADDUBSW ymm1 {k1}{z}, ymm2, ymm3/m256	C	V/V	AVX512VL AVX512BW	Multiply signed and unsigned bytes, add horizontal pair of signed words, pack saturated signed-words to ymm1 under writemask k1.
EVEX.NDS.512.66.0F38.WIG 04 /r VPMADDUBSW zmm1 {k1}{z}, zmm2, zmm3/m512	C	V/V	AVX512BW	Multiply signed and unsigned bytes, add horizontal pair of signed words, pack saturated signed-words to zmm1 under writemask k1.

NOTES:

1. See note in Section 2.4, "AVX and SSE Instruction Exception Specification" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A* and Section 22.25.3, "Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	NA	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
B	NA	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA
C	Full Vector Mem	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA

Description

(V)PMADDUBSW multiplies vertically each unsigned byte of the destination operand (first operand) with the corresponding signed byte of the source operand (second operand), producing intermediate signed 16-bit integers. Each adjacent pair of signed words is added and the saturated result is packed to the destination operand. For example, the lowest-order bytes (bits 7-0) in the source and destination operands are multiplied and the intermediate signed word result is added with the corresponding intermediate result from the 2nd lowest-order bytes (bits 15-8) of the operands; the sign-saturated result is stored in the lowest word of the destination register (15-0). The same operation is performed on the other pairs of adjacent bytes. Both operands can be MMX register or XMM registers. When the source operand is a 128-bit memory operand, the operand must be aligned on a 16-byte boundary or a general-protection exception (#GP) will be generated.

In 64-bit mode and not encoded with VEX/EVEX, use the REX prefix to access XMM8-XMM15.

128-bit Legacy SSE version: The first source and destination operands are XMM registers. The second source operand is an XMM register or a 128-bit memory location. Bits (MAXVL-1:128) of the corresponding destination register remain unchanged.

VEX.128 and EVEX.128 encoded versions: The first source and destination operands are XMM registers. The second source operand is an XMM register or a 128-bit memory location. Bits (MAXVL-1:128) of the corresponding destination register are zeroed.

VEX.256 and EVEX.256 encoded versions: The second source operand can be an YMM register or a 256-bit memory location. The first source and destination operands are YMM registers. Bits (MAXVL-1:256) of the corresponding ZMM register are zeroed.

EVEX.512 encoded version: The second source operand can be an ZMM register or a 512-bit memory location. The first source and destination operands are ZMM registers.

Operation

PMADDUBSW (with 64 bit operands)

```
DEST[15-0] = SaturateToSignedWord(SRC[15-8]*DEST[15-8]+SRC[7-0]*DEST[7-0]);
DEST[31-16] = SaturateToSignedWord(SRC[31-24]*DEST[31-24]+SRC[23-16]*DEST[23-16]);
DEST[47-32] = SaturateToSignedWord(SRC[47-40]*DEST[47-40]+SRC[39-32]*DEST[39-32]);
DEST[63-48] = SaturateToSignedWord(SRC[63-56]*DEST[63-56]+SRC[55-48]*DEST[55-48]);
```

PMADDUBSW (with 128 bit operands)

```
DEST[15-0] = SaturateToSignedWord(SRC[15-8]* DEST[15-8]+SRC[7-0]*DEST[7-0]);
// Repeat operation for 2nd through 7th word
SRC1/DEST[127-112] = SaturateToSignedWord(SRC[127-120]*DEST[127-120]+ SRC[119-112]* DEST[119-112]);
```

VPMADDUBSW (VEX.128 encoded version)

```
DEST[15:0] ← SaturateToSignedWord(SRC2[15:8]* SRC1[15:8]+SRC2[7:0]*SRC1[7:0])
// Repeat operation for 2nd through 7th word
DEST[127:112] ← SaturateToSignedWord(SRC2[127:120]*SRC1[127:120]+ SRC2[119:112]* SRC1[119:112])
DEST[MAXVL-1:128] ← 0
```

VPMADDUBSW (VEX.256 encoded version)

```
DEST[15:0] ← SaturateToSignedWord(SRC2[15:8]* SRC1[15:8]+SRC2[7:0]*SRC1[7:0])
// Repeat operation for 2nd through 15th word
DEST[255:240] ← SaturateToSignedWord(SRC2[255:248]*SRC1[255:248]+ SRC2[247:240]* SRC1[247:240])
DEST[MAXVL-1:256] ← 0
```

VPMADDUBSW (EVEX encoded versions)

```
(KL, VL) = (8, 128), (16, 256), (32, 512)

FOR j ← 0 TO KL-1
    i ← j * 16
    IF k1[j] OR *no writemask*
        THEN DEST[i+15:i] ← SaturateToSignedWord(SRC2[i+15:i+8]* SRC1[i+15:i+8] + SRC2[i+7:i]*SRC1[i+7:i])
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+15:i] remains unchanged*
                ELSE *zeroing-masking*              ; zeroing-masking
                    DEST[i+15:i] = 0
            FI
    FI;
ENDFOR;
DEST[MAXVL-1:VL] ← 0
```

Intel C/C++ Compiler Intrinsic Equivalents

VPMADDUBSW __m512i _mm512_mddubs_epi16(__m512i a, __m512i b);
VPMADDUBSW __m512i _mm512_mask_mddubs_epi16(__m512i s, __mmask32 k, __m512i a, __m512i b);
VPMADDUBSW __m512i _mm512_maskz_mddubs_epi16(__mmask32 k, __m512i a, __m512i b);
VPMADDUBSW __m256i _mm256_mask_mddubs_epi16(__m256i s, __mmask16 k, __m256i a, __m256i b);
VPMADDUBSW __m256i _mm256_maskz_mddubs_epi16(__mmask16 k, __m256i a, __m256i b);
VPMADDUBSW __m128i _mm_mask_mddubs_epi16(__m128i s, __mmask8 k, __m128i a, __m128i b);
VPMADDUBSW __m128i _mm_maskz_maddubs_epi16(__mmask8 k, __m128i a, __m128i b);
PMADDUBSW: __m64 _mm_maddubs_pi16 (__m64 a, __m64 b)
(V)PMADDUBSW: __m128i _mm_maddubs_epi16 (__m128i a, __m128i b)
VPMADDUBSW: __m256i _mm256_maddubs_epi16 (__m256i a, __m256i b)

SIMD Floating-Point Exceptions

None.

Other Exceptions

Non-EVEX-encoded instruction, see Exceptions Type 4.
EVEX-encoded instruction, see Exceptions Type E4NF.nb.

PMADDWD—Multiply and Add Packed Integers

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F F5 /r[1] PMADDWD *mm, mm/m64*	A	V/V	MMX	Multiply the packed words in *mm* by the packed words in *mm/m64*, add adjacent doubleword results, and store in *mm*.
66 0F F5 /r PMADDWD *xmm1*, *xmm2/m128*	A	V/V	SSE2	Multiply the packed word integers in *xmm1* by the packed word integers in *xmm2/m128*, add adjacent doubleword results, and store in *xmm1*.
VEX.NDS.128.66.0F.WIG F5 /r VPMADDWD *xmm1, xmm2, xmm3/m128*	B	V/V	AVX	Multiply the packed word integers in *xmm2* by the packed word integers in *xmm3/m128*, add adjacent doubleword results, and store in *xmm1*.
VEX.NDS.256.66.0F.WIG F5 /r VPMADDWD *ymm1, ymm2, ymm3/m256*	B	V/V	AVX2	Multiply the packed word integers in *ymm2* by the packed word integers in *ymm3/m256*, add adjacent doubleword results, and store in *ymm1*.
EVEX.NDS.128.66.0F.WIG F5 /r VPMADDWD xmm1 {k1}{z}, xmm2, xmm3/m128	C	V/V	AVX512VL AVX512BW	Multiply the packed word integers in xmm2 by the packed word integers in xmm3/m128, add adjacent doubleword results, and store in xmm1 under writemask k1.
EVEX.NDS.256.66.0F.WIG F5 /r VPMADDWD ymm1 {k1}{z}, ymm2, ymm3/m256	C	V/V	AVX512VL AVX512BW	Multiply the packed word integers in ymm2 by the packed word integers in ymm3/m256, add adjacent doubleword results, and store in ymm1 under writemask k1.
EVEX.NDS.512.66.0F.WIG F5 /r VPMADDWD zmm1 {k1}{z}, zmm2, zmm3/m512	C	V/V	AVX512BW	Multiply the packed word integers in zmm2 by the packed word integers in zmm3/m512, add adjacent doubleword results, and store in zmm1 under writemask k1.

NOTES:

1. See note in Section 2.4, "AVX and SSE Instruction Exception Specification" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A* and Section 22.25.3, "Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	NA	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
B	NA	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA
C	Full Vector Mem	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA

Description

Multiplies the individual signed words of the destination operand (first operand) by the corresponding signed words of the source operand (second operand), producing temporary signed, doubleword results. The adjacent double-word results are then summed and stored in the destination operand. For example, the corresponding low-order words (15-0) and (31-16) in the source and destination operands are multiplied by one another and the double-word results are added together and stored in the low doubleword of the destination register (31-0). The same operation is performed on the other pairs of adjacent words. (Figure 4-11 shows this operation when using 64-bit operands).

The (V)PMADDWD instruction wraps around only in one situation: when the 2 pairs of words being operated on in a group are all 8000H. In this case, the result wraps around to 80000000H.

In 64-bit mode and not encoded with VEX/EVEX, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

Legacy SSE version: The first source and destination operands are MMX registers. The second source operand is an MMX register or a 64-bit memory location.

128-bit Legacy SSE version: The first source and destination operands are XMM registers. The second source operand is an XMM register or a 128-bit memory location. Bits (MAXVL-1:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: The first source and destination operands are XMM registers. The second source operand is an XMM register or a 128-bit memory location. Bits (MAXVL-1:128) of the destination YMM register are zeroed.

VEX.256 encoded version: The second source operand can be an YMM register or a 256-bit memory location. The first source and destination operands are YMM registers.

EVEX.512 encoded version: The second source operand can be an ZMM register or a 512-bit memory location. The first source and destination operands are ZMM registers.

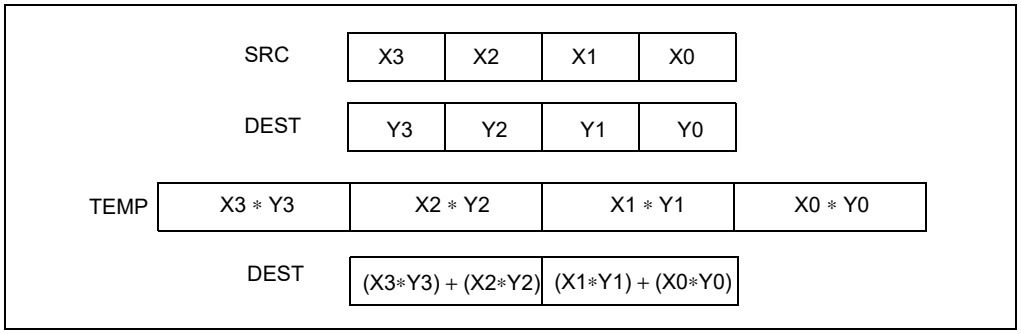


Figure 4-11. PMADDWD Execution Model Using 64-bit Operands

Operation

PMADDWD (with 64-bit operands)

```
DEST[31:0] ← (DEST[15:0] * SRC[15:0]) + (DEST[31:16] * SRC[31:16]);
DEST[63:32] ← (DEST[47:32] * SRC[47:32]) + (DEST[63:48] * SRC[63:48]);
```

PMADDWD (with 128-bit operands)

```
DEST[31:0] ← (DEST[15:0] * SRC[15:0]) + (DEST[31:16] * SRC[31:16]);
DEST[63:32] ← (DEST[47:32] * SRC[47:32]) + (DEST[63:48] * SRC[63:48]);
DEST[95:64] ← (DEST[79:64] * SRC[79:64]) + (DEST[95:80] * SRC[95:80]);
DEST[127:96] ← (DEST[111:96] * SRC[111:96]) + (DEST[127:112] * SRC[127:112]);
```

VPMADDWD (VEX.128 encoded version)

```
DEST[31:0] ← (SRC1[15:0] * SRC2[15:0]) + (SRC1[31:16] * SRC2[31:16])
DEST[63:32] ← (SRC1[47:32] * SRC2[47:32]) + (SRC1[63:48] * SRC2[63:48])
DEST[95:64] ← (SRC1[79:64] * SRC2[79:64]) + (SRC1[95:80] * SRC2[95:80])
DEST[127:96] ← (SRC1[111:96] * SRC2[111:96]) + (SRC1[127:112] * SRC2[127:112])
DEST[MAXVL-1:128] ← 0
```

VPMADDWD (VEX.256 encoded version)

```
DEST[31:0] ← (SRC1[15:0] * SRC2[15:0]) + (SRC1[31:16] * SRC2[31:16])
DEST[63:32] ← (SRC1[47:32] * SRC2[47:32]) + (SRC1[63:48] * SRC2[63:48])
DEST[95:64] ← (SRC1[79:64] * SRC2[79:64]) + (SRC1[95:80] * SRC2[95:80])
DEST[127:96] ← (SRC1[111:96] * SRC2[111:96]) + (SRC1[127:112] * SRC2[127:112])
DEST[159:128] ← (SRC1[143:128] * SRC2[143:128]) + (SRC1[159:144] * SRC2[159:144])
DEST[191:160] ← (SRC1[175:160] * SRC2[175:160]) + (SRC1[191:176] * SRC2[191:176])
DEST[223:192] ← (SRC1[207:192] * SRC2[207:192]) + (SRC1[223:208] * SRC2[223:208])
DEST[255:224] ← (SRC1[239:224] * SRC2[239:224]) + (SRC1[255:240] * SRC2[255:240])
DEST[MAXVL-1:256] ← 0
```

VPMADDWD (EVEX encoded versions)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)

FOR j ← 0 TO KL-1
    i ← j * 32
    IF k1[j] OR *no writemask*
        THEN DEST[i+31:i] ← (SRC2[i+31:i+16]* SRC1[i+31:i+16]) + (SRC2[i+15:i]*SRC1[i+15:i])
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE *zeroing-masking*              ; zeroing-masking
                    DEST[i+31:i] = 0
            FI
    FI;
ENDFOR;
DEST[MAXVL-1:VL] ← 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VPMADDWD __m512i _mm512_mdd_epi16(__m512i a, __m512i b);
VPMADDWD __m512i _mm512_mask_mdd_epi16(__m512i s, __mmask16 k, __m512i a, __m512i b);
VPMADDWD __m512i _mm512_maskz_mdd_epi16(__mmask16 k, __m512i a, __m512i b);
VPMADDWD __m256i _mm256_mask_mdd_epi16(__m256i s, __mmask8 k, __m256i a, __m256i b);
VPMADDWD __m256i _mm256_maskz_mdd_epi16(__mmask8 k, __m256i a, __m256i b);
VPMADDWD __m128i _mm_mask_mdd_epi16(__m128i s, __mmask8 k, __m128i a, __m128i b);
VPMADDWD __m128i _mm_maskz_madd_epi16(__mmask8 k, __m128i a, __m128i b);
PMADDWD:__m64 _mm_madd_pi16(__m64 m1, __m64 m2)
(V)PMADDWD:__m128i _mm_madd_epi16 (__m128i a, __m128i b)
VPMADDWD:__m256i _mm256_madd_epi16 (__m256i a, __m256i b)

Flags Affected

None.

Numeric Exceptions

None.

Other Exceptions

Non-EVEX-encoded instruction, see Exceptions Type 4.
EVEX-encoded instruction, see Exceptions Type E4NF.nb.

PMAXSB/PMAXSW/PMAXSD/PMAXSQ—Maximum of Packed Signed Integers

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F EE /r[1] PMAXSW *mm1, mm2/m64*	A	V/V	SSE	Compare signed word integers in *mm2/m64* and *mm1* and return maximum values.
66 0F 38 3C /r PMAXSB xmm1, xmm2/m128	A	V/V	SSE4_1	Compare packed signed byte integers in xmm1 and xmm2/m128 and store packed maximum values in xmm1.
66 0F EE /r PMAXSW xmm1, xmm2/m128	A	V/V	SSE2	Compare packed signed word integers in xmm2/m128 and xmm1 and stores maximum packed values in xmm1.
66 0F 38 3D /r PMAXSD xmm1, xmm2/m128	A	V/V	SSE4_1	Compare packed signed dword integers in xmm1 and xmm2/m128 and store packed maximum values in xmm1.
VEX.NDS.128.66.0F38.WIG 3C /r VPMAXSB xmm1, xmm2, xmm3/m128	B	V/V	AVX	Compare packed signed byte integers in xmm2 and xmm3/m128 and store packed maximum values in xmm1.
VEX.NDS.128.66.0F.WIG EE /r VPMAXSW xmm1, xmm2, xmm3/m128	B	V/V	AVX	Compare packed signed word integers in xmm3/m128 and xmm2 and store packed maximum values in xmm1.
VEX.NDS.128.66.0F38.WIG 3D /r VPMAXSD xmm1, xmm2, xmm3/m128	B	V/V	AVX	Compare packed signed dword integers in xmm2 and xmm3/m128 and store packed maximum values in xmm1.
VEX.NDS.256.66.0F38.WIG 3C /r VPMAXSB ymm1, ymm2, ymm3/m256	B	V/V	AVX2	Compare packed signed byte integers in ymm2 and ymm3/m256 and store packed maximum values in ymm1.
VEX.NDS.256.66.0F.WIG EE /r VPMAXSW ymm1, ymm2, ymm3/m256	B	V/V	AVX2	Compare packed signed word integers in ymm3/m256 and ymm2 and store packed maximum values in ymm1.
VEX.NDS.256.66.0F38.WIG 3D /r VPMAXSD ymm1, ymm2, ymm3/m256	B	V/V	AVX2	Compare packed signed dword integers in ymm2 and ymm3/m256 and store packed maximum values in ymm1.
EVEX.NDS.128.66.0F38.WIG 3C /r VPMAXSB xmm1{k1}{z}, xmm2, xmm3/m128	C	V/V	AVX512VL AVX512BW	Compare packed signed byte integers in xmm2 and xmm3/m128 and store packed maximum values in xmm1 under writemask k1.
EVEX.NDS.256.66.0F38.WIG 3C /r VPMAXSB ymm1{k1}{z}, ymm2, ymm3/m256	C	V/V	AVX512VL AVX512BW	Compare packed signed byte integers in ymm2 and ymm3/m256 and store packed maximum values in ymm1 under writemask k1.
EVEX.NDS.512.66.0F38.WIG 3C /r VPMAXSB zmm1{k1}{z}, zmm2, zmm3/m512	C	V/V	AVX512BW	Compare packed signed byte integers in zmm2 and zmm3/m512 and store packed maximum values in zmm1 under writemask k1.
EVEX.NDS.128.66.0F.WIG EE /r VPMAXSW xmm1{k1}{z}, xmm2, xmm3/m128	C	V/V	AVX512VL AVX512BW	Compare packed signed word integers in xmm2 and xmm3/m128 and store packed maximum values in xmm1 under writemask k1.
EVEX.NDS.256.66.0F.WIG EE /r VPMAXSW ymm1{k1}{z}, ymm2, ymm3/m256	C	V/V	AVX512VL AVX512BW	Compare packed signed word integers in ymm2 and ymm3/m256 and store packed maximum values in ymm1 under writemask k1.
EVEX.NDS.512.66.0F.WIG EE /r VPMAXSW zmm1{k1}{z}, zmm2, zmm3/m512	C	V/V	AVX512BW	Compare packed signed word integers in zmm2 and zmm3/m512 and store packed maximum values in zmm1 under writemask k1.
EVEX.NDS.128.66.0F38.W0 3D /r VPMAXSD xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst	D	V/V	AVX512VL AVX512F	Compare packed signed dword integers in xmm2 and xmm3/m128/m32bcst and store packed maximum values in xmm1 using writemask k1.

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.NDS.256.66.0F38.W0 3D /r VPMAXSD ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst	D	V/V	AVX512VL AVX512F	Compare packed signed dword integers in ymm2 and ymm3/m256/m32bcst and store packed maximum values in ymm1 using writemask k1.
EVEX.NDS.512.66.0F38.W0 3D /r VPMAXSD zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst	D	V/V	AVX512F	Compare packed signed dword integers in zmm2 and zmm3/m512/m32bcst and store packed maximum values in zmm1 using writemask k1.
EVEX.NDS.128.66.0F38.W1 3D /r VPMAXSQ xmm1 {k1}{z}, xmm2, xmm3/m128/m64bcst	D	V/V	AVX512VL AVX512F	Compare packed signed qword integers in xmm2 and xmm3/m128/m64bcst and store packed maximum values in xmm1 using writemask k1.
EVEX.NDS.256.66.0F38.W1 3D /r VPMAXSQ ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst	D	V/V	AVX512VL AVX512F	Compare packed signed qword integers in ymm2 and ymm3/m256/m64bcst and store packed maximum values in ymm1 using writemask k1.
EVEX.NDS.512.66.0F38.W1 3D /r VPMAXSQ zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst	D	V/V	AVX512F	Compare packed signed qword integers in zmm2 and zmm3/m512/m64bcst and store packed maximum values in zmm1 using writemask k1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	NA	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
B	NA	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA
C	Full Vector Mem	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA
D	Full Vector	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA

Description

Performs a SIMD compare of the packed signed byte, word, dword or qword integers in the second source operand and the first source operand and returns the maximum value for each pair of integers to the destination operand.

Legacy SSE version PMAXSW: The source operand can be an MMX technology register or a 64-bit memory location. The destination operand can be an MMX technology register.

128-bit Legacy SSE version: The first source and destination operands are XMM registers. The second source operand is an XMM register or a 128-bit memory location. Bits (MAXVL-1:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: The first source and destination operands are XMM registers. The second source operand is an XMM register or a 128-bit memory location. Bits (MAXVL-1:128) of the corresponding destination register are zeroed.

VEX.256 encoded version: The second source operand can be an YMM register or a 256-bit memory location. The first source and destination operands are YMM registers. Bits (MAXVL-1:256) of the corresponding destination register are zeroed.

EVEX encoded VPMAXSD/Q: The first source operand is a ZMM/YMM/XMM register; The second source operand is a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 32/64-bit memory location. The destination operand is conditionally updated based on writemask k1.

EVEX encoded VPMAXSB/W: The first source operand is a ZMM/YMM/XMM register; The second source operand is a ZMM/YMM/XMM register, a 512/256/128-bit memory location. The destination operand is conditionally updated based on writemask k1.

Operation

PMAXSW (64-bit operands)

```
IF DEST[15:0] > SRC[15:0]) THEN
    DEST[15:0] ← DEST[15:0];
ELSE
    DEST[15:0] ← SRC[15:0]; FI;
(* Repeat operation for 2nd and 3rd words in source and destination operands *)
IF DEST[63:48] > SRC[63:48]) THEN
    DEST[63:48] ← DEST[63:48];
ELSE
    DEST[63:48] ← SRC[63:48]; FI;
```

PMAXSB (128-bit Legacy SSE version)

```
IF DEST[7:0] >SRC[7:0] THEN
    DEST[7:0] ←DEST[7:0];
ELSE
    DEST[7:0] ←SRC[7:0]; FI;
(* Repeat operation for 2nd through 15th bytes in source and destination operands *)
IF DEST[127:120] >SRC[127:120] THEN
    DEST[127:120] ←DEST[127:120];
ELSE
    DEST[127:120] ←SRC[127:120]; FI;
DEST[MAXVL-1:128] (Unmodified)
```

VPMAXSB (VEX.128 encoded version)

```
IF SRC1[7:0] >SRC2[7:0] THEN
    DEST[7:0] ←SRC1[7:0];
ELSE
    DEST[7:0] ←SRC2[7:0]; FI;
(* Repeat operation for 2nd through 15th bytes in source and destination operands *)
IF SRC1[127:120] >SRC2[127:120] THEN
    DEST[127:120] ←SRC1[127:120];
ELSE
    DEST[127:120] ←SRC2[127:120]; FI;
DEST[MAXVL-1:128] ←0
```

VPMAXSB (VEX.256 encoded version)

```
IF SRC1[7:0] >SRC2[7:0] THEN
    DEST[7:0] ←SRC1[7:0];
ELSE
    DEST[7:0] ←SRC2[7:0]; FI;
(* Repeat operation for 2nd through 31st bytes in source and destination operands *)
IF SRC1[255:248] >SRC2[255:248] THEN
    DEST[255:248] ←SRC1[255:248];
ELSE
    DEST[255:248] ←SRC2[255:248]; FI;
DEST[MAXVL-1:256] ←0
```

VPMAXSB (EVEX encoded versions)

```
(KL, VL) = (16, 128), (32, 256), (64, 512)
FOR j ← 0 TO KL-1
    i ← j * 8
    IF k1[j] OR *no writemask* THEN
        IF SRC1[i+7:i] > SRC2[i+7:i]
            THEN DEST[i+7:i] ← SRC1[i+7:i];
            ELSE DEST[i+7:i] ← SRC2[i+7:i];
        FI;
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+7:i] remains unchanged*
                ELSE                                ; zeroing-masking
                    DEST[i+7:i] ← 0
            FI
    FI;
ENDFOR;
DEST[MAXVL-1:VL] ← 0
```

PMAXSW (128-bit Legacy SSE version)

```
    IF DEST[15:0] >SRC[15:0] THEN
        DEST[15:0] ←DEST[15:0];
    ELSE
        DEST[15:0] ←SRC[15:0]; FI;
    (* Repeat operation for 2nd through 7th words in source and destination operands *)
    IF DEST[127:112] >SRC[127:112] THEN
        DEST[127:112] ←DEST[127:112];
    ELSE
        DEST[127:112] ←SRC[127:112]; FI;
DEST[MAXVL-1:128] (Unmodified)
```

VPMAXSW (VEX.128 encoded version)

```
    IF SRC1[15:0] > SRC2[15:0] THEN
        DEST[15:0] ←SRC1[15:0];
    ELSE
        DEST[15:0] ←SRC2[15:0]; FI;
    (* Repeat operation for 2nd through 7th words in source and destination operands *)
    IF SRC1[127:112] >SRC2[127:112] THEN
        DEST[127:112] ←SRC1[127:112];
    ELSE
        DEST[127:112] ←SRC2[127:112]; FI;
DEST[MAXVL-1:128] ←0
```

VPMAXSW (VEX.256 encoded version)

```
    IF SRC1[15:0] > SRC2[15:0] THEN
        DEST[15:0] ←SRC1[15:0];
    ELSE
        DEST[15:0] ←SRC2[15:0]; FI;
    (* Repeat operation for 2nd through 15th words in source and destination operands *)
    IF SRC1[255:240] >SRC2[255:240] THEN
        DEST[255:240] ←SRC1[255:240];
    ELSE
        DEST[255:240] ←SRC2[255:240]; FI;
DEST[MAXVL-1:256] ←0
```

VPMAXSW (EVEX encoded versions)

```
(KL, VL) = (8, 128), (16, 256), (32, 512)
FOR j ← 0 TO KL-1
    i ← j * 16
    IF k1[j] OR *no writemask* THEN
        IF SRC1[i+15:i] > SRC2[i+15:i]
            THEN DEST[i+15:i] ← SRC1[i+15:i];
            ELSE DEST[i+15:i] ← SRC2[i+15:i];
        FI;
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+15:i] remains unchanged*
                ELSE                                ; zeroing-masking
                    DEST[i+15:i] ← 0
            FI
    FI;
ENDFOR;
DEST[MAXVL-1:VL] ← 0
```

PMAXSD (128-bit Legacy SSE version)

```
    IF DEST[31:0] >SRC[31:0] THEN
        DEST[31:0] ←DEST[31:0];
    ELSE
        DEST[31:0] ←SRC[31:0]; FI;
    (* Repeat operation for 2nd through 7th words in source and destination operands *)
    IF DEST[127:96] >SRC[127:96] THEN
        DEST[127:96] ←DEST[127:96];
    ELSE
        DEST[127:96] ←SRC[127:96]; FI;
DEST[MAXVL-1:128] (Unmodified)
```

VPMAXSD (VEX.128 encoded version)

```
    IF SRC1[31:0] > SRC2[31:0] THEN
        DEST[31:0] ←SRC1[31:0];
    ELSE
        DEST[31:0] ←SRC2[31:0]; FI;
    (* Repeat operation for 2nd through 3rd dwords in source and destination operands *)
    IF SRC1[127:96] > SRC2[127:96] THEN
        DEST[127:96] ←SRC1[127:96];
    ELSE
        DEST[127:96] ←SRC2[127:96]; FI;
DEST[MAXVL-1:128] ←0
```

VPMAXSD (VEX.256 encoded version)

```
    IF SRC1[31:0] > SRC2[31:0] THEN
        DEST[31:0] ←SRC1[31:0];
    ELSE
        DEST[31:0] ←SRC2[31:0]; FI;
    (* Repeat operation for 2nd through 7th dwords in source and destination operands *)
    IF SRC1[255:224] > SRC2[255:224] THEN
        DEST[255:224] ←SRC1[255:224];
    ELSE
        DEST[255:224] ←SRC2[255:224]; FI;
DEST[MAXVL-1:256] ←0
```

VPMAXSD (EVEX encoded versions)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j ← 0 TO KL-1
    i ← j * 32
    IF k1[j] OR *no writemask*THEN
        IF (EVEX.b = 1) AND (SRC2 *is memory*)
            THEN
                IF SRC1[i+31:i] > SRC2[31:0]
                    THEN DEST[i+31:i] ← SRC1[i+31:i];
                    ELSE DEST[i+31:i] ← SRC2[31:0];
                FI;
            ELSE
                IF SRC1[i+31:i] > SRC2[i+31:i]
                    THEN DEST[i+31:i] ← SRC1[i+31:i];
                    ELSE DEST[i+31:i] ← SRC2[i+31:i];
            FI;
        FI;
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE  DEST[i+31:i] ← 0              ; zeroing-masking
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0
```

VPMAXSQ (EVEX encoded versions)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j ← 0 TO KL-1
    i ← j * 64
    IF k1[j] OR *no writemask* THEN
        IF (EVEX.b = 1) AND (SRC2 *is memory*)
            THEN
                IF SRC1[i+63:i] > SRC2[63:0]
                    THEN DEST[i+63:i] ← SRC1[i+63:i];
                    ELSE DEST[i+63:i] ← SRC2[63:0];
                FI;
            ELSE
                IF SRC1[i+63:i] > SRC2[i+63:i]
                    THEN DEST[i+63:i] ← SRC1[i+63:i];
                    ELSE DEST[i+63:i] ← SRC2[i+63:i];
            FI;
        FI;
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE                                ; zeroing-masking
                    THEN DEST[i+63:i] ← 0
            FI
    FI;
ENDFOR;
DEST[MAXVL-1:VL] ← 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VPMAXSB __m512i _mm512_max_epi8(__m512i a, __m512i b);
VPMAXSB __m512i _mm512_mask_max_epi8(__m512i s, __mmask64 k, __m512i a, __m512i b);
VPMAXSB __m512i _mm512_maskz_max_epi8(__mmask64 k, __m512i a, __m512i b);
VPMAXSW __m512i _mm512_max_epi16(__m512i a, __m512i b);
VPMAXSW __m512i _mm512_mask_max_epi16(__m512i s, __mmask32 k, __m512i a, __m512i b);
VPMAXSW __m512i _mm512_maskz_max_epi16(__mmask32 k, __m512i a, __m512i b);
VPMAXSB __m256i _mm256_mask_max_epi8(__m256i s, __mmask32 k, __m256i a, __m256i b);
VPMAXSB __m256i _mm256_maskz_max_epi8(__mmask32 k, __m256i a, __m256i b);
VPMAXSW __m256i _mm256_mask_max_epi16(__m256i s, __mmask16 k, __m256i a, __m256i b);
VPMAXSW __m256i _mm256_maskz_max_epi16(__mmask16 k, __m256i a, __m256i b);
VPMAXSB __m128i _mm_mask_max_epi8(__m128i s, __mmask16 k, __m128i a, __m128i b);
VPMAXSB __m128i _mm_maskz_max_epi8(__mmask16 k, __m128i a, __m128i b);
VPMAXSW __m128i _mm_mask_max_epi16(__m128i s, __mmask8 k, __m128i a, __m128i b);
VPMAXSW __m128i _mm_maskz_max_epi16(__mmask8 k, __m128i a, __m128i b);
VPMAXSD __m256i _mm256_mask_max_epi32(__m256i s, __mmask16 k, __m256i a, __m256i b);
VPMAXSD __m256i _mm256_maskz_max_epi32(__mmask16 k, __m256i a, __m256i b);
VPMAXSQ __m256i _mm256_mask_max_epi64(__m256i s, __mmask8 k, __m256i a, __m256i b);
VPMAXSQ __m256i _mm256_maskz_max_epi64(__mmask8 k, __m256i a, __m256i b);
VPMAXSD __m128i _mm_mask_max_epi32(__m128i s, __mmask8 k, __m128i a, __m128i b);
VPMAXSD __m128i _mm_maskz_max_epi32(__mmask8 k, __m128i a, __m128i b);
VPMAXSQ __m128i _mm_mask_max_epi64(__m128i s, __mmask8 k, __m128i a, __m128i b);
VPMAXSQ __m128i _mm_maskz_max_epu64(__mmask8 k, __m128i a, __m128i b);
VPMAXSD __m512i _mm512_max_epi32(__m512i a, __m512i b);
VPMAXSD __m512i _mm512_mask_max_epi32(__m512i s, __mmask16 k, __m512i a, __m512i b);
VPMAXSD __m512i _mm512_maskz_max_epi32(__mmask16 k, __m512i a, __m512i b);
VPMAXSQ __m512i _mm512_max_epi64(__m512i a, __m512i b);
VPMAXSQ __m512i _mm512_mask_max_epi64(__m512i s, __mmask8 k, __m512i a, __m512i b);
VPMAXSQ __m512i _mm512_maskz_max_epi64(__mmask8 k, __m512i a, __m512i b);
(V)PMAXSB __m128i _mm_max_epi8 (__m128i a, __m128i b);
(V)PMAXSW __m128i _mm_max_epi16 (__m128i a, __m128i b)
(V)PMAXSD __m128i _mm_max_epi32 (__m128i a, __m128i b);
VPMAXSB __m256i _mm256_max_epi8 (__m256i a, __m256i b);
VPMAXSW __m256i _mm256_max_epi16 (__m256i a, __m256i b)
VPMAXSD __m256i _mm256_max_epi32 (__m256i a, __m256i b);
PMAXSW:__m64 _mm_max_pi16(__m64 a, __m64 b)

SIMD Floating-Point Exceptions

None

Other Exceptions

Non-EVEX-encoded instruction, see Exceptions Type 4.
EVEX-encoded VPMAXSD/Q, see Exceptions Type E4.
EVEX-encoded VPMAXSB/W, see Exceptions Type E4.nb.

PMAXUB/PMAXUW—Maximum of Packed Unsigned Integers

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F DE /r[1] PMAXUB *mm1, mm2/m64*	A	V/V	SSE	Compare unsigned byte integers in *mm2/m64* and *mm1* and returns maximum values.
66 0F DE /r PMAXUB xmm1, xmm2/m128	A	V/V	SSE2	Compare packed unsigned byte integers in xmm1 and xmm2/m128 and store packed maximum values in xmm1.
66 0F 38 3E/r PMAXUW xmm1, xmm2/m128	A	V/V	SSE4_1	Compare packed unsigned word integers in xmm2/m128 and xmm1 and stores maximum packed values in xmm1.
VEX.NDS.128.66.0F DE /r VPMAXUB xmm1, xmm2, xmm3/m128	B	V/V	AVX	Compare packed unsigned byte integers in xmm2 and xmm3/m128 and store packed maximum values in xmm1.
VEX.NDS.128.66.0F38 3E/r VPMAXUW xmm1, xmm2, xmm3/m128	B	V/V	AVX	Compare packed unsigned word integers in xmm3/m128 and xmm2 and store maximum packed values in xmm1.
VEX.NDS.256.66.0F DE /r VPMAXUB ymm1, ymm2, ymm3/m256	B	V/V	AVX2	Compare packed unsigned byte integers in ymm2 and ymm3/m256 and store packed maximum values in ymm1.
VEX.NDS.256.66.0F38 3E/r VPMAXUW ymm1, ymm2, ymm3/m256	B	V/V	AVX2	Compare packed unsigned word integers in ymm3/m256 and ymm2 and store maximum packed values in ymm1.
EVEX.NDS.128.66.0F.WIG DE /r VPMAXUB xmm1{k1}{z}, xmm2, xmm3/m128	C	V/V	AVX512VL AVX512BW	Compare packed unsigned byte integers in xmm2 and xmm3/m128 and store packed maximum values in xmm1 under writemask k1.
EVEX.NDS.256.66.0F.WIG DE /r VPMAXUB ymm1{k1}{z}, ymm2, ymm3/m256	C	V/V	AVX512VL AVX512BW	Compare packed unsigned byte integers in ymm2 and ymm3/m256 and store packed maximum values in ymm1 under writemask k1.
EVEX.NDS.512.66.0F.WIG DE /r VPMAXUB zmm1{k1}{z}, zmm2, zmm3/m512	C	V/V	AVX512BW	Compare packed unsigned byte integers in zmm2 and zmm3/m512 and store packed maximum values in zmm1 under writemask k1.
EVEX.NDS.128.66.0F38.WIG 3E /r VPMAXUW xmm1{k1}{z}, xmm2, xmm3/m128	C	V/V	AVX512VL AVX512BW	Compare packed unsigned word integers in xmm2 and xmm3/m128 and store packed maximum values in xmm1 under writemask k1.
EVEX.NDS.256.66.0F38.WIG 3E /r VPMAXUW ymm1{k1}{z}, ymm2, ymm3/m256	C	V/V	AVX512VL AVX512BW	Compare packed unsigned word integers in ymm2 and ymm3/m256 and store packed maximum values in ymm1 under writemask k1.
EVEX.NDS.512.66.0F38.WIG 3E /r VPMAXUW zmm1{k1}{z}, zmm2, zmm3/m512	C	V/V	AVX512BW	Compare packed unsigned word integers in zmm2 and zmm3/m512 and store packed maximum values in zmm1 under writemask k1.

NOTES:

1. See note in Section 2.4, "AVX and SSE Instruction Exception Specification" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A* and Section 22.25.3, "Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	NA	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
B	NA	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA
C	Full Vector Mem	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA

Description

Performs a SIMD compare of the packed unsigned byte, word integers in the second source operand and the first source operand and returns the maximum value for each pair of integers to the destination operand.

Legacy SSE version PMAXUB: The source operand can be an MMX technology register or a 64-bit memory location. The destination operand can be an MMX technology register.

128-bit Legacy SSE version: The first source and destination operands are XMM registers. The second source operand is an XMM register or a 128-bit memory location. Bits (MAXVL-1:128) of the corresponding destination register remain unchanged.

VEX.128 encoded version: The first source and destination operands are XMM registers. The second source operand is an XMM register or a 128-bit memory location. Bits (MAXVL-1:128) of the corresponding destination register are zeroed.

VEX.256 encoded version: The second source operand can be an YMM register or a 256-bit memory location. The first source and destination operands are YMM registers.

EVEX encoded versions: The first source operand is a ZMM/YMM/XMM register; The second source operand is a ZMM/YMM/XMM register or a 512/256/128-bit memory location. The destination operand is conditionally updated based on writemask k1.

Operation

PMAXUB (64-bit operands)

```
IF DEST[7:0] > SRC[17:0]) THEN
    DEST[7:0] ← DEST[7:0];
ELSE
    DEST[7:0] ← SRC[7:0]; FI;
(* Repeat operation for 2nd through 7th bytes in source and destination operands *)
IF DEST[63:56] > SRC[63:56]) THEN
    DEST[63:56] ← DEST[63:56];
ELSE
    DEST[63:56] ← SRC[63:56]; FI;
```

PMAXUB (128-bit Legacy SSE version)

```
IF DEST[7:0] >SRC[7:0] THEN
    DEST[7:0] ← DEST[7:0];
ELSE
    DEST[15:0] ← SRC[7:0]; FI;
(* Repeat operation for 2nd through 15th bytes in source and destination operands *)
IF DEST[127:120] >SRC[127:120] THEN
    DEST[127:120] ← DEST[127:120];
ELSE
    DEST[127:120] ← SRC[127:120]; FI;
DEST[MAXVL-1:128] (Unmodified)
```

VPMAXUB (VEX.128 encoded version)

```
IF SRC1[7:0] >SRC2[7:0] THEN
    DEST[7:0] ← SRC1[7:0];
ELSE
    DEST[7:0] ← SRC2[7:0]; FI;
(* Repeat operation for 2nd through 15th bytes in source and destination operands *)
IF SRC1[127:120] >SRC2[127:120] THEN
    DEST[127:120] ← SRC1[127:120];
ELSE
    DEST[127:120] ← SRC2[127:120]; FI;
DEST[MAXVL-1:128] ← 0
```

VPMAXUB (VEX.256 encoded version)

```
    IF SRC1[7:0] >SRC2[7:0] THEN
        DEST[7:0] ← SRC1[7:0];
    ELSE
        DEST[15:0] ← SRC2[7:0]; FI;
    (* Repeat operation for 2nd through 31st bytes in source and destination operands *)
    IF SRC1[255:248] >SRC2[255:248] THEN
        DEST[255:248] ← SRC1[255:248];
    ELSE
        DEST[255:248] ← SRC2[255:248]; FI;
DEST[MAXVL-1:128] ← 0
```

VPMAXUB (EVEX encoded versions)

```
(KL, VL) = (16, 128), (32, 256), (64, 512)
FOR j ← 0 TO KL-1
    i ← j * 8
    IF k1[j] OR *no writemask* THEN
        IF SRC1[i+7:i] > SRC2[i+7:i]
            THEN DEST[i+7:i] ← SRC1[i+7:i];
            ELSE DEST[i+7:i] ← SRC2[i+7:i];
        FI;
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+7:i] remains unchanged*
                ELSE                                ; zeroing-masking
                    DEST[i+7:i] ← 0
            FI
    FI;
ENDFOR;
DEST[MAXVL-1:VL] ← 0
```

PMAXUW (128-bit Legacy SSE version)

```
    IF DEST[15:0] >SRC[15:0] THEN
        DEST[15:0] ← DEST[15:0];
    ELSE
        DEST[15:0] ← SRC[15:0]; FI;
    (* Repeat operation for 2nd through 7th words in source and destination operands *)
    IF DEST[127:112] >SRC[127:112] THEN
        DEST[127:112] ← DEST[127:112];
    ELSE
        DEST[127:112] ← SRC[127:112]; FI;
DEST[MAXVL-1:128] (Unmodified)
```

VPMAXUW (VEX.128 encoded version)

```
    IF SRC1[15:0] > SRC2[15:0] THEN
        DEST[15:0] ← SRC1[15:0];
    ELSE
        DEST[15:0] ← SRC2[15:0]; FI;
    (* Repeat operation for 2nd through 7th words in source and destination operands *)
    IF SRC1[127:112] >SRC2[127:112] THEN
        DEST[127:112] ← SRC1[127:112];
    ELSE
        DEST[127:112] ← SRC2[127:112]; FI;
DEST[MAXVL-1:128] ← 0
```

VPMAXUW (VEX.256 encoded version)

```
    IF SRC1[15:0] > SRC2[15:0] THEN
        DEST[15:0] ← SRC1[15:0];
    ELSE
        DEST[15:0] ← SRC2[15:0]; FI;
    (* Repeat operation for 2nd through 15th words in source and destination operands *)
    IF SRC1[255:240] >SRC2[255:240] THEN
        DEST[255:240] ← SRC1[255:240];
    ELSE
        DEST[255:240] ← SRC2[255:240]; FI;
DEST[MAXVL-1:128] ← 0
```

VPMAXUW (EVEX encoded versions)

```
(KL, VL) = (8, 128), (16, 256), (32, 512)
FOR j ← 0 TO KL-1
    i ← j * 16
    IF k1[j] OR *no writemask* THEN
        IF SRC1[i+15:i] > SRC2[i+15:i]
            THEN DEST[i+15:i] ← SRC1[i+15:i];
            ELSE DEST[i+15:i] ← SRC2[i+15:i];
        FI;
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+15:i] remains unchanged*
                ELSE                                ; zeroing-masking
                    DEST[i+15:i] ← 0
            FI
    FI;
ENDFOR;
DEST[MAXVL-1:VL] ← 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VPMAXUB __m512i _mm512_max_epu8(__m512i a, __m512i b);
VPMAXUB __m512i _mm512_mask_max_epu8(__m512i s, __mmask64 k, __m512i a, __m512i b);
VPMAXUB __m512i _mm512_maskz_max_epu8(__mmask64 k, __m512i a, __m512i b);
VPMAXUW __m512i _mm512_max_epu16(__m512i a, __m512i b);
VPMAXUW __m512i _mm512_mask_max_epu16(__m512i s, __mmask32 k, __m512i a, __m512i b);
VPMAXUW __m512i _mm512_maskz_max_epu16(__mmask32 k, __m512i a, __m512i b);
VPMAXUB __m256i _mm256_mask_max_epu8(__m256i s, __mmask32 k, __m256i a, __m256i b);
VPMAXUB __m256i _mm256_maskz_max_epu8(__mmask32 k, __m256i a, __m256i b);
VPMAXUW __m256i _mm256_mask_max_epu16(__m256i s, __mmask16 k, __m256i a, __m256i b);
VPMAXUW __m256i _mm256_maskz_max_epu16(__mmask16 k, __m256i a, __m256i b);
VPMAXUB __m128i _mm_mask_max_epu8(__m128i s, __mmask16 k, __m128i a, __m128i b);
VPMAXUB __m128i _mm_maskz_max_epu8(__mmask16 k, __m128i a, __m128i b);
VPMAXUW __m128i _mm_mask_max_epu16(__m128i s, __mmask8 k, __m128i a, __m128i b);
VPMAXUW __m128i _mm_maskz_max_epu16(__mmask8 k, __m128i a, __m128i b);
(V)PMAXUB __m128i _mm_max_epu8 (__m128i a, __m128i b);
(V)PMAXUW __m128i _mm_max_epu16 (__m128i a, __m128i b)
VPMAXUB __m256i _mm256_max_epu8 (__m256i a, __m256i b);
VPMAXUW __m256i _mm256_max_epu16 (__m256i a, __m256i b);
PMAXUB: __m64 _mm_max_pu8(__m64 a, __m64 b);

SIMD Floating-Point Exceptions

None

Other Exceptions

Non-EVEX-encoded instruction, see Exceptions Type 4.
EVEX-encoded instruction, see Exceptions Type E4.nb.

PMAXUD/PMAXUQ—Maximum of Packed Unsigned Integers

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 38 3F /r PMAXUD xmm1, xmm2/m128	A	V/V	SSE4_1	Compare packed unsigned dword integers in xmm1 and xmm2/m128 and store packed maximum values in xmm1.
VEX.NDS.128.66.0F38.WIG 3F /r VPMAXUD xmm1, xmm2, xmm3/m128	B	V/V	AVX	Compare packed unsigned dword integers in xmm2 and xmm3/m128 and store packed maximum values in xmm1.
VEX.NDS.256.66.0F38.WIG 3F /r VPMAXUD ymm1, ymm2, ymm3/m256	B	V/V	AVX2	Compare packed unsigned dword integers in ymm2 and ymm3/m256 and store packed maximum values in ymm1.
EVEX.NDS.128.66.0F38.W0 3F /r VPMAXUD xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst	C	V/V	AVX512VL AVX512F	Compare packed unsigned dword integers in xmm2 and xmm3/m128/m32bcst and store packed maximum values in xmm1 under writemask k1.
EVEX.NDS.256.66.0F38.W0 3F /r VPMAXUD ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst	C	V/V	AVX512VL AVX512F	Compare packed unsigned dword integers in ymm2 and ymm3/m256/m32bcst and store packed maximum values in ymm1 under writemask k1.
EVEX.NDS.512.66.0F38.W0 3F /r VPMAXUD zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst	C	V/V	AVX512F	Compare packed unsigned dword integers in zmm2 and zmm3/m512/m32bcst and store packed maximum values in zmm1 under writemask k1.
EVEX.NDS.128.66.0F38.W1 3F /r VPMAXUQ xmm1 {k1}{z}, xmm2, xmm3/m128/m64bcst	C	V/V	AVX512VL AVX512F	Compare packed unsigned qword integers in xmm2 and xmm3/m128/m64bcst and store packed maximum values in xmm1 under writemask k1.
EVEX.NDS.256.66.0F38.W1 3F /r VPMAXUQ ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst	C	V/V	AVX512VL AVX512F	Compare packed unsigned qword integers in ymm2 and ymm3/m256/m64bcst and store packed maximum values in ymm1 under writemask k1.
EVEX.NDS.512.66.0F38.W1 3F /r VPMAXUQ zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst	C	V/V	AVX512F	Compare packed unsigned qword integers in zmm2 and zmm3/m512/m64bcst and store packed maximum values in zmm1 under writemask k1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	NA	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
B	NA	ModRM:reg (w)	VEX.vvvv	ModRM:r/m (r)	NA
C	Full Vector	ModRM:reg (w)	EVEX.vvvv	ModRM:r/m (r)	NA

Description

Performs a SIMD compare of the packed unsigned dword or qword integers in the second source operand and the first source operand and returns the maximum value for each pair of integers to the destination operand.

128-bit Legacy SSE version: The first source and destination operands are XMM registers. The second source operand is an XMM register or a 128-bit memory location. Bits (MAXVL-1:128) of the corresponding destination register remain unchanged.

VEX.128 encoded version: The first source and destination operands are XMM registers. The second source operand is an XMM register or a 128-bit memory location. Bits (MAXVL-1:128) of the corresponding destination register are zeroed.

VEX.256 encoded version: The first source operand is a YMM register; The second source operand is a YMM register or 256-bit memory location. Bits (MAXVL-1:256) of the corresponding destination register are zeroed.

EVEX encoded versions: The first source operand is a ZMM/YMM/XMM register; The second source operand is a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 32/64-bit memory location. The destination operand is conditionally updated based on writemask k1.

Operation

PMAXUD (128-bit Legacy SSE version)

```
IF DEST[31:0] >SRC[31:0] THEN
    DEST[31:0] ← DEST[31:0];
ELSE
    DEST[31:0] ← SRC[31:0]; FI;
(* Repeat operation for 2nd through 7th words in source and destination operands *)
IF DEST[127:96] >SRC[127:96] THEN
    DEST[127:96] ← DEST[127:96];
ELSE
    DEST[127:96] ← SRC[127:96]; FI;
DEST[MAXVL-1:128] (Unmodified)
```

VPMAXUD (VEX.128 encoded version)

```
IF SRC1[31:0] > SRC2[31:0] THEN
    DEST[31:0] ← SRC1[31:0];
ELSE
    DEST[31:0] ← SRC2[31:0]; FI;
(* Repeat operation for 2nd through 3rd dwords in source and destination operands *)
IF SRC1[127:96] > SRC2[127:96] THEN
    DEST[127:96] ← SRC1[127:96];
ELSE
    DEST[127:96] ← SRC2[127:96]; FI;
DEST[MAXVL-1:128] ← 0
```

VPMAXUD (VEX.256 encoded version)

```
IF SRC1[31:0] > SRC2[31:0] THEN
    DEST[31:0] ← SRC1[31:0];
ELSE
    DEST[31:0] ← SRC2[31:0]; FI;
(* Repeat operation for 2nd through 7th dwords in source and destination operands *)
IF SRC1[255:224] > SRC2[255:224] THEN
    DEST[255:224] ← SRC1[255:224];
ELSE
    DEST[255:224] ← SRC2[255:224]; FI;
DEST[MAXVL-1:256] ← 0
```

VPMAXUD (EVEX encoded versions)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j ← 0 TO KL-1
    i ← j * 32
    IF k1[j] OR *no writemask* THEN
        IF (EVEX.b = 1) AND (SRC2 *is memory*)
            THEN
                IF SRC1[i+31:i] > SRC2[31:0]
                    THEN DEST[i+31:i] ← SRC1[i+31:i];
                    ELSE DEST[i+31:i] ← SRC2[31:0];
                FI;
            ELSE
                IF SRC1[i+31:i] > SRC2[i+31:i]
                    THEN DEST[i+31:i] ← SRC1[i+31:i];
                    ELSE DEST[i+31:i] ← SRC2[i+31:i];
            FI;
        FI;
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE                                ; zeroing-masking
                    THEN DEST[i+31:i] ← 0
            FI
    FI;
ENDFOR;
DEST[MAXVL-1:VL] ← 0
```

VPMAXUQ (EVEX encoded versions)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j ← 0 TO KL-1
    i ← j * 64
    IF k1[j] OR *no writemask* THEN
        IF (EVEX.b = 1) AND (SRC2 *is memory*)
            THEN
                IF SRC1[i+63:i] > SRC2[63:0]
                    THEN DEST[i+63:i] ← SRC1[i+63:i];
                    ELSE DEST[i+63:i] ← SRC2[63:0];
                FI;
            ELSE
                IF SRC1[i+31:i] > SRC2[i+31:i]
                    THEN DEST[i+63:i] ← SRC1[i+63:i];
                    ELSE DEST[i+63:i] ← SRC2[i+63:i];
            FI;
        FI;
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE                                ; zeroing-masking
                    THEN DEST[i+63:i] ← 0
            FI
    FI;
ENDFOR;
DEST[MAXVL-1:VL] ← 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VPMAXUD __m512i _mm512_max_epu32(__m512i a, __m512i b);
VPMAXUD __m512i _mm512_mask_max_epu32(__m512i s, __mmask16 k, __m512i a, __m512i b);
VPMAXUD __m512i _mm512_maskz_max_epu32(__mmask16 k, __m512i a, __m512i b);
VPMAXUQ __m512i _mm512_max_epu64(__m512i a, __m512i b);
VPMAXUQ __m512i _mm512_mask_max_epu64(__m512i s, __mmask8 k, __m512i a, __m512i b);
VPMAXUQ __m512i _mm512_maskz_max_epu64(__mmask8 k, __m512i a, __m512i b);
VPMAXUD __m256i _mm256_mask_max_epu32(__m256i s, __mmask16 k, __m256i a, __m256i b);
VPMAXUD __m256i _mm256_maskz_max_epu32(__mmask16 k, __m256i a, __m256i b);
VPMAXUQ __m256i _mm256_mask_max_epu64(__m256i s, __mmask8 k, __m256i a, __m256i b);
VPMAXUQ __m256i _mm256_maskz_max_epu64(__mmask8 k, __m256i a, __m256i b);
VPMAXUD __m128i _mm_mask_max_epu32(__m128i s, __mmask8 k, __m128i a, __m128i b);
VPMAXUD __m128i _mm_maskz_max_epu32(__mmask8 k, __m128i a, __m128i b);
VPMAXUQ __m128i _mm_mask_max_epu64(__m128i s, __mmask8 k, __m128i a, __m128i b);
VPMAXUQ __m128i _mm_maskz_max_epu64(__mmask8 k, __m128i a, __m128i b);
(V)PMAXUD __m128i _mm_max_epu32 (__m128i a, __m128i b);
VPMAXUD __m256i _mm256_max_epu32 (__m256i a, __m256i b);

SIMD Floating-Point Exceptions

None

Other Exceptions

Non-EVEX-encoded instruction, see Exceptions Type 4.
EVEX-encoded instruction, see Exceptions Type E4.

PMINSB/PMINSW—Minimum of Packed Signed Integers

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F EA /r[1] PMINSW *mm1, mm2/m64*	A	V/V	SSE	Compare signed word integers in *mm2/m64* and *mm1* and return minimum values.
66 0F 38 38 /r PMINSB xmm1, xmm2/m128	A	V/V	SSE4_1	Compare packed signed byte integers in xmm1 and xmm2/m128 and store packed minimum values in xmm1.
66 0F EA /r PMINSW xmm1, xmm2/m128	A	V/V	SSE2	Compare packed signed word integers in xmm2/m128 and xmm1 and store packed minimum values in xmm1.
VEX.NDS.128.66.0F38 38 /r VPMINSB xmm1, xmm2, xmm3/m128	B	V/V	AVX	Compare packed signed byte integers in xmm2 and xmm3/m128 and store packed minimum values in xmm1.
VEX.NDS.128.66.0F EA /r VPMINSW xmm1, xmm2, xmm3/m128	B	V/V	AVX	Compare packed signed word integers in xmm3/m128 and xmm2 and return packed minimum values in xmm1.
VEX.NDS.256.66.0F38 38 /r VPMINSB ymm1, ymm2, ymm3/m256	B	V/V	AVX2	Compare packed signed byte integers in ymm2 and ymm3/m256 and store packed minimum values in ymm1.
VEX.NDS.256.66.0F EA /r VPMINSW ymm1, ymm2, ymm3/m256	B	V/V	AVX2	Compare packed signed word integers in ymm3/m256 and ymm2 and return packed minimum values in ymm1.
EVEX.NDS.128.66.0F38.WIG 38 /r VPMINSB xmm1{k1}{z}, xmm2, xmm3/m128	C	V/V	AVX512VL AVX512BW	Compare packed signed byte integers in xmm2 and xmm3/m128 and store packed minimum values in xmm1 under writemask k1.
EVEX.NDS.256.66.0F38.WIG 38 /r VPMINSB ymm1{k1}{z}, ymm2, ymm3/m256	C	V/V	AVX512VL AVX512BW	Compare packed signed byte integers in ymm2 and ymm3/m256 and store packed minimum values in ymm1 under writemask k1.
EVEX.NDS.512.66.0F38.WIG 38 /r VPMINSB zmm1{k1}{z}, zmm2, zmm3/m512	C	V/V	AVX512BW	Compare packed signed byte integers in zmm2 and zmm3/m512 and store packed minimum values in zmm1 under writemask k1.
EVEX.NDS.128.66.0F.WIG EA /r VPMINSW xmm1{k1}{z}, xmm2, xmm3/m128	C	V/V	AVX512VL AVX512BW	Compare packed signed word integers in xmm2 and xmm3/m128 and store packed minimum values in xmm1 under writemask k1.
EVEX.NDS.256.66.0F.WIG EA /r VPMINSW ymm1{k1}{z}, ymm2, ymm3/m256	C	V/V	AVX512VL AVX512BW	Compare packed signed word integers in ymm2 and ymm3/m256 and store packed minimum values in ymm1 under writemask k1.
EVEX.NDS.512.66.0F.WIG EA /r VPMINSW zmm1{k1}{z}, zmm2, zmm3/m512	C	V/V	AVX512BW	Compare packed signed word integers in zmm2 and zmm3/m512 and store packed minimum values in zmm1 under writemask k1.

NOTES:

1. See note in Section 2.4, "AVX and SSE Instruction Exception Specification" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A* and Section 22.25.3, "Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	NA	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
B	NA	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA
C	Full Vector Mem	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA

Description

Performs a SIMD compare of the packed signed byte, word, or dword integers in the second source operand and the first source operand and returns the minimum value for each pair of integers to the destination operand.

Legacy SSE version PMINSW: The source operand can be an MMX technology register or a 64-bit memory location. The destination operand can be an MMX technology register.

128-bit Legacy SSE version: The first source and destination operands are XMM registers. The second source operand is an XMM register or a 128-bit memory location. Bits (MAXVL-1:128) of the corresponding destination register remain unchanged.

VEX.128 encoded version: The first source and destination operands are XMM registers. The second source operand is an XMM register or a 128-bit memory location. Bits (MAXVL-1:128) of the corresponding destination register are zeroed.

VEX.256 encoded version: The second source operand can be an YMM register or a 256-bit memory location. The first source and destination operands are YMM registers.

EVEX encoded versions: The first source operand is a ZMM/YMM/XMM register; The second source operand is a ZMM/YMM/XMM register or a 512/256/128-bit memory location. The destination operand is conditionally updated based on writemask k1.

Operation

PMINSW (64-bit operands)

```
IF DEST[15:0] < SRC[15:0] THEN
    DEST[15:0] ← DEST[15:0];
ELSE
    DEST[15:0] ← SRC[15:0]; FI;
(* Repeat operation for 2nd and 3rd words in source and destination operands *)
IF DEST[63:48] < SRC[63:48] THEN
    DEST[63:48] ← DEST[63:48];
ELSE
    DEST[63:48] ← SRC[63:48]; FI;
```

PMINSB (128-bit Legacy SSE version)

```
IF DEST[7:0] < SRC[7:0] THEN
    DEST[7:0] ← DEST[7:0];
ELSE
    DEST[15:0] ← SRC[7:0]; FI;
(* Repeat operation for 2nd through 15th bytes in source and destination operands *)
IF DEST[127:120] < SRC[127:120] THEN
    DEST[127:120] ← DEST[127:120];
ELSE
    DEST[127:120] ← SRC[127:120]; FI;
DEST[MAXVL-1:128] (Unmodified)
```

VPMINSB (VEX.128 encoded version)

```
IF SRC1[7:0] < SRC2[7:0] THEN
    DEST[7:0] ← SRC1[7:0];
ELSE
    DEST[7:0] ← SRC2[7:0]; FI;
(* Repeat operation for 2nd through 15th bytes in source and destination operands *)
IF SRC1[127:120] < SRC2[127:120] THEN
    DEST[127:120] ← SRC1[127:120];
ELSE
    DEST[127:120] ← SRC2[127:120]; FI;
DEST[MAXVL-1:128] ← 0
```

VPMINSB (VEX.256 encoded version)

```
    IF SRC1[7:0] < SRC2[7:0] THEN
        DEST[7:0] ← SRC1[7:0];
    ELSE
        DEST[15:0] ← SRC2[7:0]; FI;
    (* Repeat operation for 2nd through 31st bytes in source and destination operands *)
    IF SRC1[255:248] < SRC2[255:248] THEN
        DEST[255:248] ← SRC1[255:248];
    ELSE
        DEST[255:248] ← SRC2[255:248]; FI;
DEST[MAXVL-1:256] ← 0
```

VPMINSB (EVEX encoded versions)

```
(KL, VL) = (16, 128), (32, 256), (64, 512)
FOR j ← 0 TO KL-1
    i ← j * 8
    IF k1[j] OR *no writemask* THEN
        IF SRC1[i+7:i] < SRC2[i+7:i]
            THEN DEST[i+7:i] ← SRC1[i+7:i];
            ELSE DEST[i+7:i] ← SRC2[i+7:i];
        FI;
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+7:i] remains unchanged*
                ELSE                            ; zeroing-masking
                    DEST[i+7:i] ← 0
            FI
    FI;
ENDFOR;
DEST[MAXVL-1:VL] ← 0
```

PMINSW (128-bit Legacy SSE version)

```
    IF DEST[15:0] < SRC[15:0] THEN
        DEST[15:0] ← DEST[15:0];
    ELSE
        DEST[15:0] ← SRC[15:0]; FI;
    (* Repeat operation for 2nd through 7th words in source and destination operands *)
    IF DEST[127:112] < SRC[127:112] THEN
        DEST[127:112] ← DEST[127:112];
    ELSE
        DEST[127:112] ← SRC[127:112]; FI;
DEST[MAXVL-1:128] (Unmodified)
```

VPMINSW (VEX.128 encoded version)

```
    IF SRC1[15:0] < SRC2[15:0] THEN
        DEST[15:0] ← SRC1[15:0];
    ELSE
        DEST[15:0] ← SRC2[15:0]; FI;
    (* Repeat operation for 2nd through 7th words in source and destination operands *)
    IF SRC1[127:112] < SRC2[127:112] THEN
        DEST[127:112] ← SRC1[127:112];
    ELSE
        DEST[127:112] ← SRC2[127:112]; FI;
DEST[MAXVL-1:128] ← 0
```

VPMINSW (VEX.256 encoded version)

```
IF SRC1[15:0] < SRC2[15:0] THEN
    DEST[15:0] ← SRC1[15:0];
ELSE
    DEST[15:0] ← SRC2[15:0]; FI;
(* Repeat operation for 2nd through 15th words in source and destination operands *)
IF SRC1[255:240] < SRC2[255:240] THEN
    DEST[255:240] ← SRC1[255:240];
ELSE
    DEST[255:240] ← SRC2[255:240]; FI;
DEST[MAXVL-1:256] ← 0
```

VPMINSW (EVEX encoded versions)

```
(KL, VL) = (8, 128), (16, 256), (32, 512)
FOR j ← 0 TO KL-1
    i ← j * 16
    IF k1[j] OR *no writemask* THEN
        IF SRC1[i+15:i] < SRC2[i+15:i]
            THEN DEST[i+15:i] ← SRC1[i+15:i];
            ELSE DEST[i+15:i] ← SRC2[i+15:i];
        FI;
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+15:i] remains unchanged*
                ELSE                                ; zeroing-masking
                    DEST[i+15:i] ← 0
            FI
    FI;
ENDFOR;
DEST[MAXVL-1:VL] ← 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VPMINSB __m512i _mm512_min_epi8(__m512i a, __m512i b);
VPMINSB __m512i _mm512_mask_min_epi8(__m512i s, __mmask64 k, __m512i a, __m512i b);
VPMINSB __m512i _mm512_maskz_min_epi8(__mmask64 k, __m512i a, __m512i b);
VPMINSW __m512i _mm512_min_epi16(__m512i a, __m512i b);
VPMINSW __m512i _mm512_mask_min_epi16(__m512i s, __mmask32 k, __m512i a, __m512i b);
VPMINSW __m512i _mm512_maskz_min_epi16(__mmask32 k, __m512i a, __m512i b);
VPMINSB __m256i _mm256_mask_min_epi8(__m256i s, __mmask32 k, __m256i a, __m256i b);
VPMINSB __m256i _mm256_maskz_min_epi8(__mmask32 k, __m256i a, __m256i b);
VPMINSW __m256i _mm256_mask_min_epi16(__m256i s, __mmask16 k, __m256i a, __m256i b);
VPMINSW __m256i _mm256_maskz_min_epi16(__mmask16 k, __m256i a, __m256i b);
VPMINSB __m128i _mm_mask_min_epi8(__m128i s, __mmask16 k, __m128i a, __m128i b);
VPMINSB __m128i _mm_maskz_min_epi8(__mmask16 k, __m128i a, __m128i b);
VPMINSW __m128i _mm_mask_min_epi16(__m128i s, __mmask8 k, __m128i a, __m128i b);
VPMINSW __m128i _mm_maskz_min_epi16(__mmask8 k, __m128i a, __m128i b);
(V)PMINSB __m128i _mm_min_epi8 (__m128i a, __m128i b);
(V)PMINSW __m128i _mm_min_epi16 (__m128i a, __m128i b)
VPMINSB __m256i _mm256_min_epi8 (__m256i a, __m256i b);
VPMINSW __m256i _mm256_min_epi16 (__m256i a, __m256i b)
PMINSW:__m64 _mm_min_pi16 (__m64 a, __m64 b)

SIMD Floating-Point Exceptions

None

Other Exceptions

Non-EVEX-encoded instruction, see Exceptions Type 4.
EVEX-encoded instruction, see Exceptions Type E4.nb.

#MF	(64-bit operations only) If there is a pending x87 FPU exception.

PMINSD/PMINSQ—Minimum of Packed Signed Integers

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 38 39 /r PMINSD xmm1, xmm2/m128	A	V/V	SSE4_1	Compare packed signed dword integers in xmm1 and xmm2/m128 and store packed minimum values in xmm1.
VEX.NDS.128.66.0F38.WIG 39 /r VPMINSD xmm1, xmm2, xmm3/m128	B	V/V	AVX	Compare packed signed dword integers in xmm2 and xmm3/m128 and store packed minimum values in xmm1.
VEX.NDS.256.66.0F38.WIG 39 /r VPMINSD ymm1, ymm2, ymm3/m256	B	V/V	AVX2	Compare packed signed dword integers in ymm2 and ymm3/m128 and store packed minimum values in ymm1.
EVEX.NDS.128.66.0F38.W0 39 /r VPMINSD xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst	C	V/V	AVX512VL AVX512F	Compare packed signed dword integers in xmm2 and xmm3/m128 and store packed minimum values in xmm1 under writemask k1.
EVEX.NDS.256.66.0F38.W0 39 /r VPMINSD ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst	C	V/V	AVX512VL AVX512F	Compare packed signed dword integers in ymm2 and ymm3/m256 and store packed minimum values in ymm1 under writemask k1.
EVEX.NDS.512.66.0F38.W0 39 /r VPMINSD zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst	C	V/V	AVX512F	Compare packed signed dword integers in zmm2 and zmm3/m512/m32bcst and store packed minimum values in zmm1 under writemask k1.
EVEX.NDS.128.66.0F38.W1 39 /r VPMINSQ xmm1 {k1}{z}, xmm2, xmm3/m128/m64bcst	C	V/V	AVX512VL AVX512F	Compare packed signed qword integers in xmm2 and xmm3/m128 and store packed minimum values in xmm1 under writemask k1.
EVEX.NDS.256.66.0F38.W1 39 /r VPMINSQ ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst	C	V/V	AVX512VL AVX512F	Compare packed signed qword integers in ymm2 and ymm3/m256 and store packed minimum values in ymm1 under writemask k1.
EVEX.NDS.512.66.0F38.W1 39 /r VPMINSQ zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst	C	V/V	AVX512F	Compare packed signed qword integers in zmm2 and zmm3/m512/m64bcst and store packed minimum values in zmm1 under writemask k1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	NA	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
B	NA	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA
C	Full Vector	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA

Description

Performs a SIMD compare of the packed signed dword or qword integers in the second source operand and the first source operand and returns the minimum value for each pair of integers to the destination operand.

128-bit Legacy SSE version: The first source and destination operands are XMM registers. The second source operand is an XMM register or a 128-bit memory location. Bits (MAXVL-1:128) of the corresponding destination register remain unchanged.

VEX.128 encoded version: The first source and destination operands are XMM registers. The second source operand is an XMM register or a 128-bit memory location. Bits (MAXVL-1:128) of the corresponding destination register are zeroed.

VEX.256 encoded version: The second source operand can be an YMM register or a 256-bit memory location. The first source and destination operands are YMM registers. Bits (MAXVL-1:256) of the corresponding destination register are zeroed.

EVEX encoded versions: The first source operand is a ZMM/YMM/XMM register; The second source operand is a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 32/64-bit memory location. The destination operand is conditionally updated based on writemask k1.

Operation

PMINSD (128-bit Legacy SSE version)

```
IF DEST[31:0] < SRC[31:0] THEN
    DEST[31:0] ← DEST[31:0];
ELSE
    DEST[31:0] ← SRC[31:0]; FI;
(* Repeat operation for 2nd through 7th words in source and destination operands *)
IF DEST[127:96] < SRC[127:96] THEN
    DEST[127:96] ← DEST[127:96];
ELSE
    DEST[127:96] ← SRC[127:96]; FI;
DEST[MAXVL-1:128] (Unmodified)
```

VPMINSD (VEX.128 encoded version)

```
IF SRC1[31:0] < SRC2[31:0] THEN
    DEST[31:0] ← SRC1[31:0];
ELSE
    DEST[31:0] ← SRC2[31:0]; FI;
(* Repeat operation for 2nd through 3rd dwords in source and destination operands *)
IF SRC1[127:96] < SRC2[127:96] THEN
    DEST[127:96] ← SRC1[127:96];
ELSE
    DEST[127:96] ← SRC2[127:96]; FI;
DEST[MAXVL-1:128] ← 0
```

VPMINSD (VEX.256 encoded version)

```
IF SRC1[31:0] < SRC2[31:0] THEN
    DEST[31:0] ← SRC1[31:0];
ELSE
    DEST[31:0] ← SRC2[31:0]; FI;
(* Repeat operation for 2nd through 7th dwords in source and destination operands *)
IF SRC1[255:224] < SRC2[255:224] THEN
    DEST[255:224] ← SRC1[255:224];
ELSE
    DEST[255:224] ← SRC2[255:224]; FI;
DEST[MAXVL-1:256] ← 0
```

VPMINSD (EVEX encoded versions)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j ← 0 TO KL-1
    i ← j * 32
    IF k1[j] OR *no writemask* THEN
        IF (EVEX.b = 1) AND (SRC2 *is memory*)
            THEN
                IF SRC1[i+31:i] < SRC2[31:0]
                    THEN DEST[i+31:i] ← SRC1[i+31:i];
                    ELSE DEST[i+31:i] ← SRC2[31:0];
                FI;
            ELSE
                IF SRC1[i+31:i] < SRC2[i+31:i]
                    THEN DEST[i+31:i] ← SRC1[i+31:i];
                    ELSE DEST[i+31:i] ← SRC2[i+31:i];
            FI;
        FI;
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE                                ; zeroing-masking
                    DEST[i+31:i] ← 0
            FI
    FI;
ENDFOR;
DEST[MAXVL-1:VL] ← 0
```

VPMINSQ (EVEX encoded versions)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j ← 0 TO KL-1
    i ← j * 64
    IF k1[j] OR *no writemask* THEN
        IF (EVEX.b = 1) AND (SRC2 *is memory*)
            THEN
                IF SRC1[i+63:i] < SRC2[63:0]
                    THEN DEST[i+63:i] ← SRC1[i+63:i];
                    ELSE DEST[i+63:i] ← SRC2[63:0];
                FI;
            ELSE
                IF SRC1[i+63:i] < SRC2[i+63:i]
                    THEN DEST[i+63:i] ← SRC1[i+63:i];
                    ELSE DEST[i+63:i] ← SRC2[i+63:i];
            FI;
        FI;
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE                                ; zeroing-masking
                    DEST[i+63:i] ← 0
            FI
    FI;
ENDFOR;
DEST[MAXVL-1:VL] ← 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VPMINSD __m512i _mm512_min_epi32(__m512i a, __m512i b);
VPMINSD __m512i _mm512_mask_min_epi32(__m512i s, __mmask16 k, __m512i a, __m512i b);
VPMINSD __m512i _mm512_maskz_min_epi32(__mmask16 k, __m512i a, __m512i b);
VPMINSQ __m512i _mm512_min_epi64(__m512i a, __m512i b);
VPMINSQ __m512i _mm512_mask_min_epi64(__m512i s, __mmask8 k, __m512i a, __m512i b);
VPMINSQ __m512i _mm512_maskz_min_epi64(__mmask8 k, __m512i a, __m512i b);
VPMINSD __m256i _mm256_mask_min_epi32(__m256i s, __mmask16 k, __m256i a, __m256i b);
VPMINSD __m256i _mm256_maskz_min_epi32(__mmask16 k, __m256i a, __m256i b);
VPMINSQ __m256i _mm256_mask_min_epi64(__m256i s, __mmask8 k, __m256i a, __m256i b);
VPMINSQ __m256i _mm256_maskz_min_epi64(__mmask8 k, __m256i a, __m256i b);
VPMINSD __m128i _mm_mask_min_epi32(__m128i s, __mmask8 k, __m128i a, __m128i b);
VPMINSD __m128i _mm_maskz_min_epi32(__mmask8 k, __m128i a, __m128i b);
VPMINSQ __m128i _mm_mask_min_epi64(__m128i s, __mmask8 k, __m128i a, __m128i b);
VPMINSQ __m128i _mm_maskz_min_epu64(__mmask8 k, __m128i a, __m128i b);
(V)PMINSD __m128i _mm_min_epi32 (__m128i a, __m128i b);
VPMINSD __m256i _mm256_min_epi32 (__m256i a, __m256i b);

SIMD Floating-Point Exceptions

None

Other Exceptions

Non-EVEX-encoded instruction, see Exceptions Type 4.
EVEX-encoded instruction, see Exceptions Type E4.

PMINUB/PMINUW—Minimum of Packed Unsigned Integers

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F DA /r[1] PMINUB *mm1, mm2/m64*	A	V/V	SSE	Compare unsigned byte integers in *mm2/m64* and *mm1* and returns minimum values.
66 0F DA /r PMINUB xmm1, xmm2/m128	A	V/V	SSE2	Compare packed unsigned byte integers in xmm1 and xmm2/m128 and store packed minimum values in xmm1.
66 0F 38 3A/r PMINUW xmm1, xmm2/m128	A	V/V	SSE4_1	Compare packed unsigned word integers in xmm2/m128 and xmm1 and store packed minimum values in xmm1.
VEX.NDS.128.66.0F DA /r VPMINUB xmm1, xmm2, xmm3/m128	B	V/V	AVX	Compare packed unsigned byte integers in xmm2 and xmm3/m128 and store packed minimum values in xmm1.
VEX.NDS.128.66.0F38 3A/r VPMINUW xmm1, xmm2, xmm3/m128	B	V/V	AVX	Compare packed unsigned word integers in xmm3/m128 and xmm2 and return packed minimum values in xmm1.
VEX.NDS.256.66.0F DA /r VPMINUB ymm1, ymm2, ymm3/m256	B	V/V	AVX2	Compare packed unsigned byte integers in ymm2 and ymm3/m256 and store packed minimum values in ymm1.
VEX.NDS.256.66.0F38 3A/r VPMINUW ymm1, ymm2, ymm3/m256	B	V/V	AVX2	Compare packed unsigned word integers in ymm3/m256 and ymm2 and return packed minimum values in ymm1.
EVEX.NDS.128.66.0F DA /r VPMINUB xmm1 {k1}{z}, xmm2, xmm3/m128	C	V/V	AVX512VL AVX512BW	Compare packed unsigned byte integers in xmm2 and xmm3/m128 and store packed minimum values in xmm1 under writemask k1.
EVEX.NDS.256.66.0F DA /r VPMINUB ymm1 {k1}{z}, ymm2, ymm3/m256	C	V/V	AVX512VL AVX512BW	Compare packed unsigned byte integers in ymm2 and ymm3/m256 and store packed minimum values in ymm1 under writemask k1.
EVEX.NDS.512.66.0F DA /r VPMINUB zmm1 {k1}{z}, zmm2, zmm3/m512	C	V/V	AVX512BW	Compare packed unsigned byte integers in zmm2 and zmm3/m512 and store packed minimum values in zmm1 under writemask k1.
EVEX.NDS.128.66.0F38 3A/r VPMINUW xmm1{k1}{z}, xmm2, xmm3/m128	C	V/V	AVX512VL AVX512BW	Compare packed unsigned word integers in xmm3/m128 and xmm2 and return packed minimum values in xmm1 under writemask k1.
EVEX.NDS.256.66.0F38 3A/r VPMINUW ymm1{k1}{z}, ymm2, ymm3/m256	C	V/V	AVX512VL AVX512BW	Compare packed unsigned word integers in ymm3/m256 and ymm2 and return packed minimum values in ymm1 under writemask k1.
EVEX.NDS.512.66.0F38 3A/r VPMINUW zmm1{k1}{z}, zmm2, zmm3/m512	C	V/V	AVX512BW	Compare packed unsigned word integers in zmm3/m512 and zmm2 and return packed minimum values in zmm1 under writemask k1.

NOTES:

1. See note in Section 2.4, "AVX and SSE Instruction Exception Specification" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A* and Section 22.25.3, "Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	NA	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
B	NA	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA
C	Full Vector Mem	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA

Description

Performs a SIMD compare of the packed unsigned byte or word integers in the second source operand and the first source operand and returns the minimum value for each pair of integers to the destination operand.

Legacy SSE version PMINUB: The source operand can be an MMX technology register or a 64-bit memory location. The destination operand can be an MMX technology register.

128-bit Legacy SSE version: The first source and destination operands are XMM registers. The second source operand is an XMM register or a 128-bit memory location. Bits (MAXVL-1:128) of the corresponding destination register remain unchanged.

VEX.128 encoded version: The first source and destination operands are XMM registers. The second source operand is an XMM register or a 128-bit memory location. Bits (MAXVL-1:128) of the corresponding destination register are zeroed.

VEX.256 encoded version: The second source operand can be an YMM register or a 256-bit memory location. The first source and destination operands are YMM registers.

EVEX encoded versions: The first source operand is a ZMM/YMM/XMM register; The second source operand is a ZMM/YMM/XMM register or a 512/256/128-bit memory location. The destination operand is conditionally updated based on writemask k1.

Operation

PMINUB (for 64-bit operands)

```
IF DEST[7:0] < SRC[17:0] THEN
    DEST[7:0] ← DEST[7:0];
ELSE
    DEST[7:0] ← SRC[7:0]; FI;
(* Repeat operation for 2nd through 7th bytes in source and destination operands *)
IF DEST[63:56] < SRC[63:56] THEN
    DEST[63:56] ← DEST[63:56];
ELSE
    DEST[63:56] ← SRC[63:56]; FI;
```

PMINUB instruction for 128-bit operands:

```
IF DEST[7:0] < SRC[7:0] THEN
    DEST[7:0] ← DEST[7:0];
ELSE
    DEST[15:0] ← SRC[7:0]; FI;
(* Repeat operation for 2nd through 15th bytes in source and destination operands *)
IF DEST[127:120] < SRC[127:120] THEN
    DEST[127:120] ← DEST[127:120];
ELSE
    DEST[127:120] ← SRC[127:120]; FI;
DEST[MAXVL-1:128] (Unmodified)
```

VPMINUB (VEX.128 encoded version)

```
IF SRC1[7:0] < SRC2[7:0] THEN
    DEST[7:0] ← SRC1[7:0];
ELSE
    DEST[7:0] ← SRC2[7:0]; FI;
(* Repeat operation for 2nd through 15th bytes in source and destination operands *)
IF SRC1[127:120] < SRC2[127:120] THEN
    DEST[127:120] ← SRC1[127:120];
ELSE
    DEST[127:120] ← SRC2[127:120]; FI;
DEST[MAXVL-1:128] ← 0
```

VPMINUB (VEX.256 encoded version)

```
    IF SRC1[7:0] < SRC2[7:0] THEN
        DEST[7:0] ← SRC1[7:0];
    ELSE
        DEST[15:0] ← SRC2[7:0]; FI;
    (* Repeat operation for 2nd through 31st bytes in source and destination operands *)
    IF SRC1[255:248] < SRC2[255:248] THEN
        DEST[255:248] ← SRC1[255:248];
    ELSE
        DEST[255:248] ← SRC2[255:248]; FI;
DEST[MAXVL-1:256] ← 0
```

VPMINUB (EVEX encoded versions)

```
(KL, VL) = (16, 128), (32, 256), (64, 512)
FOR j ← 0 TO KL-1
    i ← j * 8
    IF k1[j] OR *no writemask* THEN
        IF SRC1[i+7:i] < SRC2[i+7:i]
            THEN DEST[i+7:i] ← SRC1[i+7:i];
            ELSE DEST[i+7:i] ← SRC2[i+7:i];
        FI;
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+7:i] remains unchanged*
                ELSE                                ; zeroing-masking
                    DEST[i+7:i] ← 0
            FI
    FI;
ENDFOR;
DEST[MAXVL-1:VL] ← 0
```

PMINUW instruction for 128-bit operands:

```
    IF DEST[15:0] < SRC[15:0] THEN
        DEST[15:0] ← DEST[15:0];
    ELSE
        DEST[15:0] ← SRC[15:0]; FI;
    (* Repeat operation for 2nd through 7th words in source and destination operands *)
    IF DEST[127:112] < SRC[127:112] THEN
        DEST[127:112] ← DEST[127:112];
    ELSE
        DEST[127:112] ← SRC[127:112]; FI;
DEST[MAXVL-1:128] (Unmodified)
```

VPMINUW (VEX.128 encoded version)

```
    IF SRC1[15:0] < SRC2[15:0] THEN
        DEST[15:0] ← SRC1[15:0];
    ELSE
        DEST[15:0] ← SRC2[15:0]; FI;
    (* Repeat operation for 2nd through 7th words in source and destination operands *)
    IF SRC1[127:112] < SRC2[127:112] THEN
        DEST[127:112] ← SRC1[127:112];
    ELSE
        DEST[127:112] ← SRC2[127:112]; FI;
DEST[MAXVL-1:128] ← 0
```

VPMINUW (VEX.256 encoded version)

```
    IF SRC1[15:0] < SRC2[15:0] THEN
        DEST[15:0] ← SRC1[15:0];
    ELSE
        DEST[15:0] ← SRC2[15:0]; FI;
    (* Repeat operation for 2nd through 15th words in source and destination operands *)
    IF SRC1[255:240] < SRC2[255:240] THEN
        DEST[255:240] ← SRC1[255:240];
    ELSE
        DEST[255:240] ← SRC2[255:240]; FI;
DEST[MAXVL-1:256] ← 0
```

VPMINUW (EVEX encoded versions)

```
(KL, VL) = (8, 128), (16, 256), (32, 512)
FOR j ← 0 TO KL-1
    i ← j * 16
    IF k1[j] OR *no writemask* THEN
        IF SRC1[i+15:i] < SRC2[i+15:i]
            THEN DEST[i+15:i] ← SRC1[i+15:i];
            ELSE DEST[i+15:i] ← SRC2[i+15:i];
        FI;
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+15:i] remains unchanged*
                ELSE                                ; zeroing-masking
                    DEST[i+15:i] ← 0
            FI
    FI;
ENDFOR;
DEST[MAXVL-1:VL] ← 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VPMINUB __m512i _mm512_min_epu8(__m512i a, __m512i b);
VPMINUB __m512i _mm512_mask_min_epu8(__m512i s, __mmask64 k, __m512i a, __m512i b);
VPMINUB __m512i _mm512_maskz_min_epu8(__mmask64 k, __m512i a, __m512i b);
VPMINUW __m512i _mm512_min_epu16(__m512i a, __m512i b);
VPMINUW __m512i _mm512_mask_min_epu16(__m512i s, __mmask32 k, __m512i a, __m512i b);
VPMINUW __m512i _mm512_maskz_min_epu16(__mmask32 k, __m512i a, __m512i b);
VPMINUB __m256i _mm256_mask_min_epu8(__m256i s, __mmask32 k, __m256i a, __m256i b);
VPMINUB __m256i _mm256_maskz_min_epu8(__mmask32 k, __m256i a, __m256i b);
VPMINUW __m256i _mm256_mask_min_epu16(__m256i s, __mmask16 k, __m256i a, __m256i b);
VPMINUW __m256i _mm256_maskz_min_epu16(__mmask16 k, __m256i a, __m256i b);
VPMINUB __m128i _mm_mask_min_epu8(__m128i s, __mmask16 k, __m128i a, __m128i b);
VPMINUB __m128i _mm_maskz_min_epu8(__mmask16 k, __m128i a, __m128i b);
VPMINUW __m128i _mm_mask_min_epu16(__m128i s, __mmask8 k, __m128i a, __m128i b);
VPMINUW __m128i _mm_maskz_min_epu16(__mmask8 k, __m128i a, __m128i b);
(V)PMINUB __m128i _mm_min_epu8 (__m128i a, __m128i b)
(V)PMINUW __m128i _mm_min_epu16 (__m128i a, __m128i b);
VPMINUB __m256i _mm256_min_epu8 (__m256i a, __m256i b)
VPMINUW __m256i _mm256_min_epu16 (__m256i a, __m256i b);
PMINUB: __m64 _m_min_pu8 (__m64 a, __m64 b)

SIMD Floating-Point Exceptions

None

Other Exceptions

Non-EVEX-encoded instruction, see Exceptions Type 4.
EVEX-encoded instruction, see Exceptions Type E4.nb.

PMINUD/PMINUQ—Minimum of Packed Unsigned Integers

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 38 3B /r PMINUD xmm1, xmm2/m128	A	V/V	SSE4_1	Compare packed unsigned dword integers in xmm1 and xmm2/m128 and store packed minimum values in xmm1.
VEX.NDS.128.66.0F38.WIG 3B /r VPMINUD xmm1, xmm2, xmm3/m128	B	V/V	AVX	Compare packed unsigned dword integers in xmm2 and xmm3/m128 and store packed minimum values in xmm1.
VEX.NDS.256.66.0F38.WIG 3B /r VPMINUD ymm1, ymm2, ymm3/m256	B	V/V	AVX2	Compare packed unsigned dword integers in ymm2 and ymm3/m256 and store packed minimum values in ymm1.
EVEX.NDS.128.66.0F38.W0 3B /r VPMINUD xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst	C	V/V	AVX512VL AVX512F	Compare packed unsigned dword integers in xmm2 and xmm3/m128/m32bcst and store packed minimum values in xmm1 under writemask k1.
EVEX.NDS.256.66.0F38.W0 3B /r VPMINUD ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst	C	V/V	AVX512VL AVX512F	Compare packed unsigned dword integers in ymm2 and ymm3/m256/m32bcst and store packed minimum values in ymm1 under writemask k1.
EVEX.NDS.512.66.0F38.W0 3B /r VPMINUD zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst	C	V/V	AVX512F	Compare packed unsigned dword integers in zmm2 and zmm3/m512/m32bcst and store packed minimum values in zmm1 under writemask k1.
EVEX.NDS.128.66.0F38.W1 3B /r VPMINUQ xmm1 {k1}{z}, xmm2, xmm3/m128/m64bcst	C	V/V	AVX512VL AVX512F	Compare packed unsigned qword integers in xmm2 and xmm3/m128/m64bcst and store packed minimum values in xmm1 under writemask k1.
EVEX.NDS.256.66.0F38.W1 3B /r VPMINUQ ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst	C	V/V	AVX512VL AVX512F	Compare packed unsigned qword integers in ymm2 and ymm3/m256/m64bcst and store packed minimum values in ymm1 under writemask k1.
EVEX.NDS.512.66.0F38.W1 3B /r VPMINUQ zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst	C	V/V	AVX512F	Compare packed unsigned qword integers in zmm2 and zmm3/m512/m64bcst and store packed minimum values in zmm1 under writemask k1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	NA	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
B	NA	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA
C	Full Vector	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA

Description

Performs a SIMD compare of the packed unsigned dword/qword integers in the second source operand and the first source operand and returns the minimum value for each pair of integers to the destination operand.

128-bit Legacy SSE version: The first source and destination operands are XMM registers. The second source operand is an XMM register or a 128-bit memory location. Bits (MAXVL-1:128) of the corresponding destination register remain unchanged.

VEX.128 encoded version: The first source and destination operands are XMM registers. The second source operand is an XMM register or a 128-bit memory location. Bits (MAXVL-1:128) of the corresponding destination register are zeroed.

VEX.256 encoded version: The second source operand can be an YMM register or a 256-bit memory location. The first source and destination operands are YMM registers. Bits (MAXVL-1:256) of the corresponding destination register are zeroed.

EVEX encoded versions: The first source operand is a ZMM/YMM/XMM register; The second source operand is a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 32/64-bit memory location. The destination operand is conditionally updated based on writemask k1.

Operation

PMINUD (128-bit Legacy SSE version)

```
PMINUD instruction for 128-bit operands:
    IF DEST[31:0] < SRC[31:0] THEN
        DEST[31:0] ← DEST[31:0];
    ELSE
        DEST[31:0] ← SRC[31:0]; FI;
    (* Repeat operation for 2nd through 7th words in source and destination operands *)
    IF DEST[127:96] < SRC[127:96] THEN
        DEST[127:96] ← DEST[127:96];
    ELSE
        DEST[127:96] ← SRC[127:96]; FI;
DEST[MAXVL-1:128] (Unmodified)
```

VPMINUD (VEX.128 encoded version)

```
VPMINUD instruction for 128-bit operands:
    IF SRC1[31:0] < SRC2[31:0] THEN
        DEST[31:0] ← SRC1[31:0];
    ELSE
        DEST[31:0] ← SRC2[31:0]; FI;
    (* Repeat operation for 2nd through 3rd dwords in source and destination operands *)
    IF SRC1[127:96] < SRC2[127:96] THEN
        DEST[127:96] ← SRC1[127:96];
    ELSE
        DEST[127:96] ← SRC2[127:96]; FI;
DEST[MAXVL-1:128] ← 0
```

VPMINUD (VEX.256 encoded version)

```
VPMINUD instruction for 128-bit operands:
    IF SRC1[31:0] < SRC2[31:0] THEN
        DEST[31:0] ← SRC1[31:0];
    ELSE
        DEST[31:0] ← SRC2[31:0]; FI;
    (* Repeat operation for 2nd through 7th dwords in source and destination operands *)
    IF SRC1[255:224] < SRC2[255:224] THEN
        DEST[255:224] ← SRC1[255:224];
    ELSE
        DEST[255:224] ← SRC2[255:224]; FI;
DEST[MAXVL-1:256] ← 0
```

VPMINUD (EVEX encoded versions)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j ← 0 TO KL-1
    i ← j * 32
    IF k1[j] OR *no writemask* THEN
        IF (EVEX.b = 1) AND (SRC2 *is memory*)
            THEN
                IF SRC1[i+31:i] < SRC2[31:0]
                    THEN DEST[i+31:i] ← SRC1[i+31:i];
                    ELSE DEST[i+31:i] ← SRC2[31:0];
                FI;
            ELSE
                IF SRC1[i+31:i] < SRC2[i+31:i]
                    THEN DEST[i+31:i] ← SRC1[i+31:i];
                    ELSE DEST[i+31:i] ← SRC2[i+31:i];
            FI;
        FI;
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE                                ; zeroing-masking
                    DEST[i+31:i] ← 0
            FI
    FI;
ENDFOR;
DEST[MAXVL-1:VL] ← 0
```

VPMINUQ (EVEX encoded versions)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j ← 0 TO KL-1
    i ← j * 64
    IF k1[j] OR *no writemask* THEN
        IF (EVEX.b = 1) AND (SRC2 *is memory*)
            THEN
                IF SRC1[i+63:i] < SRC2[63:0]
                    THEN DEST[i+63:i] ← SRC1[i+63:i];
                    ELSE DEST[i+63:i] ← SRC2[63:0];
                FI;
            ELSE
                IF SRC1[i+63:i] < SRC2[i+63:i]
                    THEN DEST[i+63:i] ← SRC1[i+63:i];
                    ELSE DEST[i+63:i] ← SRC2[i+63:i];
            FI;
        FI;
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE                                ; zeroing-masking
                    DEST[i+63:i] ← 0
            FI
    FI;
ENDFOR;
DEST[MAXVL-1:VL] ← 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VPMINUD __m512i _mm512_min_epu32(__m512i a, __m512i b);
VPMINUD __m512i _mm512_mask_min_epu32(__m512i s, __mmask16 k, __m512i a, __m512i b);
VPMINUD __m512i _mm512_maskz_min_epu32(__mmask16 k, __m512i a, __m512i b);
VPMINUQ __m512i _mm512_min_epu64(__m512i a, __m512i b);
VPMINUQ __m512i _mm512_mask_min_epu64(__m512i s, __mmask8 k, __m512i a, __m512i b);
VPMINUQ __m512i _mm512_maskz_min_epu64(__mmask8 k, __m512i a, __m512i b);
VPMINUD __m256i _mm256_mask_min_epu32(__m256i s, __mmask16 k, __m256i a, __m256i b);
VPMINUD __m256i _mm256_maskz_min_epu32(__mmask16 k, __m256i a, __m256i b);
VPMINUQ __m256i _mm256_mask_min_epu64(__m256i s, __mmask8 k, __m256i a, __m256i b);
VPMINUQ __m256i _mm256_maskz_min_epu64(__mmask8 k, __m256i a, __m256i b);
VPMINUD __m128i _mm_mask_min_epu32(__m128i s, __mmask8 k, __m128i a, __m128i b);
VPMINUD __m128i _mm_maskz_min_epu32(__mmask8 k, __m128i a, __m128i b);
VPMINUQ __m128i _mm_mask_min_epu64(__m128i s, __mmask8 k, __m128i a, __m128i b);
VPMINUQ __m128i _mm_maskz_min_epu64(__mmask8 k, __m128i a, __m128i b);
(V)PMINUD __m128i _mm_min_epu32 (__m128i a, __m128i b);
VPMINUD __m256i _mm256_min_epu32 (__m256i a, __m256i b);

SIMD Floating-Point Exceptions

None

Other Exceptions

Non-EVEX-encoded instruction, see Exceptions Type 4.
EVEX-encoded instruction, see Exceptions Type E4.

PMOVMSKB—Move Byte Mask

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F D7 /r[1] PMOVMSKB *reg, mm*	RM	V/V	SSE	Move a byte mask of *mm* to *reg*. The upper bits of r32 or r64 are zeroed
66 0F D7 /r PMOVMSKB *reg, xmm*	RM	V/V	SSE2	Move a byte mask of *xmm* to *reg*. The upper bits of r32 or r64 are zeroed
VEX.128.66.0F.WIG D7 /r VPMOVMSKB *reg, xmm1*	RM	V/V	AVX	Move a byte mask of *xmm1* to *reg*. The upper bits of r32 or r64 are filled with zeros.
VEX.256.66.0F.WIG D7 /r VPMOVMSKB *reg, ymm1*	RM	V/V	AVX2	Move a 32-bit mask of *ymm1* to *reg*. The upper bits of r64 are filled with zeros.

NOTES:

1. See note in Section 2.4, "AVX and SSE Instruction Exception Specification" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A* and Section 22.25.3, "Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

Description

Creates a mask made up of the most significant bit of each byte of the source operand (second operand) and stores the result in the low byte or word of the destination operand (first operand).

The byte mask is 8 bits for 64-bit source operand, 16 bits for 128-bit source operand and 32 bits for 256-bit source operand. The destination operand is a general-purpose register.

In 64-bit mode, the instruction can access additional registers (XMM8-XMM15, R8-R15) when used with a REX.R prefix. The default operand size is 64-bit in 64-bit mode.

Legacy SSE version: The source operand is an MMX technology register.

128-bit Legacy SSE version: The source operand is an XMM register.

VEX.128 encoded version: The source operand is an XMM register.

VEX.256 encoded version: The source operand is a YMM register.

Note: VEX.vvvv is reserved and must be 1111b.

Operation

PMOVMSKB (with 64-bit source operand and r32)

```
r32[0] ← SRC[7];
r32[1] ← SRC[15];
(* Repeat operation for bytes 2 through 6 *)
r32[7] ← SRC[63];
r32[31:8] ← ZERO_FILL;
```

(V)PMOVMSKB (with 128-bit source operand and r32)

```
r32[0] ← SRC[7];
r32[1] ← SRC[15];
(* Repeat operation for bytes 2 through 14 *)
r32[15] ← SRC[127];
r32[31:16] ← ZERO_FILL;
```

VPMOVMSKB (with 256-bit source operand and r32)
r32[0] ← SRC[7];
r32[1] ← SRC[15];
(* Repeat operation for bytes 3rd through 31*)
r32[31] ← SRC[255];

PMOVMSKB (with 64-bit source operand and r64)
r64[0] ← SRC[7];
r64[1] ← SRC[15];
(* Repeat operation for bytes 2 through 6 *)
r64[7] ← SRC[63];
r64[63:8] ← ZERO_FILL;

(V)PMOVMSKB (with 128-bit source operand and r64)
r64[0] ← SRC[7];
r64[1] ← SRC[15];
(* Repeat operation for bytes 2 through 14 *)
r64[15] ← SRC[127];
r64[63:16] ← ZERO_FILL;

VPMOVMSKB (with 256-bit source operand and r64)
r64[0] ← SRC[7];
r64[1] ← SRC[15];
(* Repeat operation for bytes 2 through 31*)
r64[31] ← SRC[255];
r64[63:32] ← ZERO_FILL;

Intel C/C++ Compiler Intrinsic Equivalent

PMOVMSKB: int _mm_movemask_pi8(__m64 a)

(V)PMOVMSKB: int _mm_movemask_epi8 (__m128i a)

VPMOVMSKB: int _mm256_movemask_epi8 (__m256i a)

Flags Affected

None.

Numeric Exceptions

None.

Other Exceptions

See Exceptions Type 7; additionally

#UD	If VEX.vvvv ≠ 1111B.

PMOVSX—Packed Move with Sign Extend

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0f 38 20 /r PMOVSXBW xmm1, xmm2/m64	A	V/V	SSE4_1	Sign extend 8 packed 8-bit integers in the low 8 bytes of xmm2/m64 to 8 packed 16-bit integers in xmm1.
66 0f 38 21 /r PMOVSXBD xmm1, xmm2/m32	A	V/V	SSE4_1	Sign extend 4 packed 8-bit integers in the low 4 bytes of xmm2/m32 to 4 packed 32-bit integers in xmm1.
66 0f 38 22 /r PMOVSXBQ xmm1, xmm2/m16	A	V/V	SSE4_1	Sign extend 2 packed 8-bit integers in the low 2 bytes of xmm2/m16 to 2 packed 64-bit integers in xmm1.
66 0f 38 23/r PMOVSXWD xmm1, xmm2/m64	A	V/V	SSE4_1	Sign extend 4 packed 16-bit integers in the low 8 bytes of xmm2/m64 to 4 packed 32-bit integers in xmm1.
66 0f 38 24 /r PMOVSXWQ xmm1, xmm2/m32	A	V/V	SSE4_1	Sign extend 2 packed 16-bit integers in the low 4 bytes of xmm2/m32 to 2 packed 64-bit integers in xmm1.
66 0f 38 25 /r PMOVSXDQ xmm1, xmm2/m64	A	V/V	SSE4_1	Sign extend 2 packed 32-bit integers in the low 8 bytes of xmm2/m64 to 2 packed 64-bit integers in xmm1.
VEX.128.66.0F38.WIG 20 /r VPMOVSXBW xmm1, xmm2/m64	A	V/V	AVX	Sign extend 8 packed 8-bit integers in the low 8 bytes of xmm2/m64 to 8 packed 16-bit integers in xmm1.
VEX.128.66.0F38.WIG 21 /r VPMOVSXBD xmm1, xmm2/m32	A	V/V	AVX	Sign extend 4 packed 8-bit integers in the low 4 bytes of xmm2/m32 to 4 packed 32-bit integers in xmm1.
VEX.128.66.0F38.WIG 22 /r VPMOVSXBQ xmm1, xmm2/m16	A	V/V	AVX	Sign extend 2 packed 8-bit integers in the low 2 bytes of xmm2/m16 to 2 packed 64-bit integers in xmm1.
VEX.128.66.0F38.WIG 23 /r VPMOVSXWD xmm1, xmm2/m64	A	V/V	AVX	Sign extend 4 packed 16-bit integers in the low 8 bytes of xmm2/m64 to 4 packed 32-bit integers in xmm1.
VEX.128.66.0F38.WIG 24 /r VPMOVSXWQ xmm1, xmm2/m32	A	V/V	AVX	Sign extend 2 packed 16-bit integers in the low 4 bytes of xmm2/m32 to 2 packed 64-bit integers in xmm1.
VEX.128.66.0F38.WIG 25 /r VPMOVSXDQ xmm1, xmm2/m64	A	V/V	AVX	Sign extend 2 packed 32-bit integers in the low 8 bytes of xmm2/m64 to 2 packed 64-bit integers in xmm1.
VEX.256.66.0F38.WIG 20 /r VPMOVSXBW ymm1, xmm2/m128	A	V/V	AVX2	Sign extend 16 packed 8-bit integers in xmm2/m128 to 16 packed 16-bit integers in ymm1.
VEX.256.66.0F38.WIG 21 /r VPMOVSXBD ymm1, xmm2/m64	A	V/V	AVX2	Sign extend 8 packed 8-bit integers in the low 8 bytes of xmm2/m64 to 8 packed 32-bit integers in ymm1.
VEX.256.66.0F38.WIG 22 /r VPMOVSXBQ ymm1, xmm2/m32	A	V/V	AVX2	Sign extend 4 packed 8-bit integers in the low 4 bytes of xmm2/m32 to 4 packed 64-bit integers in ymm1.
VEX.256.66.0F38.WIG 23 /r VPMOVSXWD ymm1, xmm2/m128	A	V/V	AVX2	Sign extend 8 packed 16-bit integers in the low 16 bytes of xmm2/m128 to 8 packed 32-bit integers in ymm1.
VEX.256.66.0F38.WIG 24 /r VPMOVSXWQ ymm1, xmm2/m64	A	V/V	AVX2	Sign extend 4 packed 16-bit integers in the low 8 bytes of xmm2/m64 to 4 packed 64-bit integers in ymm1.
VEX.256.66.0F38.WIG 25 /r VPMOVSXDQ ymm1, xmm2/m128	A	V/V	AVX2	Sign extend 4 packed 32-bit integers in the low 16 bytes of xmm2/m128 to 4 packed 64-bit integers in ymm1.
EVEX.128.66.0F38.WIG 20 /r VPMOVSXBW xmm1 {k1}{z}, xmm2/m64	B	V/V	AVX512VL AVX512BW	Sign extend 8 packed 8-bit integers in xmm2/m64 to 8 packed 16-bit integers in zmm1.
EVEX.256.66.0F38.WIG 20 /r VPMOVSXBW ymm1 {k1}{z}, xmm2/m128	B	V/V	AVX512VL AVX512BW	Sign extend 16 packed 8-bit integers in xmm2/m128 to 16 packed 16-bit integers in ymm1.
EVEX.512.66.0F38.WIG 20 /r VPMOVSXBW zmm1 {k1}{z}, ymm2/m256	B	V/V	AVX512BW	Sign extend 32 packed 8-bit integers in ymm2/m256 to 32 packed 16-bit integers in zmm1.
EVEX.128.66.0F38.WIG 21 /r VPMOVSXBD xmm1 {k1}{z}, xmm2/m32	C	V/V	AVX512VL AVX512F	Sign extend 4 packed 8-bit integers in the low 4 bytes of xmm2/m32 to 4 packed 32-bit integers in xmm1 subject to writemask k1.

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.256.66.0F38.WIG 21 /r VPMOVSXBD ymm1 {k1}{z}, xmm2/m64	C	V/V	AVX512VL AVX512F	Sign extend 8 packed 8-bit integers in the low 8 bytes of xmm2/m64 to 8 packed 32-bit integers in ymm1 subject to writemask k1.
EVEX.512.66.0F38.WIG 21 /r VPMOVSXBD zmm1 {k1}{z}, xmm2/m128	C	V/V	AVX512F	Sign extend 16 packed 8-bit integers in the low 16 bytes of xmm2/m128 to 16 packed 32-bit integers in zmm1 subject to writemask k1.
EVEX.128.66.0F38.WIG 22 /r VPMOVSXBQ xmm1 {k1}{z}, xmm2/m16	D	V/V	AVX512VL AVX512F	Sign extend 2 packed 8-bit integers in the low 2 bytes of xmm2/m16 to 2 packed 64-bit integers in xmm1 subject to writemask k1.
EVEX.256.66.0F38.WIG 22 /r VPMOVSXBQ ymm1 {k1}{z}, xmm2/m32	D	V/V	AVX512VL AVX512F	Sign extend 4 packed 8-bit integers in the low 4 bytes of xmm2/m32 to 4 packed 64-bit integers in ymm1 subject to writemask k1.
EVEX.512.66.0F38.WIG 22 /r VPMOVSXBQ zmm1 {k1}{z}, xmm2/m64	D	V/V	AVX512F	Sign extend 8 packed 8-bit integers in the low 8 bytes of xmm2/m64 to 8 packed 64-bit integers in zmm1 subject to writemask k1.
EVEX.128.66.0F38.WIG 23 /r VPMOVSXWD xmm1 {k1}{z}, xmm2/m64	B	V/V	AVX512VL AVX512F	Sign extend 4 packed 16-bit integers in the low 8 bytes of ymm2/mem to 4 packed 32-bit integers in xmm1 subject to writemask k1.
EVEX.256.66.0F38.WIG 23 /r VPMOVSXWD ymm1 {k1}{z}, xmm2/m128	B	V/V	AVX512VL AVX512F	Sign extend 8 packed 16-bit integers in the low 16 bytes of ymm2/m128 to 8 packed 32-bit integers in ymm1 subject to writemask k1.
EVEX.512.66.0F38.WIG 23 /r VPMOVSXWD zmm1 {k1}{z}, ymm2/m256	B	V/V	AVX512F	Sign extend 16 packed 16-bit integers in the low 32 bytes of ymm2/m256 to 16 packed 32-bit integers in zmm1 subject to writemask k1.
EVEX.128.66.0F38.WIG 24 /r VPMOVSXWQ xmm1 {k1}{z}, xmm2/m32	C	V/V	AVX512VL AVX512F	Sign extend 2 packed 16-bit integers in the low 4 bytes of xmm2/m32 to 2 packed 64-bit integers in xmm1 subject to writemask k1.
EVEX.256.66.0F38.WIG 24 /r VPMOVSXWQ ymm1 {k1}{z}, xmm2/m64	C	V/V	AVX512VL AVX512F	Sign extend 4 packed 16-bit integers in the low 8 bytes of xmm2/m64 to 4 packed 64-bit integers in ymm1 subject to writemask k1.
EVEX.512.66.0F38.WIG 24 /r VPMOVSXWQ zmm1 {k1}{z}, xmm2/m128	C	V/V	AVX512F	Sign extend 8 packed 16-bit integers in the low 16 bytes of xmm2/m128 to 8 packed 64-bit integers in zmm1 subject to writemask k1.
EVEX.128.66.0F38.W0 25 /r VPMOVSXDQ xmm1 {k1}{z}, xmm2/m64	B	V/V	AVX512VL AVX512F	Sign extend 2 packed 32-bit integers in the low 8 bytes of xmm2/m64 to 2 packed 64-bit integers in zmm1 using writemask k1.
EVEX.256.66.0F38.W0 25 /r VPMOVSXDQ ymm1 {k1}{z}, xmm2/m128	B	V/V	AVX512VL AVX512F	Sign extend 4 packed 32-bit integers in the low 16 bytes of xmm2/m128 to 4 packed 64-bit integers in zmm1 using writemask k1.
EVEX.512.66.0F38.W0 25 /r VPMOVSXDQ zmm1 {k1}{z}, ymm2/m256	B	V/V	AVX512F	Sign extend 8 packed 32-bit integers in the low 32 bytes of ymm2/m256 to 8 packed 64-bit integers in zmm1 using writemask k1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	NA	ModRM:reg (w)	ModRM:r/m (r)	NA	NA
B	Half Vector Mem	ModRM:reg (w)	ModRM:r/m (r)	NA	NA
C	Quarter Vector Mem	ModRM:reg (w)	ModRM:r/m (r)	NA	NA
D	Oct Vector Mem	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

Description

Legacy and VEX encoded versions: Packed byte, word, or dword integers in the low bytes of the source operand (second operand) are sign extended to word, dword, or quadword integers and stored in packed signed bytes the destination operand.

128-bit Legacy SSE version: Bits (MAXVL-1:128) of the corresponding destination register remain unchanged.

VEX.128 and EVEX.128 encoded versions: Bits (MAXVL-1:128) of the corresponding destination register are zeroed.

VEX.256 and EVEX.256 encoded versions: Bits (MAXVL-1:256) of the corresponding destination register are zeroed.

EVEX encoded versions: Packed byte, word or dword integers starting from the low bytes of the source operand (second operand) are sign extended to word, dword or quadword integers and stored to the destination operand under the writemask. The destination register is XMM, YMM or ZMM Register.

Note: VEX.vvvv and EVEX.vvvv are reserved and must be 1111b otherwise instructions will #UD.

Operation

Packed_Sign_Extend_BYTE_to_WORD(DEST, SRC)
DEST[15:0] ←SignExtend(SRC[7:0]);
DEST[31:16] ←SignExtend(SRC[15:8]);
DEST[47:32] ←SignExtend(SRC[23:16]);
DEST[63:48] ←SignExtend(SRC[31:24]);
DEST[79:64] ←SignExtend(SRC[39:32]);
DEST[95:80] ←SignExtend(SRC[47:40]);
DEST[111:96] ←SignExtend(SRC[55:48]);
DEST[127:112] ←SignExtend(SRC[63:56]);

Packed_Sign_Extend_BYTE_to_DWORD(DEST, SRC)
DEST[31:0] ←SignExtend(SRC[7:0]);
DEST[63:32] ←SignExtend(SRC[15:8]);
DEST[95:64] ←SignExtend(SRC[23:16]);
DEST[127:96] ←SignExtend(SRC[31:24]);

Packed_Sign_Extend_BYTE_to_QWORD(DEST, SRC)
DEST[63:0] ←SignExtend(SRC[7:0]);
DEST[127:64] ←SignExtend(SRC[15:8]);

Packed_Sign_Extend_WORD_to_DWORD(DEST, SRC)
DEST[31:0] ←SignExtend(SRC[15:0]);
DEST[63:32] ←SignExtend(SRC[31:16]);
DEST[95:64] ←SignExtend(SRC[47:32]);
DEST[127:96] ←SignExtend(SRC[63:48]);

Packed_Sign_Extend_WORD_to_QWORD(DEST, SRC)
DEST[63:0] ←SignExtend(SRC[15:0]);
DEST[127:64] ←SignExtend(SRC[31:16]);

Packed_Sign_Extend_DWORD_to_QWORD(DEST, SRC)

```
DEST[63:0] ←SignExtend(SRC[31:0]);
DEST[127:64] ←SignExtend(SRC[63:32]);
```

VPMOVSXBW (EVEX encoded versions)

```
(KL, VL) = (8, 128), (16, 256), (32, 512)
Packed_Sign_Extend_BYTE_to_WORD(TMP_DEST[127:0], SRC[63:0])
IF VL >= 256
    Packed_Sign_Extend_BYTE_to_WORD(TMP_DEST[255:128], SRC[127:64])
FI;
IF VL >= 512
    Packed_Sign_Extend_BYTE_to_WORD(TMP_DEST[383:256], SRC[191:128])
    Packed_Sign_Extend_BYTE_to_WORD(TMP_DEST[511:384], SRC[255:192])
FI;
FOR j ← 0 TO KL-1
    i ← j * 16
    IF k1[j] OR *no writemask*
        THEN DEST[i+15:i] ← TEMP_DEST[i+15:i]
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+15:i] remains unchanged*
                ELSE *zeroing-masking*              ; zeroing-masking
                    DEST[i+15:i] ← 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0
```

VPMOVSXBD (EVEX encoded versions)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
Packed_Sign_Extend_BYTE_to_DWORD(TMP_DEST[127:0], SRC[31:0])
IF VL >= 256
    Packed_Sign_Extend_BYTE_to_DWORD(TMP_DEST[255:128], SRC[63:32])
FI;
IF VL >= 512
    Packed_Sign_Extend_BYTE_to_DWORD(TMP_DEST[383:256], SRC[95:64])
    Packed_Sign_Extend_BYTE_to_DWORD(TMP_DEST[511:384], SRC[127:96])
FI;
FOR j ← 0 TO KL-1
    i ← j * 32
    IF k1[j] OR *no writemask*
        THEN DEST[i+31:i] ← TEMP_DEST[i+31:i]
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE *zeroing-masking*              ; zeroing-masking
                    DEST[i+31:i] ← 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0
```

VPMOVSXBQ (EVEX encoded versions)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
Packed_Sign_Extend_BYTE_to_QWORD(TMP_DEST[127:0], SRC[15:0])
IF VL >= 256
    Packed_Sign_Extend_BYTE_to_QWORD(TMP_DEST[255:128], SRC[31:16])
FI;
IF VL >= 512
    Packed_Sign_Extend_BYTE_to_QWORD(TMP_DEST[383:256], SRC[47:32])
    Packed_Sign_Extend_BYTE_to_QWORD(TMP_DEST[511:384], SRC[63:48])
FI;
FOR j ← 0 TO KL-1
    i ← j * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+63:i] ← TEMP_DEST[i+63:i]
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE *zeroing-masking*              ; zeroing-masking
                    DEST[i+63:i] ← 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0
```

VPMOVSXWD (EVEX encoded versions)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
Packed_Sign_Extend_WORD_to_DWORD(TMP_DEST[127:0], SRC[63:0])
IF VL >= 256
    Packed_Sign_Extend_WORD_to_DWORD(TMP_DEST[255:128], SRC[127:64])
FI;
IF VL >= 512
    Packed_Sign_Extend_WORD_to_DWORD(TMP_DEST[383:256], SRC[191:128])
    Packed_Sign_Extend_WORD_to_DWORD(TMP_DEST[511:384], SRC[256:192])
FI;
FOR j ← 0 TO KL-1
    i ← j * 32
    IF k1[j] OR *no writemask*
        THEN DEST[i+31:i] ← TEMP_DEST[i+31:i]
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE *zeroing-masking*              ; zeroing-masking
                    DEST[i+31:i] ← 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0
```

VPMOVSXWQ (EVEX encoded versions)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
Packed_Sign_Extend_WORD_to_QWORD(TMP_DEST[127:0], SRC[31:0])
IF VL >= 256
    Packed_Sign_Extend_WORD_to_QWORD(TMP_DEST[255:128], SRC[63:32])
FI;
IF VL >= 512
    Packed_Sign_Extend_WORD_to_QWORD(TMP_DEST[383:256], SRC[95:64])
    Packed_Sign_Extend_WORD_to_QWORD(TMP_DEST[511:384], SRC[127:96])
FI;
FOR j ← 0 TO KL-1
    i ← j * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+63:i] ← TEMP_DEST[i+63:i]
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE *zeroing-masking*              ; zeroing-masking
                    DEST[i+63:i] ← 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0
```

VPMOVSXDQ (EVEX encoded versions)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
Packed_Sign_Extend_DWORD_to_QWORD(TEMP_DEST[127:0], SRC[63:0])
IF VL >= 256
    Packed_Sign_Extend_DWORD_to_QWORD(TEMP_DEST[255:128], SRC[127:64])
FI;
IF VL >= 512
    Packed_Sign_Extend_DWORD_to_QWORD(TEMP_DEST[383:256], SRC[191:128])
    Packed_Sign_Extend_DWORD_to_QWORD(TEMP_DEST[511:384], SRC[255:192])
FI;
FOR j ← 0 TO KL-1
    i ← j * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+63:i] ← TEMP_DEST[i+63:i]
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE *zeroing-masking*              ; zeroing-masking
                    DEST[i+63:i] ← 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0
```

VPMOVSXBW (VEX.256 encoded version)

```
Packed_Sign_Extend_BYTE_to_WORD(DEST[127:0], SRC[63:0])
Packed_Sign_Extend_BYTE_to_WORD(DEST[255:128], SRC[127:64])
DEST[MAXVL-1:256] ← 0
```

VPMOVSXBD (VEX.256 encoded version)
Packed_Sign_Extend_BYTE_to_DWORD(DEST[127:0], SRC[31:0])
Packed_Sign_Extend_BYTE_to_DWORD(DEST[255:128], SRC[63:32])
DEST[MAXVL-1:256] ← 0

VPMOVSXBQ (VEX.256 encoded version)
Packed_Sign_Extend_BYTE_to_QWORD(DEST[127:0], SRC[15:0])
Packed_Sign_Extend_BYTE_to_QWORD(DEST[255:128], SRC[31:16])
DEST[MAXVL-1:256] ← 0

VPMOVSXWD (VEX.256 encoded version)
Packed_Sign_Extend_WORD_to_DWORD(DEST[127:0], SRC[63:0])
Packed_Sign_Extend_WORD_to_DWORD(DEST[255:128], SRC[127:64])
DEST[MAXVL-1:256] ← 0

VPMOVSXWQ (VEX.256 encoded version)
Packed_Sign_Extend_WORD_to_QWORD(DEST[127:0], SRC[31:0])
Packed_Sign_Extend_WORD_to_QWORD(DEST[255:128], SRC[63:32])
DEST[MAXVL-1:256] ← 0

VPMOVSXDQ (VEX.256 encoded version)
Packed_Sign_Extend_DWORD_to_QWORD(DEST[127:0], SRC[63:0])
Packed_Sign_Extend_DWORD_to_QWORD(DEST[255:128], SRC[127:64])
DEST[MAXVL-1:256] ← 0

VPMOVSXBW (VEX.128 encoded version)
Packed_Sign_Extend_BYTE_to_WORDDEST[127:0], SRC[127:0]()
DEST[MAXVL-1:128] ←0

VPMOVSXBD (VEX.128 encoded version)
Packed_Sign_Extend_BYTE_to_DWORD(DEST[127:0], SRC[127:0])
DEST[MAXVL-1:128] ←0

VPMOVSXBQ (VEX.128 encoded version)
Packed_Sign_Extend_BYTE_to_QWORD(DEST[127:0], SRC[127:0])
DEST[MAXVL-1:128] ←0

VPMOVSXWD (VEX.128 encoded version)
Packed_Sign_Extend_WORD_to_DWORD(DEST[127:0], SRC[127:0])
DEST[MAXVL-1:128] ←0

VPMOVSXWQ (VEX.128 encoded version)
Packed_Sign_Extend_WORD_to_QWORD(DEST[127:0], SRC[127:0])
DEST[MAXVL-1:128] ←0

VPMOVSXDQ (VEX.128 encoded version)
Packed_Sign_Extend_DWORD_to_QWORD(DEST[127:0], SRC[127:0])
DEST[MAXVL-1:128] ←0

PMOVSXBW
Packed_Sign_Extend_BYTE_to_WORD(DEST[127:0], SRC[127:0])
DEST[MAXVL-1:128] (Unmodified)

PMOVSXBD
Packed_Sign_Extend_BYTE_to_DWORD(DEST[127:0], SRC[127:0])
DEST[MAXVL-1:128] (Unmodified)

PMOVSXBQ
Packed_Sign_Extend_BYTE_to_QWORD(DEST[127:0], SRC[127:0])
DEST[MAXVL-1:128] (Unmodified)

PMOVSXWD
Packed_Sign_Extend_WORD_to_DWORD(DEST[127:0], SRC[127:0])
DEST[MAXVL-1:128] (Unmodified)

PMOVSXWQ
Packed_Sign_Extend_WORD_to_QWORD(DEST[127:0], SRC[127:0])
DEST[MAXVL-1:128] (Unmodified)

PMOVSXDQ
Packed_Sign_Extend_DWORD_to_QWORD(DEST[127:0], SRC[127:0])
DEST[MAXVL-1:128] (Unmodified)

Intel C/C++ Compiler Intrinsic Equivalent

VPMOVSXBW __m512i _mm512_cvtepi8_epi16(__m512i a);
VPMOVSXBW __m512i _mm512_mask_cvtepi8_epi16(__m512i a, __mmask32 k, __m512i b);
VPMOVSXBW __m512i _mm512_maskz_cvtepi8_epi16(__mmask32 k, __m512i b);
VPMOVSXBD __m512i _mm512_cvtepi8_epi32(__m512i a);
VPMOVSXBD __m512i _mm512_mask_cvtepi8_epi32(__m512i a, __mmask16 k, __m512i b);
VPMOVSXBD __m512i _mm512_maskz_cvtepi8_epi32(__mmask16 k, __m512i b);
VPMOVSXBQ __m512i _mm512_cvtepi8_epi64(__m512i a);
VPMOVSXBQ __m512i _mm512_mask_cvtepi8_epi64(__m512i a, __mmask8 k, __m512i b);
VPMOVSXBQ __m512i _mm512_maskz_cvtepi8_epi64(__mmask8 k, __m512i a);
VPMOVSXDQ __m512i _mm512_cvtepi32_epi64(__m512i a);
VPMOVSXDQ __m512i _mm512_mask_cvtepi32_epi64(__m512i a, __mmask8 k, __m512i b);
VPMOVSXDQ __m512i _mm512_maskz_cvtepi32_epi64(__mmask8 k, __m512i a);
VPMOVSXWD __m512i _mm512_cvtepi16_epi32(__m512i a);
VPMOVSXWD __m512i _mm512_mask_cvtepi16_epi32(__m512i a, __mmask16 k, __m512i b);
VPMOVSXWD __m512i _mm512_maskz_cvtepi16_epi32(__mmask16 k, __m512i a);
VPMOVSXWQ __m512i _mm512_cvtepi16_epi64(__m512i a);
VPMOVSXWQ __m512i _mm512_mask_cvtepi16_epi64(__m512i a, __mmask8 k, __m512i b);
VPMOVSXWQ __m512i _mm512_maskz_cvtepi16_epi64(__mmask8 k, __m512i a);
VPMOVSXBW __m256i _mm256_cvtepi8_epi16(__m256i a);
VPMOVSXBW __m256i _mm256_mask_cvtepi8_epi16(__m256i a, __mmask16 k, __m256i b);
VPMOVSXBW __m256i _mm256_maskz_cvtepi8_epi16(__mmask16 k, __m256i b);
VPMOVSXBD __m256i _mm256_cvtepi8_epi32(__m256i a);
VPMOVSXBD __m256i _mm256_mask_cvtepi8_epi32(__m256i a, __mmask8 k, __m256i b);
VPMOVSXBD __m256i _mm256_maskz_cvtepi8_epi32(__mmask8 k, __m256i b);
VPMOVSXBQ __m256i _mm256_cvtepi8_epi64(__m256i a);
VPMOVSXBQ __m256i _mm256_mask_cvtepi8_epi64(__m256i a, __mmask8 k, __m256i b);
VPMOVSXBQ __m256i _mm256_maskz_cvtepi8_epi64(__mmask8 k, __m256i a);
VPMOVSXDQ __m256i _mm256_cvtepi32_epi64(__m256i a);
VPMOVSXDQ __m256i _mm256_mask_cvtepi32_epi64(__m256i a, __mmask8 k, __m256i b);
VPMOVSXDQ __m256i _mm256_maskz_cvtepi32_epi64(__mmask8 k, __m256i a);
VPMOVSXWD __m256i _mm256_cvtepi16_epi32(__m256i a);
VPMOVSXWD __m256i _mm256_mask_cvtepi16_epi32(__m256i a, __mmask16 k, __m256i b);
VPMOVSXWD __m256i _mm256_maskz_cvtepi16_epi32(__mmask16 k, __m256i a);

VPMOVSXWQ __m256i _mm256_cvtepi16_epi64(__m256i a);
VPMOVSXWQ __m256i _mm256_mask_cvtepi16_epi64(__m256i a, __mmask8 k, __m256i b);
VPMOVSXWQ __m256i _mm256_maskz_cvtepi16_epi64(__mmask8 k, __m256i a);
VPMOVSXBW __m128i _mm_mask_cvtepi8_epi16(__m128i a, __mmask8 k, __m128i b);
VPMOVSXBW __m128i _mm_maskz_cvtepi8_epi16(__mmask8 k, __m128i b);
VPMOVSXBD __m128i _mm_mask_cvtepi8_epi32(__m128i a, __mmask8 k, __m128i b);
VPMOVSXBD __m128i _mm_maskz_cvtepi8_epi32(__mmask8 k, __m128i b);
VPMOVSXBQ __m128i _mm_mask_cvtepi8_epi64(__m128i a, __mmask8 k, __m128i b);
VPMOVSXBQ __m128i _mm_maskz_cvtepi8_epi64(__mmask8 k, __m128i a);
VPMOVSXDQ __m128i _mm_mask_cvtepi32_epi64(__m128i a, __mmask8 k, __m128i b);
VPMOVSXDQ __m128i _mm_maskz_cvtepi32_epi64(__mmask8 k, __m128i a);
VPMOVSXWD __m128i _mm_mask_cvtepi16_epi32(__m128i a, __mmask16 k, __m128i b);
VPMOVSXWD __m128i _mm_maskz_cvtepi16_epi32(__mmask16 k, __m128i a);
VPMOVSXWQ __m128i _mm_mask_cvtepi16_epi64(__m128i a, __mmask8 k, __m128i b);
VPMOVSXWQ __m128i _mm_maskz_cvtepi16_epi64(__mmask8 k, __m128i a);
PMOVSXBW __m128i _mm_ cvtepi8_epi16 (__m128i a);
PMOVSXBD __m128i _mm_ cvtepi8_epi32 (__m128i a);
PMOVSXBQ __m128i _mm_ cvtepi8_epi64 (__m128i a);
PMOVSXWD __m128i _mm_ cvtepi16_epi32 (__m128i a);
PMOVSXWQ __m128i _mm_ cvtepi16_epi64 (__m128i a);
PMOVSXDQ __m128i _mm_ cvtepi32_epi64 (__m128i a);

SIMD Floating-Point Exceptions

None

Other Exceptions

Non-EVEX-encoded instruction, see Exceptions Type 5.

EVEX-encoded instruction, see Exceptions Type E5.

#UD	If VEX.vvvv != 1111B, or EVEX.vvvv != 1111B.

PMOVZX—Packed Move with Zero Extend

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0f 38 30 /r PMOVZXBW xmm1, xmm2/m64	A	V/V	SSE4_1	Zero extend 8 packed 8-bit integers in the low 8 bytes of xmm2/m64 to 8 packed 16-bit integers in xmm1.
66 0f 38 31 /r PMOVZXBD xmm1, xmm2/m32	A	V/V	SSE4_1	Zero extend 4 packed 8-bit integers in the low 4 bytes of xmm2/m32 to 4 packed 32-bit integers in xmm1.
66 0f 38 32 /r PMOVZXBQ xmm1, xmm2/m16	A	V/V	SSE4_1	Zero extend 2 packed 8-bit integers in the low 2 bytes of xmm2/m16 to 2 packed 64-bit integers in xmm1.
66 0f 38 33 /r PMOVZXWD xmm1, xmm2/m64	A	V/V	SSE4_1	Zero extend 4 packed 16-bit integers in the low 8 bytes of xmm2/m64 to 4 packed 32-bit integers in xmm1.
66 0f 38 34 /r PMOVZXWQ xmm1, xmm2/m32	A	V/V	SSE4_1	Zero extend 2 packed 16-bit integers in the low 4 bytes of xmm2/m32 to 2 packed 64-bit integers in xmm1.
66 0f 38 35 /r PMOVZXDQ xmm1, xmm2/m64	A	V/V	SSE4_1	Zero extend 2 packed 32-bit integers in the low 8 bytes of xmm2/m64 to 2 packed 64-bit integers in xmm1.
VEX.128.66.0F38.WIG 30 /r VPMOVZXBW xmm1, xmm2/m64	A	V/V	AVX	Zero extend 8 packed 8-bit integers in the low 8 bytes of xmm2/m64 to 8 packed 16-bit integers in xmm1.
VEX.128.66.0F38.WIG 31 /r VPMOVZXBD xmm1, xmm2/m32	A	V/V	AVX	Zero extend 4 packed 8-bit integers in the low 4 bytes of xmm2/m32 to 4 packed 32-bit integers in xmm1.
VEX.128.66.0F38.WIG 32 /r VPMOVZXBQ xmm1, xmm2/m16	A	V/V	AVX	Zero extend 2 packed 8-bit integers in the low 2 bytes of xmm2/m16 to 2 packed 64-bit integers in xmm1.
VEX.128.66.0F38.WIG 33 /r VPMOVZXWD xmm1, xmm2/m64	A	V/V	AVX	Zero extend 4 packed 16-bit integers in the low 8 bytes of xmm2/m64 to 4 packed 32-bit integers in xmm1.
VEX.128.66.0F38.WIG 34 /r VPMOVZXWQ xmm1, xmm2/m32	A	V/V	AVX	Zero extend 2 packed 16-bit integers in the low 4 bytes of xmm2/m32 to 2 packed 64-bit integers in xmm1.
VEX.128.66.0F 38.WIG 35 /r VPMOVZXDQ xmm1, xmm2/m64	A	V/V	AVX	Zero extend 2 packed 32-bit integers in the low 8 bytes of xmm2/m64 to 2 packed 64-bit integers in xmm1.
VEX.256.66.0F38.WIG 30 /r VPMOVZXBW ymm1, xmm2/m128	A	V/V	AVX2	Zero extend 16 packed 8-bit integers in xmm2/m128 to 16 packed 16-bit integers in ymm1.
VEX.256.66.0F38.WIG 31 /r VPMOVZXBD ymm1, xmm2/m64	A	V/V	AVX2	Zero extend 8 packed 8-bit integers in the low 8 bytes of xmm2/m64 to 8 packed 32-bit integers in ymm1.
VEX.256.66.0F38.WIG 32 /r VPMOVZXBQ ymm1, xmm2/m32	A	V/V	AVX2	Zero extend 4 packed 8-bit integers in the low 4 bytes of xmm2/m32 to 4 packed 64-bit integers in ymm1.
VEX.256.66.0F38.WIG 33 /r VPMOVZXWD ymm1, xmm2/m128	A	V/V	AVX2	Zero extend 8 packed 16-bit integers xmm2/m128 to 8 packed 32-bit integers in ymm1.
VEX.256.66.0F38.WIG 34 /r VPMOVZXWQ ymm1, xmm2/m64	A	V/V	AVX2	Zero extend 4 packed 16-bit integers in the low 8 bytes of xmm2/m64 to 4 packed 64-bit integers in xmm1.
VEX.256.66.0F38.WIG 35 /r VPMOVZXDQ ymm1, xmm2/m128	A	V/V	AVX2	Zero extend 4 packed 32-bit integers in xmm2/m128 to 4 packed 64-bit integers in ymm1.

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.128.66.0F38 30.WIG /r VPMOVZXBW xmm1 {k1}{z}, xmm2/m64	B	V/V	AVX512VL AVX512BW	Zero extend 8 packed 8-bit integers in the low 8 bytes of xmm2/m64 to 8 packed 16-bit integers in xmm1.
EVEX.256.66.0F38.WIG 30 /r VPMOVZXBW ymm1 {k1}{z}, xmm2/m128	B	V/V	AVX512VL AVX512BW	Zero extend 16 packed 8-bit integers in xmm2/m128 to 16 packed 16-bit integers in ymm1.
EVEX.512.66.0F38.WIG 30 /r VPMOVZXBW zmm1 {k1}{z}, ymm2/m256	B	V/V	AVX512BW	Zero extend 32 packed 8-bit integers in ymm2/m256 to 32 packed 16-bit integers in zmm1.
EVEX.128.66.0F38.WIG 31 /r VPMOVZXBD xmm1 {k1}{z}, xmm2/m32	C	V/V	AVX512VL AVX512F	Zero extend 4 packed 8-bit integers in the low 4 bytes of xmm2/m32 to 4 packed 32-bit integers in xmm1 subject to writemask k1.
EVEX.256.66.0F38.WIG 31 /r VPMOVZXBD ymm1 {k1}{z}, xmm2/m64	C	V/V	AVX512VL AVX512F	Zero extend 8 packed 8-bit integers in the low 8 bytes of xmm2/m64 to 8 packed 32-bit integers in ymm1 subject to writemask k1.
EVEX.512.66.0F38.WIG 31 /r VPMOVZXBD zmm1 {k1}{z}, xmm2/m128	C	V/V	AVX512F	Zero extend 16 packed 8-bit integers in xmm2/m128 to 16 packed 32-bit integers in zmm1 subject to writemask k1.
EVEX.128.66.0F38.WIG 32 /r VPMOVZXBQ xmm1 {k1}{z}, xmm2/m16	D	V/V	AVX512VL AVX512F	Zero extend 2 packed 8-bit integers in the low 2 bytes of xmm2/m16 to 2 packed 64-bit integers in xmm1 subject to writemask k1.
EVEX.256.66.0F38.WIG 32 /r VPMOVZXBQ ymm1 {k1}{z}, xmm2/m32	D	V/V	AVX512VL AVX512F	Zero extend 4 packed 8-bit integers in the low 4 bytes of xmm2/m32 to 4 packed 64-bit integers in ymm1 subject to writemask k1.
EVEX.512.66.0F38.WIG 32 /r VPMOVZXBQ zmm1 {k1}{z}, xmm2/m64	D	V/V	AVX512F	Zero extend 8 packed 8-bit integers in the low 8 bytes of xmm2/m64 to 8 packed 64-bit integers in zmm1 subject to writemask k1.
EVEX.128.66.0F38.WIG 33 /r VPMOVZXWD xmm1 {k1}{z}, xmm2/m64	B	V/V	AVX512VL AVX512F	Zero extend 4 packed 16-bit integers in the low 8 bytes of xmm2/m64 to 4 packed 32-bit integers in xmm1 subject to writemask k1.
EVEX.256.66.0F38.WIG 33 /r VPMOVZXWD ymm1 {k1}{z}, xmm2/m128	B	V/V	AVX512VL AVX512F	Zero extend 8 packed 16-bit integers in xmm2/m128 to 8 packed 32-bit integers in zmm1 subject to writemask k1.
EVEX.512.66.0F38.WIG 33 /r VPMOVZXWD zmm1 {k1}{z}, ymm2/m256	B	V/V	AVX512F	Zero extend 16 packed 16-bit integers in ymm2/m256 to 16 packed 32-bit integers in zmm1 subject to writemask k1.
EVEX.128.66.0F38.WIG 34 /r VPMOVZXWQ xmm1 {k1}{z}, xmm2/m32	C	V/V	AVX512VL AVX512F	Zero extend 2 packed 16-bit integers in the low 4 bytes of xmm2/m32 to 2 packed 64-bit integers in xmm1 subject to writemask k1.
EVEX.256.66.0F38.WIG 34 /r VPMOVZXWQ ymm1 {k1}{z}, xmm2/m64	C	V/V	AVX512VL AVX512F	Zero extend 4 packed 16-bit integers in the low 8 bytes of xmm2/m64 to 4 packed 64-bit integers in ymm1 subject to writemask k1.
EVEX.512.66.0F38.WIG 34 /r VPMOVZXWQ zmm1 {k1}{z}, xmm2/m128	C	V/V	AVX512F	Zero extend 8 packed 16-bit integers in xmm2/m128 to 8 packed 64-bit integers in zmm1 subject to writemask k1.

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.128.66.0F38.W0 35 /r VPMOVZXDQ xmm1 {k1}{z}, xmm2/m64	B	V/V	AVX512VL AVX512F	Zero extend 2 packed 32-bit integers in the low 8 bytes of xmm2/m64 to 2 packed 64-bit integers in zmm1 using writemask k1.
EVEX.256.66.0F38.W0 35 /r VPMOVZXDQ ymm1 {k1}{z}, xmm2/m128	B	V/V	AVX512VL AVX512F	Zero extend 4 packed 32-bit integers in xmm2/m128 to 4 packed 64-bit integers in zmm1 using writemask k1.
EVEX.512.66.0F38.W0 35 /r VPMOVZXDQ zmm1 {k1}{z}, ymm2/m256	B	V/V	AVX512F	Zero extend 8 packed 32-bit integers in ymm2/m256 to 8 packed 64-bit integers in zmm1 using writemask k1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	NA	ModRM:reg (w)	ModRM:r/m (r)	NA	NA
B	Half Vector Mem	ModRM:reg (w)	ModRM:r/m (r)	NA	NA
C	Quarter Vector Mem	ModRM:reg (w)	ModRM:r/m (r)	NA	NA
D	Oct Vector Mem	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

Description

Legacy, VEX and EVEX encoded versions: Packed byte, word, or dword integers starting from the low bytes of the source operand (second operand) are zero extended to word, dword, or quadword integers and stored in packed signed bytes the destination operand.

128-bit Legacy SSE version: Bits (MAXVL-1:128) of the corresponding destination register remain unchanged.

VEX.128 encoded version: Bits (MAXVL-1:128) of the corresponding destination register are zeroed.

VEX.256 encoded version: Bits (MAXVL-1:256) of the corresponding destination register are zeroed.

EVEX encoded versions: Packed dword integers starting from the low bytes of the source operand (second operand) are zero extended to quadword integers and stored to the destination operand under the writemask.The destination register is XMM, YMM or ZMM Register.

Note: VEX.vvvv and EVEX.vvvv are reserved and must be 1111b otherwise instructions will #UD.

Operation

Packed_Zero_Extend_BYTE_to_WORD(DEST, SRC)

```
DEST[15:0] ←ZeroExtend(SRC[7:0]);
DEST[31:16] ←ZeroExtend(SRC[15:8]);
DEST[47:32] ←ZeroExtend(SRC[23:16]);
DEST[63:48] ←ZeroExtend(SRC[31:24]);
DEST[79:64] ←ZeroExtend(SRC[39:32]);
DEST[95:80] ←ZeroExtend(SRC[47:40]);
DEST[111:96] ←ZeroExtend(SRC[55:48]);
DEST[127:112] ←ZeroExtend(SRC[63:56]);
```

Packed_Zero_Extend_BYTE_to_DWORD(DEST, SRC)

```
DEST[31:0] ←ZeroExtend(SRC[7:0]);
DEST[63:32] ←ZeroExtend(SRC[15:8]);
DEST[95:64] ←ZeroExtend(SRC[23:16]);
DEST[127:96] ←ZeroExtend(SRC[31:24]);
```

Packed_Zero_Extend_BYTE_to_QWORD(DEST, SRC)
DEST[63:0] ←ZeroExtend(SRC[7:0]);
DEST[127:64] ←ZeroExtend(SRC[15:8]);

Packed_Zero_Extend_WORD_to_DWORD(DEST, SRC)
DEST[31:0] ←ZeroExtend(SRC[15:0]);
DEST[63:32] ←ZeroExtend(SRC[31:16]);
DEST[95:64] ←ZeroExtend(SRC[47:32]);
DEST[127:96] ←ZeroExtend(SRC[63:48]);

Packed_Zero_Extend_WORD_to_QWORD(DEST, SRC)
DEST[63:0] ←ZeroExtend(SRC[15:0]);
DEST[127:64] ←ZeroExtend(SRC[31:16]);

Packed_Zero_Extend_DWORD_to_QWORD(DEST, SRC)
DEST[63:0] ←ZeroExtend(SRC[31:0]);
DEST[127:64] ←ZeroExtend(SRC[63:32]);

VPMOVZXBW (EVEX encoded versions)
(KL, VL) = (8, 128), (16, 256), (32, 512)
Packed_Zero_Extend_BYTE_to_WORD(TMP_DEST[127:0], SRC[63:0])
IF VL >= 256
    Packed_Zero_Extend_BYTE_to_WORD(TMP_DEST[255:128], SRC[127:64])
FI;
IF VL >= 512
    Packed_Zero_Extend_BYTE_to_WORD(TMP_DEST[383:256], SRC[191:128])
    Packed_Zero_Extend_BYTE_to_WORD(TMP_DEST[511:384], SRC[255:192])
FI;
FOR j ← 0 TO KL-1
    i ← j * 16
    IF k1[j] OR *no writemask*
        THEN DEST[i+15:i] ← TEMP_DEST[i+15:i]
        ELSE
            IF *merging-masking* ; merging-masking
                THEN *DEST[i+15:i] remains unchanged*
                ELSE *zeroing-masking* ; zeroing-masking
                    DEST[i+15:i] ← 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0

VPMOVZXBD (EVEX encoded versions)
(KL, VL) = (4, 128), (8, 256), (16, 512)
Packed_Zero_Extend_BYTE_to_DWORD(TMP_DEST[127:0], SRC[31:0])
IF VL >= 256
    Packed_Zero_Extend_BYTE_to_DWORD(TMP_DEST[255:128], SRC[63:32])
FI;
IF VL >= 512
    Packed_Zero_Extend_BYTE_to_DWORD(TMP_DEST[383:256], SRC[95:64])
    Packed_Zero_Extend_BYTE_to_DWORD(TMP_DEST[511:384], SRC[127:96])
FI;
FOR j ← 0 TO KL-1
    i ← j * 32

```
    IF k1[j] OR *no writemask*
        THEN DEST[i+31:i] ← TEMP_DEST[i+31:i]
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE *zeroing-masking*              ; zeroing-masking
                    DEST[i+31:i] ← 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0
```

VPMOVZXBQ (EVEX encoded versions)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
Packed_Zero_Extend_BYTE_to_QWORD(TMP_DEST[127:0], SRC[15:0])
IF VL >= 256
    Packed_Zero_Extend_BYTE_to_QWORD(TMP_DEST[255:128], SRC[31:16])
FI;
IF VL >= 512
    Packed_Zero_Extend_BYTE_to_QWORD(TMP_DEST[383:256], SRC[47:32])
    Packed_Zero_Extend_BYTE_to_QWORD(TMP_DEST[511:384], SRC[63:48])
FI;
FOR j ← 0 TO KL-1
    i ← j * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+63:i] ← TEMP_DEST[i+63:i]
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE *zeroing-masking*              ; zeroing-masking
                    DEST[i+63:i] ← 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0
```

VPMOVZXWD (EVEX encoded versions)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
Packed_Zero_Extend_WORD_to_DWORD(TMP_DEST[127:0], SRC[63:0])
IF VL >= 256
    Packed_Zero_Extend_WORD_to_DWORD(TMP_DEST[255:128], SRC[127:64])
FI;
IF VL >= 512
    Packed_Zero_Extend_WORD_to_DWORD(TMP_DEST[383:256], SRC[191:128])
    Packed_Zero_Extend_WORD_to_DWORD(TMP_DEST[511:384], SRC[256:192])
FI;
FOR j ← 0 TO KL-1
    i ← j * 32
    IF k1[j] OR *no writemask*
        THEN DEST[i+31:i] ← TEMP_DEST[i+31:i]
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE *zeroing-masking*              ; zeroing-masking
```

```
                    DEST[i+31:i] ← 0
                FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0
```

VPMOVZXWQ (EVEX encoded versions)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
Packed_Zero_Extend_WORD_to_QWORD(TMP_DEST[127:0], SRC[31:0])
IF VL >= 256
    Packed_Zero_Extend_WORD_to_QWORD(TMP_DEST[255:128], SRC[63:32])
FI;
IF VL >= 512
    Packed_Zero_Extend_WORD_to_QWORD(TMP_DEST[383:256], SRC[95:64])
    Packed_Zero_Extend_WORD_to_QWORD(TMP_DEST[511:384], SRC[127:96])
FI;
FOR j ← 0 TO KL-1
    i ← j * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+63:i] ← TEMP_DEST[i+63:i]
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE *zeroing-masking*              ; zeroing-masking
                    DEST[i+63:i] ← 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0
```

VPMOVZXDQ (EVEX encoded versions)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
Packed_Zero_Extend_DWORD_to_QWORD(TEMP_DEST[127:0], SRC[63:0])
IF VL >= 256
    Packed_Zero_Extend_DWORD_to_QWORD(TEMP_DEST[255:128], SRC[127:64])
FI;
IF VL >= 512
    Packed_Zero_Extend_DWORD_to_QWORD(TEMP_DEST[383:256], SRC[191:128])
    Packed_Zero_Extend_DWORD_to_QWORD(TEMP_DEST[511:384], SRC[255:192])
FI;
FOR j ← 0 TO KL-1
    i ← j * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+63:i] ← TEMP_DEST[i+63:i]
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE *zeroing-masking*              ; zeroing-masking
                    DEST[i+63:i] ← 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0
```

VPMOVZXBW (VEX.256 encoded version)
Packed_Zero_Extend_BYTE_to_WORD(DEST[127:0], SRC[63:0])
Packed_Zero_Extend_BYTE_to_WORD(DEST[255:128], SRC[127:64])
DEST[MAXVL-1:256] ← 0

VPMOVZXBD (VEX.256 encoded version)
Packed_Zero_Extend_BYTE_to_DWORD(DEST[127:0], SRC[31:0])
Packed_Zero_Extend_BYTE_to_DWORD(DEST[255:128], SRC[63:32])
DEST[MAXVL-1:256] ← 0

VPMOVZXBQ (VEX.256 encoded version)
Packed_Zero_Extend_BYTE_to_QWORD(DEST[127:0], SRC[15:0])
Packed_Zero_Extend_BYTE_to_QWORD(DEST[255:128], SRC[31:16])
DEST[MAXVL-1:256] ← 0

VPMOVZXWD (VEX.256 encoded version)
Packed_Zero_Extend_WORD_to_DWORD(DEST[127:0], SRC[63:0])
Packed_Zero_Extend_WORD_to_DWORD(DEST[255:128], SRC[127:64])
DEST[MAXVL-1:256] ← 0

VPMOVZXWQ (VEX.256 encoded version)
Packed_Zero_Extend_WORD_to_QWORD(DEST[127:0], SRC[31:0])
Packed_Zero_Extend_WORD_to_QWORD(DEST[255:128], SRC[63:32])
DEST[MAXVL-1:256] ← 0

VPMOVZXDQ (VEX.256 encoded version)
Packed_Zero_Extend_DWORD_to_QWORD(DEST[127:0], SRC[63:0])
Packed_Zero_Extend_DWORD_to_QWORD(DEST[255:128], SRC[127:64])
DEST[MAXVL-1:256] ← 0

VPMOVZXBW (VEX.128 encoded version)
Packed_Zero_Extend_BYTE_to_WORD()
DEST[MAXVL-1:128] ←0

VPMOVZXBD (VEX.128 encoded version)
Packed_Zero_Extend_BYTE_to_DWORD()
DEST[MAXVL-1:128] ←0

VPMOVZXBQ (VEX.128 encoded version)
Packed_Zero_Extend_BYTE_to_QWORD()
DEST[MAXVL-1:128] ←0

VPMOVZXWD (VEX.128 encoded version)
Packed_Zero_Extend_WORD_to_DWORD()
DEST[MAXVL-1:128] ←0

VPMOVZXWQ (VEX.128 encoded version)
Packed_Zero_Extend_WORD_to_QWORD()
DEST[MAXVL-1:128] ←0

VPMOVZXDQ (VEX.128 encoded version)
Packed_Zero_Extend_DWORD_to_QWORD()
DEST[MAXVL-1:128] ←0

PMOVZXBW
Packed_Zero_Extend_BYTE_to_WORD()
DEST[MAXVL-1:128] (Unmodified)

PMOVZXBD
Packed_Zero_Extend_BYTE_to_DWORD()
DEST[MAXVL-1:128] (Unmodified)

PMOVZXBQ
Packed_Zero_Extend_BYTE_to_QWORD()
DEST[MAXVL-1:128] (Unmodified)

PMOVZXWD
Packed_Zero_Extend_WORD_to_DWORD()
DEST[MAXVL-1:128] (Unmodified)

PMOVZXWQ
Packed_Zero_Extend_WORD_to_QWORD()
DEST[MAXVL-1:128] (Unmodified)

PMOVZXDQ
Packed_Zero_Extend_DWORD_to_QWORD()
DEST[MAXVL-1:128] (Unmodified)

Intel C/C++ Compiler Intrinsic Equivalent

VPMOVZXBW __m512i _mm512_cvtepu8_epi16(__m256i a);
VPMOVZXBW __m512i _mm512_mask_cvtepu8_epi16(__m512i a, __mmask32 k, __m256i b);
VPMOVZXBW __m512i _mm512_maskz_cvtepu8_epi16(__mmask32 k, __m256i b);
VPMOVZXBD __m512i _mm512_cvtepu8_epi32(__m128i a);
VPMOVZXBD __m512i _mm512_mask_cvtepu8_epi32(__m512i a, __mmask16 k, __m128i b);
VPMOVZXBD __m512i _mm512_maskz_cvtepu8_epi32(__mmask16 k, __m128i b);
VPMOVZXBQ __m512i _mm512_cvtepu8_epi64(__m128i a);
VPMOVZXBQ __m512i _mm512_mask_cvtepu8_epi64(__m512i a, __mmask8 k, __m128i b);
VPMOVZXBQ __m512i _mm512_maskz_cvtepu8_epi64(__mmask8 k, __m128i a);
VPMOVZXDQ __m512i _mm512_cvtepu32_epi64(__m256i a);
VPMOVZXDQ __m512i _mm512_mask_cvtepu32_epi64(__m512i a, __mmask8 k, __m256i b);
VPMOVZXDQ __m512i _mm512_maskz_cvtepu32_epi64(__mmask8 k, __m256i a);
VPMOVZXWD __m512i _mm512_cvtepu16_epi32(__m128i a);
VPMOVZXWD __m512i _mm512_mask_cvtepu16_epi32(__m512i a, __mmask16 k, __m128i b);
VPMOVZXWD __m512i _mm512_maskz_cvtepu16_epi32(__mmask16 k, __m128i a);
VPMOVZXWQ __m512i _mm512_cvtepu16_epi64(__m256i a);
VPMOVZXWQ __m512i _mm512_mask_cvtepu16_epi64(__m512i a, __mmask8 k, __m256i b);
VPMOVZXWQ __m512i _mm512_maskz_cvtepu16_epi64(__mmask8 k, __m256i a);
VPMOVZXBW __m256i _mm256_cvtepu8_epi16(__m256i a);
VPMOVZXBW __m256i _mm256_mask_cvtepu8_epi16(__m256i a, __mmask16 k, __m128i b);
VPMOVZXBW __m256i _mm256_maskz_cvtepu8_epi16(__mmask16 k, __m128i b);
VPMOVZXBD __m256i _mm256_cvtepu8_epi32(__m128i a);
VPMOVZXBD __m256i _mm256_mask_cvtepu8_epi32(__m256i a, __mmask8 k, __m128i b);
VPMOVZXBD __m256i _mm256_maskz_cvtepu8_epi32(__mmask8 k, __m128i b);
VPMOVZXBQ __m256i _mm256_cvtepu8_epi64(__m128i a);
VPMOVZXBQ __m256i _mm256_mask_cvtepu8_epi64(__m256i a, __mmask8 k, __m128i b);
VPMOVZXBQ __m256i _mm256_maskz_cvtepu8_epi64(__mmask8 k, __m128i a);
VPMOVZXDQ __m256i _mm256_cvtepu32_epi64(__m128i a);
VPMOVZXDQ __m256i _mm256_mask_cvtepu32_epi64(__m256i a, __mmask8 k, __m128i b);

VPMOVZXDQ __m256i _mm256_maskz_cvtepu32_epi64(__mmask8 k, __m128i a);
VPMOVZXWD __m256i _mm256_cvtepu16_epi32(__m128i a);
VPMOVZXWD __m256i _mm256_mask_cvtepu16_epi32(__m256i a, __mmask16 k, __m128i b);
VPMOVZXWD __m256i _mm256_maskz_cvtepu16_epi32(__mmask16 k, __m128i a);
VPMOVZXWQ __m256i _mm256_cvtepu16_epi64(__m128i a);
VPMOVZXWQ __m256i _mm256_mask_cvtepu16_epi64(__m256i a, __mmask8 k, __m128i b);
VPMOVZXWQ __m256i _mm256_maskz_cvtepu16_epi64(__mmask8 k, __m128i a);
VPMOVZXBW __m128i _mm_mask_cvtepu8_epi16(__m128i a, __mmask8 k, __m128i b);
VPMOVZXBW __m128i _mm_maskz_cvtepu8_epi16(__mmask8 k, __m128i b);
VPMOVZXBD __m128i _mm_mask_cvtepu8_epi32(__m128i a, __mmask8 k, __m128i b);
VPMOVZXBD __m128i _mm_maskz_cvtepu8_epi32(__mmask8 k, __m128i b);
VPMOVZXBQ __m128i _mm_mask_cvtepu8_epi64(__m128i a, __mmask8 k, __m128i b);
VPMOVZXBQ __m128i _mm_maskz_cvtepu8_epi64(__mmask8 k, __m128i a);
VPMOVZXDQ __m128i _mm_mask_cvtepu32_epi64(__m128i a, __mmask8 k, __m128i b);
VPMOVZXDQ __m128i _mm_maskz_cvtepu32_epi64(__mmask8 k, __m128i a);
VPMOVZXWD __m128i _mm_mask_cvtepu16_epi32(__m128i a, __mmask16 k, __m128i b);
VPMOVZXWD __m128i _mm_maskz_cvtepu16_epi32(__mmask8 k, __m128i a);
VPMOVZXWQ __m128i _mm_mask_cvtepu16_epi64(__m128i a, __mmask8 k, __m128i b);
VPMOVZXWQ __m128i _mm_maskz_cvtepu16_epi64(__mmask8 k, __m128i a);
PMOVZXBW __m128i _mm_ cvtepu8_epi16 (__m128i a);
PMOVZXBD __m128i _mm_ cvtepu8_epi32 (__m128i a);
PMOVZXBQ __m128i _mm_ cvtepu8_epi64 (__m128i a);
PMOVZXWD __m128i _mm_ cvtepu16_epi32 (__m128i a);
PMOVZXWQ __m128i _mm_ cvtepu16_epi64 (__m128i a);
PMOVZXDQ __m128i _mm_ cvtepu32_epi64 (__m128i a);

SIMD Floating-Point Exceptions

None

Other Exceptions

Non-EVEX-encoded instruction, see Exceptions Type 5.

EVEX-encoded instruction, see Exceptions Type E5.

#UD	If VEX.vvvv != 1111B, or EVEX.vvvv != 1111B.

PMULDQ—Multiply Packed Doubleword Integers

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 38 28 /r PMULDQ xmm1, xmm2/m128	A	V/V	SSE4_1	Multiply packed signed doubleword integers in xmm1 by packed signed doubleword integers in xmm2/m128, and store the quadword results in xmm1.
VEX.NDS.128.66.0F38.WIG 28 /r VPMULDQ xmm1, xmm2, xmm3/m128	B	V/V	AVX	Multiply packed signed doubleword integers in xmm2 by packed signed doubleword integers in xmm3/m128, and store the quadword results in xmm1.
VEX.NDS.256.66.0F38.WIG 28 /r VPMULDQ ymm1, ymm2, ymm3/m256	B	V/V	AVX2	Multiply packed signed doubleword integers in ymm2 by packed signed doubleword integers in ymm3/m256, and store the quadword results in ymm1.
EVEX.NDS.128.66.0F38.W1 28 /r VPMULDQ xmm1 {k1}{z}, xmm2, xmm3/m128/m64bcst	C	V/V	AVX512VL AVX512F	Multiply packed signed doubleword integers in xmm2 by packed signed doubleword integers in xmm3/m128/m64bcst, and store the quadword results in xmm1 using writemask k1.
EVEX.NDS.256.66.0F38.W1 28 /r VPMULDQ ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst	C	V/V	AVX512VL AVX512F	Multiply packed signed doubleword integers in ymm2 by packed signed doubleword integers in ymm3/m256/m64bcst, and store the quadword results in ymm1 using writemask k1.
EVEX.NDS.512.66.0F38.W1 28 /r VPMULDQ zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst	C	V/V	AVX512F	Multiply packed signed doubleword integers in zmm2 by packed signed doubleword integers in zmm3/m512/m64bcst, and store the quadword results in zmm1 using writemask k1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	NA	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
B	NA	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA
C	Full Vector	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA

Description

Multiplies packed signed doubleword integers in the even-numbered (zero-based reference) elements of the first source operand with the packed signed doubleword integers in the corresponding elements of the second source operand and stores packed signed quadword results in the destination operand.

128-bit Legacy SSE version: The input signed doubleword integers are taken from the even-numbered elements of the source operands, i.e. the first (low) and third doubleword element. For 128-bit memory operands, 128 bits are fetched from memory, but only the first and third doublewords are used in the computation. The first source operand and the destination XMM operand is the same. The second source operand can be an XMM register or 128-bit memory location. Bits (MAXVL-1:128) of the corresponding destination register remain unchanged.

VEX.128 encoded version: The input signed doubleword integers are taken from the even-numbered elements of the source operands, i.e., the first (low) and third doubleword element. For 128-bit memory operands, 128 bits are fetched from memory, but only the first and third doublewords are used in the computation.The first source operand and the destination operand are XMM registers. The second source operand can be an XMM register or 128-bit memory location. Bits (MAXVL-1:128) of the corresponding destination register are zeroed.

VEX.256 encoded version: The input signed doubleword integers are taken from the even-numbered elements of the source operands, i.e. the first, 3rd, 5th, 7th doubleword element. For 256-bit memory operands, 256 bits are fetched from memory, but only the four even-numbered doublewords are used in the computation. The first source operand and the destination operand are YMM registers. The second source operand can be a YMM register or 256-bit memory location. Bits (MAXVL-1:256) of the corresponding destination ZMM register are zeroed.

EVEX encoded version: The input signed doubleword integers are taken from the even-numbered elements of the source operands. The first source operand is a ZMM/YMM/XMM registers. The second source operand can be an ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 64-bit memory location. The destination is a ZMM/YMM/XMM register, and updated according to the writemask at 64-bit granularity.

Operation

VPMULDQ (EVEX encoded versions)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j ← 0 TO KL-1
    i ← j * 64
    IF k1[j] OR *no writemask*
        THEN
            IF (EVEX.b = 1) AND (SRC2 *is memory*)
                THEN DEST[i+63:i] ← SignExtend64( SRC1[i+31:i]) * SignExtend64( SRC2[31:0])
                ELSE DEST[i+63:i] ← SignExtend64( SRC1[i+31:i]) * SignExtend64( SRC2[i+31:i])
            FI;
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE *zeroing-masking*                  ; zeroing-masking
                    DEST[i+63:i] ← 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0
```

VPMULDQ (VEX.256 encoded version)

```
DEST[63:0] ←SignExtend64( SRC1[31:0]) * SignExtend64( SRC2[31:0])
DEST[127:64] ←SignExtend64( SRC1[95:64]) * SignExtend64( SRC2[95:64])
DEST[191:128] ←SignExtend64( SRC1[159:128]) * SignExtend64( SRC2[159:128])
DEST[255:192] ←SignExtend64( SRC1[223:192]) * SignExtend64( SRC2[223:192])
DEST[MAXVL-1:256] ←0
```

VPMULDQ (VEX.128 encoded version)

```
DEST[63:0] ←SignExtend64( SRC1[31:0]) * SignExtend64( SRC2[31:0])
DEST[127:64] ←SignExtend64( SRC1[95:64]) * SignExtend64( SRC2[95:64])
DEST[MAXVL-1:128] ←0
```

PMULDQ (128-bit Legacy SSE version)

```
DEST[63:0] ←SignExtend64( DEST[31:0]) * SignExtend64( SRC[31:0])
DEST[127:64] ←SignExtend64( DEST[95:64]) * SignExtend64( SRC[95:64])
DEST[MAXVL-1:128] (Unmodified)
```

Intel C/C++ Compiler Intrinsic Equivalent

VPMULDQ __m512i _mm512_mul_epi32(__m512i a, __m512i b);
VPMULDQ __m512i _mm512_mask_mul_epi32(__m512i s, __mmask8 k, __m512i a, __m512i b);
VPMULDQ __m512i _mm512_maskz_mul_epi32(__mmask8 k, __m512i a, __m512i b);
VPMULDQ __m256i _mm256_mask_mul_epi32(__m256i s, __mmask8 k, __m256i a, __m256i b);
VPMULDQ __m256i _mm256_mask_mul_epi32(__mmask8 k, __m256i a, __m256i b);
VPMULDQ __m128i _mm_mask_mul_epi32(__m128i s, __mmask8 k, __m128i a, __m128i b);
VPMULDQ __m128i _mm_mask_mul_epi32(__mmask8 k, __m128i a, __m128i b);
(V)PMULDQ __m128i _mm_mul_epi32(__m128i a, __m128i b);
VPMULDQ __m256i _mm256_mul_epi32(__m256i a, __m256i b);

SIMD Floating-Point Exceptions

None

Other Exceptions

Non-EVEX-encoded instruction, see Exceptions Type 4.
EVEX-encoded instruction, see Exceptions Type E4.

PMULHRSW — Packed Multiply High with Round and Scale

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F 38 0B /r[1] PMULHRSW *mm1, mm2/m64*	A	V/V	SSSE3	Multiply 16-bit signed words, scale and round signed doublewords, pack high 16 bits to *mm1*.
66 0F 38 0B /r PMULHRSW *xmm1, xmm2/m128*	A	V/V	SSSE3	Multiply 16-bit signed words, scale and round signed doublewords, pack high 16 bits to *xmm1*.
VEX.NDS.128.66.0F38.WIG 0B /r VPMULHRSW *xmm1, xmm2, xmm3/m128*	B	V/V	AVX	Multiply 16-bit signed words, scale and round signed doublewords, pack high 16 bits to *xmm1*.
VEX.NDS.256.66.0F38.WIG 0B /r VPMULHRSW *ymm1, ymm2, ymm3/m256*	B	V/V	AVX2	Multiply 16-bit signed words, scale and round signed doublewords, pack high 16 bits to *ymm1*.
EVEX.NDS.128.66.0F38.WIG 0B /r VPMULHRSW xmm1 {k1}{z}, xmm2, xmm3/m128	C	V/V	AVX512VL AVX512BW	Multiply 16-bit signed words, scale and round signed doublewords, pack high 16 bits to xmm1 under writemask k1.
EVEX.NDS.256.66.0F38.WIG 0B /r VPMULHRSW ymm1 {k1}{z}, ymm2, ymm3/m256	C	V/V	AVX512VL AVX512BW	Multiply 16-bit signed words, scale and round signed doublewords, pack high 16 bits to ymm1 under writemask k1.
EVEX.NDS.512.66.0F38.WIG 0B /r VPMULHRSW zmm1 {k1}{z}, zmm2, zmm3/m512	C	V/V	AVX512BW	Multiply 16-bit signed words, scale and round signed doublewords, pack high 16 bits to zmm1 under writemask k1.

NOTES:

1. See note in Section 2.4, "AVX and SSE Instruction Exception Specification" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A* and Section 22.25.3, "Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	NA	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
B	NA	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA
C	Full Vector Mem	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA

Description

PMULHRSW multiplies vertically each signed 16-bit integer from the destination operand (first operand) with the corresponding signed 16-bit integer of the source operand (second operand), producing intermediate, signed 32-bit integers. Each intermediate 32-bit integer is truncated to the 18 most significant bits. Rounding is always performed by adding 1 to the least significant bit of the 18-bit intermediate result. The final result is obtained by selecting the 16 bits immediately to the right of the most significant bit of each 18-bit intermediate result and packed to the destination operand.

When the source operand is a 128-bit memory operand, the operand must be aligned on a 16-byte boundary or a general-protection exception (#GP) will be generated.

In 64-bit mode and not encoded with VEX/EVEX, use the REX prefix to access XMM8-XMM15 registers.

Legacy SSE version 64-bit operand: Both operands can be MMX registers. The second source operand is an MMX register or a 64-bit memory location.

128-bit Legacy SSE version: The first source and destination operands are XMM registers. The second source operand is an XMM register or a 128-bit memory location. Bits (MAXVL-1:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: The first source and destination operands are XMM registers. The second source operand is an XMM register or a 128-bit memory location. Bits (MAXVL-1:128) of the destination YMM register are zeroed.

VEX.256 encoded version: The second source operand can be an YMM register or a 256-bit memory location. The first source and destination operands are YMM registers.

EVEX encoded versions: The first source operand is a ZMM/YMM/XMM register. The second source operand can be a ZMM/YMM/XMM register, a 512/256/128-bit memory location. The destination operand is a ZMM/YMM/XMM register conditionally updated with writemask k1.

Operation

PMULHRSW (with 64-bit operands)

```
temp0[31:0] = INT32 ((DEST[15:0] * SRC[15:0]) >>14) + 1;
temp1[31:0] = INT32 ((DEST[31:16] * SRC[31:16]) >>14) + 1;
temp2[31:0] = INT32 ((DEST[47:32] * SRC[47:32]) >> 14) + 1;
temp3[31:0] = INT32 ((DEST[63:48] * SRc[63:48]) >> 14) + 1;
DEST[15:0] = temp0[16:1];
DEST[31:16] = temp1[16:1];
DEST[47:32] = temp2[16:1];
DEST[63:48] = temp3[16:1];
```

PMULHRSW (with 128-bit operand)

```
temp0[31:0] = INT32 ((DEST[15:0] * SRC[15:0]) >>14) + 1;
temp1[31:0] = INT32 ((DEST[31:16] * SRC[31:16]) >>14) + 1;
temp2[31:0] = INT32 ((DEST[47:32] * SRC[47:32]) >>14) + 1;
temp3[31:0] = INT32 ((DEST[63:48] * SRC[63:48]) >>14) + 1;
temp4[31:0] = INT32 ((DEST[79:64] * SRC[79:64]) >>14) + 1;
temp5[31:0] = INT32 ((DEST[95:80] * SRC[95:80]) >>14) + 1;
temp6[31:0] = INT32 ((DEST[111:96] * SRC[111:96]) >>14) + 1;
temp7[31:0] = INT32 ((DEST[127:112] * SRC[127:112) >>14) + 1;
DEST[15:0] = temp0[16:1];
DEST[31:16] = temp1[16:1];
DEST[47:32] = temp2[16:1];
DEST[63:48] = temp3[16:1];
DEST[79:64] = temp4[16:1];
DEST[95:80] = temp5[16:1];
DEST[111:96] = temp6[16:1];
DEST[127:112] = temp7[16:1];
```

VPMULHRSW (VEX.128 encoded version)

```
temp0[31:0] ← INT32 ((SRC1[15:0] * SRC2[15:0]) >>14) + 1
temp1[31:0] ← INT32 ((SRC1[31:16] * SRC2[31:16]) >>14) + 1
temp2[31:0] ← INT32 ((SRC1[47:32] * SRC2[47:32]) >>14) + 1
temp3[31:0] ← INT32 ((SRC1[63:48] * SRC2[63:48]) >>14) + 1
temp4[31:0] ← INT32 ((SRC1[79:64] * SRC2[79:64]) >>14) + 1
temp5[31:0] ← INT32 ((SRC1[95:80] * SRC2[95:80]) >>14) + 1
temp6[31:0] ← INT32 ((SRC1[111:96] * SRC2[111:96]) >>14) + 1
temp7[31:0] ← INT32 ((SRC1[127:112] * SRC2[127:112) >>14) + 1
DEST[15:0] ← temp0[16:1]
DEST[31:16] ← temp1[16:1]
DEST[47:32] ← temp2[16:1]
```

```
DEST[63:48] ← temp3[16:1]
DEST[79:64] ← temp4[16:1]
DEST[95:80] ← temp5[16:1]
DEST[111:96] ← temp6[16:1]
DEST[127:112] ← temp7[16:1]
DEST[MAXVL-1:128] ← 0
```

VPMULHRSW (VEX.256 encoded version)

```
temp0[31:0] ← INT32 ((SRC1[15:0] * SRC2[15:0]) >>14) + 1
temp1[31:0] ← INT32 ((SRC1[31:16] * SRC2[31:16]) >>14) + 1
temp2[31:0] ← INT32 ((SRC1[47:32] * SRC2[47:32]) >>14) + 1
temp3[31:0] ← INT32 ((SRC1[63:48] * SRC2[63:48]) >>14) + 1
temp4[31:0] ← INT32 ((SRC1[79:64] * SRC2[79:64]) >>14) + 1
temp5[31:0] ← INT32 ((SRC1[95:80] * SRC2[95:80]) >>14) + 1
temp6[31:0] ← INT32 ((SRC1[111:96] * SRC2[111:96]) >>14) + 1
temp7[31:0] ← INT32 ((SRC1[127:112] * SRC2[127:112) >>14) + 1
temp8[31:0] ← INT32 ((SRC1[143:128] * SRC2[143:128]) >>14) + 1
temp9[31:0] ← INT32 ((SRC1[159:144] * SRC2[159:144]) >>14) + 1
temp10[31:0] ← INT32 ((SRC1[75:160] * SRC2[175:160]) >>14) + 1
temp11[31:0] ← INT32 ((SRC1[191:176] * SRC2[191:176]) >>14) + 1
temp12[31:0] ← INT32 ((SRC1[207:192] * SRC2[207:192]) >>14) + 1
temp13[31:0] ← INT32 ((SRC1[223:208] * SRC2[223:208]) >>14) + 1
temp14[31:0] ← INT32 ((SRC1[239:224] * SRC2[239:224]) >>14) + 1
temp15[31:0] ← INT32 ((SRC1[255:240] * SRC2[255:240) >>14) + 1

DEST[15:0] ← temp0[16:1]
DEST[31:16] ← temp1[16:1]
DEST[47:32] ← temp2[16:1]
DEST[63:48] ← temp3[16:1]
DEST[79:64] ← temp4[16:1]
DEST[95:80] ← temp5[16:1]
DEST[111:96] ← temp6[16:1]
DEST[127:112] ← temp7[16:1]
DEST[143:128] ← temp8[16:1]
DEST[159:144] ← temp9[16:1]
DEST[175:160] ← temp10[16:1]
DEST[191:176] ← temp11[16:1]
DEST[207:192] ← temp12[16:1]
DEST[223:208] ← temp13[16:1]
DEST[239:224] ← temp14[16:1]
DEST[255:240] ← temp15[16:1]
DEST[MAXVL-1:256] ← 0
```

VPMULHRSW (EVEX encoded version)

```
(KL, VL) = (8, 128), (16, 256), (32, 512)
FOR j ← 0 TO KL-1
    i ← j * 16
    IF k1[j] OR *no writemask*
        THEN
            temp[31:0] ← ((SRC1[i+15:i] * SRC2[i+15:i]) >>14) + 1
            DEST[i+15:i] ← tmp[16:1]
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+15:i] remains unchanged*
```

```
                ELSE *zeroing-masking*              ; zeroing-masking
                    DEST[i+15:i] ← 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0
```

Intel C/C++ Compiler Intrinsic Equivalents

VPMULHRSW __m512i _mm512_mulhrs_epi16(__m512i a, __m512i b);
VPMULHRSW __m512i _mm512_mask_mulhrs_epi16(__m512i s, __mmask32 k, __m512i a, __m512i b);
VPMULHRSW __m512i _mm512_maskz_mulhrs_epi16(__mmask32 k, __m512i a, __m512i b);
VPMULHRSW __m256i _mm256_mask_mulhrs_epi16(__m256i s, __mmask16 k, __m256i a, __m256i b);
VPMULHRSW __m256i _mm256_maskz_mulhrs_epi16(__mmask16 k, __m256i a, __m256i b);
VPMULHRSW __m128i _mm_mask_mulhrs_epi16(__m128i s, __mmask8 k, __m128i a, __m128i b);
VPMULHRSW __m128i _mm_maskz_mulhrs_epi16(__mmask8 k, __m128i a, __m128i b);
PMULHRSW: __m64 _mm_mulhrs_pi16 (__m64 a, __m64 b)
(V)PMULHRSW: __m128i _mm_mulhrs_epi16 (__m128i a, __m128i b)
VPMULHRSW:__m256i _mm256_mulhrs_epi16 (__m256i a, __m256i b)

SIMD Floating-Point Exceptions

None.

Other Exceptions

Non-EVEX-encoded instruction, see Exceptions Type 4.
EVEX-encoded instruction, see Exceptions Type E4.nb.

PMULHUW—Multiply Packed Unsigned Integers and Store High Result

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F E4 /r[1] PMULHUW *mm1*, *mm2/m64*	A	V/V	SSE	Multiply the packed unsigned word integers in *mm1* register and *mm2/m64*, and store the high 16 bits of the results in *mm1*.
66 0F E4 /r PMULHUW *xmm1*, *xmm2/m128*	A	V/V	SSE2	Multiply the packed unsigned word integers in *xmm1* and *xmm2/m128*, and store the high 16 bits of the results in *xmm1*.
VEX.NDS.128.66.0F.WIG E4 /r VPMULHUW *xmm1, xmm2, xmm3/m128*	B	V/V	AVX	Multiply the packed unsigned word integers in *xmm2* and *xmm3/m128*, and store the high 16 bits of the results in *xmm1*.
VEX.NDS.256.66.0F.WIG E4 /r VPMULHUW *ymm1, ymm2, ymm3/m256*	B	V/V	AVX2	Multiply the packed unsigned word integers in *ymm2* and *ymm3/m256*, and store the high 16 bits of the results in *ymm1*.
EVEX.NDS.128.66.0F.WIG E4 /r VPMULHUW xmm1 {k1}{z}, xmm2, xmm3/m128	C	V/V	AVX512VL AVX512BW	Multiply the packed unsigned word integers in xmm2 and xmm3/m128, and store the high 16 bits of the results in xmm1 under writemask k1.
EVEX.NDS.256.66.0F.WIG E4 /r VPMULHUW ymm1 {k1}{z}, ymm2, ymm3/m256	C	V/V	AVX512VL AVX512BW	Multiply the packed unsigned word integers in ymm2 and ymm3/m256, and store the high 16 bits of the results in ymm1 under writemask k1.
EVEX.NDS.512.66.0F.WIG E4 /r VPMULHUW zmm1 {k1}{z}, zmm2, zmm3/m512	C	V/V	AVX512BW	Multiply the packed unsigned word integers in zmm2 and zmm3/m512, and store the high 16 bits of the results in zmm1 under writemask k1.

NOTES:

1. See note in Section 2.4, "AVX and SSE Instruction Exception Specification" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A* and Section 22.25.3, "Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	NA	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
B	NA	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA
C	Full Vector Mem	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA

Description

Performs a SIMD unsigned multiply of the packed unsigned word integers in the destination operand (first operand) and the source operand (second operand), and stores the high 16 bits of each 32-bit intermediate results in the destination operand. (Figure 4-12 shows this operation when using 64-bit operands.)

In 64-bit mode and not encoded with VEX/EVEX, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

Legacy SSE version 64-bit operand: The source operand can be an MMX technology register or a 64-bit memory location. The destination operand is an MMX technology register.

128-bit Legacy SSE version: The first source and destination operands are XMM registers. The second source operand is an XMM register or a 128-bit memory location. Bits (MAXVL-1:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: The first source and destination operands are XMM registers. The second source operand is an XMM register or a 128-bit memory location. Bits (MAXVL-1:128) of the destination YMM register are zeroed. VEX.L must be 0, otherwise the instruction will #UD.

VEX.256 encoded version: The second source operand can be an YMM register or a 256-bit memory location. The first source and destination operands are YMM registers.

EVEX encoded versions: The first source operand is a ZMM/YMM/XMM register. The second source operand can be a ZMM/YMM/XMM register, a 512/256/128-bit memory location. The destination operand is a ZMM/YMM/XMM register conditionally updated with writemask k1.

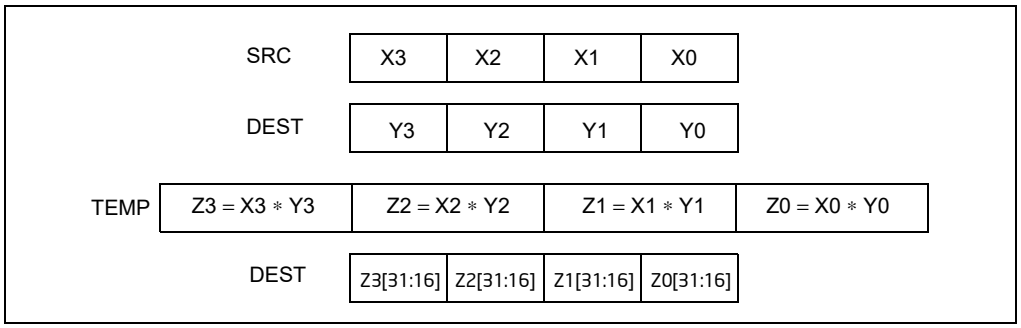


Figure 4-12. PMULHUW and PMULHW Instruction Operation Using 64-bit Operands

Operation

PMULHUW (with 64-bit operands)

```
TEMP0[31:0] ← DEST[15:0] * SRC[15:0]; (* Unsigned multiplication *)
TEMP1[31:0] ← DEST[31:16] * SRC[31:16];
TEMP2[31:0] ← DEST[47:32] * SRC[47:32];
TEMP3[31:0] ← DEST[63:48] * SRC[63:48];
DEST[15:0] ← TEMP0[31:16];
DEST[31:16] ← TEMP1[31:16];
DEST[47:32] ← TEMP2[31:16];
DEST[63:48] ← TEMP3[31:16];
```

PMULHUW (with 128-bit operands)

```
TEMP0[31:0] ← DEST[15:0] * SRC[15:0]; (* Unsigned multiplication *)
TEMP1[31:0] ← DEST[31:16] * SRC[31:16];
TEMP2[31:0] ← DEST[47:32] * SRC[47:32];
TEMP3[31:0] ← DEST[63:48] * SRC[63:48];
TEMP4[31:0] ← DEST[79:64] * SRC[79:64];
TEMP5[31:0] ← DEST[95:80] * SRC[95:80];
TEMP6[31:0] ← DEST[111:96] * SRC[111:96];
TEMP7[31:0] ← DEST[127:112] * SRC[127:112];
DEST[15:0] ← TEMP0[31:16];
DEST[31:16] ← TEMP1[31:16];
DEST[47:32] ← TEMP2[31:16];
DEST[63:48] ← TEMP3[31:16];
DEST[79:64] ← TEMP4[31:16];
DEST[95:80] ← TEMP5[31:16];
DEST[111:96] ← TEMP6[31:16];
DEST[127:112] ← TEMP7[31:16];
```

VPMULHUW (VEX.128 encoded version)

```
TEMP0[31:0] ← SRC1[15:0] * SRC2[15:0]
TEMP1[31:0] ← SRC1[31:16] * SRC2[31:16]
TEMP2[31:0] ← SRC1[47:32] * SRC2[47:32]
TEMP3[31:0] ← SRC1[63:48] * SRC2[63:48]
TEMP4[31:0] ← SRC1[79:64] * SRC2[79:64]
TEMP5[31:0] ← SRC1[95:80] * SRC2[95:80]
TEMP6[31:0] ← SRC1[111:96] * SRC2[111:96]
TEMP7[31:0] ← SRC1[127:112] * SRC2[127:112]
DEST[15:0] ← TEMP0[31:16]
DEST[31:16] ← TEMP1[31:16]
DEST[47:32] ← TEMP2[31:16]
DEST[63:48] ← TEMP3[31:16]
DEST[79:64] ← TEMP4[31:16]
DEST[95:80] ← TEMP5[31:16]
DEST[111:96] ← TEMP6[31:16]
DEST[127:112] ← TEMP7[31:16]
DEST[MAXVL-1:128] ← 0
```

PMULHUW (VEX.256 encoded version)

```
TEMP0[31:0] ← SRC1[15:0] * SRC2[15:0]
TEMP1[31:0] ← SRC1[31:16] * SRC2[31:16]
TEMP2[31:0] ← SRC1[47:32] * SRC2[47:32]
TEMP3[31:0] ← SRC1[63:48] * SRC2[63:48]
TEMP4[31:0] ← SRC1[79:64] * SRC2[79:64]
TEMP5[31:0] ← SRC1[95:80] * SRC2[95:80]
TEMP6[31:0] ← SRC1[111:96] * SRC2[111:96]
TEMP7[31:0] ← SRC1[127:112] * SRC2[127:112]
TEMP8[31:0] ← SRC1[143:128] * SRC2[143:128]
TEMP9[31:0] ← SRC1[159:144] * SRC2[159:144]
TEMP10[31:0] ← SRC1[175:160] * SRC2[175:160]
TEMP11[31:0] ← SRC1[191:176] * SRC2[191:176]
TEMP12[31:0] ← SRC1[207:192] * SRC2[207:192]
TEMP13[31:0] ← SRC1[223:208] * SRC2[223:208]
TEMP14[31:0] ← SRC1[239:224] * SRC2[239:224]
TEMP15[31:0] ← SRC1[255:240] * SRC2[255:240]
DEST[15:0] ← TEMP0[31:16]
DEST[31:16] ← TEMP1[31:16]
DEST[47:32] ← TEMP2[31:16]
DEST[63:48] ← TEMP3[31:16]
DEST[79:64] ← TEMP4[31:16]
DEST[95:80] ← TEMP5[31:16]
DEST[111:96] ← TEMP6[31:16]
DEST[127:112] ← TEMP7[31:16]
DEST[143:128] ← TEMP8[31:16]
DEST[159:144] ← TEMP9[31:16]
DEST[175:160] ← TEMP10[31:16]
DEST[191:176] ← TEMP11[31:16]
DEST[207:192] ← TEMP12[31:16]
DEST[223:208] ← TEMP13[31:16]
DEST[239:224] ← TEMP14[31:16]
DEST[255:240] ← TEMP15[31:16]
DEST[MAXVL-1:256] ← 0
```

PMULHUW (EVEX encoded versions)

```
(KL, VL) = (8, 128), (16, 256), (32, 512)
FOR j ← 0 TO KL-1
    i ← j * 16
    IF k1[j] OR *no writemask*
        THEN
            temp[31:0] ← SRC1[i+15:i] * SRC2[i+15:i]
            DEST[i+15:i] ← tmp[31:16]
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+15:i] remains unchanged*
                ELSE *zeroing-masking*                  ; zeroing-masking
                    DEST[i+15:i] ← 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VPMULHUW __m512i _mm512_mulhi_epu16(__m512i a, __m512i b);
VPMULHUW __m512i _mm512_mask_mulhi_epu16(__m512i s, __mmask32 k, __m512i a, __m512i b);
VPMULHUW __m512i _mm512_maskz_mulhi_epu16(__mmask32 k, __m512i a, __m512i b);
VPMULHUW __m256i _mm256_mask_mulhi_epu16(__m256i s, __mmask16 k, __m256i a, __m256i b);
VPMULHUW __m256i _mm256_maskz_mulhi_epu16(__mmask16 k, __m256i a, __m256i b);
VPMULHUW __m128i _mm_mask_mulhi_epu16(__m128i s, __mmask8 k, __m128i a, __m128i b);
VPMULHUW __m128i _mm_maskz_mulhi_epu16(__mmask8 k, __m128i a, __m128i b);
PMULHUW:__m64 _mm_mulhi_pu16(__m64 a, __m64 b)
(V)PMULHUW:__m128i _mm_mulhi_epu16 (__m128i a, __m128i b)
VPMULHUW:__m256i _mm256_mulhi_epu16 (__m256i a, __m256i b)

Flags Affected

None.

Numeric Exceptions

None.

Other Exceptions

Non-EVEX-encoded instruction, see Exceptions Type 4.
EVEX-encoded instruction, see Exceptions Type E4.nb.

PMULHW—Multiply Packed Signed Integers and Store High Result

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F E5 /r[1] PMULHW *mm, mm/m64*	A	V/V	MMX	Multiply the packed signed word integers in *mm1* register and *mm2/m64*, and store the high 16 bits of the results in *mm1*.
66 0F E5 /r PMULHW *xmm1, xmm2/m128*	A	V/V	SSE2	Multiply the packed signed word integers in *xmm1* and *xmm2/m128*, and store the high 16 bits of the results in *xmm1*.
VEX.NDS.128.66.0F.WIG E5 /r VPMULHW *xmm1, xmm2, xmm3/m128*	B	V/V	AVX	Multiply the packed signed word integers in *xmm2* and *xmm3/m128*, and store the high 16 bits of the results in *xmm1*.
VEX.NDS.256.66.0F.WIG E5 /r VPMULHW *ymm1, ymm2, ymm3/m256*	B	V/V	AVX2	Multiply the packed signed word integers in *ymm2* and *ymm3/m256*, and store the high 16 bits of the results in *ymm1*.
EVEX.NDS.128.66.0F.WIG E5 /r VPMULHW xmm1 {k1}{z}, xmm2, xmm3/m128	C	V/V	AVX512VL AVX512BW	Multiply the packed signed word integers in xmm2 and xmm3/m128, and store the high 16 bits of the results in xmm1 under writemask k1.
EVEX.NDS.256.66.0F.WIG E5 /r VPMULHW ymm1 {k1}{z}, ymm2, ymm3/m256	C	V/V	AVX512VL AVX512BW	Multiply the packed signed word integers in ymm2 and ymm3/m256, and store the high 16 bits of the results in ymm1 under writemask k1.
EVEX.NDS.512.66.0F.WIG E5 /r VPMULHW zmm1 {k1}{z}, zmm2, zmm3/m512	C	V/V	AVX512BW	Multiply the packed signed word integers in zmm2 and zmm3/m512, and store the high 16 bits of the results in zmm1 under writemask k1.

NOTES:

1. See note in Section 2.4, "AVX and SSE Instruction Exception Specification" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A* and Section 22.25.3, "Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	NA	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
B	NA	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA
C	Full Vector Mem	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA

Description

Performs a SIMD signed multiply of the packed signed word integers in the destination operand (first operand) and the source operand (second operand), and stores the high 16 bits of each intermediate 32-bit result in the destination operand. (Figure 4-12 shows this operation when using 64-bit operands.)

n 64-bit mode and not encoded with VEX/EVEX, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

Legacy SSE version 64-bit operand: The source operand can be an MMX technology register or a 64-bit memory location. The destination operand is an MMX technology register.

128-bit Legacy SSE version: The first source and destination operands are XMM registers. The second source operand is an XMM register or a 128-bit memory location. Bits (MAXVL-1:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: The first source and destination operands are XMM registers. The second source operand is an XMM register or a 128-bit memory location. Bits (MAXVL-1:128) of the destination YMM register are zeroed. VEX.L must be 0, otherwise the instruction will #UD.

VEX.256 encoded version: The second source operand can be an YMM register or a 256-bit memory location. The first source and destination operands are YMM registers.

EVEX encoded versions: The first source operand is a ZMM/YMM/XMM register. The second source operand can be a ZMM/YMM/XMM register, a 512/256/128-bit memory location. The destination operand is a ZMM/YMM/XMM register conditionally updated with writemask k1.

Operation

PMULHW (with 64-bit operands)

```
TEMP0[31:0] ← DEST[15:0] * SRC[15:0]; (* Signed multiplication *)
TEMP1[31:0] ← DEST[31:16] * SRC[31:16];
TEMP2[31:0] ← DEST[47:32] * SRC[47:32];
TEMP3[31:0] ← DEST[63:48] * SRC[63:48];
DEST[15:0] ← TEMP0[31:16];
DEST[31:16] ← TEMP1[31:16];
DEST[47:32] ← TEMP2[31:16];
DEST[63:48] ← TEMP3[31:16];
```

PMULHW (with 128-bit operands)

```
TEMP0[31:0] ← DEST[15:0] * SRC[15:0]; (* Signed multiplication *)
TEMP1[31:0] ← DEST[31:16] * SRC[31:16];
TEMP2[31:0] ← DEST[47:32] * SRC[47:32];
TEMP3[31:0] ← DEST[63:48] * SRC[63:48];
TEMP4[31:0] ← DEST[79:64] * SRC[79:64];
TEMP5[31:0] ← DEST[95:80] * SRC[95:80];
TEMP6[31:0] ← DEST[111:96] * SRC[111:96];
TEMP7[31:0] ← DEST[127:112] * SRC[127:112];
DEST[15:0] ← TEMP0[31:16];
DEST[31:16] ← TEMP1[31:16];
DEST[47:32] ← TEMP2[31:16];
DEST[63:48] ← TEMP3[31:16];
DEST[79:64] ← TEMP4[31:16];
DEST[95:80] ← TEMP5[31:16];
DEST[111:96] ← TEMP6[31:16];
DEST[127:112] ← TEMP7[31:16];
```

VPMULHW (VEX.128 encoded version)

```
TEMP0[31:0] ← SRC1[15:0] * SRC2[15:0] (*Signed Multiplication*)
TEMP1[31:0] ← SRC1[31:16] * SRC2[31:16]
TEMP2[31:0] ← SRC1[47:32] * SRC2[47:32]
TEMP3[31:0] ← SRC1[63:48] * SRC2[63:48]
TEMP4[31:0] ← SRC1[79:64] * SRC2[79:64]
TEMP5[31:0] ← SRC1[95:80] * SRC2[95:80]
TEMP6[31:0] ← SRC1[111:96] * SRC2[111:96]
TEMP7[31:0] ← SRC1[127:112] * SRC2[127:112]
DEST[15:0] ← TEMP0[31:16]
DEST[31:16] ← TEMP1[31:16]
DEST[47:32] ← TEMP2[31:16]
DEST[63:48] ← TEMP3[31:16]
DEST[79:64] ← TEMP4[31:16]
DEST[95:80] ← TEMP5[31:16]
DEST[111:96] ← TEMP6[31:16]
DEST[127:112] ← TEMP7[31:16]
DEST[MAXVL-1:128] ← 0
```

PMULHW (VEX.256 encoded version)

```
TEMP0[31:0] ← SRC1[15:0] * SRC2[15:0] (*Signed Multiplication*)
TEMP1[31:0] ← SRC1[31:16] * SRC2[31:16]
TEMP2[31:0] ← SRC1[47:32] * SRC2[47:32]
TEMP3[31:0] ← SRC1[63:48] * SRC2[63:48]
TEMP4[31:0] ← SRC1[79:64] * SRC2[79:64]
TEMP5[31:0] ← SRC1[95:80] * SRC2[95:80]
TEMP6[31:0] ← SRC1[111:96] * SRC2[111:96]
TEMP7[31:0] ← SRC1[127:112] * SRC2[127:112]
TEMP8[31:0] ← SRC1[143:128] * SRC2[143:128]
TEMP9[31:0] ← SRC1[159:144] * SRC2[159:144]
TEMP10[31:0] ← SRC1[175:160] * SRC2[175:160]
TEMP11[31:0] ← SRC1[191:176] * SRC2[191:176]
TEMP12[31:0] ← SRC1[207:192] * SRC2[207:192]
TEMP13[31:0] ← SRC1[223:208] * SRC2[223:208]
TEMP14[31:0] ← SRC1[239:224] * SRC2[239:224]
TEMP15[31:0] ← SRC1[255:240] * SRC2[255:240]
DEST[15:0] ← TEMP0[31:16]
DEST[31:16] ← TEMP1[31:16]
DEST[47:32] ← TEMP2[31:16]
DEST[63:48] ← TEMP3[31:16]
DEST[79:64] ← TEMP4[31:16]
DEST[95:80] ← TEMP5[31:16]
DEST[111:96] ← TEMP6[31:16]
DEST[127:112] ← TEMP7[31:16]
DEST[143:128] ← TEMP8[31:16]
DEST[159:144] ← TEMP9[31:16]
DEST[175:160] ← TEMP10[31:16]
DEST[191:176] ← TEMP11[31:16]
DEST[207:192] ← TEMP12[31:16]
DEST[223:208] ← TEMP13[31:16]
DEST[239:224] ← TEMP14[31:16]
DEST[255:240] ← TEMP15[31:16]
DEST[MAXVL-1:256] ← 0
```

PMULHW (EVEX encoded versions)

```
(KL, VL) = (8, 128), (16, 256), (32, 512)
FOR j ← 0 TO KL-1
    i ← j * 16
    IF k1[j] OR *no writemask*
        THEN
            temp[31:0] ← SRC1[i+15:i] * SRC2[i+15:i]
            DEST[i+15:i] ← tmp[31:16]
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+15:i] remains unchanged*
                ELSE *zeroing-masking*              ; zeroing-masking
                    DEST[i+15:i] ← 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VPMULHW __m512i _mm512_mulhi_epi16(__m512i a, __m512i b);
VPMULHW __m512i _mm512_mask_mulhi_epi16(__m512i s, __mmask32 k, __m512i a, __m512i b);
VPMULHW __m512i _mm512_maskz_mulhi_epi16(__mmask32 k, __m512i a, __m512i b);
VPMULHW __m256i _mm256_mask_mulhi_epi16(__m256i s, __mmask16 k, __m256i a, __m256i b);
VPMULHW __m256i _mm256_maskz_mulhi_epi16(__mmask16 k, __m256i a, __m256i b);
VPMULHW __m128i _mm_mask_mulhi_epi16(__m128i s, __mmask8 k, __m128i a, __m128i b);
VPMULHW __m128i _mm_maskz_mulhi_epi16(__mmask8 k, __m128i a, __m128i b);
PMULHW:__m64 _mm_mulhi_pi16 (__m64 m1, __m64 m2)
(V)PMULHW:__m128i _mm_mulhi_epi16 (__m128i a, __m128i b)
VPMULHW:__m256i _mm256_mulhi_epi16 (__m256i a, __m256i b)

Flags Affected

None.

SIMD Floating-Point Exceptions

None.

Other Exceptions

Non-EVEX-encoded instruction, see Exceptions Type 4.

EVEX-encoded instruction, see Exceptions Type E4.nb.

PMULLD/PMULLQ—Multiply Packed Integers and Store Low Result

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 38 40 /r PMULLD xmm1, xmm2/m128	A	V/V	SSE4_1	Multiply the packed dword signed integers in xmm1 and xmm2/m128 and store the low 32 bits of each product in xmm1.
VEX.NDS.128.66.0F38.WIG 40 /r VPMULLD xmm1, xmm2, xmm3/m128	B	V/V	AVX	Multiply the packed dword signed integers in xmm2 and xmm3/m128 and store the low 32 bits of each product in xmm1.
VEX.NDS.256.66.0F38.WIG 40 /r VPMULLD ymm1, ymm2, ymm3/m256	B	V/V	AVX2	Multiply the packed dword signed integers in ymm2 and ymm3/m256 and store the low 32 bits of each product in ymm1.
EVEX.NDS.128.66.0F38.W0 40 /r VPMULLD xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst	C	V/V	AVX512VL AVX512F	Multiply the packed dword signed integers in xmm2 and xmm3/m128/m32bcst and store the low 32 bits of each product in xmm1 under writemask k1.
EVEX.NDS.256.66.0F38.W0 40 /r VPMULLD ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst	C	V/V	AVX512VL AVX512F	Multiply the packed dword signed integers in ymm2 and ymm3/m256/m32bcst and store the low 32 bits of each product in ymm1 under writemask k1.
EVEX.NDS.512.66.0F38.W0 40 /r VPMULLD zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst	C	V/V	AVX512F	Multiply the packed dword signed integers in zmm2 and zmm3/m512/m32bcst and store the low 32 bits of each product in zmm1 under writemask k1.
EVEX.NDS.128.66.0F38.W1 40 /r VPMULLQ xmm1 {k1}{z}, xmm2, xmm3/m128/m64bcst	C	V/V	AVX512VL AVX512DQ	Multiply the packed qword signed integers in xmm2 and xmm3/m128/m64bcst and store the low 64 bits of each product in xmm1 under writemask k1.
EVEX.NDS.256.66.0F38.W1 40 /r VPMULLQ ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst	C	V/V	AVX512VL AVX512DQ	Multiply the packed qword signed integers in ymm2 and ymm3/m256/m64bcst and store the low 64 bits of each product in ymm1 under writemask k1.
EVEX.NDS.512.66.0F38.W1 40 /r VPMULLQ zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst	C	V/V	AVX512DQ	Multiply the packed qword signed integers in zmm2 and zmm3/m512/m64bcst and store the low 64 bits of each product in zmm1 under writemask k1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	NA	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
B	NA	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA
C	Full Vector	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA

Description

Performs a SIMD signed multiply of the packed signed dword/qword integers from each element of the first source operand with the corresponding element in the second source operand. The low 32/64 bits of each 64/128-bit intermediate results are stored to the destination operand.

128-bit Legacy SSE version: The first source and destination operands are XMM registers. The second source operand is an XMM register or a 128-bit memory location. Bits (MAXVL-1:128) of the corresponding ZMM destination register remain unchanged.

VEX.128 encoded version: The first source and destination operands are XMM registers. The second source operand is an XMM register or a 128-bit memory location. Bits (MAXVL-1:128) of the corresponding ZMM register are zeroed.

VEX.256 encoded version: The first source operand is a YMM register; The second source operand is a YMM register or 256-bit memory location. Bits (MAXVL-1:256) of the corresponding destination ZMM register are zeroed.

EVEX encoded versions: The first source operand is a ZMM/YMM/XMM register. The second source operand is a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 32/64-bit memory location. The destination operand is conditionally updated based on writemask k1.

Operation

VPMULLQ (EVEX encoded versions)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j ← 0 TO KL-1
    i ← j * 64
    IF k1[j] OR *no writemask* THEN
            IF (EVEX.b == 1) AND (SRC2 *is memory*)
                THEN Temp[127:0] ← SRC1[i+63:i] * SRC2[63:0]
                ELSE Temp[127:0] ← SRC1[i+63:i] * SRC2[i+63:i]
            FI;
            DEST[i+63:i] ← Temp[63:0]
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE                                ; zeroing-masking
                    DEST[i+63:i] ← 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0
```

VPMULLD (EVEX encoded versions)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j ← 0 TO KL-1
    i ← j * 32
    IF k1[j] OR *no writemask* THEN
            IF (EVEX.b = 1) AND (SRC2 *is memory*)
                THEN Temp[63:0] ← SRC1[i+31:i] * SRC2[31:0]
                ELSE Temp[63:0] ← SRC1[i+31:i] * SRC2[i+31:i]
            FI;
            DEST[i+31:i] ← Temp[31:0]
        ELSE
            IF *merging-masking*                    ; merging-masking
                *DEST[i+31:i] remains unchanged*
                ELSE                                ; zeroing-masking
                    DEST[i+31:i] ← 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0
```

VPMULLD (VEX.256 encoded version)
Temp0[63:0] ← SRC1[31:0] * SRC2[31:0]
Temp1[63:0] ← SRC1[63:32] * SRC2[63:32]
Temp2[63:0] ← SRC1[95:64] * SRC2[95:64]
Temp3[63:0] ← SRC1[127:96] * SRC2[127:96]
Temp4[63:0] ← SRC1[159:128] * SRC2[159:128]
Temp5[63:0] ← SRC1[191:160] * SRC2[191:160]
Temp6[63:0] ← SRC1[223:192] * SRC2[223:192]
Temp7[63:0] ← SRC1[255:224] * SRC2[255:224]

DEST[31:0] ← Temp0[31:0]
DEST[63:32] ← Temp1[31:0]
DEST[95:64] ← Temp2[31:0]
DEST[127:96] ← Temp3[31:0]
DEST[159:128] ← Temp4[31:0]
DEST[191:160] ← Temp5[31:0]
DEST[223:192] ← Temp6[31:0]
DEST[255:224] ← Temp7[31:0]
DEST[MAXVL-1:256] ← 0

VPMULLD (VEX.128 encoded version)
Temp0[63:0] ← SRC1[31:0] * SRC2[31:0]
Temp1[63:0] ← SRC1[63:32] * SRC2[63:32]
Temp2[63:0] ← SRC1[95:64] * SRC2[95:64]
Temp3[63:0] ← SRC1[127:96] * SRC2[127:96]
DEST[31:0] ← Temp0[31:0]
DEST[63:32] ← Temp1[31:0]
DEST[95:64] ← Temp2[31:0]
DEST[127:96] ← Temp3[31:0]
DEST[MAXVL-1:128] ← 0

PMULLD (128-bit Legacy SSE version)
Temp0[63:0] ← DEST[31:0] * SRC[31:0]
Temp1[63:0] ← DEST[63:32] * SRC[63:32]
Temp2[63:0] ← DEST[95:64] * SRC[95:64]
Temp3[63:0] ← DEST[127:96] * SRC[127:96]
DEST[31:0] ← Temp0[31:0]
DEST[63:32] ← Temp1[31:0]
DEST[95:64] ← Temp2[31:0]
DEST[127:96] ← Temp3[31:0]
DEST[MAXVL-1:128] (Unmodified)

Intel C/C++ Compiler Intrinsic Equivalent

VPMULLD __m512i _mm512_mullo_epi32(__m512i a, __m512i b);
VPMULLD __m512i _mm512_mask_mullo_epi32(__m512i s, __mmask16 k, __m512i a, __m512i b);
VPMULLD __m512i _mm512_maskz_mullo_epi32(__mmask16 k, __m512i a, __m512i b);
VPMULLD __m256i _mm256_mask_mullo_epi32(__m256i s, __mmask8 k, __m256i a, __m256i b);
VPMULLD __m256i _mm256_maskz_mullo_epi32(__mmask8 k, __m256i a, __m256i b);
VPMULLD __m128i _mm_mask_mullo_epi32(__m128i s, __mmask8 k, __m128i a, __m128i b);
VPMULLD __m128i _mm_maskz_mullo_epi32(__mmask8 k, __m128i a, __m128i b);
VPMULLD __m256i _mm256_mullo_epi32(__m256i a, __m256i b);
PMULLD __m128i _mm_mullo_epi32(__m128i a, __m128i b);
VPMULLQ __m512i _mm512_mullo_epi64(__m512i a, __m512i b);
VPMULLQ __m512i _mm512_mask_mullo_epi64(__m512i s, __mmask8 k, __m512i a, __m512i b);

VPMULLQ __m512i _mm512_maskz_mullo_epi64(__mmask8 k, __m512i a, __m512i b);
VPMULLQ __m256i _mm256_mullo_epi64(__m256i a, __m256i b);
VPMULLQ __m256i _mm256_mask_mullo_epi64(__m256i s, __mmask8 k, __m256i a, __m256i b);
VPMULLQ __m256i _mm256_maskz_mullo_epi64(__mmask8 k, __m256i a, __m256i b);
VPMULLQ __m128i _mm_mullo_epi64(__m128i a, __m128i b);
VPMULLQ __m128i _mm_mask_mullo_epi64(__m128i s, __mmask8 k, __m128i a, __m128i b);
VPMULLQ __m128i _mm_maskz_mullo_epi64(__mmask8 k, __m128i a, __m128i b);

SIMD Floating-Point Exceptions

None

Other Exceptions

Non-EVEX-encoded instruction, see Exceptions Type 4.
EVEX-encoded instruction, see Exceptions Type E4.

PMULLW—Multiply Packed Signed Integers and Store Low Result

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F D5 /r[1] PMULLW *mm, mm/m64*	A	V/V	MMX	Multiply the packed signed word integers in *mm1* register and *mm2/m64*, and store the low 16 bits of the results in *mm1*.
66 0F D5 /r PMULLW *xmm1*, *xmm2/m128*	A	V/V	SSE2	Multiply the packed signed word integers in *xmm1* and *xmm2/m128*, and store the low 16 bits of the results in *xmm1*.
VEX.NDS.128.66.0F.WIG D5 /r VPMULLW *xmm1, xmm2, xmm3/m128*	B	V/V	AVX	Multiply the packed dword signed integers in *xmm2* and *xmm3/m128* and store the low 32 bits of each product in *xmm1*.
VEX.NDS.256.66.0F.WIG D5 /r VPMULLW *ymm1, ymm2, ymm3/m256*	B	V/V	AVX2	Multiply the packed signed word integers in *ymm2* and *ymm3/m256*, and store the low 16 bits of the results in *ymm1*.
EVEX.NDS.128.66.0F.WIG D5 /r VPMULLW xmm1 {k1}{z}, xmm2, xmm3/m128	C	V/V	AVX512VL AVX512BW	Multiply the packed signed word integers in xmm2 and xmm3/m128, and store the low 16 bits of the results in xmm1 under writemask k1.
EVEX.NDS.256.66.0F.WIG D5 /r VPMULLW ymm1 {k1}{z}, ymm2, ymm3/m256	C	V/V	AVX512VL AVX512BW	Multiply the packed signed word integers in ymm2 and ymm3/m256, and store the low 16 bits of the results in ymm1 under writemask k1.
EVEX.NDS.512.66.0F.WIG D5 /r VPMULLW zmm1 {k1}{z}, zmm2, zmm3/m512	C	V/V	AVX512BW	Multiply the packed signed word integers in zmm2 and zmm3/m512, and store the low 16 bits of the results in zmm1 under writemask k1.

NOTES:

1. See note in Section 2.4, "AVX and SSE Instruction Exception Specification" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A* and Section 22.25.3, "Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	NA	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
B	NA	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA
C	Full Vector Mem	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA

Description

Performs a SIMD signed multiply of the packed signed word integers in the destination operand (first operand) and the source operand (second operand), and stores the low 16 bits of each intermediate 32-bit result in the destination operand. (Figure 4-12 shows this operation when using 64-bit operands.)

In 64-bit mode and not encoded with VEX/EVEX, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

Legacy SSE version 64-bit operand: The source operand can be an MMX technology register or a 64-bit memory location. The destination operand is an MMX technology register.

128-bit Legacy SSE version: The first source and destination operands are XMM registers. The second source operand is an XMM register or a 128-bit memory location. Bits (MAXVL-1:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: The first source and destination operands are XMM registers. The second source operand is an XMM register or a 128-bit memory location. Bits (MAXVL-1:128) of the destination YMM register are zeroed. VEX.L must be 0, otherwise the instruction will #UD.

VEX.256 encoded version: The second source operand can be an YMM register or a 256-bit memory location. The first source and destination operands are YMM registers.

EVEX encoded versions: The first source operand is a ZMM/YMM/XMM register. The second source operand is a ZMM/YMM/XMM register, a 512/256/128-bit memory location. The destination operand is conditionally updated based on writemask k1.

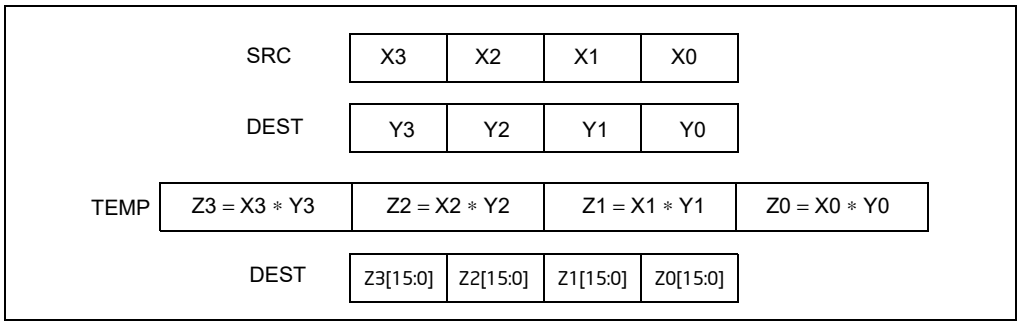


Figure 4-13. PMULLU Instruction Operation Using 64-bit Operands

Operation

PMULLW (with 64-bit operands)

```
TEMP0[31:0] ← DEST[15:0] * SRC[15:0]; (* Signed multiplication *)
TEMP1[31:0] ← DEST[31:16] * SRC[31:16];
TEMP2[31:0] ← DEST[47:32] * SRC[47:32];
TEMP3[31:0] ← DEST[63:48] * SRC[63:48];
DEST[15:0] ← TEMP0[15:0];
DEST[31:16] ← TEMP1[15:0];
DEST[47:32] ← TEMP2[15:0];
DEST[63:48] ← TEMP3[15:0];
```

PMULLW (with 128-bit operands)

```
TEMP0[31:0] ← DEST[15:0] * SRC[15:0]; (* Signed multiplication *)
TEMP1[31:0] ← DEST[31:16] * SRC[31:16];
TEMP2[31:0] ← DEST[47:32] * SRC[47:32];
TEMP3[31:0] ← DEST[63:48] * SRC[63:48];
TEMP4[31:0] ← DEST[79:64] * SRC[79:64];
TEMP5[31:0] ← DEST[95:80] * SRC[95:80];
TEMP6[31:0] ← DEST[111:96] * SRC[111:96];
TEMP7[31:0] ← DEST[127:112] * SRC[127:112];
DEST[15:0] ← TEMP0[15:0];
DEST[31:16] ← TEMP1[15:0];
DEST[47:32] ← TEMP2[15:0];
DEST[63:48] ← TEMP3[15:0];
DEST[79:64] ← TEMP4[15:0];
DEST[95:80] ← TEMP5[15:0];
DEST[111:96] ← TEMP6[15:0];
DEST[127:112] ← TEMP7[15:0];
DEST[MAXVL-1:256] ← 0
```

VPMULLW (VEX.128 encoded version)

```
Temp0[31:0] ← SRC1[15:0] * SRC2[15:0]
Temp1[31:0] ← SRC1[31:16] * SRC2[31:16]
Temp2[31:0] ← SRC1[47:32] * SRC2[47:32]
Temp3[31:0] ← SRC1[63:48] * SRC2[63:48]
Temp4[31:0] ← SRC1[79:64] * SRC2[79:64]
Temp5[31:0] ← SRC1[95:80] * SRC2[95:80]
Temp6[31:0] ← SRC1[111:96] * SRC2[111:96]
Temp7[31:0] ← SRC1[127:112] * SRC2[127:112]
DEST[15:0] ← Temp0[15:0]
DEST[31:16] ← Temp1[15:0]
DEST[47:32] ← Temp2[15:0]
DEST[63:48] ← Temp3[15:0]
DEST[79:64] ← Temp4[15:0]
DEST[95:80] ← Temp5[15:0]
DEST[111:96] ← Temp6[15:0]
DEST[127:112] ← Temp7[15:0]
DEST[MAXVL-1:128] ← 0
```

PMULLW (EVEX encoded versions)

```
(KL, VL) = (8, 128), (16, 256), (32, 512)
FOR j ← 0 TO KL-1
    i ← j * 16
    IF k1[j] OR *no writemask*
        THEN
            temp[31:0] ← SRC1[i+15:i] * SRC2[i+15:i]
            DEST[i+15:i] ← temp[15:0]
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+15:i] remains unchanged*
                ELSE *zeroing-masking*              ; zeroing-masking
                    DEST[i+15:i] ← 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VPMULLW __m512i _mm512_mullo_epi16(__m512i a, __m512i b);
VPMULLW __m512i _mm512_mask_mullo_epi16(__m512i s, __mmask32 k, __m512i a, __m512i b);
VPMULLW __m512i _mm512_maskz_mullo_epi16(__mmask32 k, __m512i a, __m512i b);
VPMULLW __m256i _mm256_mask_mullo_epi16(__m256i s, __mmask16 k, __m256i a, __m256i b);
VPMULLW __m256i _mm256_maskz_mullo_epi16(__mmask16 k, __m256i a, __m256i b);
VPMULLW __m128i _mm_mask_mullo_epi16(__m128i s, __mmask8 k, __m128i a, __m128i b);
VPMULLW __m128i _mm_maskz_mullo_epi16(__mmask8 k, __m128i a, __m128i b);
PMULLW: __m64 _mm_mullo_pi16(__m64 m1, __m64 m2)
(V)PMULLW: __m128i _mm_mullo_epi16 (__m128i a, __m128i b)
VPMULLW:__m256i _mm256_mullo_epi16 (__m256i a, __m256i b);

Flags Affected

None.

SIMD Floating-Point Exceptions

None.

Other Exceptions

Non-EVEX-encoded instruction, see Exceptions Type 4.
EVEX-encoded instruction, see Exceptions Type E4.nb.

PMULUDQ—Multiply Packed Unsigned Doubleword Integers

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F F4 /r[1] PMULUDQ *mm1*, *mm2/m64*	A	V/V	SSE2	Multiply unsigned doubleword integer in *mm1* by unsigned doubleword integer in *mm2/m64*, and store the quadword result in *mm1*.
66 0F F4 /r PMULUDQ *xmm1*, *xmm2/m128*	A	V/V	SSE2	Multiply packed unsigned doubleword integers in *xmm1* by packed unsigned doubleword integers in *xmm2/m128*, and store the quadword results in *xmm1*.
VEX.NDS.128.66.0F.WIG F4 /r VPMULUDQ *xmm1, xmm2, xmm3/m128*	B	V/V	AVX	Multiply packed unsigned doubleword integers in *xmm2* by packed unsigned doubleword integers in *xmm3/m128*, and store the quadword results in *xmm1*.
VEX.NDS.256.66.0F.WIG F4 /r VPMULUDQ *ymm1, ymm2, ymm3/m256*	B	V/V	AVX2	Multiply packed unsigned doubleword integers in *ymm2* by packed unsigned doubleword integers in *ymm3/m256*, and store the quadword results in *ymm1*.
EVEX.NDS.128.66.0F.W1 F4 /r VPMULUDQ xmm1 {k1}{z}, xmm2, xmm3/m128/m64bcst	C	V/V	AVX512VL AVX512F	Multiply packed unsigned doubleword integers in xmm2 by packed unsigned doubleword integers in xmm3/m128/m64bcst, and store the quadword results in xmm1 under writemask k1.
EVEX.NDS.256.66.0F.W1 F4 /r VPMULUDQ ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst	C	V/V	AVX512VL AVX512F	Multiply packed unsigned doubleword integers in ymm2 by packed unsigned doubleword integers in ymm3/m256/m64bcst, and store the quadword results in ymm1 under writemask k1.
EVEX.NDS.512.66.0F.W1 F4 /r VPMULUDQ zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst	C	V/V	AVX512F	Multiply packed unsigned doubleword integers in zmm2 by packed unsigned doubleword integers in zmm3/m512/m64bcst, and store the quadword results in zmm1 under writemask k1.

NOTES:

1. See note in Section 2.4, "AVX and SSE Instruction Exception Specification" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A* and Section 22.25.3, "Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	NA	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
B	NA	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA
C	Full Vector	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA

Description

Multiplies the first operand (destination operand) by the second operand (source operand) and stores the result in the destination operand.

In 64-bit mode and not encoded with VEX/EVEX, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

Legacy SSE version 64-bit operand: The source operand can be an unsigned doubleword integer stored in the low doubleword of an MMX technology register or a 64-bit memory location. The destination operand can be an unsigned doubleword integer stored in the low doubleword an MMX technology register. The result is an unsigned

quadword integer stored in the destination an MMX technology register. When a quadword result is too large to be represented in 64 bits (overflow), the result is wrapped around and the low 64 bits are written to the destination element (that is, the carry is ignored).

For 64-bit memory operands, 64 bits are fetched from memory, but only the low doubleword is used in the computation.

128-bit Legacy SSE version: The second source operand is two packed unsigned doubleword integers stored in the first (low) and third doublewords of an XMM register or a 128-bit memory location. For 128-bit memory operands, 128 bits are fetched from memory, but only the first and third doublewords are used in the computation. The first source operand is two packed unsigned doubleword integers stored in the first and third doublewords of an XMM register. The destination contains two packed unsigned quadword integers stored in an XMM register. Bits (MAXVL-1:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: The second source operand is two packed unsigned doubleword integers stored in the first (low) and third doublewords of an XMM register or a 128-bit memory location. For 128-bit memory operands, 128 bits are fetched from memory, but only the first and third doublewords are used in the computation. The first source operand is two packed unsigned doubleword integers stored in the first and third doublewords of an XMM register. The destination contains two packed unsigned quadword integers stored in an XMM register. Bits (MAXVL-1:128) of the destination YMM register are zeroed.

VEX.256 encoded version: The second source operand is four packed unsigned doubleword integers stored in the first (low), third, fifth and seventh doublewords of a YMM register or a 256-bit memory location. For 256-bit memory operands, 256 bits are fetched from memory, but only the first, third, fifth and seventh doublewords are used in the computation. The first source operand is four packed unsigned doubleword integers stored in the first, third, fifth and seventh doublewords of an YMM register. The destination contains four packed unaligned quadword integers stored in an YMM register.

EVEX encoded version: The input unsigned doubleword integers are taken from the even-numbered elements of the source operands. The first source operand is a ZMM/YMM/XMM registers. The second source operand can be an ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 64-bit memory location. The destination is a ZMM/YMM/XMM register, and updated according to the writemask at 64-bit granularity.

Operation

PMULUDQ (with 64-Bit operands)

```
DEST[63:0] ← DEST[31:0] * SRC[31:0];
```

PMULUDQ (with 128-Bit operands)

```
DEST[63:0] ← DEST[31:0] * SRC[31:0];
DEST[127:64] ← DEST[95:64] * SRC[95:64];
```

VPMULUDQ (VEX.128 encoded version)

```
DEST[63:0] ← SRC1[31:0] * SRC2[31:0]
DEST[127:64] ← SRC1[95:64] * SRC2[95:64]
DEST[MAXVL-1:128] ← 0
```

VPMULUDQ (VEX.256 encoded version)

```
DEST[63:0] ← SRC1[31:0] * SRC2[31:0]
DEST[127:64] ← SRC1[95:64] * SRC2[95:64
DEST[191:128] ← SRC1[159:128] * SRC2[159:128]
DEST[255:192] ← SRC1[223:192] * SRC2[223:192]
DEST[MAXVL-1:256] ← 0
```

VPMULUDQ (EVEX encoded versions)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j ← 0 TO KL-1
    i ← j * 64
    IF k1[j] OR *no writemask* THEN
            IF (EVEX.b = 1) AND (SRC2 *is memory*)
                THEN DEST[i+63:i] ← ZeroExtend64( SRC1[i+31:i]) * ZeroExtend64( SRC2[31:0] )
                ELSE DEST[i+63:i] ← ZeroExtend64( SRC1[i+31:i]) * ZeroExtend64( SRC2[i+31:i] )
            FI;
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE *zeroing-masking*                  ; zeroing-masking
                    DEST[i+63:i] ← 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VPMULUDQ __m512i _mm512_mul_epu32(__m512i a, __m512i b);
VPMULUDQ __m512i _mm512_mask_mul_epu32(__m512i s, __mmask8 k, __m512i a, __m512i b);
VPMULUDQ __m512i _mm512_maskz_mul_epu32(__mmask8 k, __m512i a, __m512i b);
VPMULUDQ __m256i _mm256_mask_mul_epu32(__m256i s, __mmask8 k, __m256i a, __m256i b);
VPMULUDQ __m256i _mm256_maskz_mul_epu32(__mmask8 k, __m256i a, __m256i b);
VPMULUDQ __m128i _mm_mask_mul_epu32(__m128i s, __mmask8 k, __m128i a, __m128i b);
VPMULUDQ __m128i _mm_maskz_mul_epu32(__mmask8 k, __m128i a, __m128i b);
PMULUDQ:__m64 _mm_mul_su32 (__m64 a, __m64 b)
(V)PMULUDQ:__m128i _mm_mul_epu32 (__m128i a, __m128i b)
VPMULUDQ:__m256i _mm256_mul_epu32(__m256i a, __m256i b);

Flags Affected

None.

SIMD Floating-Point Exceptions

None.

Other Exceptions

Non-EVEX-encoded instruction, see Exceptions Type 4.

EVEX-encoded instruction, see Exceptions Type E4.

POP—Pop a Value from the Stack

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
8F /0	POP r/*m16*	M	Valid	Valid	Pop top of stack into *m16*; increment stack pointer.
8F /0	POP r/*m32*	M	N.E.	Valid	Pop top of stack into *m32*; increment stack pointer.
8F /0	POP r/*m64*	M	Valid	N.E.	Pop top of stack into *m64*; increment stack pointer. Cannot encode 32-bit operand size.
58+ *rw*	POP *r16*	O	Valid	Valid	Pop top of stack into *r16*; increment stack pointer.
58+ *rd*	POP *r32*	O	N.E.	Valid	Pop top of stack into *r32*; increment stack pointer.
58+ *rd*	POP *r64*	O	Valid	N.E.	Pop top of stack into *r64*; increment stack pointer. Cannot encode 32-bit operand size.
1F	POP DS	ZO	Invalid	Valid	Pop top of stack into DS; increment stack pointer.
07	POP ES	ZO	Invalid	Valid	Pop top of stack into ES; increment stack pointer.
17	POP SS	ZO	Invalid	Valid	Pop top of stack into SS; increment stack pointer.
0F A1	POP FS	ZO	Valid	Valid	Pop top of stack into FS; increment stack pointer by 16 bits.
0F A1	POP FS	ZO	N.E.	Valid	Pop top of stack into FS; increment stack pointer by 32 bits.
0F A1	POP FS	ZO	Valid	N.E.	Pop top of stack into FS; increment stack pointer by 64 bits.
0F A9	POP GS	ZO	Valid	Valid	Pop top of stack into GS; increment stack pointer by 16 bits.
0F A9	POP GS	ZO	N.E.	Valid	Pop top of stack into GS; increment stack pointer by 32 bits.
0F A9	POP GS	ZO	Valid	N.E.	Pop top of stack into GS; increment stack pointer by 64 bits.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
M	ModRM:r/m (w)	NA	NA	NA
O	opcode + rd (w)	NA	NA	NA
ZO	NA	NA	NA	NA

Description

Loads the value from the top of the stack to the location specified with the destination operand (or explicit opcode) and then increments the stack pointer. The destination operand can be a general-purpose register, memory location, or segment register.

Address and operand sizes are determined and used as follows:

- Address size. The D flag in the current code-segment descriptor determines the default address size; it may be overridden by an instruction prefix (67H).

The address size is used only when writing to a destination operand in memory.

- Operand size. The D flag in the current code-segment descriptor determines the default operand size; it may be overridden by instruction prefixes (66H or REX.W).

  The operand size (16, 32, or 64 bits) determines the amount by which the stack pointer is incremented (2, 4 or 8).

- Stack-address size. Outside of 64-bit mode, the B flag in the current stack-segment descriptor determines the size of the stack pointer (16 or 32 bits); in 64-bit mode, the size of the stack pointer is always 64 bits.

  The stack-address size determines the width of the stack pointer when reading from the stack in memory and when incrementing the stack pointer. (As stated above, the amount by which the stack pointer is incremented is determined by the operand size.)

If the destination operand is one of the segment registers DS, ES, FS, GS, or SS, the value loaded into the register must be a valid segment selector. In protected mode, popping a segment selector into a segment register automatically causes the descriptor information associated with that segment selector to be loaded into the hidden (shadow) part of the segment register and causes the selector and the descriptor information to be validated (see the "Operation" section below).

A NULL value (0000-0003) may be popped into the DS, ES, FS, or GS register without causing a general protection fault. However, any subsequent attempt to reference a segment whose corresponding segment register is loaded with a NULL value causes a general protection exception (#GP). In this situation, no memory reference occurs and the saved value of the segment register is NULL.

The POP instruction cannot pop a value into the CS register. To load the CS register from the stack, use the RET instruction.

If the ESP register is used as a base register for addressing a destination operand in memory, the POP instruction computes the effective address of the operand after it increments the ESP register. For the case of a 16-bit stack where ESP wraps to 0H as a result of the POP instruction, the resulting location of the memory write is processor-family-specific.

The POP ESP instruction increments the stack pointer (ESP) before data at the old top of stack is written into the destination.

A POP SS instruction inhibits all interrupts, including the NMI interrupt, until after execution of the next instruction. This action allows sequential execution of POP SS and MOV ESP, EBP instructions without the danger of having an invalid stack during an interrupt[1]. However, use of the LSS instruction is the preferred method of loading the SS and ESP registers.

In 64-bit mode, using a REX prefix in the form of REX.R permits access to additional registers (R8-R15). When in 64-bit mode, POPs using 32-bit operands are not encodable and POPs to DS, ES, SS are not valid. See the summary chart at the beginning of this section for encoding data and limits.

Operation

```
IF StackAddrSize = 32
    THEN
        IF OperandSize = 32
            THEN
                DEST ← SS:ESP; (* Copy a doubleword *)
                ESP ← ESP + 4;
            ELSE (* OperandSize = 16*)
                DEST ← SS:ESP; (* Copy a word *)
```

1. If a code instruction breakpoint (for debug) is placed on an instruction located immediately after a POP SS instruction, the breakpoint may not be triggered. However, in a sequence of instructions that POP the SS register, only the first instruction in the sequence is guaranteed to delay an interrupt.

   In the following sequence, interrupts may be recognized before POP ESP executes:

   ```
   POP SS
   POP SS
   POP ESP
   ```

```
                ESP ← ESP + 2;
        FI;
    ELSE IF StackAddrSize = 64
        THEN
            IF OperandSize = 64
                THEN
                    DEST ← SS:RSP; (* Copy quadword *)
                    RSP ← RSP + 8;
                ELSE (* OperandSize = 16*)
                    DEST ← SS:RSP; (* Copy a word *)
                    RSP ← RSP + 2;
            FI;
        FI;
    ELSE StackAddrSize = 16
        THEN
            IF OperandSize = 16
                THEN
                    DEST ← SS:SP; (* Copy a word *)
                    SP ← SP + 2;
                ELSE (* OperandSize = 32 *)
                    DEST ← SS:SP; (* Copy a doubleword *)
                    SP ← SP + 4;
            FI;

FI;
```

Loading a segment register while in protected mode results in special actions, as described in the following listing. These checks are performed on the segment selector and the segment descriptor it points to.

```
64-BIT_MODE
IF FS, or GS is loaded with non-NULL selector;
    THEN
        IF segment selector index is outside descriptor table limits
            OR segment is not a data or readable code segment
            OR ((segment is a data or nonconforming code segment)
                AND (both RPL and CPL > DPL))
                    THEN #GP(selector);
            IF segment not marked present
                THEN #NP(selector);
        ELSE
            SegmentRegister ← segment selector;
            SegmentRegister ← segment descriptor;
        FI;
FI;
IF FS, or GS is loaded with a NULL selector;
        THEN
            SegmentRegister ← segment selector;
            SegmentRegister ← segment descriptor;
FI;


PREOTECTED MODE OR COMPATIBILITY MODE;

IF SS is loaded;
```

```
    THEN
        IF segment selector is NULL
            THEN #GP(0);
        FI;
        IF segment selector index is outside descriptor table limits
            or segment selector's RPL ≠ CPL
            or segment is not a writable data segment
            or DPL ≠ CPL
                THEN #GP(selector);
        FI;
        IF segment not marked present
            THEN #SS(selector);
            ELSE
                SS ← segment selector;
                SS ← segment descriptor;
        FI;
FI;

IF DS, ES, FS, or GS is loaded with non-NULL selector;
    THEN
        IF segment selector index is outside descriptor table limits
            or segment is not a data or readable code segment
            or ((segment is a data or nonconforming code segment)
            and (both RPL and CPL > DPL))
                THEN #GP(selector);
        FI;
        IF segment not marked present
            THEN #NP(selector);
            ELSE
                SegmentRegister ← segment selector;
                SegmentRegister ← segment descriptor;
         FI;
FI;

IF DS, ES, FS, or GS is loaded with a NULL selector
    THEN
        SegmentRegister ← segment selector;
        SegmentRegister ← segment descriptor;
FI;
```

Flags Affected

None.

Protected Mode Exceptions

#GP(0)	If attempt is made to load SS register with NULL segment selector.
	If the destination operand is in a non-writable segment.
	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
	If the DS, ES, FS, or GS register is used to access memory and it contains a NULL segment selector.
#GP(selector)	If segment selector index is outside descriptor table limits.
	If the SS register is being loaded and the segment selector's RPL and the segment descriptor's DPL are not equal to the CPL.

	If the SS register is being loaded and the segment pointed to is a non-writable data segment.
	If the DS, ES, FS, or GS register is being loaded and the segment pointed to is not a data or readable code segment.
	If the DS, ES, FS, or GS register is being loaded and the segment pointed to is a data or nonconforming code segment, but both the RPL and the CPL are greater than the DPL.
#SS(0)	If the current top of stack is not within the stack segment.
	If a memory operand effective address is outside the SS segment limit.
#SS(selector)	If the SS register is being loaded and the segment pointed to is marked not present.
#NP	If the DS, ES, FS, or GS register is being loaded and the segment pointed to is marked not present.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If an unaligned memory reference is made while the current privilege level is 3 and alignment checking is enabled.
#UD	If the LOCK prefix is used.

Real-Address Mode Exceptions

#GP	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#UD	If the LOCK prefix is used.

Virtual-8086 Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If an unaligned memory reference is made while alignment checking is enabled.
#UD	If the LOCK prefix is used.

Compatibility Mode Exceptions

Same as for protected mode exceptions.

64-Bit Mode Exceptions

#GP(0)	If the memory address is in a non-canonical form.
#SS(0)	If the stack address is in a non-canonical form.
#GP(selector)	If the descriptor is outside the descriptor table limit.
	If the FS or GS register is being loaded and the segment pointed to is not a data or readable code segment.
	If the FS or GS register is being loaded and the segment pointed to is a data or nonconforming code segment, but both the RPL and the CPL are greater than the DPL.
#AC(0)	If an unaligned memory reference is made while alignment checking is enabled.
#PF(fault-code)	If a page fault occurs.
#NP	If the FS or GS register is being loaded and the segment pointed to is marked not present.
#UD	If the LOCK prefix is used.

POPA/POPAD—Pop All General-Purpose Registers

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
61	POPA	ZO	Invalid	Valid	Pop DI, SI, BP, BX, DX, CX, and AX.
61	POPAD	ZO	Invalid	Valid	Pop EDI, ESI, EBP, EBX, EDX, ECX, and EAX.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
ZO	NA	NA	NA	NA

Description

Pops doublewords (POPAD) or words (POPA) from the stack into the general-purpose registers. The registers are loaded in the following order: EDI, ESI, EBP, EBX, EDX, ECX, and EAX (if the operand-size attribute is 32) and DI, SI, BP, BX, DX, CX, and AX (if the operand-size attribute is 16). (These instructions reverse the operation of the PUSHA/PUSHAD instructions.) The value on the stack for the ESP or SP register is ignored. Instead, the ESP or SP register is incremented after each register is loaded.

The POPA (pop all) and POPAD (pop all double) mnemonics reference the same opcode. The POPA instruction is intended for use when the operand-size attribute is 16 and the POPAD instruction for when the operand-size attribute is 32. Some assemblers may force the operand size to 16 when POPA is used and to 32 when POPAD is used (using the operand-size override prefix [66H] if necessary). Others may treat these mnemonics as synonyms (POPA/POPAD) and use the current setting of the operand-size attribute to determine the size of values to be popped from the stack, regardless of the mnemonic used. (The D flag in the current code segment's segment descriptor determines the operand-size attribute.)

This instruction executes as described in non-64-bit modes. It is not valid in 64-bit mode.

Operation

```
IF 64-Bit Mode
    THEN
        #UD;
ELSE
    IF OperandSize = 32 (* Instruction = POPAD *)
    THEN
        EDI ← Pop();
        ESI ← Pop();
        EBP ← Pop();
        Increment ESP by 4; (* Skip next 4 bytes of stack *)
        EBX ← Pop();
        EDX ← Pop();
        ECX ← Pop();
        EAX ← Pop();
    ELSE (* OperandSize = 16, instruction = POPA *)
        DI ← Pop();
        SI ← Pop();
        BP ← Pop();
        Increment ESP by 2; (* Skip next 2 bytes of stack *)
        BX ← Pop();
        DX ← Pop();
        CX ← Pop();
        AX ← Pop();
    FI;
FI;
```

Flags Affected

None.

Protected Mode Exceptions

#SS(0)	If the starting or ending stack address is not within the stack segment.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If an unaligned memory reference is made while the current privilege level is 3 and alignment checking is enabled.
#UD	If the LOCK prefix is used.

Real-Address Mode Exceptions

#SS	If the starting or ending stack address is not within the stack segment.
#UD	If the LOCK prefix is used.

Virtual-8086 Mode Exceptions

#SS(0)	If the starting or ending stack address is not within the stack segment.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If an unaligned memory reference is made while alignment checking is enabled.
#UD	If the LOCK prefix is used.

Compatibility Mode Exceptions

Same as for protected mode exceptions.

64-Bit Mode Exceptions

#UD	If in 64-bit mode.

POPCNT — Return the Count of Number of Bits Set to 1

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
F3 0F B8 /r	POPCNT *r16, r/m16*	RM	Valid	Valid	POPCNT on *r/m16*
F3 0F B8 /r	POPCNT *r32, r/m32*	RM	Valid	Valid	POPCNT on *r/m32*
F3 REX.W 0F B8 */r*	POPCNT *r64, r/m64*	RM	Valid	N.E.	POPCNT on *r/m64*

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

Description

This instruction calculates the number of bits set to 1 in the second operand (source) and returns the count in the first operand (a destination register).

Operation

```
Count = 0;
For (i=0; i < OperandSize; i++)
{       IF (SRC[ i] = 1) // i'th bit
        THEN Count++; FI;
}
DEST ← Count;
```

Flags Affected

OF, SF, ZF, AF, CF, PF are all cleared. ZF is set if SRC = 0, otherwise ZF is cleared.

Intel C/C++ Compiler Intrinsic Equivalent

POPCNT: int _mm_popcnt_u32(unsigned int a);

POPCNT: int64_t _mm_popcnt_u64(unsigned __int64 a);

Protected Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS or GS segments.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF (fault-code)	For a page fault.
#AC(0)	If an unaligned memory reference is made while the current privilege level is 3 and alignment checking is enabled.
#UD	If CPUID.01H:ECX.POPCNT [Bit 23] = 0.
	If LOCK prefix is used.

Real-Address Mode Exceptions

#GP(0)	If any part of the operand lies outside of the effective address space from 0 to 0FFFFH.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#UD	If CPUID.01H:ECX.POPCNT [Bit 23] = 0.
	If LOCK prefix is used.

Virtual 8086 Mode Exceptions

#GP(0)	If any part of the operand lies outside of the effective address space from 0 to 0FFFFH.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF (fault-code)	For a page fault.
#AC(0)	If an unaligned memory reference is made while alignment checking is enabled.
#UD	If CPUID.01H:ECX.POPCNT [Bit 23] = 0.
	If LOCK prefix is used.

Compatibility Mode Exceptions

Same exceptions as in Protected Mode.

64-Bit Mode Exceptions

#GP(0)	If the memory address is in a non-canonical form.
#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#PF (fault-code)	For a page fault.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If CPUID.01H:ECX.POPCNT [Bit 23] = 0.
	If LOCK prefix is used.

POPF/POPFD/POPFQ—Pop Stack into EFLAGS Register

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
9D	POPF	ZO	Valid	Valid	Pop top of stack into lower 16 bits of EFLAGS.
9D	POPFD	ZO	N.E.	Valid	Pop top of stack into EFLAGS.
9D	POPFQ	ZO	Valid	N.E.	Pop top of stack and zero-extend into RFLAGS.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
ZO	NA	NA	NA	NA

Description

Pops a doubleword (POPFD) from the top of the stack (if the current operand-size attribute is 32) and stores the value in the EFLAGS register, or pops a word from the top of the stack (if the operand-size attribute is 16) and stores it in the lower 16 bits of the EFLAGS register (that is, the FLAGS register). These instructions reverse the operation of the PUSHF/PUSHFD/PUSHFQ instructions.

The POPF (pop flags) and POPFD (pop flags double) mnemonics reference the same opcode. The POPF instruction is intended for use when the operand-size attribute is 16; the POPFD instruction is intended for use when the operand-size attribute is 32. Some assemblers may force the operand size to 16 for POPF and to 32 for POPFD. Others may treat the mnemonics as synonyms (POPF/POPFD) and use the setting of the operand-size attribute to determine the size of values to pop from the stack.

The effect of POPF/POPFD on the EFLAGS register changes, depending on the mode of operation. See Table 4-15 and the key below for details.

When operating in protected, compatibility, or 64-bit mode at privilege level 0 (or in real-address mode, the equivalent to privilege level 0), all non-reserved flags in the EFLAGS register except RF[1], VIP, VIF, and VM may be modified. VIP, VIF and VM remain unaffected.

When operating in protected, compatibility, or 64-bit mode with a privilege level greater than 0, but less than or equal to IOPL, all flags can be modified except the IOPL field and RF, IF, VIP, VIF, and VM; these remain unaffected. The AC and ID flags can only be modified if the operand-size attribute is 32. The interrupt flag (IF) is altered only when executing at a level at least as privileged as the IOPL. If a POPF/POPFD instruction is executed with insufficient privilege, an exception does not occur but privileged bits do not change.

When operating in virtual-8086 mode (EFLAGS.VM = 1) without the virtual-8086 mode extensions (CR4.VME = 0), the POPF/POPFD instructions can be used only if IOPL = 3; otherwise, a general-protection exception (#GP) occurs. If the virtual-8086 mode extensions are enabled (CR4.VME = 1), POPF (but not POPFD) can be executed in virtual-8086 mode with IOPL < 3.

(The protected-mode virtual-interrupt feature — enabled by setting CR4.PVI — affects the CLI and STI instructions in the same manner as the virtual-8086 mode extensions. POPF, however, is not affected by CR4.PVI.)

In 64-bit mode, the mnemonic assigned is POPFQ (note that the 32-bit operand is not encodable). POPFQ pops 64 bits from the stack. Reserved bits of RFLAGS (including the upper 32 bits of RFLAGS) are not affected.

See Chapter 3 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*, for more information about the EFLAGS registers.

1. RF is always zero after the execution of POPF. This is because POPF, like all instructions, clears RF as it begins to execute.

Table 4-15. Effect of POPF/POPFD on the EFLAGS Register

Mode	Operand Size	CPL	IOPL	Flags																Notes
				21	20	19	18	17	16	14	13:12	11	10	9	8	7	6	4	2	0	
				ID	VIP	VIF	AC	VM	RF	NT	IOPL	OF	DF	IF	TF	SF	ZF	AF	PF	CF	
Real-Address Mode (CR0.PE = 0)	16	0	0-3	N	N	N	N	N	0	S	S	S	S	S	S	S	S	S	S	S	
	32	0	0-3	S	N	N	S	N	0	S	S	S	S	S	S	S	S	S	S	S	
Protected, Compatibility, and 64-Bit Modes (CR0.PE = 1 EFLAGS.VM = 0)	16	0	0-3	N	N	N	N	N	0	S	S	S	S	S	S	S	S	S	S	S	
	16	1-3	<CPL	N	N	N	N	N	0	S	N	S	S	N	S	S	S	S	S	S	
	16	1-3	≥CPL	N	N	N	N	N	0	S	N	S	S	S	S	S	S	S	S	S	
	32, 64	0	0-3	S	N	N	S	N	0	S	S	S	S	S	S	S	S	S	S	S	
	32, 64	1-3	<CPL	S	N	N	S	N	0	S	N	S	S	N	S	S	S	S	S	S	
	32, 64	1-3	≥CPL	S	N	N	S	N	0	S	N	S	S	S	S	S	S	S	S	S	
Virtual-8086 (CR0.PE = 1 EFLAGS.VM = 1 CR4.VME = 0)	16	3	0-2	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	1
	16	3	3	N	N	N	N	N	0	S	N	S	S	S	S	S	S	S	S	S	
	32	3	0-2	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	1
	32	3	3	S	N	N	S	N	0	S	N	S	S	S	S	S	S	S	S	S	
VME (CR0.PE = 1 EFLAGS.VM = 1 CR4.VME = 1)	16	3	0-2	N/X	N/X	SV/X	N/X	N/X	0/X	S/X	N/X	S/X	S/X	N/X	S/X	S/X	S/X	S/X	S/X	S/X	2,3
	16	3	3	N	N	N	N	N	0	S	N	S	S	S	S	S	S	S	S	S	
	32	3	0-2	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	1
	32	3	3	S	N	N	S	N	0	S	N	S	S	S	S	S	S	S	S	S	

NOTES:

1. #GP fault - no flag update
2. #GP fault with no flag update if VIP=1 in EFLAGS register and IF=1 in FLAGS value on stack
3. #GP fault with no flag update if TF=1 in FLAGS value on stack

Key	
S	Updated from stack
SV	Updated from IF (bit 9) in FLAGS value on stack
N	No change in value
X	No EFLAGS update
0	Value is cleared

Operation

```
IF EFLAGS.VM = 0 (* Not in Virtual-8086 Mode *)
    THEN IF CPL = 0 OR CR0.PE = 0
        THEN
            IF OperandSize = 32;
                THEN
                    EFLAGS ← Pop(); (* 32-bit pop *)
                    (* All non-reserved flags except RF, VIP, VIF, and VM can be modified;
                    VIP, VIF, VM, and all reserved bits are unaffected. RF is cleared. *)
                ELSE IF (Operandsize = 64)
                    RFLAGS = Pop(); (* 64-bit pop *)
                    (* All non-reserved flags except RF, VIP, VIF, and VM can be modified;
                    VIP, VIF, VM, and all reserved bits are unaffected. RF is cleared. *)
```

```
            ELSE (* OperandSize = 16 *)
                EFLAGS[15:0] ← Pop(); (* 16-bit pop *)
                (* All non-reserved flags can be modified. *)
        FI;
    ELSE (* CPL > 0 *)
        IF OperandSize = 32
            THEN
                IF CPL > IOPL
                    THEN
                        EFLAGS ← Pop(); (* 32-bit pop *)
                        (* All non-reserved bits except IF, IOPL, VIP, VIF, VM and RF can be modified;
                        IF, IOPL, VIP, VIF, VM and all reserved bits are unaffected; RF is cleared. *)
                    ELSE
                        EFLAGS ← Pop(); (* 32-bit pop *)
                        (* All non-reserved bits except IOPL, VIP, VIF, VM and RF can be modified;
                        IOPL, VIP, VIF, VM and all reserved bits are unaffected; RF is cleared. *)
                FI;
            ELSE IF (Operandsize = 64)
                IF CPL > IOPL
                    THEN
                        RFLAGS ← Pop(); (* 64-bit pop *)
                        (* All non-reserved bits except IF, IOPL, VIP, VIF, VM and RF can be modified;
                        IF, IOPL, VIP, VIF, VM and all reserved bits are unaffected; RF is cleared. *)
                    ELSE
                        RFLAGS ← Pop(); (* 64-bit pop *)
                        (* All non-reserved bits except IOPL, VIP, VIF, VM and RF can be modified;
                        IOPL, VIP, VIF, VM and all reserved bits are unaffected; RF is cleared. *)
                FI;
            ELSE (* OperandSize = 16 *)
                EFLAGS[15:0] ← Pop(); (* 16-bit pop *)
                (* All non-reserved bits except IOPL can be modified; IOPL and all
                reserved bits are unaffected. *)
        FI;
    FI;
ELSE (* In virtual-8086 mode *)
    IF IOPL = 3
        THEN
            IF OperandSize = 32
                THEN
                    EFLAGS ← Pop();
                    (* All non-reserved bits except IOPL, VIP, VIF, VM, and RF can be modified;
                    VIP, VIF, VM, IOPL and all reserved bits are unaffected. RF is cleared. *)
                ELSE
                    EFLAGS[15:0] ← Pop(); FI;
                    (* All non-reserved bits except IOPL can be modified; IOPL and all reserved bits are unaffected. *)
            FI;
        ELSE (* IOPL < 3 *)
            IF (Operandsize = 32) OR (CR4.VME = 0)
                THEN #GP(0); (* Trap to virtual-8086 monitor. *)
                ELSE (* Operandsize = 16 and CR4.VME = 1 *)
                    tempFLAGS ← Pop();
                    IF (EFLAGS.VIP = 1 AND tempFLAGS[9] = 1) OR tempFLAGS[8] = 1
                        THEN #GP(0);
                        ELSE
```

```
                                    EFLAGS.VIF ← tempFLAGS[9];
                                    EFLAGS[15:0] ← tempFLAGS;
                                    (* All non-reserved bits except IOPL and IF can be modified;
                                    IOPL, IF, and all reserved bits are unaffected. *)
                            FI;
                    FI;
            FI;
FI;
```

Flags Affected

All flags may be affected; see the Operation section for details.

Protected Mode Exceptions

#SS(0)	If the top of stack is not within the stack segment.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If an unaligned memory reference is made while CPL = 3 and alignment checking is enabled.
#UD	If the LOCK prefix is used.

Real-Address Mode Exceptions

#SS	If the top of stack is not within the stack segment.
#UD	If the LOCK prefix is used.

Virtual-8086 Mode Exceptions

#GP(0)	If IOPL < 3 and VME is not enabled. If IOPL < 3 and the 32-bit operand size is used. If IOPL < 3, EFLAGS.VIP = 1, and bit 9 (IF) is set in the FLAGS value on the stack. If IOPL < 3 and bit 8 (TF) is set in the FLAGS value on the stack. If an attempt is made to execute the POPF/POPFD instruction with an operand-size override prefix.
#SS(0)	If the top of stack is not within the stack segment.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If an unaligned memory reference is made while alignment checking is enabled.
#UD	If the LOCK prefix is used.

Compatibility Mode Exceptions

Same as for protected mode exceptions.

64-Bit Mode Exceptions

#GP(0)	If the memory address is in a non-canonical form.
#SS(0)	If the stack address is in a non-canonical form.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If the LOCK prefix is used.

POR—Bitwise Logical OR

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F EB /r[1] POR *mm, mm/m64*	A	V/V	MMX	Bitwise OR of *mm/m64* and *mm*.
66 0F EB /r POR *xmm1, xmm2/m128*	A	V/V	SSE2	Bitwise OR of *xmm2/m128* and *xmm1*.
VEX.NDS.128.66.0F.WIG EB /r VPOR *xmm1, xmm2, xmm3/m128*	B	V/V	AVX	Bitwise OR of *xmm2/m128* and *xmm3*.
VEX.NDS.256.66.0F.WIG EB /r VPOR *ymm1, ymm2, ymm3/m256*	B	V/V	AVX2	Bitwise OR of *ymm2/m256* and *ymm3*.
EVEX.NDS.128.66.0F.W0 EB /r VPORD xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst	C	V/V	AVX512VL AVX512F	Bitwise OR of packed doubleword integers in xmm2 and xmm3/m128/m32bcst using writemask k1.
EVEX.NDS.256.66.0F.W0 EB /r VPORD ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst	C	V/V	AVX512VL AVX512F	Bitwise OR of packed doubleword integers in ymm2 and ymm3/m256/m32bcst using writemask k1.
EVEX.NDS.512.66.0F.W0 EB /r VPORD zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst	C	V/V	AVX512F	Bitwise OR of packed doubleword integers in zmm2 and zmm3/m512/m32bcst using writemask k1.
EVEX.NDS.128.66.0F.W1 EB /r VPORQ xmm1 {k1}{z}, xmm2, xmm3/m128/m64bcst	C	V/V	AVX512VL AVX512F	Bitwise OR of packed quadword integers in xmm2 and xmm3/m128/m64bcst using writemask k1.
EVEX.NDS.256.66.0F.W1 EB /r VPORQ ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst	C	V/V	AVX512VL AVX512F	Bitwise OR of packed quadword integers in ymm2 and ymm3/m256/m64bcst using writemask k1.
EVEX.NDS.512.66.0F.W1 EB /r VPORQ zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst	C	V/V	AVX512F	Bitwise OR of packed quadword integers in zmm2 and zmm3/m512/m64bcst using writemask k1.

NOTES:

1. See note in Section 2.4, "AVX and SSE Instruction Exception Specification" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A* and Section 22.25.3, "Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	NA	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
B	NA	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA
C	Full Vector	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA

Description

Performs a bitwise logical OR operation on the source operand (second operand) and the destination operand (first operand) and stores the result in the destination operand. Each bit of the result is set to 1 if either or both of the corresponding bits of the first and second operands are 1; otherwise, it is set to 0.

In 64-bit mode and not encoded with VEX/EVEX, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

Legacy SSE version: The source operand can be an MMX technology register or a 64-bit memory location. The destination operand is an MMX technology register.

128-bit Legacy SSE version: The second source operand is an XMM register or a 128-bit memory location. The first source and destination operands can be XMM registers. Bits (MAXVL-1:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: The second source operand is an XMM register or a 128-bit memory location. The first source and destination operands can be XMM registers. Bits (MAXVL-1:128) of the destination YMM register are zeroed.

VEX.256 encoded version: The second source operand is an YMM register or a 256-bit memory location. The first source and destination operands can be YMM registers.

EVEX encoded version: The first source operand is a ZMM/YMM/XMM register. The second source operand can be a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 32/64-bit memory location. The destination operand is a ZMM/YMM/XMM register conditionally updated with writemask k1 at 32/64-bit granularity.

Operation

POR (64-bit operand)

```
DEST ← DEST OR SRC
```

POR (128-bit Legacy SSE version)

```
DEST ← DEST OR SRC
DEST[MAXVL-1:128] (Unmodified)
```

VPOR (VEX.128 encoded version)

```
DEST ← SRC1 OR SRC2
DEST[MAXVL-1:128] ← 0
```

VPOR (VEX.256 encoded version)

```
DEST ← SRC1 OR SRC2
DEST[MAXVL-1:256] ← 0
```

VPORD (EVEX encoded versions)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j ← 0 TO KL-1
    i ← j * 32
    IF k1[j] OR *no writemask* THEN
            IF (EVEX.b = 1) AND (SRC2 *is memory*)
                THEN DEST[i+31:i] ← SRC1[i+31:i] BITWISE OR SRC2[31:0]
                ELSE DEST[i+31:i] ← SRC1[i+31:i] BITWISE OR SRC2[i+31:i]
            FI;
        ELSE
            IF *merging-masking*                    ; merging-masking
                *DEST[i+31:i] remains unchanged*
                ELSE                                ; zeroing-masking
                    DEST[i+31:i] ← 0
            FI;
    FI;
ENDFOR;
DEST[MAXVL-1:VL] ← 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VPORD __m512i _mm512_or_epi32(__m512i a, __m512i b);
VPORD __m512i _mm512_mask_or_epi32(__m512i s, __mmask16 k, __m512i a, __m512i b);
VPORD __m512i _mm512_maskz_or_epi32(__mmask16 k, __m512i a, __m512i b);
VPORD __m256i _mm256_or_epi32(__m256i a, __m256i b);
VPORD __m256i _mm256_mask_or_epi32(__m256i s, __mmask8 k, __m256i a, __m256i b,);
VPORD __m256i _mm256_maskz_or_epi32(__mmask8 k, __m256i a, __m256i b);
VPORD __m128i _mm_or_epi32(__m128i a, __m128i b);
VPORD __m128i _mm_mask_or_epi32(__m128i s, __mmask8 k, __m128i a, __m128i b);
VPORD __m128i _mm_maskz_or_epi32(__mmask8 k, __m128i a, __m128i b);
VPORQ __m512i _mm512_or_epi64(__m512i a, __m512i b);
VPORQ __m512i _mm512_mask_or_epi64(__m512i s, __mmask8 k, __m512i a, __m512i b);
VPORQ __m512i _mm512_maskz_or_epi64(__mmask8 k, __m512i a, __m512i b);
VPORQ __m256i _mm256_or_epi64(__m256i a, int imm);
VPORQ __m256i _mm256_mask_or_epi64(__m256i s, __mmask8 k, __m256i a, __m256i b);
VPORQ __m256i _mm256_maskz_or_epi64(__mmask8 k, __m256i a, __m256i b);
VPORQ __m128i _mm_or_epi64(__m128i a, __m128i b);
VPORQ __m128i _mm_mask_or_epi64(__m128i s, __mmask8 k, __m128i a, __m128i b);
VPORQ __m128i _mm_maskz_or_epi64(__mmask8 k, __m128i a, __m128i b);
POR __m64 _mm_or_si64(__m64 m1, __m64 m2)
(V)POR: __m128i _mm_or_si128(__m128i m1, __m128i m2)
VPOR: __m256i _mm256_or_si256 (__m256i a, __m256i b)

Flags Affected

None.

SIMD Floating-Point Exceptions

None.

Other Exceptions

Non-EVEX-encoded instruction, see Exceptions Type 4.
EVEX-encoded instruction, see Exceptions Type E4.

PREFETCH*h*—Prefetch Data Into Caches

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
0F 18 /1	PREFETCHT0 *m8*	M	Valid	Valid	Move data from *m8* closer to the processor using T0 hint.
0F 18 /2	PREFETCHT1 *m8*	M	Valid	Valid	Move data from *m8* closer to the processor using T1 hint.
0F 18 /3	PREFETCHT2 *m8*	M	Valid	Valid	Move data from *m8* closer to the processor using T2 hint.
0F 18 /0	PREFETCHNTA *m8*	M	Valid	Valid	Move data from *m8* closer to the processor using NTA hint.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
M	ModRM:r/m (r)	NA	NA	NA

Description

Fetches the line of data from memory that contains the byte specified with the source operand to a location in the cache hierarchy specified by a locality hint:

- T0 (temporal data)—prefetch data into all levels of the cache hierarchy.
- T1 (temporal data with respect to first level cache misses)—prefetch data into level 2 cache and higher.
- T2 (temporal data with respect to second level cache misses)—prefetch data into level 3 cache and higher, or an implementation-specific choice.
- NTA (non-temporal data with respect to all cache levels)—prefetch data into non-temporal cache structure and into a location close to the processor, minimizing cache pollution.

The source operand is a byte memory location. (The locality hints are encoded into the machine level instruction using bits 3 through 5 of the ModR/M byte.)

If the line selected is already present in the cache hierarchy at a level closer to the processor, no data movement occurs. Prefetches from uncacheable or WC memory are ignored.

The PREFETCH*h* instruction is merely a hint and does not affect program behavior. If executed, this instruction moves data closer to the processor in anticipation of future use.

The implementation of prefetch locality hints is implementation-dependent, and can be overloaded or ignored by a processor implementation. The amount of data prefetched is also processor implementation-dependent. It will, however, be a minimum of 32 bytes. Additional details of the implementation-dependent locality hints are described in Section 7.4 of *Intel® 64 and IA-32 Architectures Optimization Reference Manual*.

It should be noted that processors are free to speculatively fetch and cache data from system memory regions that are assigned a memory-type that permits speculative reads (that is, the WB, WC, and WT memory types). A PREFETCH*h* instruction is considered a hint to this speculative behavior. Because this speculative fetching can occur at any time and is not tied to instruction execution, a PREFETCH*h* instruction is not ordered with respect to the fence instructions (MFENCE, SFENCE, and LFENCE) or locked memory references. A PREFETCH*h* instruction is also unordered with respect to CLFLUSH and CLFLUSHOPT instructions, other PREFETCH*h* instructions, or any other general instruction. It is ordered with respect to serializing instructions such as CPUID, WRMSR, OUT, and MOV CR.

This instruction's operation is the same in non-64-bit modes and 64-bit mode.

Operation

```
FETCH (m8);
```

Intel C/C++ Compiler Intrinsic Equivalent

void _mm_prefetch(char *p, int i)

The argument "*p" gives the address of the byte (and corresponding cache line) to be prefetched. The value "i" gives a constant (_MM_HINT_T0, _MM_HINT_T1, _MM_HINT_T2, or _MM_HINT_NTA) that specifies the type of prefetch operation to be performed.

Numeric Exceptions

None.

Exceptions (All Operating Modes)

#UD	If the LOCK prefix is used.

PREFETCHW—Prefetch Data into Caches in Anticipation of a Write

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
0F 0D /1 PREFETCHW m8	A	V/V	PRFCHW	Move data from m8 closer to the processor in anticipation of a write.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
M	ModRM:r/m (r)	NA	NA	NA

Description

Fetches the cache line of data from memory that contains the byte specified with the source operand to a location in the 1st or 2nd level cache and invalidates other cached instances of the line.

The source operand is a byte memory location. If the line selected is already present in the lowest level cache and is already in an exclusively owned state, no data movement occurs. Prefetches from non-writeback memory are ignored.

The PREFETCHW instruction is merely a hint and does not affect program behavior. If executed, this instruction moves data closer to the processor and invalidates other cached copies in anticipation of the line being written to in the future.

The characteristic of prefetch locality hints is implementation-dependent, and can be overloaded or ignored by a processor implementation. The amount of data prefetched is also processor implementation-dependent. It will, however, be a minimum of 32 bytes. Additional details of the implementation-dependent locality hints are described in Section 7.4 of *Intel® 64 and IA-32 Architectures Optimization Reference Manual*.

It should be noted that processors are free to speculatively fetch and cache data with exclusive ownership from system memory regions that permit such accesses (that is, the WB memory type). A PREFETCHW instruction is considered a hint to this speculative behavior. Because this speculative fetching can occur at any time and is not tied to instruction execution, a PREFETCHW instruction is not ordered with respect to the fence instructions (MFENCE, SFENCE, and LFENCE) or locked memory references. A PREFETCHW instruction is also unordered with respect to CLFLUSH and CLFLUSHOPT instructions, other PREFETCHW instructions, or any other general instruction

It is ordered with respect to serializing instructions such as CPUID, WRMSR, OUT, and MOV CR.

This instruction's operation is the same in non-64-bit modes and 64-bit mode.

Operation

```
FETCH_WITH_EXCLUSIVE_OWNERSHIP (m8);
```

Flags Affected

All flags are affected

C/C++ Compiler Intrinsic Equivalent

```
void _m_prefetchw( void * );
```

Protected Mode Exceptions

#UD	If the LOCK prefix is used.

Real-Address Mode Exceptions

#UD	If the LOCK prefix is used.

Virtual-8086 Mode Exceptions

#UD	If the LOCK prefix is used.

Compatibility Mode Exceptions

#UD	If the LOCK prefix is used.

64-Bit Mode Exceptions

#UD	If the LOCK prefix is used.

PREFETCHWT1—Prefetch Vector Data Into Caches with Intent to Write and T1 Hint

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
0F 0D /2 PREFETCHWT1 m8	M	V/V	PREFETCHWT1	Move data from m8 closer to the processor using T1 hint with intent to write.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
M	ModRM:r/m (r)	NA	NA	NA

Description

Fetches the line of data from memory that contains the byte specified with the source operand to a location in the cache hierarchy specified by an intent to write hint (so that data is brought into 'Exclusive' state via a request for ownership) and a locality hint:

• T1 (temporal data with respect to first level cache)—prefetch data into the second level cache.

The source operand is a byte memory location. (The locality hints are encoded into the machine level instruction using bits 3 through 5 of the ModR/M byte. Use of any ModR/M value other than the specified ones will lead to unpredictable behavior.)

If the line selected is already present in the cache hierarchy at a level closer to the processor, no data movement occurs. Prefetches from uncacheable or WC memory are ignored.

The PREFETCHh instruction is merely a hint and does not affect program behavior. If executed, this instruction moves data closer to the processor in anticipation of future use.

The implementation of prefetch locality hints is implementation-dependent, and can be overloaded or ignored by a processor implementation. The amount of data prefetched is also processor implementation-dependent. It will, however, be a minimum of 32 bytes.

It should be noted that processors are free to speculatively fetch and cache data from system memory regions that are assigned a memory-type that permits speculative reads (that is, the WB, WC, and WT memory types). A PREFETCHh instruction is considered a hint to this speculative behavior. Because this speculative fetching can occur at any time and is not tied to instruction execution, a PREFETCHh instruction is not ordered with respect to the fence instructions (MFENCE, SFENCE, and LFENCE) or locked memory references. A PREFETCHh instruction is also unordered with respect to CLFLUSH and CLFLUSHOPT instructions, other PREFETCHh instructions, or any other general instruction. It is ordered with respect to serializing instructions such as CPUID, WRMSR, OUT, and MOV CR.

This instruction's operation is the same in non-64-bit modes and 64-bit mode.

Operation

PREFETCH(mem, Level, State) Prefetches a byte memory location pointed by 'mem' into the cache level specified by 'Level'; a request for exclusive/ownership is done if 'State' is 1. Note that the memory location ignore cache line splits. This operation is considered a hint for the processor and may be skipped depending on implementation.

```
Prefetch (m8, Level = 1, EXCLUSIVE=1);
```

Flags Affected

All flags are affected

C/C++ Compiler Intrinsic Equivalent

void _mm_prefetch(char const *, int hint= _MM_HINT_ET1);

Protected Mode Exceptions

#UD	If the LOCK prefix is used.

Real-Address Mode Exceptions

#UD	If the LOCK prefix is used.

Virtual-8086 Mode Exceptions

#UD	If the LOCK prefix is used.

Compatibility Mode Exceptions

#UD	If the LOCK prefix is used.

64-Bit Mode Exceptions

#UD	If the LOCK prefix is used.

PSADBW—Compute Sum of Absolute Differences

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F F6 /r[1] PSADBW *mm1, mm2/m64*	A	V/V	SSE	Computes the absolute differences of the packed unsigned byte integers from *mm2 /m64* and *mm1*; differences are then summed to produce an unsigned word integer result.
66 0F F6 /r PSADBW *xmm1, xmm2/m128*	A	V/V	SSE2	Computes the absolute differences of the packed unsigned byte integers from *xmm2 /m128* and *xmm1*; the 8 low differences and 8 high differences are then summed separately to produce two unsigned word integer results.
VEX.NDS.128.66.0F.WIG F6 /r VPSADBW *xmm1, xmm2, xmm3/m128*	B	V/V	AVX	Computes the absolute differences of the packed unsigned byte integers from *xmm3 /m128* and *xmm2*; the 8 low differences and 8 high differences are then summed separately to produce two unsigned word integer results.
VEX.NDS.256.66.0F.WIG F6 /r VPSADBW *ymm1, ymm2, ymm3/m256*	B	V/V	AVX2	Computes the absolute differences of the packed unsigned byte integers from *ymm3 /m256* and *ymm2*; then each consecutive 8 differences are summed separately to produce four unsigned word integer results.
EVEX.NDS.128.66.0F.WIG F6 /r VPSADBW xmm1, xmm2, xmm3/m128	C	V/V	AVX512VL AVX512BW	Computes the absolute differences of the packed unsigned byte integers from xmm3 /m128 and xmm2; then each consecutive 8 differences are summed separately to produce four unsigned word integer results.
EVEX.NDS.256.66.0F.WIG F6 /r VPSADBW ymm1, ymm2, ymm3/m256	C	V/V	AVX512VL AVX512BW	Computes the absolute differences of the packed unsigned byte integers from ymm3 /m256 and ymm2; then each consecutive 8 differences are summed separately to produce four unsigned word integer results.
EVEX.NDS.512.66.0F.WIG F6 /r VPSADBW zmm1, zmm2, zmm3/m512	C	V/V	AVX512BW	Computes the absolute differences of the packed unsigned byte integers from zmm3 /m512 and zmm2; then each consecutive 8 differences are summed separately to produce four unsigned word integer results.

NOTES:

1. See note in Section 2.4, "AVX and SSE Instruction Exception Specification" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A* and Section 22.25.3, "Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	NA	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
B	NA	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA
C	Full Vector Mem	ModRM:reg (w)	EVEX.vvvv	ModRM:r/m (r)	NA

Description

Computes the absolute value of the difference of 8 unsigned byte integers from the source operand (second operand) and from the destination operand (first operand). These 8 differences are then summed to produce an unsigned word integer result that is stored in the destination operand. Figure 4-14 shows the operation of the PSADBW instruction when using 64-bit operands.

When operating on 64-bit operands, the word integer result is stored in the low word of the destination operand, and the remaining bytes in the destination operand are cleared to all 0s.

When operating on 128-bit operands, two packed results are computed. Here, the 8 low-order bytes of the source and destination operands are operated on to produce a word result that is stored in the low word of the destination operand, and the 8 high-order bytes are operated on to produce a word result that is stored in bits 64 through 79 of the destination operand. The remaining bytes of the destination operand are cleared.

For 256-bit version, the third group of 8 differences are summed to produce an unsigned word in bits[143:128] of the destination register and the fourth group of 8 differences are summed to produce an unsigned word in bits[207:192] of the destination register. The remaining words of the destination are set to 0.

For 512-bit version, the fifth group result is stored in bits [271:256] of the destination. The result from the sixth group is stored in bits [335:320]. The results for the seventh and eighth group are stored respectively in bits [399:384] and bits [463:447], respectively. The remaining bits in the destination are set to 0.

In 64-bit mode and not encoded by VEX/EVEX prefix, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

Legacy SSE version: The source operand can be an MMX technology register or a 64-bit memory location. The destination operand is an MMX technology register.

128-bit Legacy SSE version: The first source operand and destination register are XMM registers. The second source operand is an XMM register or a 128-bit memory location. Bits (MAXVL-1:128) of the corresponding ZMM destination register remain unchanged.

VEX.128 and EVEX.128 encoded versions: The first source operand and destination register are XMM registers. The second source operand is an XMM register or a 128-bit memory location. Bits (MAXVL-1:128) of the corresponding ZMM register are zeroed.

VEX.256 and EVEX.256 encoded versions: The first source operand and destination register are YMM registers. The second source operand is an YMM register or a 256-bit memory location. Bits (MAXVL-1:256) of the corresponding ZMM register are zeroed.

EVEX.512 encoded version: The first source operand and destination register are ZMM registers. The second source operand is a ZMM register or a 512-bit memory location.

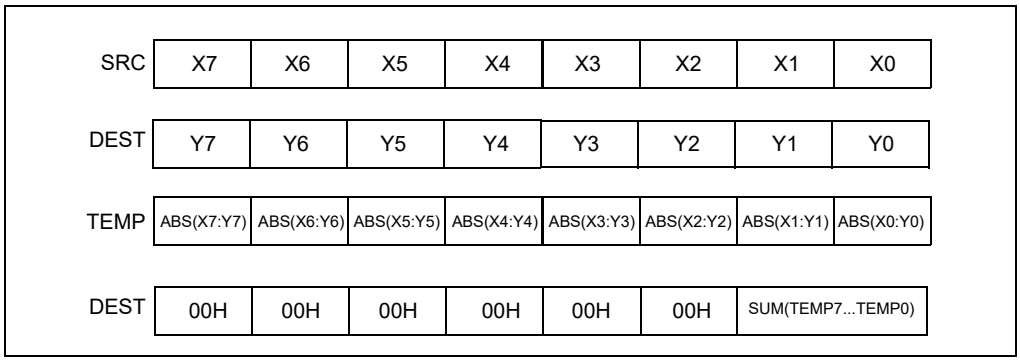


Figure 4-14. PSADBW Instruction Operation Using 64-bit Operands

Operation

VPSADBW (EVEX encoded versions)

```
VL = 128, 256, 512
TEMP0 ← ABS(SRC1[7:0] - SRC2[7:0])
(* Repeat operation for bytes 1 through 15 *)
TEMP15 ← ABS(SRC1[127:120] - SRC2[127:120])
DEST[15:0] ←SUM(TEMP0:TEMP7)
DEST[63:16] ← 000000000000H
DEST[79:64] ← SUM(TEMP8:TEMP15)
DEST[127:80] ← 00000000000H

IF VL >= 256
    (* Repeat operation for bytes 16 through 31*)
    TEMP31 ← ABS(SRC1[255:248] - SRC2[255:248])
    DEST[143:128] ←SUM(TEMP16:TEMP23)
    DEST[191:144] ← 000000000000H
    DEST[207:192] ← SUM(TEMP24:TEMP31)
    DEST[223:208] ← 00000000000H
FI;
IF VL >= 512
(* Repeat operation for bytes 32 through 63*)
    TEMP63 ← ABS(SRC1[511:504] - SRC2[511:504])
    DEST[271:256] ←SUM(TEMP0:TEMP7)
    DEST[319:272] ← 000000000000H
    DEST[335:320] ← SUM(TEMP8:TEMP15)
    DEST[383:336] ← 00000000000H
    DEST[399:384] ←SUM(TEMP16:TEMP23)
    DEST[447:400] ← 000000000000H
    DEST[463:448] ← SUM(TEMP24:TEMP31)
    DEST[511:464] ← 00000000000H
FI;
DEST[MAXVL-1:VL] ← 0
```

VPSADBW (VEX.256 encoded version)

```
TEMP0 ← ABS(SRC1[7:0] - SRC2[7:0])
(* Repeat operation for bytes 2 through 30*)
TEMP31 ← ABS(SRC1[255:248] - SRC2[255:248])
DEST[15:0] ←SUM(TEMP0:TEMP7)
DEST[63:16] ← 000000000000H
DEST[79:64] ← SUM(TEMP8:TEMP15)
DEST[127:80] ← 00000000000H
DEST[143:128] ←SUM(TEMP16:TEMP23)
DEST[191:144] ← 000000000000H
DEST[207:192] ← SUM(TEMP24:TEMP31)
DEST[223:208] ← 00000000000H
DEST[MAXVL-1:256] ← 0
```

VPSADBW (VEX.128 encoded version)
TEMP0 ← ABS(SRC1[7:0] - SRC2[7:0])
(* Repeat operation for bytes 2 through 14 *)
TEMP15 ← ABS(SRC1[127:120] - SRC2[127:120])
DEST[15:0] ←SUM(TEMP0:TEMP7)
DEST[63:16] ← 000000000000H
DEST[79:64] ← SUM(TEMP8:TEMP15)
DEST[127:80] ← 00000000000H
DEST[MAXVL-1:128] ← 0

PSADBW (128-bit Legacy SSE version)
TEMP0 ← ABS(DEST[7:0] - SRC[7:0])
(* Repeat operation for bytes 2 through 14 *)
TEMP15 ← ABS(DEST[127:120] - SRC[127:120])
DEST[15:0] ←SUM(TEMP0:TEMP7)
DEST[63:16] ← 000000000000H
DEST[79:64] ← SUM(TEMP8:TEMP15)
DEST[127:80] ← 00000000000
DEST[MAXVL-1:128] (Unmodified)

PSADBW (64-bit operand)
TEMP0 ← ABS(DEST[7:0] - SRC[7:0])
(* Repeat operation for bytes 2 through 6 *)
TEMP7 ← ABS(DEST[63:56] - SRC[63:56])
DEST[15:0] ←SUM(TEMP0:TEMP7)
DEST[63:16] ← 000000000000H

Intel C/C++ Compiler Intrinsic Equivalent

VPSADBW __m512i _mm512_sad_epu8(__m512i a, __m512i b)
PSADBW:__m64 _mm_sad_pu8(__m64 a,__m64 b)
(V)PSADBW:__m128i _mm_sad_epu8(__m128i a, __m128i b)
VPSADBW:__m256i _mm256_sad_epu8(__m256i a, __m256i b)

Flags Affected

None.

SIMD Floating-Point Exceptions

None.

Other Exceptions

Non-EVEX-encoded instruction, see Exceptions Type 4.
EVEX-encoded instruction, see Exceptions Type E4NF.nb.

PSHUFB — Packed Shuffle Bytes

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F 38 00 /r[1] PSHUFB *mm1, mm2/m64*	A	V/V	SSSE3	Shuffle bytes in *mm1* according to contents of *mm2/m64*.
66 0F 38 00 /r PSHUFB *xmm1, xmm2/m128*	A	V/V	SSSE3	Shuffle bytes in *xmm1* according to contents of *xmm2/m128*.
VEX.NDS.128.66.0F38.WIG 00 /r VPSHUFB *xmm1, xmm2, xmm3/m128*	B	V/V	AVX	Shuffle bytes in *xmm2* according to contents of *xmm3/m128*.
VEX.NDS.256.66.0F38.WIG 00 /r VPSHUFB *ymm1, ymm2, ymm3/m256*	B	V/V	AVX2	Shuffle bytes in *ymm2* according to contents of *ymm3/m256*.
EVEX.NDS.128.66.0F38.WIG 00 /r VPSHUFB xmm1 {k1}{z}, xmm2, xmm3/m128	C	V/V	AVX512VL AVX512BW	Shuffle bytes in xmm2 according to contents of xmm3/m128 under write mask k1.
EVEX.NDS.256.66.0F38.WIG 00 /r VPSHUFB ymm1 {k1}{z}, ymm2, ymm3/m256	C	V/V	AVX512VL AVX512BW	Shuffle bytes in ymm2 according to contents of ymm3/m256 under write mask k1.
EVEX.NDS.512.66.0F38.WIG 00 /r VPSHUFB zmm1 {k1}{z}, zmm2, zmm3/m512	C	V/V	AVX512BW	Shuffle bytes in zmm2 according to contents of zmm3/m512 under write mask k1.

NOTES:

1. See note in Section 2.4, "AVX and SSE Instruction Exception Specification" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A* and Section 22.25.3, "Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	NA	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
B	NA	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA
C	Full Vector Mem	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA

Description

PSHUFB performs in-place shuffles of bytes in the destination operand (the first operand) according to the shuffle control mask in the source operand (the second operand). The instruction permutes the data in the destination operand, leaving the shuffle mask unaffected. If the most significant bit (bit[7]) of each byte of the shuffle control mask is set, then constant zero is written in the result byte. Each byte in the shuffle control mask forms an index to permute the corresponding byte in the destination operand. The value of each index is the least significant 4 bits (128-bit operation) or 3 bits (64-bit operation) of the shuffle control byte. When the source operand is a 128-bit memory operand, the operand must be aligned on a 16-byte boundary or a general-protection exception (#GP) will be generated.

In 64-bit mode and not encoded with VEX/EVEX, use the REX prefix to access XMM8-XMM15 registers.

Legacy SSE version 64-bit operand: Both operands can be MMX registers.

128-bit Legacy SSE version: The first source operand and the destination operand are the same. Bits (MAXVL-1:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: The destination operand is the first operand, the first source operand is the second operand, the second source operand is the third operand. Bits (MAXVL-1:128) of the destination YMM register are zeroed.

VEX.256 encoded version: Bits (255:128) of the destination YMM register stores the 16-byte shuffle result of the upper 16 bytes of the first source operand, using the upper 16-bytes of the second source operand as control mask.

The value of each index is for the high 128-bit lane is the least significant 4 bits of the respective shuffle control byte. The index value selects a source data element within each 128-bit lane.

EVEX encoded version: The second source operand is an ZMM/YMM/XMM register or an 512/256/128-bit memory location. The first source operand and destination operands are ZMM/YMM/XMM registers. The destination is conditionally updated with writemask k1.

EVEX and VEX encoded version: Four/two in-lane 128-bit shuffles.

Operation

PSHUFB (with 64 bit operands)

```
TEMP ← DEST
for i = 0 to 7 {
    if (SRC[(i * 8)+7] = 1 ) then
        DEST[(i*8)+7...(i*8)+0] ← 0;
    else
        index[2..0] ← SRC[(i*8)+2 .. (i*8)+0];
        DEST[(i*8)+7...(i*8)+0] ← TEMP[(index*8+7)..(index*8+0)];
    endif;
}
```

PSHUFB (with 128 bit operands)

```
TEMP ← DEST
for i = 0 to 15 {
    if (SRC[(i * 8)+7] = 1 ) then
        DEST[(i*8)+7..(i*8)+0] ← 0;
     else
        index[3..0] ← SRC[(i*8)+3 .. (i*8)+0];
        DEST[(i*8)+7..(i*8)+0] ← TEMP[(index*8+7)..(index*8+0)];
    endif
}
```

VPSHUFB (VEX.128 encoded version)

```
for i = 0 to 15 {
    if (SRC2[(i * 8)+7] = 1) then
        DEST[(i*8)+7..(i*8)+0] ← 0;
        else
        index[3..0] ← SRC2[(i*8)+3 .. (i*8)+0];
        DEST[(i*8)+7..(i*8)+0] ← SRC1[(index*8+7)..(index*8+0)];
    endif
}
DEST[MAXVL-1:128] ← 0
```

VPSHUFB (VEX.256 encoded version)

```
for i = 0 to 15 {
    if (SRC2[(i * 8)+7] == 1 ) then
        DEST[(i*8)+7..(i*8)+0] ← 0;
        else
        index[3..0] ← SRC2[(i*8)+3 .. (i*8)+0];
        DEST[(i*8)+7..(i*8)+0] ← SRC1[(index*8+7)..(index*8+0)];
    endif
    if (SRC2[128 + (i * 8)+7] == 1 ) then
        DEST[128 + (i*8)+7..(i*8)+0] ← 0;
        else
        index[3..0] ← SRC2[128 + (i*8)+3 .. (i*8)+0];
        DEST[128 + (i*8)+7..(i*8)+0] ← SRC1[128 + (index*8+7)..(index*8+0)];
```

```
    endif
}
```

VPSHUFB (EVEX encoded versions)

```
(KL, VL) = (16, 128), (32, 256), (64, 512)
jmask ← (KL-1) & ~0xF                          // 0x00, 0x10, 0x30 depending on the VL
FOR j = 0 TO KL-1                              // dest
    IF kl[ i ] or no_masking
        index ← src.byte[ j ];
        IF index & 0x80
            Dest.byte[ j ] ← 0;
        ELSE
            index ← (index & 0xF) + (j & jmask);   // 16-element in-lane lookup
            Dest.byte[ j ] ← src.byte[ index ];
    ELSE if zeroing
        Dest.byte[ j ] ← 0;
DEST[MAXVL-1:VL] ← 0;
```

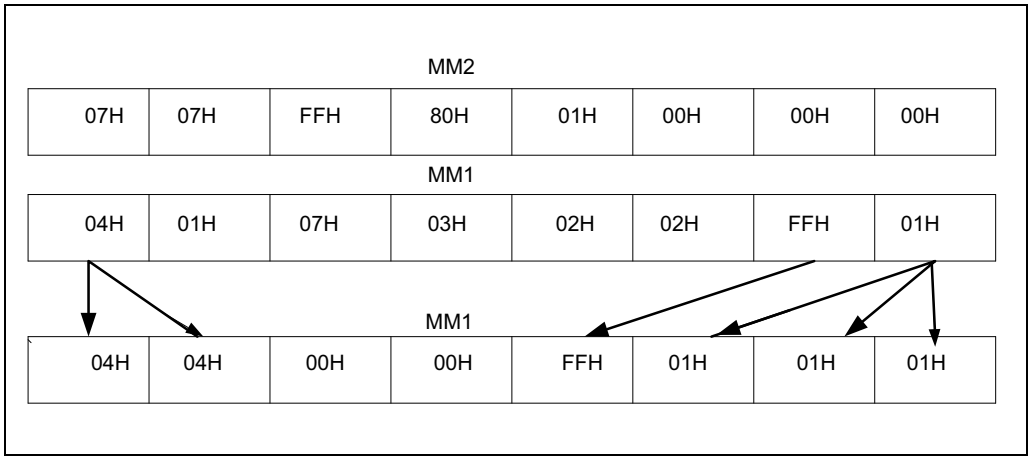


Figure 4-15. PSHUFB with 64-Bit Operands

Intel C/C++ Compiler Intrinsic Equivalent

VPSHUFB __m512i _mm512_shuffle_epi8(__m512i a, __m512i b);
VPSHUFB __m512i _mm512_mask_shuffle_epi8(__m512i s, __mmask64 k, __m512i a, __m512i b);
VPSHUFB __m512i _mm512_maskz_shuffle_epi8(__mmask64 k, __m512i a, __m512i b);
VPSHUFB __m256i _mm256_mask_shuffle_epi8(__m256i s, __mmask32 k, __m256i a, __m256i b);
VPSHUFB __m256i _mm256_maskz_shuffle_epi8(__mmask32 k, __m256i a, __m256i b);
VPSHUFB __m128i _mm_mask_shuffle_epi8(__m128i s, __mmask16 k, __m128i a, __m128i b);
VPSHUFB __m128i _mm_maskz_shuffle_epi8(__mmask16 k, __m128i a, __m128i b);
PSHUFB: __m64 _mm_shuffle_pi8 (__m64 a, __m64 b)
(V)PSHUFB: __m128i _mm_shuffle_epi8 (__m128i a, __m128i b)
VPSHUFB:__m256i _mm256_shuffle_epi8(__m256i a, __m256i b)

SIMD Floating-Point Exceptions

None.

Other Exceptions

Non-EVEX-encoded instruction, see Exceptions Type 4.
EVEX-encoded instruction, see Exceptions Type E4NF.nb.

PSHUFD—Shuffle Packed Doublewords

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 70 /r ib PSHUFD *xmm1, xmm2/m128, imm8*	A	V/V	SSE2	Shuffle the doublewords in *xmm2/m128* based on the encoding in *imm8* and store the result in *xmm1*.
VEX.128.66.0F.WIG 70 /r ib VPSHUFD *xmm1, xmm2/m128, imm8*	A	V/V	AVX	Shuffle the doublewords in *xmm2/m128* based on the encoding in *imm8* and store the result in *xmm1*.
VEX.256.66.0F.WIG 70 /r ib VPSHUFD *ymm1, ymm2/m256, imm8*	A	V/V	AVX2	Shuffle the doublewords in *ymm2/m256* based on the encoding in *imm8* and store the result in *ymm1*.
EVEX.128.66.0F.W0 70 /r ib VPSHUFD xmm1 {k1}{z}, xmm2/m128/m32bcst, imm8	B	V/V	AVX512VL AVX512F	Shuffle the doublewords in xmm2/m128/m32bcst based on the encoding in imm8 and store the result in xmm1 using writemask k1.
EVEX.256.66.0F.W0 70 /r ib VPSHUFD ymm1 {k1}{z}, ymm2/m256/m32bcst, imm8	B	V/V	AVX512VL AVX512F	Shuffle the doublewords in ymm2/m256/m32bcst based on the encoding in imm8 and store the result in ymm1 using writemask k1.
EVEX.512.66.0F.W0 70 /r ib VPSHUFD zmm1 {k1}{z}, zmm2/m512/m32bcst, imm8	B	V/V	AVX512F	Shuffle the doublewords in zmm2/m512/m32bcst based on the encoding in imm8 and store the result in zmm1 using writemask k1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	NA	ModRM:reg (w)	ModRM:r/m (r)	imm8	NA
B	Full Vector	ModRM:reg (w)	ModRM:r/m (r)	Imm8	NA

Description

Copies doublewords from source operand (second operand) and inserts them in the destination operand (first operand) at the locations selected with the order operand (third operand). Figure 4-16 shows the operation of the 256-bit VPSHUFD instruction and the encoding of the order operand. Each 2-bit field in the order operand selects the contents of one doubleword location within a 128-bit lane and copy to the target element in the destination operand. For example, bits 0 and 1 of the order operand targets the first doubleword element in the low and high 128-bit lane of the destination operand for 256-bit VPSHUFD. The encoded value of bits 1:0 of the order operand (see the field encoding in Figure 4-16) determines which doubleword element (from the respective 128-bit lane) of the source operand will be copied to doubleword 0 of the destination operand.

For 128-bit operation, only the low 128-bit lane are operative. The source operand can be an XMM register or a 128-bit memory location. The destination operand is an XMM register. The order operand is an 8-bit immediate. Note that this instruction permits a doubleword in the source operand to be copied to more than one doubleword location in the destination operand.

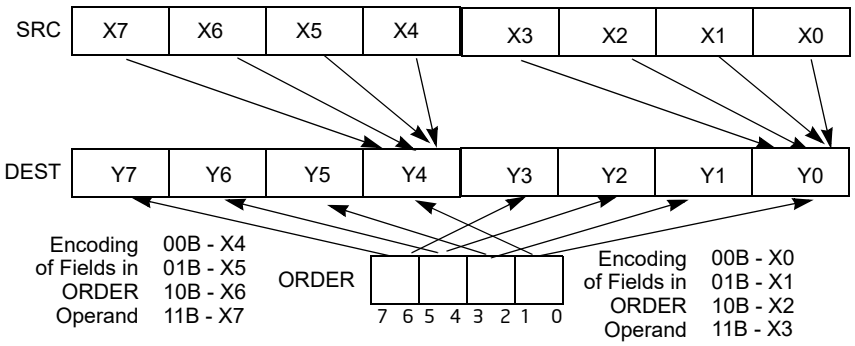


Figure 4-16. 256-bit VPSHUFD Instruction Operation

The source operand can be an XMM register or a 128-bit memory location. The destination operand is an XMM register. The order operand is an 8-bit immediate. Note that this instruction permits a doubleword in the source operand to be copied to more than one doubleword location in the destination operand.

In 64-bit mode and not encoded in VEX/EVEX, using REX.R permits this instruction to access XMM8-XMM15.

128-bit Legacy SSE version: Bits (MAXVL-1:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: The source operand can be an XMM register or a 128-bit memory location. The destination operand is an XMM register. Bits (MAXVL-1:128) of the corresponding ZMM register are zeroed.

VEX.256 encoded version: The source operand can be an YMM register or a 256-bit memory location. The destination operand is an YMM register. Bits (MAXVL-1:256) of the corresponding ZMM register are zeroed. Bits (255-1:128) of the destination stores the shuffled results of the upper 16 bytes of the source operand using the immediate byte as the order operand.

EVEX encoded version: The source operand can be an ZMM/YMM/XMM register, a 512/256/128-bit memory location, or a 512/256/128-bit vector broadcasted from a 32-bit memory location. The destination operand is a ZMM/YMM/XMM register updated according to the writemask.

Each 128-bit lane of the destination stores the shuffled results of the respective lane of the source operand using the immediate byte as the order operand.

Note: EVEX.vvvv and VEX.vvvv are reserved and must be 1111b otherwise instructions will #UD.

Operation

PSHUFD (128-bit Legacy SSE version)

```
DEST[31:0] ← (SRC >> (ORDER[1:0] * 32))[31:0];
DEST[63:32] ← (SRC >> (ORDER[3:2] * 32))[31:0];
DEST[95:64] ← (SRC >> (ORDER[5:4] * 32))[31:0];
DEST[127:96] ← (SRC >> (ORDER[7:6] * 32))[31:0];
DEST[MAXVL-1:128] (Unmodified)
```

VPSHUFD (VEX.128 encoded version)

```
DEST[31:0] ← (SRC >> (ORDER[1:0] * 32))[31:0];
DEST[63:32] ← (SRC >> (ORDER[3:2] * 32))[31:0];
DEST[95:64] ← (SRC >> (ORDER[5:4] * 32))[31:0];
DEST[127:96] ← (SRC >> (ORDER[7:6] * 32))[31:0];
DEST[MAXVL-1:128] ← 0
```

VPSHUFD (VEX.256 encoded version)

```
DEST[31:0] ← (SRC[127:0] >> (ORDER[1:0] * 32))[31:0];
DEST[63:32] ← (SRC[127:0] >> (ORDER[3:2] * 32))[31:0];
DEST[95:64] ← (SRC[127:0] >> (ORDER[5:4] * 32))[31:0];
DEST[127:96] ← (SRC[127:0] >> (ORDER[7:6] * 32))[31:0];
DEST[159:128] ← (SRC[255:128] >> (ORDER[1:0] * 32))[31:0];
DEST[191:160] ← (SRC[255:128] >> (ORDER[3:2] * 32))[31:0];
DEST[223:192] ← (SRC[255:128] >> (ORDER[5:4] * 32))[31:0];
DEST[255:224] ← (SRC[255:128] >> (ORDER[7:6] * 32))[31:0];
DEST[MAXVL-1:256] ← 0
```

VPSHUFD (EVEX encoded versions)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j ← 0 TO KL-1
    i ← j * 32
    IF (EVEX.b = 1) AND (SRC *is memory*)
        THEN TMP_SRC[i+31:i] ← SRC[31:0]
        ELSE TMP_SRC[i+31:i] ← SRC[i+31:i]
    FI;
ENDFOR;
IF VL >= 128
    TMP_DEST[31:0] ← (TMP_SRC[127:0] >> (ORDER[1:0] * 32))[31:0];
    TMP_DEST[63:32] ← (TMP_SRC[127:0] >> (ORDER[3:2] * 32))[31:0];
    TMP_DEST[95:64] ← (TMP_SRC[127:0] >> (ORDER[5:4] * 32))[31:0];
    TMP_DEST[127:96] ← (TMP_SRC[127:0] >> (ORDER[7:6] * 32))[31:0];
FI;
IF VL >= 256
    TMP_DEST[159:128] ← (TMP_SRC[255:128] >> (ORDER[1:0] * 32))[31:0];
    TMP_DEST[191:160] ← (TMP_SRC[255:128] >> (ORDER[3:2] * 32))[31:0];
    TMP_DEST[223:192] ← (TMP_SRC[255:128] >> (ORDER[5:4] * 32))[31:0];
    TMP_DEST[255:224] ← (TMP_SRC[255:128] >> (ORDER[7:6] * 32))[31:0];
FI;
IF VL >= 512
    TMP_DEST[287:256] ← (TMP_SRC[383:256] >> (ORDER[1:0] * 32))[31:0];
    TMP_DEST[319:288] ← (TMP_SRC[383:256] >> (ORDER[3:2] * 32))[31:0];
    TMP_DEST[351:320] ← (TMP_SRC[383:256] >> (ORDER[5:4] * 32))[31:0];
    TMP_DEST[383:352] ← (TMP_SRC[383:256] >> (ORDER[7:6] * 32))[31:0];
    TMP_DEST[415:384] ← (TMP_SRC[511:384] >> (ORDER[1:0] * 32))[31:0];
    TMP_DEST[447:416] ← (TMP_SRC[511:384] >> (ORDER[3:2] * 32))[31:0];
    TMP_DEST[479:448] ←(TMP_SRC[511:384] >> (ORDER[5:4] * 32))[31:0];
    TMP_DEST[511:480] ← (TMP_SRC[511:384] >> (ORDER[7:6] * 32))[31:0];
FI;
FOR j ← 0 TO KL-1
    i ← j * 32
    IF k1[j] OR *no writemask*
        THEN DEST[i+31:i] ← TMP_DEST[i+31:i]
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE *zeroing-masking*              ; zeroing-masking
                    DEST[i+31:i] ← 0
            FI
    FI;
ENDFOR
```

DEST[MAXVL-1:VL] ← 0

Intel C/C++ Compiler Intrinsic Equivalent

VPSHUFD __m512i _mm512_shuffle_epi32(__m512i a, int n);
VPSHUFD __m512i _mm512_mask_shuffle_epi32(__m512i s, __mmask16 k, __m512i a, int n);
VPSHUFD __m512i _mm512_maskz_shuffle_epi32(__mmask16 k, __m512i a, int n);
VPSHUFD __m256i _mm256_mask_shuffle_epi32(__m256i s, __mmask8 k, __m256i a, int n);
VPSHUFD __m256i _mm256_maskz_shuffle_epi32(__mmask8 k, __m256i a, int n);
VPSHUFD __m128i _mm_mask_shuffle_epi32(__m128i s, __mmask8 k, __m128i a, int n);
VPSHUFD __m128i _mm_maskz_shuffle_epi32(__mmask8 k, __m128i a, int n);
(V)PSHUFD:__m128i _mm_shuffle_epi32(__m128i a, int n)
VPSHUFD:__m256i _mm256_shuffle_epi32(__m256i a, const int n)

Flags Affected

None.

SIMD Floating-Point Exceptions

None.

Other Exceptions

Non-EVEX-encoded instruction, see Exceptions Type 4.
EVEX-encoded instruction, see Exceptions Type E4NF.

#UD	If VEX.vvvv ≠ 1111B or EVEX.vvvv ≠ 1111B.

PSHUFHW—Shuffle Packed High Words

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
F3 0F 70 /*r* ib PSHUFHW *xmm1, xmm2/m128, imm8*	A	V/V	SSE2	Shuffle the high words in *xmm2/m128* based on the encoding in *imm8* and store the result in *xmm1*.
VEX.128.F3.0F.WIG 70 /r ib VPSHUFHW *xmm1, xmm2/m128, imm8*	A	V/V	AVX	Shuffle the high words in *xmm2/m128* based on the encoding in *imm8* and store the result in *xmm1*.
VEX.256.F3.0F.WIG 70 /r ib VPSHUFHW *ymm1, ymm2/m256, imm8*	A	V/V	AVX2	Shuffle the high words in *ymm2/m256* based on the encoding in *imm8* and store the result in *ymm1*.
EVEX.128.F3.0F.WIG 70 /r ib VPSHUFHW xmm1 {k1}{z}, xmm2/m128, imm8	B	V/V	AVX512VL AVX512BW	Shuffle the high words in xmm2/m128 based on the encoding in imm8 and store the result in xmm1 under write mask k1.
EVEX.256.F3.0F.WIG 70 /r ib VPSHUFHW ymm1 {k1}{z}, ymm2/m256, imm8	B	V/V	AVX512VL AVX512BW	Shuffle the high words in ymm2/m256 based on the encoding in imm8 and store the result in ymm1 under write mask k1.
EVEX.512.F3.0F.WIG 70 /r ib VPSHUFHW zmm1 {k1}{z}, zmm2/m512, imm8	B	V/V	AVX512BW	Shuffle the high words in zmm2/m512 based on the encoding in imm8 and store the result in zmm1 under write mask k1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	NA	ModRM:reg (w)	ModRM:r/m (r)	imm8	NA
B	Full Vector Mem	ModRM:reg (w)	ModRM:r/m (r)	Imm8	NA

Description

Copies words from the high quadword of a 128-bit lane of the source operand and inserts them in the high quadword of the destination operand at word locations (of the respective lane) selected with the immediate operand. This 256-bit operation is similar to the in-lane operation used by the 256-bit VPSHUFD instruction, which is illustrated in Figure 4-16. For 128-bit operation, only the low 128-bit lane is operative. Each 2-bit field in the immediate operand selects the contents of one word location in the high quadword of the destination operand. The binary encodings of the immediate operand fields select words (0, 1, 2 or 3, 4) from the high quadword of the source operand to be copied to the destination operand. The low quadword of the source operand is copied to the low quadword of the destination operand, for each 128-bit lane.

Note that this instruction permits a word in the high quadword of the source operand to be copied to more than one word location in the high quadword of the destination operand.

In 64-bit mode and not encoded with VEX/EVEX, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

128-bit Legacy SSE version: The destination operand is an XMM register. The source operand can be an XMM register or a 128-bit memory location. Bits (MAXVL-1:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: The destination operand is an XMM register. The source operand can be an XMM register or a 128-bit memory location. Bits (MAXVL-1:128) of the destination YMM register are zeroed. VEX.vvvv is reserved and must be 1111b, VEX.L must be 0, otherwise the instruction will #UD.

VEX.256 encoded version: The destination operand is an YMM register. The source operand can be an YMM register or a 256-bit memory location.

EVEX encoded version: The destination operand is a ZMM/YMM/XMM registers. The source operand can be a ZMM/YMM/XMM register, a 512/256/128-bit memory location. The destination is updated according to the writemask.

Note: In VEX encoded versions, VEX.vvvv is reserved and must be 1111b otherwise instructions will #UD.

Operation

PSHUFHW (128-bit Legacy SSE version)

```
DEST[63:0] ← SRC[63:0]
DEST[79:64] ← (SRC >> (imm[1:0] *16))[79:64]
DEST[95:80] ← (SRC >> (imm[3:2] * 16))[79:64]
DEST[111:96] ← (SRC >> (imm[5:4] * 16))[79:64]
DEST[127:112] ← (SRC >> (imm[7:6] * 16))[79:64]
DEST[MAXVL-1:128] (Unmodified)
```

VPSHUFHW (VEX.128 encoded version)

```
DEST[63:0] ← SRC1[63:0]
DEST[79:64] ← (SRC1 >> (imm[1:0] *16))[79:64]
DEST[95:80] ← (SRC1 >> (imm[3:2] * 16))[79:64]
DEST[111:96] ← (SRC1 >> (imm[5:4] * 16))[79:64]
DEST[127:112] ← (SRC1 >> (imm[7:6] * 16))[79:64]
DEST[MAXVL-1:128] ← 0
```

VPSHUFHW (VEX.256 encoded version)

```
DEST[63:0] ← SRC1[63:0]
DEST[79:64] ← (SRC1 >> (imm[1:0] *16))[79:64]
DEST[95:80] ← (SRC1 >> (imm[3:2] * 16))[79:64]
DEST[111:96] ← (SRC1 >> (imm[5:4] * 16))[79:64]
DEST[127:112] ← (SRC1 >> (imm[7:6] * 16))[79:64]
DEST[191:128] ← SRC1[191:128]
DEST[207192] ← (SRC1 >> (imm[1:0] *16))[207:192]
DEST[223:208] ← (SRC1 >> (imm[3:2] * 16))[207:192]
DEST[239:224] ← (SRC1 >> (imm[5:4] * 16))[207:192]
DEST[255:240] ← (SRC1 >> (imm[7:6] * 16))[207:192]
DEST[MAXVL-1:256] ← 0
```

VPSHUFHW (EVEX encoded versions)

```
(KL, VL) = (8, 128), (16, 256), (32, 512)
IF VL >= 128
    TMP_DEST[63:0] ← SRC1[63:0]
    TMP_DEST[79:64] ← (SRC1 >> (imm[1:0] *16))[79:64]
    TMP_DEST[95:80] ← (SRC1 >> (imm[3:2] * 16))[79:64]
    TMP_DEST[111:96] ← (SRC1 >> (imm[5:4] * 16))[79:64]
    TMP_DEST[127:112] ← (SRC1 >> (imm[7:6] * 16))[79:64]
FI;
IF VL >= 256
    TMP_DEST[191:128] ← SRC1[191:128]
    TMP_DEST[207:192] ← (SRC1 >> (imm[1:0] *16))[207:192]
    TMP_DEST[223:208] ← (SRC1 >> (imm[3:2] * 16))[207:192]
    TMP_DEST[239:224] ← (SRC1 >> (imm[5:4] * 16))[207:192]
    TMP_DEST[255:240] ← (SRC1 >> (imm[7:6] * 16))[207:192]
FI;
IF VL >= 512
    TMP_DEST[319:256] ← SRC1[319:256]
    TMP_DEST[335:320] ← (SRC1 >> (imm[1:0] *16))[335:320]
```

```
    TMP_DEST[351:336] ← (SRC1 >> (imm[3:2] * 16))[335:320]
    TMP_DEST[367:352] ← (SRC1 >> (imm[5:4] * 16))[335:320]
    TMP_DEST[383:368] ← (SRC1 >> (imm[7:6] * 16))[335:320]
    TMP_DEST[447:384] ← SRC1[447:384]
    TMP_DEST[463:448] ← (SRC1 >> (imm[1:0] *16))[463:448]
    TMP_DEST[479:464] ← (SRC1 >> (imm[3:2] * 16))[463:448]
    TMP_DEST[495:480] ← (SRC1 >> (imm[5:4] * 16))[463:448]
    TMP_DEST[511:496] ← (SRC1 >> (imm[7:6] * 16))[463:448]
FI;

FOR j ← 0 TO KL-1
    i ← j * 16
    IF k1[j] OR *no writemask*
        THEN DEST[i+15:i] ← TMP_DEST[i+15:i];
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+15:i] remains unchanged*
                ELSE *zeroing-masking*              ; zeroing-masking
                    DEST[i+15:i] ← 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VPSHUFHW __m512i _mm512_shufflehi_epi16(__m512i a, int n);
VPSHUFHW __m512i _mm512_mask_shufflehi_epi16(__m512i s, __mmask16 k, __m512i a, int n);
VPSHUFHW __m512i _mm512_maskz_shufflehi_epi16(__mmask16 k, __m512i a, int n);
VPSHUFHW __m256i _mm256_mask_shufflehi_epi16(__m256i s, __mmask8 k, __m256i a, int n);
VPSHUFHW __m256i _mm256_maskz_shufflehi_epi16(__mmask8 k, __m256i a, int n);
VPSHUFHW __m128i _mm_mask_shufflehi_epi16(__m128i s, __mmask8 k, __m128i a, int n);
VPSHUFHW __m128i _mm_maskz_shufflehi_epi16(__mmask8 k, __m128i a, int n);
(V)PSHUFHW:__m128i _mm_shufflehi_epi16(__m128i a, int n)
VPSHUFHW:__m256i _mm256_shufflehi_epi16(__m256i a, const int n)

Flags Affected

None.

SIMD Floating-Point Exceptions

None.

Other Exceptions

Non-EVEX-encoded instruction, see Exceptions Type 4;
EVEX-encoded instruction, see Exceptions Type E4NF.nb

#UD	If VEX.vvvv != 1111B, or EVEX.vvvv != 1111B.

PSHUFLW—Shuffle Packed Low Words

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
F2 0F 70 /r ib PSHUFLW *xmm1, xmm2/m128, imm8*	A	V/V	SSE2	Shuffle the low words in *xmm2/m128* based on the encoding in *imm8* and store the result in *xmm1*.
VEX.128.F2.0F.WIG 70 /r ib VPSHUFLW *xmm1, xmm2/m128, imm8*	A	V/V	AVX	Shuffle the low words in *xmm2/m128* based on the encoding in *imm8* and store the result in *xmm1*.
VEX.256.F2.0F.WIG 70 /r ib VPSHUFLW *ymm1, ymm2/m256, imm8*	A	V/V	AVX2	Shuffle the low words in *ymm2/m256* based on the encoding in *imm8* and store the result in *ymm1*.
EVEX.128.F2.0F.WIG 70 /r ib VPSHUFLW xmm1 {k1}{z}, xmm2/m128, imm8	B	V/V	AVX512VL AVX512BW	Shuffle the low words in xmm2/m128 based on the encoding in imm8 and store the result in xmm1 under write mask k1.
EVEX.256.F2.0F.WIG 70 /r ib VPSHUFLW ymm1 {k1}{z}, ymm2/m256, imm8	B	V/V	AVX512VL AVX512BW	Shuffle the low words in ymm2/m256 based on the encoding in imm8 and store the result in ymm1 under write mask k1.
EVEX.512.F2.0F.WIG 70 /r ib VPSHUFLW zmm1 {k1}{z}, zmm2/m512, imm8	B	V/V	AVX512BW	Shuffle the low words in zmm2/m512 based on the encoding in imm8 and store the result in zmm1 under write mask k1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	NA	ModRM:reg (w)	ModRM:r/m (r)	imm8	NA
B	Full Vector Mem	ModRM:reg (w)	ModRM:r/m (r)	Imm8	NA

Description

Copies words from the low quadword of a 128-bit lane of the source operand and inserts them in the low quadword of the destination operand at word locations (of the respective lane) selected with the immediate operand. The 256-bit operation is similar to the in-lane operation used by the 256-bit VPSHUFD instruction, which is illustrated in Figure 4-16. For 128-bit operation, only the low 128-bit lane is operative. Each 2-bit field in the immediate operand selects the contents of one word location in the low quadword of the destination operand. The binary encodings of the immediate operand fields select words (0, 1, 2 or 3) from the low quadword of the source operand to be copied to the destination operand. The high quadword of the source operand is copied to the high quadword of the destination operand, for each 128-bit lane.

Note that this instruction permits a word in the low quadword of the source operand to be copied to more than one word location in the low quadword of the destination operand.

In 64-bit mode and not encoded with VEX/EVEX, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

128-bit Legacy SSE version: The destination operand is an XMM register. The source operand can be an XMM register or a 128-bit memory location. Bits (MAXVL-1:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: The destination operand is an XMM register. The source operand can be an XMM register or a 128-bit memory location. Bits (MAXVL-1:128) of the destination YMM register are zeroed.

VEX.256 encoded version: The destination operand is an YMM register. The source operand can be an YMM register or a 256-bit memory location.

EVEX encoded version: The destination operand is a ZMM/YMM/XMM registers. The source operand can be a ZMM/YMM/XMM register, a 512/256/128-bit memory location. The destination is updated according to the writemask.

Note: In VEX encoded versions, VEX.vvvv is reserved and must be 1111b otherwise instructions will #UD.

Operation

PSHUFLW (128-bit Legacy SSE version)

```
DEST[15:0] ← (SRC >> (imm[1:0] *16))[15:0]
DEST[31:16] ← (SRC >> (imm[3:2] * 16))[15:0]
DEST[47:32] ← (SRC >> (imm[5:4] * 16))[15:0]
DEST[63:48] ← (SRC >> (imm[7:6] * 16))[15:0]
DEST[127:64] ← SRC[127:64]
DEST[MAXVL-1:128] (Unmodified)
```

VPSHUFLW (VEX.128 encoded version)

```
DEST[15:0] ← (SRC1 >> (imm[1:0] *16))[15:0]
DEST[31:16] ← (SRC1 >> (imm[3:2] * 16))[15:0]
DEST[47:32] ← (SRC1 >> (imm[5:4] * 16))[15:0]
DEST[63:48] ← (SRC1 >> (imm[7:6] * 16))[15:0]
DEST[127:64] ← SRC[127:64]
DEST[MAXVL-1:128] ← 0
```

VPSHUFLW (VEX.256 encoded version)

```
DEST[15:0] ← (SRC1 >> (imm[1:0] *16))[15:0]
DEST[31:16] ← (SRC1 >> (imm[3:2] * 16))[15:0]
DEST[47:32] ← (SRC1 >> (imm[5:4] * 16))[15:0]
DEST[63:48] ← (SRC1 >> (imm[7:6] * 16))[15:0]
DEST[127:64] ← SRC1[127:64]
DEST[143:128] ← (SRC1 >> (imm[1:0] *16))[143:128]
DEST[159:144] ← (SRC1 >> (imm[3:2] * 16))[143:128]
DEST[175:160] ← (SRC1 >> (imm[5:4] * 16))[143:128]
DEST[191:176] ← (SRC1 >> (imm[7:6] * 16))[143:128]
DEST[255:192] ← SRC1[255:192]
DEST[MAXVL-1:256] ← 0
```

VPSHUFLW (EVEX.U1.512 encoded version)

```
(KL, VL) = (8, 128), (16, 256), (32, 512)
IF VL >= 128
    TMP_DEST[15:0] ← (SRC1 >> (imm[1:0] *16))[15:0]
    TMP_DEST[31:16] ← (SRC1 >> (imm[3:2] * 16))[15:0]
    TMP_DEST[47:32] ← (SRC1 >> (imm[5:4] * 16))[15:0]
    TMP_DEST[63:48] ← (SRC1 >> (imm[7:6] * 16))[15:0]
    TMP_DEST[127:64] ← SRC1[127:64]
FI;
IF VL >= 256
    TMP_DEST[143:128] ← (SRC1 >> (imm[1:0] *16))[143:128]
    TMP_DEST[159:144] ← (SRC1 >> (imm[3:2] * 16))[143:128]
    TMP_DEST[175:160] ← (SRC1 >> (imm[5:4] * 16))[143:128]
    TMP_DEST[191:176] ← (SRC1 >> (imm[7:6] * 16))[143:128]
    TMP_DEST[255:192] ← SRC1[255:192]
FI;
IF VL >= 512
    TMP_DEST[271:256] ← (SRC1 >> (imm[1:0] *16))[271:256]
    TMP_DEST[287:272] ← (SRC1 >> (imm[3:2] * 16))[271:256]
    TMP_DEST[303:288] ← (SRC1 >> (imm[5:4] * 16))[271:256]
    TMP_DEST[319:304] ← (SRC1 >> (imm[7:6] * 16))[271:256]
    TMP_DEST[383:320] ← SRC1[383:320]
```

```
    TMP_DEST[399:384] ← (SRC1 >> (imm[1:0] *16))[399:384]
    TMP_DEST[415:400] ← (SRC1 >> (imm[3:2] * 16))[399:384]
    TMP_DEST[431:416] ← (SRC1 >> (imm[5:4] * 16))[399:384]
    TMP_DEST[447:432] ← (SRC1 >> (imm[7:6] * 16))[399:384]
    TMP_DEST[511:448] ← SRC1[511:448]
FI;

FOR j ← 0 TO KL-1
    i ← j * 16
    IF k1[j] OR *no writemask*
        THEN DEST[i+15:i] ← TMP_DEST[i+15:i];
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+15:i] remains unchanged*
                ELSE *zeroing-masking*              ; zeroing-masking
                    DEST[i+15:i] ← 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VPSHUFLW __m512i _mm512_shufflelo_epi16(__m512i a, int n);
VPSHUFLW __m512i _mm512_mask_shufflelo_epi16(__m512i s, __mmask16 k, __m512i a, int n);
VPSHUFLW __m512i _mm512_maskz_shufflelo_epi16(__mmask16 k, __m512i a, int n);
VPSHUFLW __m256i _mm256_mask_shufflelo_epi16(__m256i s, __mmask8 k, __m256i a, int n);
VPSHUFLW __m256i _mm256_maskz_shufflelo_epi16(__mmask8 k, __m256i a, int n);
VPSHUFLW __m128i _mm_mask_shufflelo_epi16(__m128i s, __mmask8 k, __m128i a, int n);
VPSHUFLW __m128i _mm_maskz_shufflelo_epi16(__mmask8 k, __m128i a, int n);
(V)PSHUFLW:__m128i _mm_shufflelo_epi16(__m128i a, int n)
VPSHUFLW:__m256i _mm256_shufflelo_epi16(__m256i a, const int n)

Flags Affected

None.

SIMD Floating-Point Exceptions

None.

Other Exceptions

Non-EVEX-encoded instruction, see Exceptions Type 4;
EVEX-encoded instruction, see Exceptions Type E4NF.nb

#UD	If VEX.vvvv != 1111B, or EVEX.vvvv != 1111B.

PSHUFW—Shuffle Packed Words

Opcode/ Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
NP 0F 70 /*r* ib PSHUFW *mm1, mm2/m64, imm8*	RMI	Valid	Valid	Shuffle the words in *mm2/m64* based on the encoding in *imm8* and store the result in *mm1*.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RMI	ModRM:reg (w)	ModRM:r/m (r)	imm8	NA

Description

Copies words from the source operand (second operand) and inserts them in the destination operand (first operand) at word locations selected with the order operand (third operand). This operation is similar to the operation used by the PSHUFD instruction, which is illustrated in Figure 4-16. For the PSHUFW instruction, each 2-bit field in the order operand selects the contents of one word location in the destination operand. The encodings of the order operand fields select words from the source operand to be copied to the destination operand.

The source operand can be an MMX technology register or a 64-bit memory location. The destination operand is an MMX technology register. The order operand is an 8-bit immediate. Note that this instruction permits a word in the source operand to be copied to more than one word location in the destination operand.

In 64-bit mode, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

Operation

```
DEST[15:0] ← (SRC >> (ORDER[1:0] * 16))[15:0];
DEST[31:16] ← (SRC >> (ORDER[3:2] * 16))[15:0];
DEST[47:32] ← (SRC >> (ORDER[5:4] * 16))[15:0];
DEST[63:48] ← (SRC >> (ORDER[7:6] * 16))[15:0];
```

Intel C/C++ Compiler Intrinsic Equivalent

PSHUFW: __m64 _mm_shuffle_pi16(__m64 a, int n)

Flags Affected

None.

Numeric Exceptions

None.

Other Exceptions

See Table 22-7, "Exception Conditions for SIMD/MMX Instructions with Memory Reference," in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*.

PSIGNB/PSIGNW/PSIGND — Packed SIGN

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F 38 08 /r[1] PSIGNB *mm1, mm2/m64*	RM	V/V	SSSE3	Negate/zero/preserve packed byte integers in *mm1* depending on the corresponding sign in *mm2/m64*.
66 0F 38 08 /r PSIGNB *xmm1, xmm2/m128*	RM	V/V	SSSE3	Negate/zero/preserve packed byte integers in *xmm1* depending on the corresponding sign in *xmm2/m128*.
NP 0F 38 09 /r[1] PSIGNW *mm1, mm2/m64*	RM	V/V	SSSE3	Negate/zero/preserve packed word integers in *mm1* depending on the corresponding sign in *mm2/m128*.
66 0F 38 09 /r PSIGNW *xmm1, xmm2/m128*	RM	V/V	SSSE3	Negate/zero/preserve packed word integers in *xmm1* depending on the corresponding sign in *xmm2/m128*.
NP 0F 38 0A /r[1] PSIGND *mm1, mm2/m64*	RM	V/V	SSSE3	Negate/zero/preserve packed doubleword integers in *mm1* depending on the corresponding sign in *mm2/m128*.
66 0F 38 0A /r PSIGND *xmm1, xmm2/m128*	RM	V/V	SSSE3	Negate/zero/preserve packed doubleword integers in *xmm1* depending on the corresponding sign in *xmm2/m128*.
VEX.NDS.128.66.0F38.WIG 08 /r VPSIGNB *xmm1, xmm2, xmm3/m128*	RVM	V/V	AVX	Negate/zero/preserve packed byte integers in *xmm2* depending on the corresponding sign in *xmm3/m128*.
VEX.NDS.128.66.0F38.WIG 09 /r VPSIGNW *xmm1, xmm2, xmm3/m128*	RVM	V/V	AVX	Negate/zero/preserve packed word integers in *xmm2* depending on the corresponding sign in *xmm3/m128*.
VEX.NDS.128.66.0F38.WIG 0A /r VPSIGND *xmm1, xmm2, xmm3/m128*	RVM	V/V	AVX	Negate/zero/preserve packed doubleword integers in *xmm2* depending on the corresponding sign in *xmm3/m128*.
VEX.NDS.256.66.0F38.WIG 08 /r VPSIGNB *ymm1, ymm2, ymm3/m256*	RVM	V/V	AVX2	Negate packed byte integers in *ymm2* if the corresponding sign in *ymm3/m256* is less than zero.
VEX.NDS.256.66.0F38.WIG 09 /r VPSIGNW *ymm1, ymm2, ymm3/m256*	RVM	V/V	AVX2	Negate packed 16-bit integers in *ymm2* if the corresponding sign in *ymm3/m256* is less than zero.
VEX.NDS.256.66.0F38.WIG 0A /r VPSIGND *ymm1, ymm2, ymm3/m256*	RVM	V/V	AVX2	Negate packed doubleword integers in *ymm2* if the corresponding sign in *ymm3/m256* is less than zero.

NOTES:

1. See note in Section 2.4, "AVX and SSE Instruction Exception Specification" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A* and Section 22.25.3, "Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
RVM	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA

Description

(V)PSIGNB/(V)PSIGNW/(V)PSIGND negates each data element of the destination operand (the first operand) if the signed integer value of the corresponding data element in the source operand (the second operand) is less than zero. If the signed integer value of a data element in the source operand is positive, the corresponding data element in the destination operand is unchanged. If a data element in the source operand is zero, the corresponding data element in the destination operand is set to zero.

(V)PSIGNB operates on signed bytes. (V)PSIGNW operates on 16-bit signed words. (V)PSIGND operates on signed 32-bit integers. When the source operand is a 128bit memory operand, the operand must be aligned on a 16-byte boundary or a general-protection exception (#GP) will be generated.

Legacy SSE instructions: Both operands can be MMX registers. In 64-bit mode, use the REX prefix to access additional registers.

128-bit Legacy SSE version: The first source and destination operands are XMM registers. The second source operand is an XMM register or a 128-bit memory location. Bits (MAXVL-1:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: The first source and destination operands are XMM registers. The second source operand is an XMM register or a 128-bit memory location. Bits (MAXVL-1:128) of the destination YMM register are zeroed. VEX.L must be 0, otherwise instructions will #UD.

VEX.256 encoded version: The first source and destination operands are YMM registers. The second source operand is an YMM register or a 256-bit memory location.

Operation

PSIGNB (with 64 bit operands)

```
IF (SRC[7:0] < 0 )
    DEST[7:0] ← Neg(DEST[7:0])
ELSEIF (SRC[7:0] = 0 )
    DEST[7:0] ← 0
ELSEIF (SRC[7:0] > 0 )
    DEST[7:0] ← DEST[7:0]
Repeat operation for 2nd through 7th bytes

IF (SRC[63:56] < 0 )
    DEST[63:56] ← Neg(DEST[63:56])
ELSEIF (SRC[63:56] = 0 )
    DEST[63:56] ← 0
ELSEIF (SRC[63:56] > 0 )
    DEST[63:56] ← DEST[63:56]
```

PSIGNB (with 128 bit operands)

```
IF (SRC[7:0] < 0 )
    DEST[7:0] ← Neg(DEST[7:0])
ELSEIF (SRC[7:0] = 0 )
    DEST[7:0] ← 0
ELSEIF (SRC[7:0] > 0 )
    DEST[7:0] ← DEST[7:0]
Repeat operation for 2nd through 15th bytes
IF (SRC[127:120] < 0 )
    DEST[127:120] ← Neg(DEST[127:120])
ELSEIF (SRC[127:120] = 0 )
    DEST[127:120] ← 0
ELSEIF (SRC[127:120] > 0 )
    DEST[127:120] ← DEST[127:120]
```

VPSIGNB (VEX.128 encoded version)
DEST[127:0] ←BYTE_SIGN(SRC1, SRC2)
DEST[MAXVL-1:128] ← 0

VPSIGNB (VEX.256 encoded version)
DEST[255:0] ←BYTE_SIGN_256b(SRC1, SRC2)

PSIGNW (with 64 bit operands)
```
IF (SRC[15:0] < 0 )
    DEST[15:0] ← Neg(DEST[15:0])
ELSEIF (SRC[15:0] = 0 )
    DEST[15:0] ← 0
ELSEIF (SRC[15:0] > 0 )
    DEST[15:0] ← DEST[15:0]
Repeat operation for 2nd through 3rd words
IF (SRC[63:48] < 0 )
    DEST[63:48] ← Neg(DEST[63:48])
ELSEIF (SRC[63:48] = 0 )
    DEST[63:48] ← 0
ELSEIF (SRC[63:48] > 0 )
    DEST[63:48] ← DEST[63:48]
```

PSIGNW (with 128 bit operands)
```
IF (SRC[15:0] < 0 )
    DEST[15:0] ← Neg(DEST[15:0])
ELSEIF (SRC[15:0] = 0 )
    DEST[15:0] ← 0
ELSEIF (SRC[15:0] > 0 )
    DEST[15:0] ← DEST[15:0]
Repeat operation for 2nd through 7th words
IF (SRC[127:112] < 0 )
    DEST[127:112] ← Neg(DEST[127:112])
ELSEIF (SRC[127:112] = 0 )
    DEST[127:112] ← 0
ELSEIF (SRC[127:112] > 0 )
    DEST[127:112] ← DEST[127:112]
```

VPSIGNW (VEX.128 encoded version)
DEST[127:0] ←WORD_SIGN(SRC1, SRC2)
DEST[MAXVL-1:128] ← 0

VPSIGNW (VEX.256 encoded version)
DEST[255:0] ←WORD_SIGN(SRC1, SRC2)

PSIGND (with 64 bit operands)
```
IF (SRC[31:0] < 0 )
    DEST[31:0] ← Neg(DEST[31:0])
ELSEIF (SRC[31:0] = 0 )
    DEST[31:0] ← 0
ELSEIF (SRC[31:0] > 0 )
    DEST[31:0] ← DEST[31:0]
IF (SRC[63:32] < 0 )
    DEST[63:32] ← Neg(DEST[63:32])
ELSEIF (SRC[63:32] = 0 )
    DEST[63:32] ← 0
```

```
    ELSEIF (SRC[63:32] > 0 )
        DEST[63:32] ← DEST[63:32]
```

PSIGND (with 128 bit operands)

```
    IF (SRC[31:0] < 0 )
        DEST[31:0] ← Neg(DEST[31:0])
    ELSEIF (SRC[31:0] = 0 )
        DEST[31:0] ← 0
    ELSEIF (SRC[31:0] > 0 )
        DEST[31:0] ← DEST[31:0]
    Repeat operation for 2nd through 3rd double words
    IF (SRC[127:96] < 0 )
        DEST[127:96] ← Neg(DEST[127:96])
    ELSEIF (SRC[127:96] = 0 )
        DEST[127:96] ← 0
    ELSEIF (SRC[127:96] > 0 )
        DEST[127:96] ← DEST[127:96]
```

VPSIGND (VEX.128 encoded version)

```
DEST[127:0] ←DWORD_SIGN(SRC1, SRC2)
DEST[MAXVL-1:128] ← 0
```

VPSIGND (VEX.256 encoded version)

```
DEST[255:0] ←DWORD_SIGN(SRC1, SRC2)
```

Intel C/C++ Compiler Intrinsic Equivalent

PSIGNB: __m64 _mm_sign_pi8 (__m64 a, __m64 b)

(V)PSIGNB: __m128i _mm_sign_epi8 (__m128i a, __m128i b)

VPSIGNB: __m256i _mm256_sign_epi8 (__m256i a, __m256i b)

PSIGNW: __m64 _mm_sign_pi16 (__m64 a, __m64 b)

(V)PSIGNW: __m128i _mm_sign_epi16 (__m128i a, __m128i b)

VPSIGNW: __m256i _mm256_sign_epi16 (__m256i a, __m256i b)

PSIGND: __m64 _mm_sign_pi32 (__m64 a, __m64 b)

(V)PSIGND: __m128i _mm_sign_epi32 (__m128i a, __m128i b)

VPSIGND: __m256i _mm256_sign_epi32 (__m256i a, __m256i b)

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Exceptions Type 4; additionally

#UD	If VEX.L = 1.

PSLLDQ—Shift Double Quadword Left Logical

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 73 /7 ib PSLLDQ *xmm1, imm8*	A	V/V	SSE2	Shift *xmm1* left by *imm8* bytes while shifting in 0s.
VEX.NDD.128.66.0F.WIG 73 /7 ib VPSLLDQ *xmm1, xmm2, imm8*	B	V/V	AVX	Shift *xmm2* left by *imm8* bytes while shifting in 0s and store result in *xmm1*.
VEX.NDD.256.66.0F.WIG 73 /7 ib VPSLLDQ *ymm1, ymm2, imm8*	B	V/V	AVX2	Shift *ymm2* left by *imm8* bytes while shifting in 0s and store result in *ymm1*.
EVEX.NDD.128.66.0F.WIG 73 /7 ib VPSLLDQ xmm1,xmm2/ m128, imm8	C	V/V	AVX512VL AVX512BW	Shift xmm2/m128 left by imm8 bytes while shifting in 0s and store result in xmm1.
EVEX.NDD.256.66.0F.WIG 73 /7 ib VPSLLDQ ymm1, ymm2/m256, imm8	C	V/V	AVX512VL AVX512BW	Shift ymm2/m256 left by imm8 bytes while shifting in 0s and store result in ymm1.
EVEX.NDD.512.66.0F.WIG 73 /7 ib VPSLLDQ zmm1, zmm2/m512, imm8	C	V/V	AVX512BW	Shift zmm2/m512 left by imm8 bytes while shifting in 0s and store result in zmm1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	NA	ModRM:r/m (r, w)	imm8	NA	NA
B	NA	VEX.vvvv (w)	ModRM:r/m (r)	imm8	NA
C	Full Vector Mem	EVEX.vvvv (w)	ModRM:r/m (R)	Imm8	NA

Description

Shifts the destination operand (first operand) to the left by the number of bytes specified in the count operand (second operand). The empty low-order bytes are cleared (set to all 0s). If the value specified by the count operand is greater than 15, the destination operand is set to all 0s. The count operand is an 8-bit immediate.

128-bit Legacy SSE version: The source and destination operands are the same. Bits (MAXVL-1:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: The source and destination operands are XMM registers. Bits (MAXVL-1:128) of the destination YMM register are zeroed.

VEX.256 encoded version: The source operand is YMM register. The destination operand is an YMM register. Bits (MAXVL-1:256) of the corresponding ZMM register are zeroed. The count operand applies to both the low and high 128-bit lanes.

EVEX encoded versions: The source operand is a ZMM/YMM/XMM register or a 512/256/128-bit memory location. The destination operand is a ZMM/YMM/XMM register. The count operand applies to each 128-bit lanes.

Operation

VPSLLDQ (EVEX.U1.512 encoded version)

```
TEMP ← COUNT
IF (TEMP > 15) THEN TEMP ← 16; FI
DEST[127:0] ← SRC[127:0] << (TEMP * 8)
DEST[255:128] ← SRC[255:128] << (TEMP * 8)
DEST[383:256] ← SRC[383:256] << (TEMP * 8)
DEST[511:384] ← SRC[511:384] << (TEMP * 8)
DEST[MAXVL-1:512] ← 0
```

VPSLLDQ (VEX.256 and EVEX.256 encoded version)
TEMP ← COUNT
IF (TEMP > 15) THEN TEMP ← 16; FI
DEST[127:0] ← SRC[127:0] << (TEMP * 8)
DEST[255:128] ← SRC[255:128] << (TEMP * 8)
DEST[MAXVL-1:256] ← 0

VPSLLDQ (VEX.128 and EVEX.128 encoded version)
TEMP ← COUNT
IF (TEMP > 15) THEN TEMP ← 16; FI
DEST ← SRC << (TEMP * 8)
DEST[MAXVL-1:128] ← 0

PSLLDQ(128-bit Legacy SSE version)
TEMP ← COUNT
IF (TEMP > 15) THEN TEMP ← 16; FI
DEST ← DEST << (TEMP * 8)
DEST[MAXVL-1:128] (Unmodified)

Intel C/C++ Compiler Intrinsic Equivalent

(V)PSLLDQ:__m128i _mm_slli_si128 (__m128i a, int imm)
VPSLLDQ:__m256i _mm256_slli_si256 (__m256i a, const int imm)
VPSLLDQ __m512i _mm512_bslli_epi128 (__m512i a, const int imm)

Flags Affected

None.

Numeric Exceptions

None.

Other Exceptions

Non-EVEX-encoded instruction, see Exceptions Type 7.
EVEX-encoded instruction, see Exceptions Type E4NF.nb.

PSLLW/PSLLD/PSLLQ—Shift Packed Data Left Logical

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F F1 /r[1] PSLLW *mm, mm/m64*	A	V/V	MMX	Shift words in *mm* left *mm/m64* while shifting in 0s.
66 0F F1 /r PSLLW *xmm1, xmm2/m128*	A	V/V	SSE2	Shift words in *xmm1* left by *xmm2/m128* while shifting in 0s.
NP 0F 71 /6 ib PSLLW *mm1, imm8*	B	V/V	MMX	Shift words in *mm* left by *imm8* while shifting in 0s.
66 0F 71 /6 ib PSLLW *xmm1, imm8*	B	V/V	SSE2	Shift words in *xmm1* left by *imm8* while shifting in 0s.
NP 0F F2 /r[1] PSLLD *mm, mm/m64*	A	V/V	MMX	Shift doublewords in *mm* left by *mm/m64* while shifting in 0s.
66 0F F2 /r PSLLD *xmm1, xmm2/m128*	A	V/V	SSE2	Shift doublewords in *xmm1* left by *xmm2/m128* while shifting in 0s.
NP 0F 72 /6 ib[1] PSLLD *mm, imm8*	B	V/V	MMX	Shift doublewords in *mm* left by *imm8* while shifting in 0s.
66 0F 72 /6 ib PSLLD *xmm1, imm8*	B	V/V	SSE2	Shift doublewords in *xmm1* left by *imm8* while shifting in 0s.
NP 0F F3 /r[1] PSLLQ *mm, mm/m64*	A	V/V	MMX	Shift quadword in *mm* left by *mm/m64* while shifting in 0s.
66 0F F3 /r PSLLQ *xmm1, xmm2/m128*	A	V/V	SSE2	Shift quadwords in *xmm1* left by *xmm2/m128* while shifting in 0s.
NP 0F 73 /6 ib[1] PSLLQ *mm, imm8*	B	V/V	MMX	Shift quadword in *mm* left by *imm8* while shifting in 0s.
66 0F 73 /6 ib PSLLQ *xmm1, imm8*	B	V/V	SSE2	Shift quadwords in *xmm1* left by *imm8* while shifting in 0s.
VEX.NDS.128.66.0F.WIG F1 /r VPSLLW *xmm1, xmm2, xmm3/m128*	C	V/V	AVX	Shift words in *xmm2* left by amount specified in *xmm3/m128* while shifting in 0s.
VEX.NDD.128.66.0F.WIG 71 /6 ib VPSLLW *xmm1, xmm2, imm8*	D	V/V	AVX	Shift words in *xmm2* left by *imm8* while shifting in 0s.
VEX.NDS.128.66.0F.WIG F2 /r VPSLLD *xmm1, xmm2, xmm3/m128*	C	V/V	AVX	Shift doublewords in *xmm2* left by amount specified in *xmm3/m128* while shifting in 0s.
VEX.NDD.128.66.0F.WIG 72 /6 ib VPSLLD *xmm1, xmm2, imm8*	D	V/V	AVX	Shift doublewords in *xmm2* left by *imm8* while shifting in 0s.
VEX.NDS.128.66.0F.WIG F3 /r VPSLLQ *xmm1, xmm2, xmm3/m128*	C	V/V	AVX	Shift quadwords in *xmm2* left by amount specified in *xmm3/m128* while shifting in 0s.
VEX.NDD.128.66.0F.WIG 73 /6 ib VPSLLQ *xmm1, xmm2, imm8*	D	V/V	AVX	Shift quadwords in *xmm2* left by *imm8* while shifting in 0s.
VEX.NDS.256.66.0F.WIG F1 /r VPSLLW *ymm1, ymm2, xmm3/m128*	C	V/V	AVX2	Shift words in *ymm2* left by amount specified in *xmm3/m128* while shifting in 0s.
VEX.NDD.256.66.0F.WIG 71 /6 ib VPSLLW *ymm1, ymm2, imm8*	D	V/V	AVX2	Shift words in *ymm2* left by *imm8* while shifting in 0s.

VEX.NDS.256.66.0F.WIG F2 /r VPSLLD *ymm1, ymm2, xmm3/m128*	C	V/V	AVX2	Shift doublewords in *ymm2* left by amount specified in *xmm3/m128* while shifting in 0s.
VEX.NDD.256.66.0F.WIG 72 /6 ib VPSLLD *ymm1, ymm2, imm8*	D	V/V	AVX2	Shift doublewords in *ymm2* left by *imm8* while shifting in 0s.
VEX.NDS.256.66.0F.WIG F3 /r VPSLLQ *ymm1, ymm2, xmm3/m128*	C	V/V	AVX2	Shift quadwords in *ymm2* left by amount specified in *xmm3/m128* while shifting in 0s.
VEX.NDD.256.66.0F.WIG 73 /6 ib VPSLLQ *ymm1, ymm2, imm8*	D	V/V	AVX2	Shift quadwords in *ymm2* left by *imm8* while shifting in 0s.
EVEX.NDS.128.66.0F.WIG F1 /r VPSLLW xmm1 {k1}{z}, xmm2, xmm3/m128	G	V/V	AVX512VL AVX512BW	Shift words in xmm2 left by amount specified in xmm3/m128 while shifting in 0s using writemask k1.
EVEX.NDS.256.66.0F.WIG F1 /r VPSLLW ymm1 {k1}{z}, ymm2, xmm3/m128	G	V/V	AVX512VL AVX512BW	Shift words in ymm2 left by amount specified in xmm3/m128 while shifting in 0s using writemask k1.
EVEX.NDS.512.66.0F.WIG F1 /r VPSLLW zmm1 {k1}{z}, zmm2, xmm3/m128	G	V/V	AVX512BW	Shift words in zmm2 left by amount specified in xmm3/m128 while shifting in 0s using writemask k1.
EVEX.NDD.128.66.0F.WIG 71 /6 ib VPSLLW xmm1 {k1}{z}, xmm2/m128, imm8	E	V/V	AVX512VL AVX512BW	Shift words in xmm2/m128 left by imm8 while shifting in 0s using writemask k1.
EVEX.NDD.256.66.0F.WIG 71 /6 ib VPSLLW ymm1 {k1}{z}, ymm2/m256, imm8	E	V/V	AVX512VL AVX512BW	Shift words in ymm2/m256 left by imm8 while shifting in 0s using writemask k1.
EVEX.NDD.512.66.0F.WIG 71 /6 ib VPSLLW zmm1 {k1}{z}, zmm2/m512, imm8	E	V/V	AVX512BW	Shift words in zmm2/m512 left by imm8 while shifting in 0 using writemask k1.
EVEX.NDS.128.66.0F.W0 F2 /r VPSLLD xmm1 {k1}{z}, xmm2, xmm3/m128	G	V/V	AVX512VL AVX512F	Shift doublewords in xmm2 left by amount specified in xmm3/m128 while shifting in 0s under writemask k1.
EVEX.NDS.256.66.0F.W0 F2 /r VPSLLD ymm1 {k1}{z}, ymm2, xmm3/m128	G	V/V	AVX512VL AVX512F	Shift doublewords in ymm2 left by amount specified in xmm3/m128 while shifting in 0s under writemask k1.
EVEX.NDS.512.66.0F.W0 F2 /r VPSLLD zmm1 {k1}{z}, zmm2, xmm3/m128	G	V/V	AVX512F	Shift doublewords in zmm2 left by amount specified in xmm3/m128 while shifting in 0s under writemask k1.
EVEX.NDD.128.66.0F.W0 72 /6 ib VPSLLD xmm1 {k1}{z}, xmm2/m128/m32bcst, imm8	F	V/V	AVX512VL AVX512F	Shift doublewords in xmm2/m128/m32bcst left by imm8 while shifting in 0s using writemask k1.
EVEX.NDD.256.66.0F.W0 72 /6 ib VPSLLD ymm1 {k1}{z}, ymm2/m256/m32bcst, imm8	F	V/V	AVX512VL AVX512F	Shift doublewords in ymm2/m256/m32bcst left by imm8 while shifting in 0s using writemask k1.
EVEX.NDD.512.66.0F.W0 72 /6 ib VPSLLD zmm1 {k1}{z}, zmm2/m512/m32bcst, imm8	F	V/V	AVX512F	Shift doublewords in zmm2/m512/m32bcst left by imm8 while shifting in 0s using writemask k1.
EVEX.NDS.128.66.0F.W1 F3 /r VPSLLQ xmm1 {k1}{z}, xmm2, xmm3/m128	G	V/V	AVX512VL AVX512F	Shift quadwords in xmm2 left by amount specified in xmm3/m128 while shifting in 0s using writemask k1.
EVEX.NDS.256.66.0F.W1 F3 /r VPSLLQ ymm1 {k1}{z}, ymm2, xmm3/m128	G	V/V	AVX512VL AVX512F	Shift quadwords in ymm2 left by amount specified in xmm3/m128 while shifting in 0s using writemask k1.
EVEX.NDS.512.66.0F.W1 F3 /r VPSLLQ zmm1 {k1}{z}, zmm2, xmm3/m128	G	V/V	AVX512F	Shift quadwords in zmm2 left by amount specified in xmm3/m128 while shifting in 0s using writemask k1.

EVEX.NDD.128.66.0F.W1 73 /6 ib VPSLLQ xmm1 {k1}{z}, xmm2/m128/m64bcst, imm8	F	V/V	AVX512VL AVX512F	Shift quadwords in xmm2/m128/m64bcst left by imm8 while shifting in 0s using writemask k1.
EVEX.NDD.256.66.0F.W1 73 /6 ib VPSLLQ ymm1 {k1}{z}, ymm2/m256/m64bcst, imm8	F	V/V	AVX512VL AVX512F	Shift quadwords in ymm2/m256/m64bcst left by imm8 while shifting in 0s using writemask k1.
EVEX.NDD.512.66.0F.W1 73 /6 ib VPSLLQ zmm1 {k1}{z}, zmm2/m512/m64bcst, imm8	F	V/V	AVX512F	Shift quadwords in zmm2/m512/m64bcst left by imm8 while shifting in 0s using writemask k1.

NOTES:

1. See note in Section 2.4, "AVX and SSE Instruction Exception Specification" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A* and Section 22.25.3, "Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	NA	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
B	NA	ModRM:r/m (r, w)	imm8	NA	NA
C	NA	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA
D	NA	VEX.vvvv (w)	ModRM:r/m (r)	imm8	NA
E	Full Vector Mem	EVEX.vvvv (w)	ModRM:r/m (R)	Imm8	NA
F	Full Vector	EVEX.vvvv (w)	ModRM:r/m (R)	Imm8	NA
G	Mem128	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA

Description

Shifts the bits in the individual data elements (words, doublewords, or quadword) in the destination operand (first operand) to the left by the number of bits specified in the count operand (second operand). As the bits in the data elements are shifted left, the empty low-order bits are cleared (set to 0). If the value specified by the count operand is greater than 15 (for words), 31 (for doublewords), or 63 (for a quadword), then the destination operand is set to all 0s. Figure 4-17 gives an example of shifting words in a 64-bit operand.

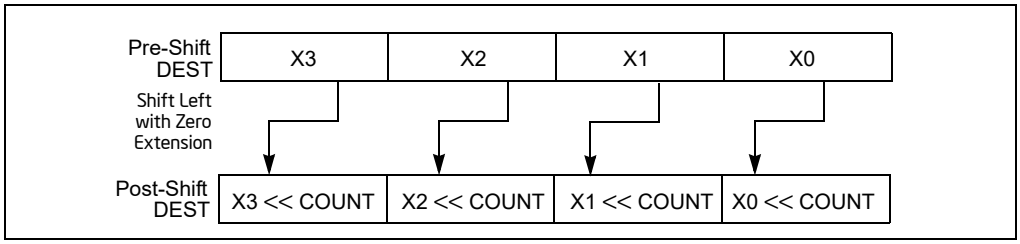


Figure 4-17. PSLLW, PSLLD, and PSLLQ Instruction Operation Using 64-bit Operand

The (V)PSLLW instruction shifts each of the words in the destination operand to the left by the number of bits specified in the count operand; the (V)PSLLD instruction shifts each of the doublewords in the destination operand; and the (V)PSLLQ instruction shifts the quadword (or quadwords) in the destination operand.

In 64-bit mode and not encoded with VEX/EVEX, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

Legacy SSE instructions 64-bit operand: The destination operand is an MMX technology register; the count operand can be either an MMX technology register or an 64-bit memory location.

128-bit Legacy SSE version: The destination and first source operands are XMM registers. Bits (MAXVL-1:128) of the corresponding YMM destination register remain unchanged. The count operand can be either an XMM register or a 128-bit memory location or an 8-bit immediate. If the count operand is a memory address, 128 bits are loaded but the upper 64 bits are ignored.

VEX.128 encoded version: The destination and first source operands are XMM registers. Bits (MAXVL-1:128) of the destination YMM register are zeroed. The count operand can be either an XMM register or a 128-bit memory location or an 8-bit immediate. If the count operand is a memory address, 128 bits are loaded but the upper 64 bits are ignored.

VEX.256 encoded version: The destination operand is a YMM register. The source operand is a YMM register or a memory location. The count operand can come either from an XMM register or a memory location or an 8-bit immediate. Bits (MAXVL-1:256) of the corresponding ZMM register are zeroed.

EVEX encoded versions: The destination operand is a ZMM register updated according to the writemask. The count operand is either an 8-bit immediate (the immediate count version) or an 8-bit value from an XMM register or a memory location (the variable count version). For the immediate count version, the source operand (the second operand) can be a ZMM register, a 512-bit memory location or a 512-bit vector broadcasted from a 32/64-bit memory location. For the variable count version, the first source operand (the second operand) is a ZMM register, the second source operand (the third operand, 8-bit variable count) can be an XMM register or a memory location.

Note: In VEX/EVEX encoded versions of shifts with an immediate count, vvvv of VEX/EVEX encode the destination register, and VEX.B/EVEX.B + ModRM.r/m encodes the source register.

Note: For shifts with an immediate count (VEX.128.66.0F 71-73 /6, or EVEX.128.66.0F 71-73 /6), VEX.vvvv/EVEX.vvvv encodes the destination register.

Operation

PSLLW (with 64-bit operand)

```
IF (COUNT > 15)
THEN
    DEST[64:0] ← 0000000000000000H;
ELSE
    DEST[15:0] ← ZeroExtend(DEST[15:0] << COUNT);
    (* Repeat shift operation for 2nd and 3rd words *)
    DEST[63:48] ← ZeroExtend(DEST[63:48] << COUNT);
FI;
```

PSLLD (with 64-bit operand)

```
IF (COUNT > 31)
THEN
    DEST[64:0] ← 0000000000000000H;
ELSE
    DEST[31:0] ← ZeroExtend(DEST[31:0] << COUNT);
    DEST[63:32] ← ZeroExtend(DEST[63:32] << COUNT);
FI;
```

PSLLQ (with 64-bit operand)

```
IF (COUNT > 63)
THEN
    DEST[64:0] ← 0000000000000000H;
ELSE
    DEST ← ZeroExtend(DEST << COUNT);
FI;
```

```
LOGICAL_LEFT_SHIFT_WORDS(SRC, COUNT_SRC)
COUNT ←COUNT_SRC[63:0];
IF (COUNT > 15)
THEN
```

```
    DEST[127:0] ←00000000000000000000000000000000H
ELSE
    DEST[15:0] ←ZeroExtend(SRC[15:0] << COUNT);
    (* Repeat shift operation for 2nd through 7th words *)
    DEST[127:112] ←ZeroExtend(SRC[127:112] << COUNT);
FI;

LOGICAL_LEFT_SHIFT_DWORDS1(SRC, COUNT_SRC)
COUNT ← COUNT_SRC[63:0];
IF (COUNT > 31)
THEN
    DEST[31:0] ← 0
ELSE
    DEST[31:0] ← ZeroExtend(SRC[31:0] << COUNT);
FI;

LOGICAL_LEFT_SHIFT_DWORDS(SRC, COUNT_SRC)
COUNT ←COUNT_SRC[63:0];
IF (COUNT > 31)
THEN
    DEST[127:0] ←00000000000000000000000000000000H
ELSE
    DEST[31:0] ←ZeroExtend(SRC[31:0] << COUNT);
    (* Repeat shift operation for 2nd through 3rd words *)
    DEST[127:96] ←ZeroExtend(SRC[127:96] << COUNT);
FI;

LOGICAL_LEFT_SHIFT_QWORDS1(SRC, COUNT_SRC)
COUNT ← COUNT_SRC[63:0];
IF (COUNT > 63)
THEN
    DEST[63:0] ← 0
ELSE
    DEST[63:0] ← ZeroExtend(SRC[63:0] << COUNT);
FI;

LOGICAL_LEFT_SHIFT_QWORDS(SRC, COUNT_SRC)
COUNT ←COUNT_SRC[63:0];
IF (COUNT > 63)
THEN
    DEST[127:0] ←00000000000000000000000000000000H
ELSE
    DEST[63:0] ←ZeroExtend(SRC[63:0] << COUNT);
    DEST[127:64] ←ZeroExtend(SRC[127:64] << COUNT);
FI;
LOGICAL_LEFT_SHIFT_WORDS_256b(SRC, COUNT_SRC)
COUNT ←COUNT_SRC[63:0];
IF (COUNT > 15)
THEN
    DEST[127:0] ←00000000000000000000000000000000H
    DEST[255:128] ←00000000000000000000000000000000H
ELSE
    DEST[15:0] ←ZeroExtend(SRC[15:0] << COUNT);
    (* Repeat shift operation for 2nd through 15th words *)
```

```
    DEST[255:240] ←ZeroExtend(SRC[255:240] << COUNT);
FI;

LOGICAL_LEFT_SHIFT_DWORDS_256b(SRC, COUNT_SRC)
COUNT ←COUNT_SRC[63:0];
IF (COUNT > 31)
THEN
    DEST[127:0] ←000000000000000000000000000000000H
    DEST[255:128] ←000000000000000000000000000000000H
ELSE
    DEST[31:0] ←ZeroExtend(SRC[31:0] << COUNT);
    (* Repeat shift operation for 2nd through 7th words *)
    DEST[255:224] ←ZeroExtend(SRC[255:224] << COUNT);
FI;

LOGICAL_LEFT_SHIFT_QWORDS_256b(SRC, COUNT_SRC)
COUNT ←COUNT_SRC[63:0];
IF (COUNT > 63)
THEN
    DEST[127:0] ←000000000000000000000000000000000H
    DEST[255:128] ←000000000000000000000000000000000H
ELSE
    DEST[63:0] ←ZeroExtend(SRC[63:0] << COUNT);
    DEST[127:64] ←ZeroExtend(SRC[127:64] << COUNT)
    DEST[191:128] ←ZeroExtend(SRC[191:128] << COUNT);
    DEST[255:192] ←ZeroExtend(SRC[255:192] << COUNT);
FI;
```

VPSLLW (EVEX versions, xmm/m128)

```
(KL, VL) = (8, 128), (16, 256), (32, 512)
IF VL = 128
    TMP_DEST[127:0] ← LOGICAL_LEFT_SHIFT_WORDS_128b(SRC1[127:0], SRC2)
FI;
IF VL = 256
    TMP_DEST[255:0] ← LOGICAL_LEFT_SHIFT_WORDS_256b(SRC1[255:0], SRC2)
FI;
IF VL = 512
    TMP_DEST[255:0] ← LOGICAL_LEFT_SHIFT_WORDS_256b(SRC1[255:0], SRC2)
    TMP_DEST[511:256] ← LOGICAL_LEFT_SHIFT_WORDS_256b(SRC1[511:256], SRC2)
FI;

FOR j ← 0 TO KL-1
    i ← j * 16
    IF k1[j] OR *no writemask*
        THEN DEST[i+15:i] ← TMP_DEST[i+15:i]
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+15:i] remains unchanged*
                ELSE *zeroing-masking*              ; zeroing-masking
                    DEST[i+15:i] = 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0
```

VPSLLW (EVEX versions, imm8)

```
(KL, VL) = (8, 128), (16, 256), (32, 512)
IF VL = 128
    TMP_DEST[127:0] ← LOGICAL_LEFT_SHIFT_WORDS_128b(SRC1[127:0], imm8)
FI;
IF VL = 256
    TMP_DEST[255:0] ← LOGICAL_RIGHT_SHIFT_WORDS_256b(SRC1[255:0], imm8)
FI;
IF VL = 512
    TMP_DEST[255:0] ← LOGICAL_LEFT_SHIFT_WORDS_256b(SRC1[255:0], imm8)
    TMP_DEST[511:256] ← LOGICAL_LEFT_SHIFT_WORDS_256b(SRC1[511:256], imm8)
FI;

FOR j ← 0 TO KL-1
    i ← j * 16
    IF k1[j] OR *no writemask*
        THEN DEST[i+15:i] ← TMP_DEST[i+15:i]
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+15:i] remains unchanged*
                ELSE *zeroing-masking*              ; zeroing-masking
                    DEST[i+15:i] = 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0
```

VPSLLW (ymm, ymm, xmm/m128) - VEX.256 encoding

```
DEST[255:0] ←LOGICAL_LEFT_SHIFT_WORDS_256b(SRC1, SRC2)
DEST[MAXVL-1:256] ←0;
```

VPSLLW (ymm, imm8) - VEX.256 encoding

```
DEST[255:0] ←LOGICAL_LEFT_SHIFT_WORD_256b(SRC1, imm8)
DEST[MAXVL-1:256] ←0;
```

VPSLLW (xmm, xmm, xmm/m128) - VEX.128 encoding

```
DEST[127:0] ←LOGICAL_LEFT_SHIFT_WORDS(SRC1, SRC2)
DEST[MAXVL-1:128] ←0
```

VPSLLW (xmm, imm8) - VEX.128 encoding

```
DEST[127:0] ←LOGICAL_LEFT_SHIFT_WORDS(SRC1, imm8)
DEST[MAXVL-1:128] ←0
```

PSLLW (xmm, xmm, xmm/m128)

```
DEST[127:0] ←LOGICAL_LEFT_SHIFT_WORDS(DEST, SRC)
DEST[MAXVL-1:128] (Unmodified)
```

PSLLW (xmm, imm8)

```
DEST[127:0] ←LOGICAL_LEFT_SHIFT_WORDS(DEST, imm8)
DEST[MAXVL-1:128] (Unmodified)
```

VPSLLD (EVEX versions, imm8)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j ← 0 TO KL-1
    i ← j * 32
    IF k1[j] OR *no writemask* THEN
            IF (EVEX.b = 1) AND (SRC1 *is memory*)
                THEN DEST[i+31:i] ← LOGICAL_LEFT_SHIFT_DWORDS1(SRC1[31:0], imm8)
                ELSE DEST[i+31:i] ← LOGICAL_LEFT_SHIFT_DWORDS1(SRC1[i+31:i], imm8)
            FI;
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE *zeroing-masking*                  ; zeroing-masking
                    DEST[i+31:i] ← 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0
```

VPSLLD (EVEX versions, xmm/m128)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
IF VL = 128
    TMP_DEST[127:0] ← LOGICAL_LEFT_SHIFT_DWORDS_128b(SRC1[127:0], SRC2)
FI;
IF VL = 256
    TMP_DEST[255:0] ← LOGICAL_LEFT_SHIFT_DWORDS_256b(SRC1[255:0], SRC2)
FI;
IF VL = 512
    TMP_DEST[255:0] ← LOGICAL_LEFT_SHIFT_DWORDS_256b(SRC1[255:0], SRC2)
    TMP_DEST[511:256] ← LOGICAL_LEFT_SHIFT_DWORDS_256b(SRC1[511:256], SRC2)
FI;

FOR j ← 0 TO KL-1
    i ← j * 32
    IF k1[j] OR *no writemask*
        THEN DEST[i+31:i] ← TMP_DEST[i+31:i]
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE *zeroing-masking*                  ; zeroing-masking
                    DEST[i+31:i] ← 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0
```

VPSLLD (ymm, ymm, xmm/m128) - VEX.256 encoding

```
DEST[255:0] ←LOGICAL_LEFT_SHIFT_DWORDS_256b(SRC1, SRC2)
DEST[MAXVL-1:256] ←0;
```

VPSLLD (ymm, imm8) - VEX.256 encoding

```
DEST[255:0] ←LOGICAL_LEFT_SHIFT_DWORDS_256b(SRC1, imm8)
DEST[MAXVL-1:256] ←0;
```

VPSLLD (xmm, xmm, xmm/m128) - VEX.128 encoding

```
DEST[127:0] ←LOGICAL_LEFT_SHIFT_DWORDS(SRC1, SRC2)
DEST[MAXVL-1:128] ←0
```

VPSLLD (xmm, imm8) - VEX.128 encoding

```
DEST[127:0] ←LOGICAL_LEFT_SHIFT_DWORDS(SRC1, imm8)
DEST[MAXVL-1:128] ←0
```

PSLLD (xmm, xmm, xmm/m128)

```
DEST[127:0] ←LOGICAL_LEFT_SHIFT_DWORDS(DEST, SRC)
DEST[MAXVL-1:128] (Unmodified)
```

PSLLD (xmm, imm8)

```
DEST[127:0] ←LOGICAL_LEFT_SHIFT_DWORDS(DEST, imm8)
DEST[MAXVL-1:128] (Unmodified)
```

VPSLLQ (EVEX versions, imm8)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j ← 0 TO KL-1
    i ← j * 64
    IF k1[j] OR *no writemask* THEN
            IF (EVEX.b = 1) AND (SRC1 *is memory*)
                THEN DEST[i+63:i] ← LOGICAL_LEFT_SHIFT_QWORDS1(SRC1[63:0], imm8)
                ELSE DEST[i+63:i] ← LOGICAL_LEFT_SHIFT_QWORDS1(SRC1[i+63:i], imm8)
            FI;
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE *zeroing-masking*                  ; zeroing-masking
                    DEST[i+63:i] ← 0
            FI
    FI;
ENDFOR
```

VPSLLQ (EVEX versions, xmm/m128)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
IF VL = 128
    TMP_DEST[127:0] ← LOGICAL_LEFT_SHIFT_QWORDS_128b(SRC1[127:0], SRC2)
FI;
IF VL = 256
    TMP_DEST[255:0] ← LOGICAL_LEFT_SHIFT_QWORDS_256b(SRC1[255:0], SRC2)
FI;
IF VL = 512
    TMP_DEST[255:0] ←LOGICAL_LEFT_SHIFT_QWORDS_256b(SRC1[255:0], SRC2)
    TMP_DEST[511:256] ←LOGICAL_LEFT_SHIFT_QWORDS_256b(SRC1[511:256], SRC2)
FI;
```

```
FOR j ← 0 TO KL-1
    i ← j * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+63:i] ← TMP_DEST[i+63:i]
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE *zeroing-masking*              ; zeroing-masking
                    DEST[i+63:i] ← 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] ←0
```

VPSLLQ (ymm, ymm, xmm/m128) - VEX.256 encoding

```
DEST[255:0] ←LOGICAL_LEFT_SHIFT_QWORDS_256b(SRC1, SRC2)
DEST[MAXVL-1:256] ←0;
```

VPSLLQ (ymm, imm8) - VEX.256 encoding

```
DEST[255:0] ←LOGICAL_LEFT_SHIFT_QWORDS_256b(SRC1, imm8)
DEST[MAXVL-1:256] ←0;
```

VPSLLQ (xmm, xmm, xmm/m128) - VEX.128 encoding

```
DEST[127:0] ←LOGICAL_LEFT_SHIFT_QWORDS(SRC1, SRC2)
DEST[MAXVL-1:128] ←0
```

VPSLLQ (xmm, imm8) - VEX.128 encoding

```
DEST[127:0] ←LOGICAL_LEFT_SHIFT_QWORDS(SRC1, imm8)
DEST[MAXVL-1:128] ←0
```

PSLLQ (xmm, xmm, xmm/m128)

```
DEST[127:0] ←LOGICAL_LEFT_SHIFT_QWORDS(DEST, SRC)
DEST[MAXVL-1:128] (Unmodified)
```

PSLLQ (xmm, imm8)

```
DEST[127:0] ←LOGICAL_LEFT_SHIFT_QWORDS(DEST, imm8)
DEST[MAXVL-1:128] (Unmodified)
```

Intel C/C++ Compiler Intrinsic Equivalents

VPSLLD __m512i _mm512_slli_epi32(__m512i a, unsigned int imm);
VPSLLD __m512i _mm512_mask_slli_epi32(__m512i s, __mmask16 k, __m512i a, unsigned int imm);
VPSLLD __m512i _mm512_maskz_slli_epi32(__mmask16 k, __m512i a, unsigned int imm);
VPSLLD __m256i _mm256_mask_slli_epi32(__m256i s, __mmask8 k, __m256i a, unsigned int imm);
VPSLLD __m256i _mm256_maskz_slli_epi32(__mmask8 k, __m256i a, unsigned int imm);
VPSLLD __m128i _mm_mask_slli_epi32(__m128i s, __mmask8 k, __m128i a, unsigned int imm);
VPSLLD __m128i _mm_maskz_slli_epi32(__mmask8 k, __m128i a, unsigned int imm);
VPSLLD __m512i _mm512_sll_epi32(__m512i a, __m128i cnt);
VPSLLD __m512i _mm512_mask_sll_epi32(__m512i s, __mmask16 k, __m512i a, __m128i cnt);
VPSLLD __m512i _mm512_maskz_sll_epi32(__mmask16 k, __m512i a, __m128i cnt);
VPSLLD __m256i _mm256_mask_sll_epi32(__m256i s, __mmask8 k, __m256i a, __m128i cnt);
VPSLLD __m256i _mm256_maskz_sll_epi32(__mmask8 k, __m256i a, __m128i cnt);
VPSLLD __m128i _mm_mask_sll_epi32(__m128i s, __mmask8 k, __m128i a, __m128i cnt);
VPSLLD __m128i _mm_maskz_sll_epi32(__mmask8 k, __m128i a, __m128i cnt);

VPSLLQ __m512i _mm512_mask_slli_epi64(__m512i a, unsigned int imm);
VPSLLQ __m512i _mm512_mask_slli_epi64(__m512i s, __mmask8 k, __m512i a, unsigned int imm);
VPSLLQ __m512i _mm512_maskz_slli_epi64(__mmask8 k, __m512i a, unsigned int imm);
VPSLLQ __m256i _mm256_mask_slli_epi64(__m256i s, __mmask8 k, __m256i a, unsigned int imm);
VPSLLQ __m256i _mm256_maskz_slli_epi64(__mmask8 k, __m256i a, unsigned int imm);
VPSLLQ __m128i _mm_mask_slli_epi64(__m128i s, __mmask8 k, __m128i a, unsigned int imm);
VPSLLQ __m128i _mm_maskz_slli_epi64(__mmask8 k, __m128i a, unsigned int imm);
VPSLLQ __m512i _mm512_mask_sll_epi64(__m512i a, __m128i cnt);
VPSLLQ __m512i _mm512_mask_sll_epi64(__m512i s, __mmask8 k, __m512i a, __m128i cnt);
VPSLLQ __m512i _mm512_maskz_sll_epi64(__mmask8 k, __m512i a, __m128i cnt);
VPSLLQ __m256i _mm256_mask_sll_epi64(__m256i s, __mmask8 k, __m256i a, __m128i cnt);
VPSLLQ __m256i _mm256_maskz_sll_epi64(__mmask8 k, __m256i a, __m128i cnt);
VPSLLQ __m128i _mm_mask_sll_epi64(__m128i s, __mmask8 k, __m128i a, __m128i cnt);
VPSLLQ __m128i _mm_maskz_sll_epi64(__mmask8 k, __m128i a, __m128i cnt);
VPSLLW __m512i _mm512_slli_epi16(__m512i a, unsigned int imm);
VPSLLW __m512i _mm512_mask_slli_epi16(__m512i s, __mmask32 k, __m512i a, unsigned int imm);
VPSLLW __m512i _mm512_maskz_slli_epi16(__mmask32 k, __m512i a, unsigned int imm);
VPSLLW __m256i _mm256_mask_sllii_epi16(__m256i s, __mmask16 k, __m256i a, unsigned int imm);
VPSLLW __m256i _mm256_maskz_slli_epi16(__mmask16 k, __m256i a, unsigned int imm);
VPSLLW __m128i _mm_mask_slli_epi16(__m128i s, __mmask8 k, __m128i a, unsigned int imm);
VPSLLW __m128i _mm_maskz_slli_epi16(__mmask8 k, __m128i a, unsigned int imm);
VPSLLW __m512i _mm512_sll_epi16(__m512i a, __m128i cnt);
VPSLLW __m512i _mm512_mask_sll_epi16(__m512i s, __mmask32 k, __m512i a, __m128i cnt);
VPSLLW __m512i _mm512_maskz_sll_epi16(__mmask32 k, __m512i a, __m128i cnt);
VPSLLW __m256i _mm256_mask_sll_epi16(__m256i s, __mmask16 k, __m256i a, __m128i cnt);
VPSLLW __m256i _mm256_maskz_sll_epi16(__mmask16 k, __m256i a, __m128i cnt);
VPSLLW __m128i _mm_mask_sll_epi16(__m128i s, __mmask8 k, __m128i a, __m128i cnt);
VPSLLW __m128i _mm_maskz_sll_epi16(__mmask8 k, __m128i a, __m128i cnt);
PSLLW:__m64 _mm_slli_pi16 (__m64 m, int count)
PSLLW:__m64 _mm_sll_pi16(__m64 m, __m64 count)
(V)PSLLW:__m128i _mm_slli_epi16(__m64 m, int count)
(V)PSLLW:__m128i _mm_sll_epi16(__m128i m, __m128i count)
VPSLLW:__m256i _mm256_slli_epi16 (__m256i m, int count)
VPSLLW:__m256i _mm256_sll_epi16 (__m256i m, __m128i count)
PSLLD:__m64 _mm_slli_pi32(__m64 m, int count)
PSLLD:__m64 _mm_sll_pi32(__m64 m, __m64 count)
(V)PSLLD:__m128i _mm_slli_epi32(__m128i m, int count)
(V)PSLLD:__m128i _mm_sll_epi32(__m128i m, __m128i count)
VPSLLD:__m256i _mm256_slli_epi32 (__m256i m, int count)
VPSLLD:__m256i _mm256_sll_epi32 (__m256i m, __m128i count)
PSLLQ:__m64 _mm_slli_si64(__m64 m, int count)
PSLLQ:__m64 _mm_sll_si64(__m64 m, __m64 count)
(V)PSLLQ:__m128i _mm_slli_epi64(__m128i m, int count)
(V)PSLLQ:__m128i _mm_sll_epi64(__m128i m, __m128i count)
VPSLLQ:__m256i _mm256_slli_epi64 (__m256i m, int count)
VPSLLQ:__m256i _mm256_sll_epi64 (__m256i m, __m128i count)

Flags Affected

None.

Numeric Exceptions

None.

Other Exceptions

VEX-encoded instructions:

Syntax with RM/RVM operand encoding, see Exceptions Type 4.

Syntax with MI/VMI operand encoding, see Exceptions Type 7.

EVEX-encoded VPSLLW, see Exceptions Type E4NF.nb.

EVEX-encoded VPSLLD/Q:

Syntax with M128 operand encoding, see Exceptions Type E4NF.nb.

Syntax with FVI operand encoding, see Exceptions Type E4.

PSRAW/PSRAD/PSRAQ—Shift Packed Data Right Arithmetic

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F E1 /r[1] PSRAW *mm, mm/m64*	A	V/V	MMX	Shift words in *mm* right by *mm/m64* while shifting in sign bits.
66 0F E1 /r PSRAW *xmm1, xmm2/m128*	A	V/V	SSE2	Shift words in *xmm1* right by *xmm2/m128* while shifting in sign bits.
NP 0F 71 /4 ib[1] PSRAW *mm, imm8*	B	V/V	MMX	Shift words in *mm* right by *imm8* while shifting in sign bits
66 0F 71 /4 ib PSRAW *xmm1*, imm8	B	V/V	SSE2	Shift words in *xmm1* right by imm8 while shifting in sign bits
NP 0F E2 /r[1] PSRAD *mm, mm/m64*	A	V/V	MMX	Shift doublewords in *mm* right by *mm/m64* while shifting in sign bits.
66 0F E2 /r PSRAD *xmm1, xmm2/m128*	A	V/V	SSE2	Shift doubleword in *xmm1* right by *xmm2 /m128* while shifting in sign bits.
NP 0F 72 /4 ib[1] PSRAD *mm, imm8*	B	V/V	MMX	Shift doublewords in *mm* right by *imm8* while shifting in sign bits.
66 0F 72 /4 ib PSRAD *xmm1*, imm8	B	V/V	SSE2	Shift doublewords in *xmm1* right by *imm8* while shifting in sign bits.
VEX.NDS.128.66.0F.WIG E1 /r VPSRAW *xmm1, xmm2, xmm3/m128*	C	V/V	AVX	Shift words in *xmm2* right by amount specified in *xmm3/m128* while shifting in sign bits.
VEX.NDD.128.66.0F.WIG 71 /4 ib VPSRAW *xmm1, xmm2, imm8*	D	V/V	AVX	Shift words in *xmm2* right by *imm8* while shifting in sign bits.
VEX.NDS.128.66.0F.WIG E2 /r VPSRAD *xmm1, xmm2, xmm3/m128*	C	V/V	AVX	Shift doublewords in *xmm2* right by amount specified in *xmm3/m128* while shifting in sign bits.
VEX.NDD.128.66.0F.WIG 72 /4 ib VPSRAD *xmm1, xmm2, imm8*	D	V/V	AVX	Shift doublewords in *xmm2* right by *imm8* while shifting in sign bits.
VEX.NDS.256.66.0F.WIG E1 /r VPSRAW *ymm1, ymm2, xmm3/m128*	C	V/V	AVX2	Shift words in *ymm2* right by amount specified in *xmm3/m128* while shifting in sign bits.
VEX.NDD.256.66.0F.WIG 71 /4 ib VPSRAW *ymm1, ymm2, imm8*	D	V/V	AVX2	Shift words in *ymm2* right by *imm8* while shifting in sign bits.
VEX.NDS.256.66.0F.WIG E2 /r VPSRAD *ymm1, ymm2, xmm3/m128*	C	V/V	AVX2	Shift doublewords in *ymm2* right by amount specified in *xmm3/m128* while shifting in sign bits.
VEX.NDD.256.66.0F.WIG 72 /4 ib VPSRAD *ymm1, ymm2, imm8*	D	V/V	AVX2	Shift doublewords in *ymm2* right by *imm8* while shifting in sign bits.
EVEX.NDS.128.66.0F.WIG E1 /r VPSRAW xmm1 {k1}{z}, xmm2, xmm3/m128	G	V/V	AVX512VL AVX512BW	Shift words in xmm2 right by amount specified in xmm3/m128 while shifting in sign bits using writemask k1.
EVEX.NDS.256.66.0F.WIG E1 /r VPSRAW ymm1 {k1}{z}, ymm2, xmm3/m128	G	V/V	AVX512VL AVX512BW	Shift words in ymm2 right by amount specified in xmm3/m128 while shifting in sign bits using writemask k1.
EVEX.NDS.512.66.0F.WIG E1 /r VPSRAW zmm1 {k1}{z}, zmm2, xmm3/m128	G	V/V	AVX512BW	Shift words in zmm2 right by amount specified in xmm3/m128 while shifting in sign bits using writemask k1.

EVEX.NDD.128.66.0F.WIG 71 /4 ib VPSRAW xmm1 {k1}{z}, xmm2/m128, imm8	E	V/V	AVX512VL AVX512BW	Shift words in xmm2/m128 right by imm8 while shifting in sign bits using writemask k1.
EVEX.NDD.256.66.0F.WIG 71 /4 ib VPSRAW ymm1 {k1}{z}, ymm2/m256, imm8	E	V/V	AVX512VL AVX512BW	Shift words in ymm2/m256 right by imm8 while shifting in sign bits using writemask k1.
EVEX.NDD.512.66.0F.WIG 71 /4 ib VPSRAW zmm1 {k1}{z}, zmm2/m512, imm8	E	V/V	AVX512BW	Shift words in zmm2/m512 right by imm8 while shifting in sign bits using writemask k1.
EVEX.NDS.128.66.0F.W0 E2 /r VPSRAD xmm1 {k1}{z}, xmm2, xmm3/m128	G	V/V	AVX512VL AVX512F	Shift doublewords in xmm2 right by amount specified in xmm3/m128 while shifting in sign bits using writemask k1.
EVEX.NDS.256.66.0F.W0 E2 /r VPSRAD ymm1 {k1}{z}, ymm2, xmm3/m128	G	V/V	AVX512VL AVX512F	Shift doublewords in ymm2 right by amount specified in xmm3/m128 while shifting in sign bits using writemask k1.
EVEX.NDS.512.66.0F.W0 E2 /r VPSRAD zmm1 {k1}{z}, zmm2, xmm3/m128	G	V/V	AVX512F	Shift doublewords in zmm2 right by amount specified in xmm3/m128 while shifting in sign bits using writemask k1.
EVEX.NDD.128.66.0F.W0 72 /4 ib VPSRAD xmm1 {k1}{z}, xmm2/m128/m32bcst, imm8	F	V/V	AVX512VL AVX512F	Shift doublewords in xmm2/m128/m32bcst right by imm8 while shifting in sign bits using writemask k1.
EVEX.NDD.256.66.0F.W0 72 /4 ib VPSRAD ymm1 {k1}{z}, ymm2/m256/m32bcst, imm8	F	V/V	AVX512VL AVX512F	Shift doublewords in ymm2/m256/m32bcst right by imm8 while shifting in sign bits using writemask k1.
EVEX.NDD.512.66.0F.W0 72 /4 ib VPSRAD zmm1 {k1}{z}, zmm2/m512/m32bcst, imm8	F	V/V	AVX512F	Shift doublewords in zmm2/m512/m32bcst right by imm8 while shifting in sign bits using writemask k1.
EVEX.NDS.128.66.0F.W1 E2 /r VPSRAQ xmm1 {k1}{z}, xmm2, xmm3/m128	G	V/V	AVX512VL AVX512F	Shift quadwords in xmm2 right by amount specified in xmm3/m128 while shifting in sign bits using writemask k1.
EVEX.NDS.256.66.0F.W1 E2 /r VPSRAQ ymm1 {k1}{z}, ymm2, xmm3/m128	G	V/V	AVX512VL AVX512F	Shift quadwords in ymm2 right by amount specified in xmm3/m128 while shifting in sign bits using writemask k1.
EVEX.NDS.512.66.0F.W1 E2 /r VPSRAQ zmm1 {k1}{z}, zmm2, xmm3/m128	G	V/V	AVX512F	Shift quadwords in zmm2 right by amount specified in xmm3/m128 while shifting in sign bits using writemask k1.
EVEX.NDD.128.66.0F.W1 72 /4 ib VPSRAQ xmm1 {k1}{z}, xmm2/m128/m64bcst, imm8	F	V/V	AVX512VL AVX512F	Shift quadwords in xmm2/m128/m64bcst right by imm8 while shifting in sign bits using writemask k1.
EVEX.NDD.256.66.0F.W1 72 /4 ib VPSRAQ ymm1 {k1}{z}, ymm2/m256/m64bcst, imm8	F	V/V	AVX512VL AVX512F	Shift quadwords in ymm2/m256/m64bcst right by imm8 while shifting in sign bits using writemask k1.
EVEX.NDD.512.66.0F.W1 72 /4 ib VPSRAQ zmm1 {k1}{z}, zmm2/m512/m64bcst, imm8	F	V/V	AVX512F	Shift quadwords in zmm2/m512/m64bcst right by imm8 while shifting in sign bits using writemask k1.

NOTES:

1. See note in Section 2.4, "AVX and SSE Instruction Exception Specification" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A* and Section 22.25.3, "Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	NA	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
B	NA	ModRM:r/m (r, w)	imm8	NA	NA
C	NA	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA
D	NA	VEX.vvvv (w)	ModRM:r/m (r)	imm8	NA
E	Full Vector Mem	EVEX.vvvv (w)	ModRM:r/m (R)	Imm8	NA
F	Full Vector	EVEX.vvvv (w)	ModRM:r/m (R)	Imm8	NA
G	Mem128	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA

Description

Shifts the bits in the individual data elements (words, doublewords or quadwords) in the destination operand (first operand) to the right by the number of bits specified in the count operand (second operand). As the bits in the data elements are shifted right, the empty high-order bits are filled with the initial value of the sign bit of the data element. If the value specified by the count operand is greater than 15 (for words), 31 (for doublewords), or 63 (for quadwords), each destination data element is filled with the initial value of the sign bit of the element. (Figure 4-18 gives an example of shifting words in a 64-bit operand.)

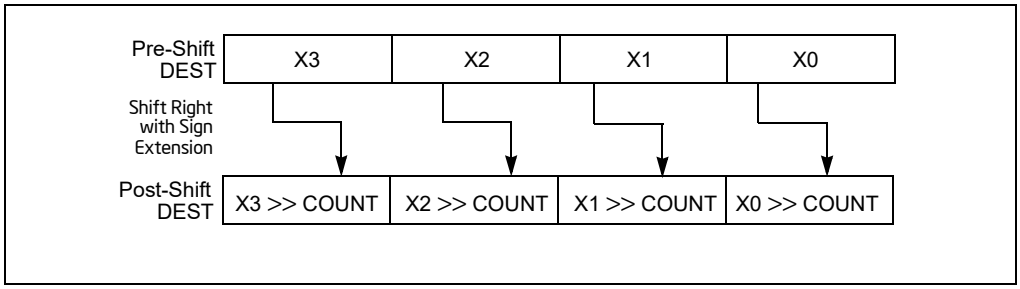


Figure 4-18. PSRAW and PSRAD Instruction Operation Using a 64-bit Operand

Note that only the first 64-bits of a 128-bit count operand are checked to compute the count. If the second source operand is a memory address, 128 bits are loaded.

The (V)PSRAW instruction shifts each of the words in the destination operand to the right by the number of bits specified in the count operand, and the (V)PSRAD instruction shifts each of the doublewords in the destination operand.

In 64-bit mode and not encoded with VEX/EVEX, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

Legacy SSE instructions 64-bit operand: The destination operand is an MMX technology register; the count operand can be either an MMX technology register or an 64-bit memory location.

128-bit Legacy SSE version: The destination and first source operands are XMM registers. Bits (MAXVL-1:128) of the corresponding YMM destination register remain unchanged. The count operand can be either an XMM register or a 128-bit memory location or an 8-bit immediate. If the count operand is a memory address, 128 bits are loaded but the upper 64 bits are ignored.

VEX.128 encoded version: The destination and first source operands are XMM registers. Bits (MAXVL-1:128) of the destination YMM register are zeroed. The count operand can be either an XMM register or a 128-bit memory location or an 8-bit immediate. If the count operand is a memory address, 128 bits are loaded but the upper 64 bits are ignored.

VEX.256 encoded version: The destination operand is a YMM register. The source operand is a YMM register or a memory location. The count operand can come either from an XMM register or a memory location or an 8-bit immediate. Bits (MAXVL-1:256) of the corresponding ZMM register are zeroed.

EVEX encoded versions: The destination operand is a ZMM register updated according to the writemask. The count operand is either an 8-bit immediate (the immediate count version) or an 8-bit value from an XMM register or a memory location (the variable count version). For the immediate count version, the source operand (the second operand) can be a ZMM register, a 512-bit memory location or a 512-bit vector broadcasted from a 32/64-bit memory location. For the variable count version, the first source operand (the second operand) is a ZMM register, the second source operand (the third operand, 8-bit variable count) can be an XMM register or a memory location.

Note: In VEX/EVEX encoded versions of shifts with an immediate count, vvvv of VEX/EVEX encode the destination register, and VEX.B/EVEX.B + ModRM.r/m encodes the source register.

Note: For shifts with an immediate count (VEX.128.66.0F 71-73 /4, EVEX.128.66.0F 71-73 /4), VEX.vvvv/EVEX.vvvv encodes the destination register.

Operation

PSRAW (with 64-bit operand)

```
    IF (COUNT > 15)
        THEN COUNT ← 16;
    FI;
    DEST[15:0] ← SignExtend(DEST[15:0] >> COUNT);
    (* Repeat shift operation for 2nd and 3rd words *)
    DEST[63:48] ← SignExtend(DEST[63:48] >> COUNT);
```

PSRAD (with 64-bit operand)

```
    IF (COUNT > 31)
        THEN COUNT ← 32;
    FI;
    DEST[31:0] ← SignExtend(DEST[31:0] >> COUNT);
    DEST[63:32] ← SignExtend(DEST[63:32] >> COUNT);

ARITHMETIC_RIGHT_SHIFT_DWORDS1(SRC, COUNT_SRC)
COUNT ← COUNT_SRC[63:0];
IF (COUNT > 31)
THEN
    DEST[31:0] ← SignBit
ELSE
    DEST[31:0] ← SignExtend(SRC[31:0] >> COUNT);
FI;

ARITHMETIC_RIGHT_SHIFT_QWORDS1(SRC, COUNT_SRC)
COUNT ← COUNT_SRC[63:0];
IF (COUNT > 63)
THEN
    DEST[63:0] ← SignBit
ELSE
    DEST[63:0] ← SignExtend(SRC[63:0] >> COUNT);
FI;

ARITHMETIC_RIGHT_SHIFT_WORDS_256b(SRC, COUNT_SRC)
COUNT ← COUNT_SRC[63:0];
IF (COUNT > 15)
    THEN     COUNT ← 16;
FI;
DEST[15:0] ← SignExtend(SRC[15:0] >> COUNT);
    (* Repeat shift operation for 2nd through 15th words *)
DEST[255:240] ← SignExtend(SRC[255:240] >> COUNT);
```

```
ARITHMETIC_RIGHT_SHIFT_DWORDS_256b(SRC, COUNT_SRC)
COUNT ← COUNT_SRC[63:0];
IF (COUNT > 31)
    THEN    COUNT ← 32;
FI;
DEST[31:0] ← SignExtend(SRC[31:0] >> COUNT);
    (* Repeat shift operation for 2nd through 7th words *)
DEST[255:224] ← SignExtend(SRC[255:224] >> COUNT);

ARITHMETIC_RIGHT_SHIFT_QWORDS(SRC, COUNT_SRC, VL)          ; VL: 128b, 256b or 512b
COUNT ← COUNT_SRC[63:0];
IF (COUNT > 63)
    THEN    COUNT ← 64;
FI;
DEST[63:0] ← SignExtend(SRC[63:0] >> COUNT);
    (* Repeat shift operation for 2nd through 7th words *)
DEST[VL-1:VL-64] ← SignExtend(SRC[VL-1:VL-64] >> COUNT);


ARITHMETIC_RIGHT_SHIFT_WORDS(SRC, COUNT_SRC)
COUNT ← COUNT_SRC[63:0];
IF (COUNT > 15)
    THEN    COUNT ← 16;
FI;
DEST[15:0] ← SignExtend(SRC[15:0] >> COUNT);
    (* Repeat shift operation for 2nd through 7th words *)
DEST[127:112] ← SignExtend(SRC[127:112] >> COUNT);

ARITHMETIC_RIGHT_SHIFT_DWORDS(SRC, COUNT_SRC)
COUNT ← COUNT_SRC[63:0];
IF (COUNT > 31)
    THEN    COUNT ← 32;
FI;
DEST[31:0] ← SignExtend(SRC[31:0] >> COUNT);
    (* Repeat shift operation for 2nd through 3rd words *)
DEST[127:96] ← SignExtend(SRC[127:96] >> COUNT);
```

VPSRAW (EVEX versions, xmm/m128)

```
(KL, VL) = (8, 128), (16, 256), (32, 512)
IF VL = 128
    TMP_DEST[127:0] ← ARITHMETIC_RIGHT_SHIFT_WORDS_128b(SRC1[127:0], SRC2)
FI;
IF VL = 256
    TMP_DEST[255:0] ← ARITHMETIC_RIGHT_SHIFT_WORDS_256b(SRC1[255:0], SRC2)
FI;
IF VL = 512
    TMP_DEST[255:0] ← ARITHMETIC_RIGHT_SHIFT_WORDS_256b(SRC1[255:0], SRC2)
    TMP_DEST[511:256] ← ARITHMETIC_RIGHT_SHIFT_WORDS_256b(SRC1[511:256], SRC2)
FI;

FOR j ← 0 TO KL-1
    i ← j * 16
    IF k1[j] OR *no writemask*
        THEN DEST[i+15:i] ← TMP_DEST[i+15:i]
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+15:i] remains unchanged*
                ELSE *zeroing-masking*              ; zeroing-masking
                    DEST[i+15:i] = 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0
```

VPSRAW (EVEX versions, imm8)

```
(KL, VL) = (8, 128), (16, 256), (32, 512)
IF VL = 128
    TMP_DEST[127:0] ← ARITHMETIC_RIGHT_SHIFT_WORDS_128b(SRC1[127:0], imm8)
FI;
IF VL = 256
    TMP_DEST[255:0] ← ARITHMETIC_RIGHT_SHIFT_WORDS_256b(SRC1[255:0], imm8)
FI;
IF VL = 512
    TMP_DEST[255:0] ← ARITHMETIC_RIGHT_SHIFT_WORDS_256b(SRC1[255:0], imm8)
    TMP_DEST[511:256] ← ARITHMETIC_RIGHT_SHIFT_WORDS_256b(SRC1[511:256], imm8)
FI;

FOR j ← 0 TO KL-1
    i ← j * 16
    IF k1[j] OR *no writemask*
        THEN DEST[i+15:i] ← TMP_DEST[i+15:i]
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+15:i] remains unchanged*
                ELSE *zeroing-masking*              ; zeroing-masking
                    DEST[i+15:i] = 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0
```

VPSRAW (ymm, ymm, xmm/m128) - VEX

```
DEST[255:0] ← ARITHMETIC_RIGHT_SHIFT_WORDS_256b(SRC1, SRC2)
DEST[MAXVL-1:256] ← 0
```

VPSRAW (ymm, imm8) - VEX

```
DEST[255:0] ← ARITHMETIC_RIGHT_SHIFT_WORDS_256b(SRC1, imm8)
DEST[MAXVL-1:256] ← 0
```

VPSRAW (xmm, xmm, xmm/m128) - VEX

```
DEST[127:0] ← ARITHMETIC_RIGHT_SHIFT_WORDS(SRC1, SRC2)
DEST[MAXVL-1:128] ← 0
```

VPSRAW (xmm, imm8) - VEX

```
DEST[127:0] ← ARITHMETIC_RIGHT_SHIFT_WORDS(SRC1, imm8)
DEST[MAXVL-1:128] ← 0
```

PSRAW (xmm, xmm, xmm/m128)

```
DEST[127:0] ←ARITHMETIC_RIGHT_SHIFT_WORDS(DEST, SRC)
DEST[MAXVL-1:128] (Unmodified)
```

PSRAW (xmm, imm8)

```
DEST[127:0] ←ARITHMETIC_RIGHT_SHIFT_WORDS(DEST, imm8)
DEST[MAXVL-1:128] (Unmodified)
```

VPSRAD (EVEX versions, imm8)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j ← 0 TO KL-1
    i ← j * 32
    IF k1[j] OR *no writemask* THEN
            IF (EVEX.b = 1) AND (SRC1 *is memory*)
                THEN DEST[i+31:i] ← ARITHMETIC_RIGHT_SHIFT_DWORDS1(SRC1[31:0], imm8)
                ELSE DEST[i+31:i] ← ARITHMETIC_RIGHT_SHIFT_DWORDS1(SRC1[i+31:i], imm8)
            FI;
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE *zeroing-masking*                  ; zeroing-masking
                    DEST[i+31:i] ← 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0
```

VPSRAD (EVEX versions, xmm/m128)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
IF VL = 128
    TMP_DEST[127:0] ← ARITHMETIC_RIGHT_SHIFT_DWORDS_128b(SRC1[127:0], SRC2)
FI;
IF VL = 256
    TMP_DEST[255:0] ← ARITHMETIC_RIGHT_SHIFT_DWORDS_256b(SRC1[255:0], SRC2)
FI;
IF VL = 512
    TMP_DEST[255:0] ← ARITHMETIC_RIGHT_SHIFT_DWORDS_256b(SRC1[255:0], SRC2)
    TMP_DEST[511:256] ← ARITHMETIC_RIGHT_SHIFT_DWORDS_256b(SRC1[511:256], SRC2)
```

```
FI;

FOR j ← 0 TO KL-1
    i ← j * 32
    IF k1[j] OR *no writemask*
        THEN DEST[i+31:i] ← TMP_DEST[i+31:i]
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE *zeroing-masking*              ; zeroing-masking
                    DEST[i+31:i] ← 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0
```

VPSRAD (ymm, ymm, xmm/m128) - VEX

```
DEST[255:0] ←ARITHMETIC_RIGHT_SHIFT_DWORDS_256b(SRC1, SRC2)
DEST[MAXVL-1:256] ← 0
```

VPSRAD (ymm, imm8) - VEX

```
DEST[255:0] ←ARITHMETIC_RIGHT_SHIFT_DWORDS_256b(SRC1, imm8)
DEST[MAXVL-1:256] ← 0
```

VPSRAD (xmm, xmm, xmm/m128) - VEX

```
DEST[127:0] ←ARITHMETIC_RIGHT_SHIFT_DWORDS(SRC1, SRC2)
DEST[MAXVL-1:128] ←0
```

VPSRAD (xmm, imm8) - VEX

```
DEST[127:0] ←ARITHMETIC_RIGHT_SHIFT_DWORDS(SRC1, imm8)
DEST[MAXVL-1:128] ←0
```

PSRAD (xmm, xmm, xmm/m128)

```
DEST[127:0] ←ARITHMETIC_RIGHT_SHIFT_DWORDS(DEST, SRC)
DEST[MAXVL-1:128] (Unmodified)
```

PSRAD (xmm, imm8)

```
DEST[127:0] ←ARITHMETIC_RIGHT_SHIFT_DWORDS(DEST, imm8)
DEST[MAXVL-1:128] (Unmodified)
```

VPSRAQ (EVEX versions, imm8)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j ← 0 TO KL-1
    i ← j * 64
    IF k1[j] OR *no writemask* THEN
            IF (EVEX.b = 1) AND (SRC1 *is memory*)
                THEN DEST[i+63:i] ← ARITHMETIC_RIGHT_SHIFT_QWORDS1(SRC1[63:0], imm8)
                ELSE DEST[i+63:i] ← ARITHMETIC_RIGHT_SHIFT_QWORDS1(SRC1[i+63:i], imm8)
            FI;
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE *zeroing-masking*              ; zeroing-masking
                    DEST[i+63:i] ← 0
```

```
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0
```

VPSRAQ (EVEX versions, xmm/m128)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
TMP_DEST[VL-1:0] ← ARITHMETIC_RIGHT_SHIFT_QWORDS(SRC1[VL-1:0], SRC2, VL)

FOR j ← 0 TO 7
    i ← j * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+63:i] ← TMP_DEST[i+63:i]
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE *zeroing-masking*              ; zeroing-masking
                    DEST[i+63:i] ← 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0
```

Intel C/C++ Compiler Intrinsic Equivalents

VPSRAD __m512i _mm512_srai_epi32(__m512i a, unsigned int imm);
VPSRAD __m512i _mm512_mask_srai_epi32(__m512i s, __mmask16 k, __m512i a, unsigned int imm);
VPSRAD __m512i _mm512_maskz_srai_epi32(__mmask16 k, __m512i a, unsigned int imm);
VPSRAD __m256i _mm256_mask_srai_epi32(__m256i s, __mmask8 k, __m256i a, unsigned int imm);
VPSRAD __m256i _mm256_maskz_srai_epi32(__mmask8 k, __m256i a, unsigned int imm);
VPSRAD __m128i _mm_mask_srai_epi32(__m128i s, __mmask8 k, __m128i a, unsigned int imm);
VPSRAD __m128i _mm_maskz_srai_epi32(__mmask8 k, __m128i a, unsigned int imm);
VPSRAD __m512i _mm512_sra_epi32(__m512i a, __m128i cnt);
VPSRAD __m512i _mm512_mask_sra_epi32(__m512i s, __mmask16 k, __m512i a, __m128i cnt);
VPSRAD __m512i _mm512_maskz_sra_epi32(__mmask16 k, __m512i a, __m128i cnt);
VPSRAD __m256i _mm256_mask_sra_epi32(__m256i s, __mmask8 k, __m256i a, __m128i cnt);
VPSRAD __m256i _mm256_maskz_sra_epi32(__mmask8 k, __m256i a, __m128i cnt);
VPSRAD __m128i _mm_mask_sra_epi32(__m128i s, __mmask8 k, __m128i a, __m128i cnt);
VPSRAD __m128i _mm_maskz_sra_epi32(__mmask8 k, __m128i a, __m128i cnt);
VPSRAQ __m512i _mm512_srai_epi64(__m512i a, unsigned int imm);
VPSRAQ __m512i _mm512_mask_srai_epi64(__m512i s, __mmask8 k, __m512i a, unsigned int imm)
VPSRAQ __m512i _mm512_maskz_srai_epi64(__mmask8 k, __m512i a, unsigned int imm)
VPSRAQ __m256i _mm256_mask_srai_epi64(__m256i s, __mmask8 k, __m256i a, unsigned int imm);
VPSRAQ __m256i _mm256_maskz_srai_epi64(__mmask8 k, __m256i a, unsigned int imm);
VPSRAQ __m128i _mm_mask_srai_epi64(__m128i s, __mmask8 k, __m128i a, unsigned int imm);
VPSRAQ __m128i _mm_maskz_srai_epi64(__mmask8 k, __m128i a, unsigned int imm);
VPSRAQ __m512i _mm512_sra_epi64(__m512i a, __m128i cnt);
VPSRAQ __m512i _mm512_mask_sra_epi64(__m512i s, __mmask8 k, __m512i a, __m128i cnt)
VPSRAQ __m512i _mm512_maskz_sra_epi64(__mmask8 k, __m512i a, __m128i cnt)
VPSRAQ __m256i _mm256_mask_sra_epi64(__m256i s, __mmask8 k, __m256i a, __m128i cnt);
VPSRAQ __m256i _mm256_maskz_sra_epi64(__mmask8 k, __m256i a, __m128i cnt);
VPSRAQ __m128i _mm_mask_sra_epi64(__m128i s, __mmask8 k, __m128i a, __m128i cnt);
VPSRAQ __m128i _mm_maskz_sra_epi64(__mmask8 k, __m128i a, __m128i cnt);
VPSRAW __m512i _mm512_srai_epi16(__m512i a, unsigned int imm);
VPSRAW __m512i _mm512_mask_srai_epi16(__m512i s, __mmask32 k, __m512i a, unsigned int imm);

VPSRAW __m512i _mm512_maskz_srai_epi16(__mmask32 k, __m512i a, unsigned int imm);
VPSRAW __m256i _mm256_mask_srai_epi16(__m256i s, __mmask16 k, __m256i a, unsigned int imm);
VPSRAW __m256i _mm256_maskz_srai_epi16(__mmask16 k, __m256i a, unsigned int imm);
VPSRAW __m128i _mm_mask_srai_epi16(__m128i s, __mmask8 k, __m128i a, unsigned int imm);
VPSRAW __m128i _mm_maskz_srai_epi16(__mmask8 k, __m128i a, unsigned int imm);
VPSRAW __m512i _mm512_sra_epi16(__m512i a, __m128i cnt);
VPSRAW __m512i _mm512_mask_sra_epi16(__m512i s, __mmask16 k, __m512i a, __m128i cnt);
VPSRAW __m512i _mm512_maskz_sra_epi16(__mmask16 k, __m512i a, __m128i cnt);
VPSRAW __m256i _mm256_mask_sra_epi16(__m256i s, __mmask8 k, __m256i a, __m128i cnt);
VPSRAW __m256i _mm256_maskz_sra_epi16(__mmask8 k, __m256i a, __m128i cnt);
VPSRAW __m128i _mm_mask_sra_epi16(__m128i s, __mmask8 k, __m128i a, __m128i cnt);
VPSRAW __m128i _mm_maskz_sra_epi16(__mmask8 k, __m128i a, __m128i cnt);
PSRAW:__m64 _mm_srai_pi16 (__m64 m, int count)
PSRAW:__m64 _mm_sra_pi16 (__m64 m, __m64 count)
(V)PSRAW:__m128i _mm_srai_epi16(__m128i m, int count)
(V)PSRAW:__m128i _mm_sra_epi16(__m128i m, __m128i count)
VPSRAW:__m256i _mm256_srai_epi16 (__m256i m, int count)
VPSRAW:__m256i _mm256_sra_epi16 (__m256i m, __m128i count)
PSRAD:__m64 _mm_srai_pi32 (__m64 m, int count)
PSRAD:__m64 _mm_sra_pi32 (__m64 m, __m64 count)
(V)PSRAD:__m128i _mm_srai_epi32 (__m128i m, int count)
(V)PSRAD:__m128i _mm_sra_epi32 (__m128i m, __m128i count)
VPSRAD:__m256i _mm256_srai_epi32 (__m256i m, int count)
VPSRAD:__m256i _mm256_sra_epi32 (__m256i m, __m128i count)

Flags Affected

None.

Numeric Exceptions

None.

Other Exceptions

VEX-encoded instructions:

Syntax with RM/RVM operand encoding, see Exceptions Type 4.

Syntax with MI/VMI operand encoding, see Exceptions Type 7.

EVEX-encoded VPSRAW, see Exceptions Type E4NF.nb.

EVEX-encoded VPSRAD/Q:

Syntax with M128 operand encoding, see Exceptions Type E4NF.nb.

Syntax with FVI operand encoding, see Exceptions Type E4.

PSRLDQ—Shift Double Quadword Right Logical

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 73 /3 ib PSRLDQ *xmm1, imm8*	A	V/V	SSE2	Shift *xmm1* right by *imm8* while shifting in 0s.
VEX.NDD.128.66.0F.WIG 73 /3 ib VPSRLDQ *xmm1, xmm2, imm8*	B	V/V	AVX	Shift *xmm2* right by *imm8* bytes while shifting in 0s.
VEX.NDD.256.66.0F.WIG 73 /3 ib VPSRLDQ *ymm1, ymm2, imm8*	B	V/V	AVX2	Shift *ymm1* right by *imm8* bytes while shifting in 0s.
EVEX.NDD.128.66.0F.WIG 73 /3 ib VPSRLDQ xmm1, xmm2/m128, imm8	C	V/V	AVX512VL AVX512BW	Shift xmm2/m128 right by imm8 bytes while shifting in 0s and store result in xmm1.
EVEX.NDD.256.66.0F.WIG 73 /3 ib VPSRLDQ ymm1, ymm2/m256, imm8	C	V/V	AVX512VL AVX512BW	Shift ymm2/m256 right by imm8 bytes while shifting in 0s and store result in ymm1.
EVEX.NDD.512.66.0F.WIG 73 /3 ib VPSRLDQ zmm1, zmm2/m512, imm8	C	V/V	AVX512BW	Shift zmm2/m512 right by imm8 bytes while shifting in 0s and store result in zmm1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	NA	ModRM:r/m (r, w)	imm8	NA	NA
B	NA	VEX.vvvv (w)	ModRM:r/m (r)	imm8	NA
C	Full Vector Mem	EVEX.vvvv (w)	ModRM:r/m (R)	Imm8	NA

Description

Shifts the destination operand (first operand) to the right by the number of bytes specified in the count operand (second operand). The empty high-order bytes are cleared (set to all 0s). If the value specified by the count operand is greater than 15, the destination operand is set to all 0s. The count operand is an 8-bit immediate.

In 64-bit mode and not encoded with VEX/EVEX, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

128-bit Legacy SSE version: The source and destination operands are the same. Bits (MAXVL-1:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: The source and destination operands are XMM registers. Bits (MAXVL-1:128) of the destination YMM register are zeroed.

VEX.256 encoded version: The source operand is a YMM register. The destination operand is a YMM register. The count operand applies to both the low and high 128-bit lanes.

VEX.256 encoded version: The source operand is YMM register. The destination operand is an YMM register. Bits (MAXVL-1:256) of the corresponding ZMM register are zeroed. The count operand applies to both the low and high 128-bit lanes.

EVEX encoded versions: The source operand is a ZMM/YMM/XMM register or a 512/256/128-bit memory location. The destination operand is a ZMM/YMM/XMM register. The count operand applies to each 128-bit lanes.

Note: VEX.vvvv/EVEX.vvvv encodes the destination register.

Operation

VPSRLDQ (EVEX.512 encoded version)

```
TEMP ← COUNT
IF (TEMP > 15) THEN TEMP ← 16; FI
DEST[127:0] ← SRC[127:0] >> (TEMP * 8)
DEST[255:128] ← SRC[255:128] >> (TEMP * 8)
DEST[383:256] ← SRC[383:256] >> (TEMP * 8)
DEST[511:384] ← SRC[511:384] >> (TEMP * 8)
DEST[MAXVL-1:512] ← 0;
```

VPSRLDQ (VEX.256 and EVEX.256 encoded version)

```
TEMP ← COUNT
IF (TEMP > 15) THEN TEMP ← 16; FI
DEST[127:0] ← SRC[127:0] >> (TEMP * 8)
DEST[255:128] ← SRC[255:128] >> (TEMP * 8)
DEST[MAXVL-1:256] ← 0;
```

VPSRLDQ (VEX.128 and EVEX.128 encoded version)

```
TEMP ← COUNT
IF (TEMP > 15) THEN TEMP ← 16; FI
DEST ← SRC >> (TEMP * 8)
DEST[MAXVL-1:128] ← 0;
```

PSRLDQ(128-bit Legacy SSE version)

```
TEMP ← COUNT
IF (TEMP > 15) THEN TEMP ← 16; FI
DEST ← DEST >> (TEMP * 8)
DEST[MAXVL-1:128] (Unmodified)
```

Intel C/C++ Compiler Intrinsic Equivalents

(V)PSRLDQ __m128i _mm_srli_si128 (__m128i a, int imm)
VPSRLDQ __m256i _mm256_bsrli_epi128 (__m256i, const int)
VPSRLDQ __m512i _mm512_bsrli_epi128 (__m512i, int)

Flags Affected

None.

Numeric Exceptions

None.

Other Exceptions

Non-EVEX-encoded instruction, see Exceptions Type 7.
EVEX-encoded instruction, see Exceptions Type E4NF.nb.

PSRLW/PSRLD/PSRLQ—Shift Packed Data Right Logical

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F D1 /r[1] PSRLW *mm, mm/m64*	A	V/V	MMX	Shift words in *mm* right by amount specified in *mm/m64* while shifting in 0s.
66 0F D1 /r PSRLW *xmm1, xmm2/m128*	A	V/V	SSE2	Shift words in *xmm1* right by amount specified in *xmm2/m128* while shifting in 0s.
NP 0F 71 /2 ib[1] PSRLW *mm, imm8*	B	V/V	MMX	Shift words in *mm* right by *imm8* while shifting in 0s.
66 0F 71 /2 ib PSRLW *xmm1, imm8*	B	V/V	SSE2	Shift words in *xmm1* right by *imm8* while shifting in 0s.
NP 0F D2 /r[1] PSRLD *mm, mm/m64*	A	V/V	MMX	Shift doublewords in *mm* right by amount specified in *mm/m64* while shifting in 0s.
66 0F D2 /r PSRLD *xmm1, xmm2/m128*	A	V/V	SSE2	Shift doublewords in *xmm1* right by amount specified in *xmm2 /m128* while shifting in 0s.
NP 0F 72 /2 ib[1] PSRLD *mm, imm8*	B	V/V	MMX	Shift doublewords in *mm* right by *imm8* while shifting in 0s.
66 0F 72 /2 ib PSRLD *xmm1*, imm8	B	V/V	SSE2	Shift doublewords in *xmm1* right by *imm8* while shifting in 0s.
NP 0F D3 /r[1] PSRLQ *mm, mm/m64*	A	V/V	MMX	Shift *mm* right by amount specified in *mm/m64* while shifting in 0s.
66 0F D3 /r PSRLQ *xmm1, xmm2/m128*	A	V/V	SSE2	Shift quadwords in *xmm1* right by amount specified in *xmm2/m128* while shifting in 0s.
NP 0F 73 /2 ib[1] PSRLQ *mm, imm8*	B	V/V	MMX	Shift *mm* right by *imm8* while shifting in 0s.
66 0F 73 /2 ib PSRLQ *xmm1, imm8*	B	V/V	SSE2	Shift quadwords in *xmm1* right by *imm8* while shifting in 0s.
VEX.NDS.128.66.0F.WIG D1 /r VPSRLW *xmm1, xmm2, xmm3/m128*	C	V/V	AVX	Shift words in *xmm2* right by amount specified in *xmm3/m128* while shifting in 0s.
VEX.NDD.128.66.0F.WIG 71 /2 ib VPSRLW *xmm1, xmm2, imm8*	D	V/V	AVX	Shift words in *xmm2* right by *imm8* while shifting in 0s.
VEX.NDS.128.66.0F.WIG D2 /r VPSRLD *xmm1, xmm2, xmm3/m128*	C	V/V	AVX	Shift doublewords in *xmm2* right by amount specified in *xmm3/m128* while shifting in 0s.
VEX.NDD.128.66.0F.WIG 72 /2 ib VPSRLD *xmm1, xmm2, imm8*	D	V/V	AVX	Shift doublewords in *xmm2* right by *imm8* while shifting in 0s.
VEX.NDS.128.66.0F.WIG D3 /r VPSRLQ *xmm1, xmm2, xmm3/m128*	C	V/V	AVX	Shift quadwords in *xmm2* right by amount specified in *xmm3/m128* while shifting in 0s.
VEX.NDD.128.66.0F.WIG 73 /2 ib VPSRLQ *xmm1, xmm2, imm8*	D	V/V	AVX	Shift quadwords in *xmm2* right by *imm8* while shifting in 0s.
VEX.NDS.256.66.0F.WIG D1 /r VPSRLW *ymm1, ymm2, xmm3/m128*	C	V/V	AVX2	Shift words in *ymm2* right by amount specified in *xmm3/m128* while shifting in 0s.
VEX.NDD.256.66.0F.WIG 71 /2 ib VPSRLW *ymm1, ymm2, imm8*	D	V/V	AVX2	Shift words in *ymm2* right by *imm8* while shifting in 0s.

VEX.NDS.256.66.0F.WIG D2 /r VPSRLD *ymm1, ymm2, xmm3/m128*	C	V/V	AVX2	Shift doublewords in *ymm2* right by amount specified in *xmm3/m128* while shifting in 0s.
VEX.NDD.256.66.0F.WIG 72 /2 ib VPSRLD *ymm1, ymm2, imm8*	D	V/V	AVX2	Shift doublewords in *ymm2* right by *imm8* while shifting in 0s.
VEX.NDS.256.66.0F.WIG D3 /r VPSRLQ *ymm1, ymm2, xmm3/m128*	C	V/V	AVX2	Shift quadwords in *ymm2* right by amount specified in *xmm3/m128* while shifting in 0s.
VEX.NDD.256.66.0F.WIG 73 /2 ib VPSRLQ *ymm1, ymm2, imm8*	D	V/V	AVX2	Shift quadwords in *ymm2* right by *imm8* while shifting in 0s.
EVEX.NDS.128.66.0F.WIG D1 /r VPSRLW xmm1 {k1}{z}, xmm2, xmm3/m128	G	V/V	AVX512VL AVX512BW	Shift words in xmm2 right by amount specified in xmm3/m128 while shifting in 0s using writemask k1.
EVEX.NDS.256.66.0F.WIG D1 /r VPSRLW ymm1 {k1}{z}, ymm2, xmm3/m128	G	V/V	AVX512VL AVX512BW	Shift words in ymm2 right by amount specified in xmm3/m128 while shifting in 0s using writemask k1.
EVEX.NDS.512.66.0F.WIG D1 /r VPSRLW zmm1 {k1}{z}, zmm2, xmm3/m128	G	V/V	AVX512BW	Shift words in zmm2 right by amount specified in xmm3/m128 while shifting in 0s using writemask k1.
EVEX.NDD.128.66.0F.WIG 71 /2 ib VPSRLW xmm1 {k1}{z}, xmm2/m128, imm8	E	V/V	AVX512VL AVX512BW	Shift words in xmm2/m128 right by imm8 while shifting in 0s using writemask k1.
EVEX.NDD.256.66.0F.WIG 71 /2 ib VPSRLW ymm1 {k1}{z}, ymm2/m256, imm8	E	V/V	AVX512VL AVX512BW	Shift words in ymm2/m256 right by imm8 while shifting in 0s using writemask k1.
EVEX.NDD.512.66.0F.WIG 71 /2 ib VPSRLW zmm1 {k1}{z}, zmm2/m512, imm8	E	V/V	AVX512BW	Shift words in zmm2/m512 right by imm8 while shifting in 0s using writemask k1.
EVEX.NDS.128.66.0F.W0 D2 /r VPSRLD xmm1 {k1}{z}, xmm2, xmm3/m128	G	V/V	AVX512VL AVX512F	Shift doublewords in xmm2 right by amount specified in xmm3/m128 while shifting in 0s using writemask k1.
EVEX.NDS.256.66.0F.W0 D2 /r VPSRLD ymm1 {k1}{z}, ymm2, xmm3/m128	G	V/V	AVX512VL AVX512F	Shift doublewords in ymm2 right by amount specified in xmm3/m128 while shifting in 0s using writemask k1.
EVEX.NDS.512.66.0F.W0 D2 /r VPSRLD zmm1 {k1}{z}, zmm2, xmm3/m128	G	V/V	AVX512F	Shift doublewords in zmm2 right by amount specified in xmm3/m128 while shifting in 0s using writemask k1.
EVEX.NDD.128.66.0F.W0 72 /2 ib VPSRLD xmm1 {k1}{z}, xmm2/m128/m32bcst, imm8	F	V/V	AVX512VL AVX512F	Shift doublewords in xmm2/m128/m32bcst right by imm8 while shifting in 0s using writemask k1.
EVEX.NDD.256.66.0F.W0 72 /2 ib VPSRLD ymm1 {k1}{z}, ymm2/m256/m32bcst, imm8	F	V/V	AVX512VL AVX512F	Shift doublewords in ymm2/m256/m32bcst right by imm8 while shifting in 0s using writemask k1.
EVEX.NDD.512.66.0F.W0 72 /2 ib VPSRLD zmm1 {k1}{z}, zmm2/m512/m32bcst, imm8	F	V/V	AVX512F	Shift doublewords in zmm2/m512/m32bcst right by imm8 while shifting in 0s using writemask k1.
EVEX.NDS.128.66.0F.W1 D3 /r VPSRLQ xmm1 {k1}{z}, xmm2, xmm3/m128	G	V/V	AVX512VL AVX512F	Shift quadwords in xmm2 right by amount specified in xmm3/m128 while shifting in 0s using writemask k1.
EVEX.NDS.256.66.0F.W1 D3 /r VPSRLQ ymm1 {k1}{z}, ymm2, xmm3/m128	G	V/V	AVX512VL AVX512F	Shift quadwords in ymm2 right by amount specified in xmm3/m128 while shifting in 0s using writemask k1.
EVEX.NDS.512.66.0F.W1 D3 /r VPSRLQ zmm1 {k1}{z}, zmm2, xmm3/m128	G	V/V	AVX512F	Shift quadwords in zmm2 right by amount specified in xmm3/m128 while shifting in 0s using writemask k1.

EVEX.NDD.128.66.0F.W1 73 /2 ib VPSRLQ xmm1 {k1}{z}, xmm2/m128/m64bcst, imm8	F	V/V	AVX512VL AVX512F	Shift quadwords in xmm2/m128/m64bcst right by imm8 while shifting in 0s using writemask k1.
EVEX.NDD.256.66.0F.W1 73 /2 ib VPSRLQ ymm1 {k1}{z}, ymm2/m256/m64bcst, imm8	F	V/V	AVX512VL AVX512F	Shift quadwords in ymm2/m256/m64bcst right by imm8 while shifting in 0s using writemask k1.
EVEX.NDD.512.66.0F.W1 73 /2 ib VPSRLQ zmm1 {k1}{z}, zmm2/m512/m64bcst, imm8	F	V/V	AVX512F	Shift quadwords in zmm2/m512/m64bcst right by imm8 while shifting in 0s using writemask k1.

NOTES:

1. See note in Section 2.4, "AVX and SSE Instruction Exception Specification" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A* and Section 22.25.3, "Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	NA	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
B	NA	ModRM:r/m (r, w)	imm8	NA	NA
C	NA	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA
D	NA	VEX.vvvv (w)	ModRM:r/m (r)	imm8	NA
E	Full Vector Mem	EVEX.vvvv (w)	ModRM:r/m (R)	Imm8	NA
F	Full Vector	EVEX.vvvv (w)	ModRM:r/m (R)	Imm8	NA
G	Mem128	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA

Description

Shifts the bits in the individual data elements (words, doublewords, or quadword) in the destination operand (first operand) to the right by the number of bits specified in the count operand (second operand). As the bits in the data elements are shifted right, the empty high-order bits are cleared (set to 0). If the value specified by the count operand is greater than 15 (for words), 31 (for doublewords), or 63 (for a quadword), then the destination operand is set to all 0s. Figure 4-19 gives an example of shifting words in a 64-bit operand.

Note that only the low 64-bits of a 128-bit count operand are checked to compute the count.

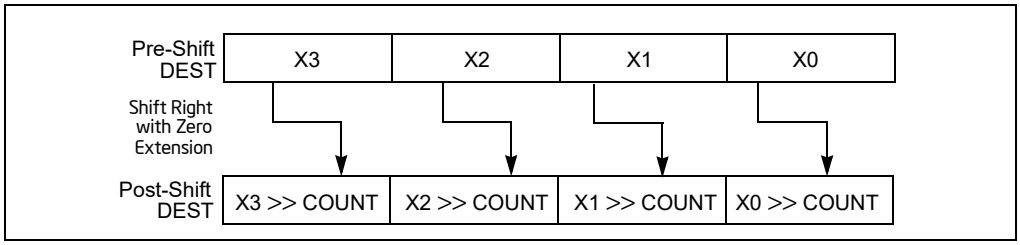


Figure 4-19. PSRLW, PSRLD, and PSRLQ Instruction Operation Using 64-bit Operand

The (V)PSRLW instruction shifts each of the words in the destination operand to the right by the number of bits specified in the count operand; the (V)PSRLD instruction shifts each of the doublewords in the destination operand; and the PSRLQ instruction shifts the quadword (or quadwords) in the destination operand.

In 64-bit mode and not encoded with VEX/EVEX, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

Legacy SSE instruction 64-bit operand: The destination operand is an MMX technology register; the count operand can be either an MMX technology register or an 64-bit memory location.

128-bit Legacy SSE version: The destination operand is an XMM register; the count operand can be either an XMM register or a 128-bit memory location, or an 8-bit immediate. If the count operand is a memory address, 128 bits are loaded but the upper 64 bits are ignored. Bits (MAXVL-1:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: The destination operand is an XMM register; the count operand can be either an XMM register or a 128-bit memory location, or an 8-bit immediate. If the count operand is a memory address, 128 bits are loaded but the upper 64 bits are ignored. Bits (MAXVL-1:128) of the destination YMM register are zeroed.

VEX.256 encoded version: The destination operand is a YMM register. The source operand is a YMM register or a memory location. The count operand can come either from an XMM register or a memory location or an 8-bit immediate. Bits (MAXVL-1:256) of the corresponding ZMM register are zeroed.

EVEX encoded versions: The destination operand is a ZMM register updated according to the writemask. The count operand is either an 8-bit immediate (the immediate count version) or an 8-bit value from an XMM register or a memory location (the variable count version). For the immediate count version, the source operand (the second operand) can be a ZMM register, a 512-bit memory location or a 512-bit vector broadcasted from a 32/64-bit memory location. For the variable count version, the first source operand (the second operand) is a ZMM register, the second source operand (the third operand, 8-bit variable count) can be an XMM register or a memory location.

Note: In VEX/EVEX encoded versions of shifts with an immediate count, vvvv of VEX/EVEX encode the destination register, and VEX.B/EVEX.B + ModRM.r/m encodes the source register.

Note: For shifts with an immediate count (VEX.128.66.0F 71-73 /2, or EVEX.128.66.0F 71-73 /2), VEX.vvvv/EVEX.vvvv encodes the destination register.

Operation

PSRLW (with 64-bit operand)

```
    IF (COUNT > 15)
    THEN
        DEST[64:0] ← 0000000000000000H
    ELSE
        DEST[15:0] ← ZeroExtend(DEST[15:0] >> COUNT);
        (* Repeat shift operation for 2nd and 3rd words *)
        DEST[63:48] ← ZeroExtend(DEST[63:48] >> COUNT);
    FI;
```

PSRLD (with 64-bit operand)

```
    IF (COUNT > 31)
    THEN
        DEST[64:0] ← 0000000000000000H
    ELSE
        DEST[31:0] ← ZeroExtend(DEST[31:0] >> COUNT);
        DEST[63:32] ← ZeroExtend(DEST[63:32] >> COUNT);
    FI;
```

PSRLQ (with 64-bit operand)

```
    IF (COUNT > 63)
    THEN
        DEST[64:0] ← 0000000000000000H
    ELSE
        DEST ← ZeroExtend(DEST >> COUNT);
    FI;
LOGICAL_RIGHT_SHIFT_DWORDS1(SRC, COUNT_SRC)
COUNT ← COUNT_SRC[63:0];
IF (COUNT > 31)
THEN
    DEST[31:0] ← 0
ELSE
```

```
    DEST[31:0] ← ZeroExtend(SRC[31:0] >> COUNT);
FI;

LOGICAL_RIGHT_SHIFT_QWORDS1(SRC, COUNT_SRC)
COUNT ← COUNT_SRC[63:0];
IF (COUNT > 63)
THEN
    DEST[63:0] ← 0
ELSE
    DEST[63:0] ← ZeroExtend(SRC[63:0] >> COUNT);
FI;
LOGICAL_RIGHT_SHIFT_WORDS_256b(SRC, COUNT_SRC)
COUNT ←COUNT_SRC[63:0];
IF (COUNT > 15)
THEN
    DEST[255:0] ←0
ELSE
    DEST[15:0] ←ZeroExtend(SRC[15:0] >> COUNT);
    (* Repeat shift operation for 2nd through 15th words *)
    DEST[255:240] ←ZeroExtend(SRC[255:240] >> COUNT);
FI;

LOGICAL_RIGHT_SHIFT_WORDS(SRC, COUNT_SRC)
COUNT ←COUNT_SRC[63:0];
IF (COUNT > 15)
THEN
    DEST[127:0] ←00000000000000000000000000000000H
ELSE
    DEST[15:0] ←ZeroExtend(SRC[15:0] >> COUNT);
    (* Repeat shift operation for 2nd through 7th words *)
    DEST[127:112] ←ZeroExtend(SRC[127:112] >> COUNT);
FI;

LOGICAL_RIGHT_SHIFT_DWORDS_256b(SRC, COUNT_SRC)
COUNT ←COUNT_SRC[63:0];
IF (COUNT > 31)
THEN
    DEST[255:0] ←0
ELSE
    DEST[31:0] ←ZeroExtend(SRC[31:0] >> COUNT);
    (* Repeat shift operation for 2nd through 3rd words *)
    DEST[255:224] ←ZeroExtend(SRC[255:224] >> COUNT);
FI;

LOGICAL_RIGHT_SHIFT_DWORDS(SRC, COUNT_SRC)
COUNT ←COUNT_SRC[63:0];
IF (COUNT > 31)
THEN
    DEST[127:0] ←00000000000000000000000000000000H
ELSE
    DEST[31:0] ←ZeroExtend(SRC[31:0] >> COUNT);
    (* Repeat shift operation for 2nd through 3rd words *)
    DEST[127:96] ←ZeroExtend(SRC[127:96] >> COUNT);
FI;
```

```
LOGICAL_RIGHT_SHIFT_QWORDS_256b(SRC, COUNT_SRC)
COUNT ←COUNT_SRC[63:0];
IF (COUNT > 63)
THEN
    DEST[255:0] ←0
ELSE
    DEST[63:0] ←ZeroExtend(SRC[63:0] >> COUNT);
    DEST[127:64] ←ZeroExtend(SRC[127:64] >> COUNT);
    DEST[191:128] ←ZeroExtend(SRC[191:128] >> COUNT);
    DEST[255:192] ←ZeroExtend(SRC[255:192] >> COUNT);
FI;

LOGICAL_RIGHT_SHIFT_QWORDS(SRC, COUNT_SRC)
COUNT ←COUNT_SRC[63:0];
IF (COUNT > 63)
THEN
    DEST[127:0] ←00000000000000000000000000000000H
ELSE
    DEST[63:0] ←ZeroExtend(SRC[63:0] >> COUNT);
    DEST[127:64] ←ZeroExtend(SRC[127:64] >> COUNT);
FI;
```

VPSRLW (EVEX versions, xmm/m128)

```
(KL, VL) = (8, 128), (16, 256), (32, 512)
IF VL = 128
    TMP_DEST[127:0] ← LOGICAL_RIGHT_SHIFT_WORDS_128b(SRC1[127:0], SRC2)
FI;
IF VL = 256
    TMP_DEST[255:0] ← LOGICAL_RIGHT_SHIFT_WORDS_256b(SRC1[255:0], SRC2)
FI;
IF VL = 512
    TMP_DEST[255:0] ← LOGICAL_RIGHT_SHIFT_WORDS_256b(SRC1[255:0], SRC2)
    TMP_DEST[511:256] ← LOGICAL_RIGHT_SHIFT_WORDS_256b(SRC1[511:256], SRC2)
FI;

FOR j ← 0 TO KL-1
    i ← j * 16
    IF k1[j] OR *no writemask*
        THEN DEST[i+15:i] ← TMP_DEST[i+15:i]
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+15:i] remains unchanged*
                ELSE *zeroing-masking*              ; zeroing-masking
                    DEST[i+15:i] = 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0
```

VPSRLW (EVEX versions, imm8)

```
(KL, VL) = (8, 128), (16, 256), (32, 512)
IF VL = 128
    TMP_DEST[127:0] ← LOGICAL_RIGHT_SHIFT_WORDS_128b(SRC1[127:0], imm8)
FI;
IF VL = 256
    TMP_DEST[255:0] ← LOGICAL_RIGHT_SHIFT_WORDS_256b(SRC1[255:0], imm8)
FI;
IF VL = 512
    TMP_DEST[255:0] ← LOGICAL_RIGHT_SHIFT_WORDS_256b(SRC1[255:0], imm8)
    TMP_DEST[511:256] ← LOGICAL_RIGHT_SHIFT_WORDS_256b(SRC1[511:256], imm8)
FI;

FOR j ← 0 TO KL-1
    i ← j * 16
    IF k1[j] OR *no writemask*
        THEN DEST[i+15:i] ← TMP_DEST[i+15:i]
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+15:i] remains unchanged*
                ELSE *zeroing-masking*              ; zeroing-masking
                    DEST[i+15:i] = 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0
```

VPSRLW (ymm, ymm, xmm/m128) - VEX.256 encoding

```
DEST[255:0] ←LOGICAL_RIGHT_SHIFT_WORDS_256b(SRC1, SRC2)
DEST[MAXVL-1:256] ←0;
```

VPSRLW (ymm, imm8) - VEX.256 encoding

```
DEST[255:0] ←LOGICAL_RIGHT_SHIFT_WORDS_256b(SRC1, imm8)
DEST[MAXVL-1:256] ←0;
```

VPSRLW (xmm, xmm, xmm/m128) - VEX.128 encoding

```
DEST[127:0] ←LOGICAL_RIGHT_SHIFT_WORDS(SRC1, SRC2)
DEST[MAXVL-1:128] ←0
```

VPSRLW (xmm, imm8) - VEX.128 encoding

```
DEST[127:0] ←LOGICAL_RIGHT_SHIFT_WORDS(SRC1, imm8)
DEST[MAXVL-1:128] ←0
```

PSRLW (xmm, xmm, xmm/m128)

```
DEST[127:0] ←LOGICAL_RIGHT_SHIFT_WORDS(DEST, SRC)
DEST[MAXVL-1:128] (Unmodified)
```

PSRLW (xmm, imm8)

```
DEST[127:0] ←LOGICAL_RIGHT_SHIFT_WORDS(DEST, imm8)
DEST[MAXVL-1:128] (Unmodified)
```

VPSRLD (EVEX versions, xmm/m128)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
IF VL = 128
    TMP_DEST[127:0] ← LOGICAL_RIGHT_SHIFT_DWORDS_128b(SRC1[127:0], SRC2)
FI;
IF VL = 256
    TMP_DEST[255:0] ← LOGICAL_RIGHT_SHIFT_DWORDS_256b(SRC1[255:0], SRC2)
FI;
IF VL = 512
    TMP_DEST[255:0] ← LOGICAL_RIGHT_SHIFT_DWORDS_256b(SRC1[255:0], SRC2)
    TMP_DEST[511:256] ← LOGICAL_RIGHT_SHIFT_DWORDS_256b(SRC1[511:256], SRC2)
FI;

FOR j ← 0 TO KL-1
    i ← j * 32
    IF k1[j] OR *no writemask*
        THEN DEST[i+31:i] ← TMP_DEST[i+31:i]
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE *zeroing-masking*              ; zeroing-masking
                    DEST[i+31:i] ← 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0
```

VPSRLD (EVEX versions, imm8)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j ← 0 TO KL-1
    i ← j * 32
    IF k1[j] OR *no writemask* THEN
            IF (EVEX.b = 1) AND (SRC1 *is memory*)
                THEN DEST[i+31:i] ← LOGICAL_RIGHT_SHIFT_DWORDS1(SRC1[31:0], imm8)
                ELSE DEST[i+31:i] ← LOGICAL_RIGHT_SHIFT_DWORDS1(SRC1[i+31:i], imm8)
            FI;
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE *zeroing-masking*              ; zeroing-masking
                    DEST[i+31:i] ← 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0
```

VPSRLD (ymm, ymm, xmm/m128) - VEX.256 encoding

```
DEST[255:0] ←LOGICAL_RIGHT_SHIFT_DWORDS_256b(SRC1, SRC2)
DEST[MAXVL-1:256] ←0;
```

VPSRLD (ymm, imm8) - VEX.256 encoding

```
DEST[255:0] ←LOGICAL_RIGHT_SHIFT_DWORDS_256b(SRC1, imm8)
DEST[MAXVL-1:256] ←0;
```

VPSRLD (xmm, xmm, xmm/m128) - VEX.128 encoding

```
DEST[127:0] ←LOGICAL_RIGHT_SHIFT_DWORDS(SRC1, SRC2)
DEST[MAXVL-1:128] ←0
```

VPSRLD (xmm, imm8) - VEX.128 encoding

```
DEST[127:0] ←LOGICAL_RIGHT_SHIFT_DWORDS(SRC1, imm8)
DEST[MAXVL-1:128] ←0
```

PSRLD (xmm, xmm, xmm/m128)

```
DEST[127:0] ←LOGICAL_RIGHT_SHIFT_DWORDS(DEST, SRC)
DEST[MAXVL-1:128] (Unmodified)
```

PSRLD (xmm, imm8)

```
DEST[127:0] ←LOGICAL_RIGHT_SHIFT_DWORDS(DEST, imm8)
DEST[MAXVL-1:128] (Unmodified)
```

VPSRLQ (EVEX versions, xmm/m128)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
TMP_DEST[255:0] ← LOGICAL_RIGHT_SHIFT_QWORDS_256b(SRC1[255:0], SRC2)
TMP_DEST[511:256] ← LOGICAL_RIGHT_SHIFT_QWORDS_256b(SRC1[511:256], SRC2)
IF VL = 128
    TMP_DEST[127:0] ← LOGICAL_RIGHT_SHIFT_QWORDS_128b(SRC1[127:0], SRC2)
FI;
IF VL = 256
    TMP_DEST[255:0] ← LOGICAL_RIGHT_SHIFT_QWORDS_256b(SRC1[255:0], SRC2)
FI;
IF VL = 512
    TMP_DEST[255:0] ← LOGICAL_RIGHT_SHIFT_QWORDS_256b(SRC1[255:0], SRC2)
    TMP_DEST[511:256] ← LOGICAL_RIGHT_SHIFT_QWORDS_256b(SRC1[511:256], SRC2)
FI;
FOR j ← 0 TO KL-1
    i ← j * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+63:i] ← TMP_DEST[i+63:i]
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE *zeroing-masking*              ; zeroing-masking
                    DEST[i+63:i] ← 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0
```

VPSRLQ (EVEX versions, imm8)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j ← 0 TO KL-1
    i ← j * 64
    IF k1[j] OR *no writemask* THEN
            IF (EVEX.b = 1) AND (SRC1 *is memory*)
                THEN DEST[i+63:i] ← LOGICAL_RIGHT_SHIFT_QWORDS1(SRC1[63:0], imm8)
                ELSE DEST[i+63:i] ← LOGICAL_RIGHT_SHIFT_QWORDS1(SRC1[i+63:i], imm8)
            FI;
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE *zeroing-masking*                  ; zeroing-masking
                    DEST[i+63:i] ← 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0
```

VPSRLQ (ymm, ymm, xmm/m128) - VEX.256 encoding

```
DEST[255:0] ←LOGICAL_RIGHT_SHIFT_QWORDS_256b(SRC1, SRC2)
DEST[MAXVL-1:256] ←0;
```

VPSRLQ (ymm, imm8) - VEX.256 encoding

```
DEST[255:0] ←LOGICAL_RIGHT_SHIFT_QWORDS_256b(SRC1, imm8)
DEST[MAXVL-1:256] ←0;
```

VPSRLQ (xmm, xmm, xmm/m128) - VEX.128 encoding

```
DEST[127:0] ←LOGICAL_RIGHT_SHIFT_QWORDS(SRC1, SRC2)
DEST[MAXVL-1:128] ←0
```

VPSRLQ (xmm, imm8) - VEX.128 encoding

```
DEST[127:0] ←LOGICAL_RIGHT_SHIFT_QWORDS(SRC1, imm8)
DEST[MAXVL-1:128] ←0
```

PSRLQ (xmm, xmm, xmm/m128)

```
DEST[127:0] ←LOGICAL_RIGHT_SHIFT_QWORDS(DEST, SRC)
DEST[MAXVL-1:128] (Unmodified)
```

PSRLQ (xmm, imm8)

```
DEST[127:0] ←LOGICAL_RIGHT_SHIFT_QWORDS(DEST, imm8)
DEST[MAXVL-1:128] (Unmodified)
```

Intel C/C++ Compiler Intrinsic Equivalents

VPSRLD __m512i _mm512_srli_epi32(__m512i a, unsigned int imm);
VPSRLD __m512i _mm512_mask_srli_epi32(__m512i s, __mmask16 k, __m512i a, unsigned int imm);
VPSRLD __m512i _mm512_maskz_srli_epi32(__mmask16 k, __m512i a, unsigned int imm);
VPSRLD __m256i _mm256_mask_srli_epi32(__m256i s, __mmask8 k, __m256i a, unsigned int imm);
VPSRLD __m256i _mm256_maskz_srli_epi32(__mmask8 k, __m256i a, unsigned int imm);
VPSRLD __m128i _mm_mask_srli_epi32(__m128i s, __mmask8 k, __m128i a, unsigned int imm);
VPSRLD __m128i _mm_maskz_srli_epi32(__mmask8 k, __m128i a, unsigned int imm);
VPSRLD __m512i _mm512_srl_epi32(__m512i a, __m128i cnt);
VPSRLD __m512i _mm512_mask_srl_epi32(__m512i s, __mmask16 k, __m512i a, __m128i cnt);
VPSRLD __m512i _mm512_maskz_srl_epi32(__mmask16 k, __m512i a, __m128i cnt);
VPSRLD __m256i _mm256_mask_srl_epi32(__m256i s, __mmask8 k, __m256i a, __m128i cnt);

VPSRLD __m256i _mm256_maskz_srl_epi32(__mmask8 k, __m256i a, __m128i cnt);
VPSRLD __m128i _mm_mask_srl_epi32(__m128i s, __mmask8 k, __m128i a, __m128i cnt);
VPSRLD __m128i _mm_maskz_srl_epi32(__mmask8 k, __m128i a, __m128i cnt);
VPSRLQ __m512i _mm512_srli_epi64(__m512i a, unsigned int imm);
VPSRLQ __m512i _mm512_mask_srli_epi64(__m512i s, __mmask8 k, __m512i a, unsigned int imm);
VPSRLQ __m512i _mm512_mask_srli_epi64(__mmask8 k, __m512i a, unsigned int imm);
VPSRLQ __m256i _mm256_mask_srli_epi64(__m256i s, __mmask8 k, __m256i a, unsigned int imm);
VPSRLQ __m256i _mm256_maskz_srli_epi64(__mmask8 k, __m256i a, unsigned int imm);
VPSRLQ __m128i _mm_mask_srli_epi64(__m128i s, __mmask8 k, __m128i a, unsigned int imm);
VPSRLQ __m128i _mm_maskz_srli_epi64(__mmask8 k, __m128i a, unsigned int imm);
VPSRLQ __m512i _mm512_srl_epi64(__m512i a, __m128i cnt);
VPSRLQ __m512i _mm512_mask_srl_epi64(__m512i s, __mmask8 k, __m512i a, __m128i cnt);
VPSRLQ __m512i _mm512_mask_srl_epi64(__mmask8 k, __m512i a, __m128i cnt);
VPSRLQ __m256i _mm256_mask_srl_epi64(__m256i s, __mmask8 k, __m256i a, __m128i cnt);
VPSRLQ __m256i _mm256_maskz_srl_epi64(__mmask8 k, __m256i a, __m128i cnt);
VPSRLQ __m128i _mm_mask_srl_epi64(__m128i s, __mmask8 k, __m128i a, __m128i cnt);
VPSRLQ __m128i _mm_maskz_srl_epi64(__mmask8 k, __m128i a, __m128i cnt);
VPSRLW __m512i _mm512_srli_epi16(__m512i a, unsigned int imm);
VPSRLW __m512i _mm512_mask_srli_epi16(__m512i s, __mmask32 k, __m512i a, unsigned int imm);
VPSRLW __m512i _mm512_maskz_srli_epi16(__mmask32 k, __m512i a, unsigned int imm);
VPSRLW __m256i _mm256_mask_srlii_epi16(__m256i s, __mmask16 k, __m256i a, unsigned int imm);
VPSRLW __m256i _mm256_maskz_srli_epi16(__mmask16 k, __m256i a, unsigned int imm);
VPSRLW __m128i _mm_mask_srli_epi16(__m128i s, __mmask8 k, __m128i a, unsigned int imm);
VPSRLW __m128i _mm_maskz_srli_epi16(__mmask8 k, __m128i a, unsigned int imm);
VPSRLW __m512i _mm512_srl_epi16(__m512i a, __m128i cnt);
VPSRLW __m512i _mm512_mask_srl_epi16(__m512i s, __mmask32 k, __m512i a, __m128i cnt);
VPSRLW __m512i _mm512_maskz_srl_epi16(__mmask32 k, __m512i a, __m128i cnt);
VPSRLW __m256i _mm256_mask_srl_epi16(__m256i s, __mmask16 k, __m256i a, __m128i cnt);
VPSRLW __m256i _mm256_maskz_srl_epi16(__mmask8 k, __mmask16 a, __m128i cnt);
VPSRLW __m128i _mm_mask_srl_epi16(__m128i s, __mmask8 k, __m128i a, __m128i cnt);
VPSRLW __m128i _mm_maskz_srl_epi16(__mmask8 k, __m128i a, __m128i cnt);
PSRLW:__m64 _mm_srli_pi16(__m64 m, int count)
PSRLW:__m64 _mm_srl_pi16 (__m64 m, __m64 count)
(V)PSRLW:__m128i _mm_srli_epi16 (__m128i m, int count)
(V)PSRLW:__m128i _mm_srl_epi16 (__m128i m, __m128i count)
VPSRLW:__m256i _mm256_srli_epi16 (__m256i m, int count)
VPSRLW:__m256i _mm256_srl_epi16 (__m256i m, __m128i count)
PSRLD:__m64 _mm_srli_pi32 (__m64 m, int count)
PSRLD:__m64 _mm_srl_pi32 (__m64 m, __m64 count)
(V)PSRLD:__m128i _mm_srli_epi32 (__m128i m, int count)
(V)PSRLD:__m128i _mm_srl_epi32 (__m128i m, __m128i count)
VPSRLD:__m256i _mm256_srli_epi32 (__m256i m, int count)
VPSRLD:__m256i _mm256_srl_epi32 (__m256i m, __m128i count)
PSRLQ:__m64 _mm_srli_si64 (__m64 m, int count)
PSRLQ:__m64 _mm_srl_si64 (__m64 m, __m64 count)
(V)PSRLQ:__m128i _mm_srli_epi64 (__m128i m, int count)
(V)PSRLQ:__m128i _mm_srl_epi64 (__m128i m, __m128i count)
VPSRLQ:__m256i _mm256_srli_epi64 (__m256i m, int count)
VPSRLQ:__m256i _mm256_srl_epi64 (__m256i m, __m128i count)

Flags Affected

None.

Numeric Exceptions

None.

Other Exceptions

VEX-encoded instructions:

Syntax with RM/RVM operand encoding, see Exceptions Type 4.
Syntax with MI/VMI operand encoding, see Exceptions Type 7.

EVEX-encoded VPSRLW, see Exceptions Type E4NF.nb.

EVEX-encoded VPSRLD/Q:

Syntax with M128 operand encoding, see Exceptions Type E4NF.nb.
Syntax with FVI operand encoding, see Exceptions Type E4.

PSUBB/PSUBW/PSUBD—Subtract Packed Integers

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F F8 /r[1] PSUBB *mm, mm/m64*	A	V/V	MMX	Subtract packed byte integers in *mm/m64* from packed byte integers in *mm*.
66 0F F8 /r PSUBB *xmm1, xmm2/m128*	A	V/V	SSE2	Subtract packed byte integers in *xmm2/m128* from packed byte integers in *xmm1*.
NP 0F F9 /r[1] PSUBW *mm, mm/m64*	A	V/V	MMX	Subtract packed word integers in *mm/m64* from packed word integers in *mm*.
66 0F F9 /r PSUBW *xmm1, xmm2/m128*	A	V/V	SSE2	Subtract packed word integers in *xmm2/m128* from packed word integers in *xmm1*.
NP 0F FA /r[1] PSUBD *mm, mm/m64*	A	V/V	MMX	Subtract packed doubleword integers in *mm/m64* from packed doubleword integers in *mm*.
66 0F FA /r PSUBD *xmm1, xmm2/m128*	A	V/V	SSE2	Subtract packed doubleword integers in *xmm2/mem128* from packed doubleword integers in *xmm1*.
VEX.NDS.128.66.0F.WIG F8 /r VPSUBB *xmm1, xmm2, xmm3/m128*	B	V/V	AVX	Subtract packed byte integers in *xmm3/m128* from *xmm2*.
VEX.NDS.128.66.0F.WIG F9 /r VPSUBW *xmm1, xmm2, xmm3/m128*	B	V/V	AVX	Subtract packed word integers in *xmm3/m128* from *xmm2*.
VEX.NDS.128.66.0F.WIG FA /r VPSUBD *xmm1, xmm2, xmm3/m128*	B	V/V	AVX	Subtract packed doubleword integers in *xmm3/m128* from *xmm2*.
VEX.NDS.256.66.0F.WIG F8 /r VPSUBB *ymm1, ymm2, ymm3/m256*	B	V/V	AVX2	Subtract packed byte integers in *ymm3/m256* from *ymm2*.
VEX.NDS.256.66.0F.WIG F9 /r VPSUBW *ymm1, ymm2, ymm3/m256*	B	V/V	AVX2	Subtract packed word integers in *ymm3/m256* from *ymm2*.
VEX.NDS.256.66.0F.WIG FA /r VPSUBD *ymm1, ymm2, ymm3/m256*	B	V/V	AVX2	Subtract packed doubleword integers in *ymm3/m256* from *ymm2*.
EVEX.NDS.128.66.0F.WIG F8 /r VPSUBB xmm1 {k1}{z}, xmm2, xmm3/m128	C	V/V	AVX512VL AVX512BW	Subtract packed byte integers in xmm3/m128 from xmm2 and store in xmm1 using writemask k1.
EVEX.NDS.256.66.0F.WIG F8 /r VPSUBB ymm1 {k1}{z}, ymm2, ymm3/m256	C	V/V	AVX512VL AVX512BW	Subtract packed byte integers in ymm3/m256 from ymm2 and store in ymm1 using writemask k1.
EVEX.NDS.512.66.0F.WIG F8 /r VPSUBB zmm1 {k1}{z}, zmm2, zmm3/m512	C	V/V	AVX512BW	Subtract packed byte integers in zmm3/m512 from zmm2 and store in zmm1 using writemask k1.
EVEX.NDS.128.66.0F.WIG F9 /r VPSUBW xmm1 {k1}{z}, xmm2, xmm3/m128	C	V/V	AVX512VL AVX512BW	Subtract packed word integers in xmm3/m128 from xmm2 and store in xmm1 using writemask k1.
EVEX.NDS.256.66.0F.WIG F9 /r VPSUBW ymm1 {k1}{z}, ymm2, ymm3/m256	C	V/V	AVX512VL AVX512BW	Subtract packed word integers in ymm3/m256 from ymm2 and store in ymm1 using writemask k1.
EVEX.NDS.512.66.0F.WIG F9 /r VPSUBW zmm1 {k1}{z}, zmm2, zmm3/m512	C	V/V	AVX512BW	Subtract packed word integers in zmm3/m512 from zmm2 and store in zmm1 using writemask k1.

EVEX.NDS.128.66.0F.W0 FA /r VPSUBD xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst	D	V/V	AVX512VL AVX512F	Subtract packed doubleword integers in xmm3/m128/m32bcst from xmm2 and store in xmm1 using writemask k1.
EVEX.NDS.256.66.0F.W0 FA /r VPSUBD ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst	D	V/V	AVX512VL AVX512F	Subtract packed doubleword integers in ymm3/m256/m32bcst from ymm2 and store in ymm1 using writemask k1.
EVEX.NDS.512.66.0F.W0 FA /r VPSUBD zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst	D	V/V	AVX512F	Subtract packed doubleword integers in zmm3/m512/m32bcst from zmm2 and store in zmm1 using writemask k1

NOTES:

1. See note in Section 2.4, "AVX and SSE Instruction Exception Specification" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A* and Section 22.25.3, "Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	NA	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
B	NA	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA
C	Full Vector Mem	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA
D	Full Vector	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA

Description

Performs a SIMD subtract of the packed integers of the source operand (second operand) from the packed integers of the destination operand (first operand), and stores the packed integer results in the destination operand. See Figure 9-4 in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*, for an illustration of a SIMD operation. Overflow is handled with wraparound, as described in the following paragraphs.

The (V)PSUBB instruction subtracts packed byte integers. When an individual result is too large or too small to be represented in a byte, the result is wrapped around and the low 8 bits are written to the destination element.

The (V)PSUBW instruction subtracts packed word integers. When an individual result is too large or too small to be represented in a word, the result is wrapped around and the low 16 bits are written to the destination element.

The (V)PSUBD instruction subtracts packed doubleword integers. When an individual result is too large or too small to be represented in a doubleword, the result is wrapped around and the low 32 bits are written to the destination element.

Note that the (V)PSUBB, (V)PSUBW, and (V)PSUBD instructions can operate on either unsigned or signed (two's complement notation) packed integers; however, it does not set bits in the EFLAGS register to indicate overflow and/or a carry. To prevent undetected overflow conditions, software must control the ranges of values upon which it operates.

In 64-bit mode and not encoded with VEX/EVEX, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

Legacy SSE version 64-bit operand: The destination operand must be an MMX technology register and the source operand can be either an MMX technology register or a 64-bit memory location.

128-bit Legacy SSE version: The second source operand is an XMM register or a 128-bit memory location. The first source operand and destination operands are XMM registers. Bits (MAXVL-1:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: The second source operand is an XMM register or a 128-bit memory location. The first source operand and destination operands are XMM registers. Bits (MAXVL-1:128) of the destination YMM register are zeroed.

VEX.256 encoded versions: The second source operand is an YMM register or an 256-bit memory location. The first source operand and destination operands are YMM registers. Bits (MAXVL-1:256) of the corresponding ZMM register are zeroed.

EVEX encoded VPSUBD: The second source operand is a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 32/64-bit memory location. The first source operand and destination operands are ZMM/YMM/XMM registers. The destination is conditionally updated with writemask k1.

EVEX encoded VPSUBB/W: The second source operand is a ZMM/YMM/XMM register, a 512/256/128-bit memory location. The first source operand and destination operands are ZMM/YMM/XMM registers. The destination is conditionally updated with writemask k1.

Operation

PSUBB (with 64-bit operands)

```
DEST[7:0] ← DEST[7:0] – SRC[7:0];
(* Repeat subtract operation for 2nd through 7th byte *)
DEST[63:56] ← DEST[63:56] – SRC[63:56];
```

PSUBW (with 64-bit operands)

```
DEST[15:0] ← DEST[15:0] – SRC[15:0];
(* Repeat subtract operation for 2nd and 3rd word *)
DEST[63:48] ← DEST[63:48] – SRC[63:48];
```

PSUBD (with 64-bit operands)

```
DEST[31:0] ← DEST[31:0] – SRC[31:0];
DEST[63:32] ← DEST[63:32] – SRC[63:32];
```

PSUBD (with 128-bit operands)

```
DEST[31:0]  ← DEST[31:0] – SRC[31:0];
(* Repeat subtract operation for 2nd and 3rd doubleword *)
DEST[127:96] ← DEST[127:96] – SRC[127:96];
```

VPSUBB (EVEX encoded versions)

```
(KL, VL) = (16, 128), (32, 256), (64, 512)
FOR j ← 0 TO KL-1
    i ← j * 8
    IF k1[j] OR *no writemask*
        THEN DEST[i+7:i] ← SRC1[i+7:i] - SRC2[i+7:i]
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+7:i] remains unchanged*
                ELSE *zeroing-masking*              ; zeroing-masking
                    DEST[i+7:i] = 0
            FI
    FI;
ENDFOR;
DEST[MAXVL-1:VL] ← 0
```

VPSUBW (EVEX encoded versions)

```
(KL, VL) = (8, 128), (16, 256), (32, 512)
FOR j ← 0 TO KL-1
    i ← j * 16
    IF k1[j] OR *no writemask*
        THEN DEST[i+15:i] ← SRC1[i+15:i] - SRC2[i+15:i]
        ELSE
            IF *merging-masking*                    ; merging-masking
```

```
                THEN *DEST[i+15:i] remains unchanged*
                ELSE *zeroing-masking*                  ; zeroing-masking
                    DEST[i+15:i] = 0
            FI
    FI;
ENDFOR;
DEST[MAXVL-1:VL] ← 0
```

VPSUBD (EVEX encoded versions)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j ← 0 TO KL-1
    i ← j * 32
    IF k1[j] OR *no writemask* THEN
            IF (EVEX.b = 1) AND (SRC2 *is memory*)
                THEN DEST[i+31:i] ← SRC1[i+31:i] - SRC2[31:0]
                ELSE DEST[i+31:i] ← SRC1[i+31:i] - SRC2[i+31:i]
            FI;
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE *zeroing-masking*                  ; zeroing-masking
                    DEST[i+31:i] ← 0
            FI
    FI;
ENDFOR;
DEST[MAXVL-1:VL] ← 0
```

VPSUBB (VEX.256 encoded version)

```
DEST[7:0] ←SRC1[7:0]-SRC2[7:0]
DEST[15:8] ←SRC1[15:8]-SRC2[15:8]
DEST[23:16] ←SRC1[23:16]-SRC2[23:16]
DEST[31:24] ←SRC1[31:24]-SRC2[31:24]
DEST[39:32] ←SRC1[39:32]-SRC2[39:32]
DEST[47:40] ←SRC1[47:40]-SRC2[47:40]
DEST[55:48] ←SRC1[55:48]-SRC2[55:48]
DEST[63:56] ←SRC1[63:56]-SRC2[63:56]
DEST[71:64] ←SRC1[71:64]-SRC2[71:64]
DEST[79:72] ←SRC1[79:72]-SRC2[79:72]
DEST[87:80] ←SRC1[87:80]-SRC2[87:80]
DEST[95:88] ←SRC1[95:88]-SRC2[95:88]
DEST[103:96] ←SRC1[103:96]-SRC2[103:96]
DEST[111:104] ←SRC1[111:104]-SRC2[111:104]
DEST[119:112] ←SRC1[119:112]-SRC2[119:112]
DEST[127:120] ←SRC1[127:120]-SRC2[127:120]
DEST[135:128] ←SRC1[135:128]-SRC2[135:128]
DEST[143:136] ←SRC1[143:136]-SRC2[143:136]
DEST[151:144] ←SRC1[151:144]-SRC2[151:144]
DEST[159:152] ←SRC1[159:152]-SRC2[159:152]
DEST[167:160] ←SRC1[167:160]-SRC2[167:160]
DEST[175:168] ←SRC1[175:168]-SRC2[175:168]
DEST[183:176] ←SRC1[183:176]-SRC2[183:176]
DEST[191:184] ←SRC1[191:184]-SRC2[191:184]
DEST[199:192] ←SRC1[199:192]-SRC2[199:192]
DEST[207:200] ←SRC1[207:200]-SRC2[207:200]
```

DEST[215:208] ←SRC1[215:208]-SRC2[215:208]
DEST[223:216] ←SRC1[223:216]-SRC2[223:216]
DEST[231:224] ←SRC1[231:224]-SRC2[231:224]
DEST[239:232] ←SRC1[239:232]-SRC2[239:232]
DEST[247:240] ←SRC1[247:240]-SRC2[247:240]
DEST[255:248] ←SRC1[255:248]-SRC2[255:248]
DEST[MAXVL-1:256] ←0

VPSUBB (VEX.128 encoded version)
DEST[7:0] ←SRC1[7:0]-SRC2[7:0]
DEST[15:8] ←SRC1[15:8]-SRC2[15:8]
DEST[23:16] ←SRC1[23:16]-SRC2[23:16]
DEST[31:24] ←SRC1[31:24]-SRC2[31:24]
DEST[39:32] ←SRC1[39:32]-SRC2[39:32]
DEST[47:40] ←SRC1[47:40]-SRC2[47:40]
DEST[55:48] ←SRC1[55:48]-SRC2[55:48]
DEST[63:56] ←SRC1[63:56]-SRC2[63:56]
DEST[71:64] ←SRC1[71:64]-SRC2[71:64]
DEST[79:72] ←SRC1[79:72]-SRC2[79:72]
DEST[87:80] ←SRC1[87:80]-SRC2[87:80]
DEST[95:88] ←SRC1[95:88]-SRC2[95:88]
DEST[103:96] ←SRC1[103:96]-SRC2[103:96]
DEST[111:104] ←SRC1[111:104]-SRC2[111:104]
DEST[119:112] ←SRC1[119:112]-SRC2[119:112]
DEST[127:120] ←SRC1[127:120]-SRC2[127:120]
DEST[MAXVL-1:128] ←0

PSUBB (128-bit Legacy SSE version)
DEST[7:0] ←DEST[7:0]-SRC[7:0]
DEST[15:8] ←DEST[15:8]-SRC[15:8]
DEST[23:16] ←DEST[23:16]-SRC[23:16]
DEST[31:24] ←DEST[31:24]-SRC[31:24]
DEST[39:32] ←DEST[39:32]-SRC[39:32]
DEST[47:40] ←DEST[47:40]-SRC[47:40]
DEST[55:48] ←DEST[55:48]-SRC[55:48]
DEST[63:56] ←DEST[63:56]-SRC[63:56]
DEST[71:64] ←DEST[71:64]-SRC[71:64]
DEST[79:72] ←DEST[79:72]-SRC[79:72]
DEST[87:80] ←DEST[87:80]-SRC[87:80]
DEST[95:88] ←DEST[95:88]-SRC[95:88]
DEST[103:96] ←DEST[103:96]-SRC[103:96]
DEST[111:104] ←DEST[111:104]-SRC[111:104]
DEST[119:112] ←DEST[119:112]-SRC[119:112]
DEST[127:120] ←DEST[127:120]-SRC[127:120]
DEST[MAXVL-1:128] (Unmodified)

VPSUBW (VEX.256 encoded version)
DEST[15:0] ←SRC1[15:0]-SRC2[15:0]
DEST[31:16] ←SRC1[31:16]-SRC2[31:16]
DEST[47:32] ←SRC1[47:32]-SRC2[47:32]
DEST[63:48] ←SRC1[63:48]-SRC2[63:48]
DEST[79:64] ←SRC1[79:64]-SRC2[79:64]
DEST[95:80] ←SRC1[95:80]-SRC2[95:80]
DEST[111:96] ←SRC1[111:96]-SRC2[111:96]

DEST[127:112] ←SRC1[127:112]-SRC2[127:112]
DEST[143:128] ←SRC1[143:128]-SRC2[143:128]
DEST[159:144] ←SRC1[159:144]-SRC2[159:144]
DEST[175:160] ←SRC1[175:160]-SRC2[175:160]
DEST[191:176] ←SRC1[191:176]-SRC2[191:176]
DEST[207:192] ←SRC1207:192]-SRC2[207:192]
DEST[223:208] ←SRC1[223:208]-SRC2[223:208]
DEST[239:224] ←SRC1[239:224]-SRC2[239:224]
DEST[255:240] ←SRC1[255:240]-SRC2[255:240]
DEST[MAXVL-1:256] ←0

VPSUBW (VEX.128 encoded version)
DEST[15:0] ←SRC1[15:0]-SRC2[15:0]
DEST[31:16] ←SRC1[31:16]-SRC2[31:16]
DEST[47:32] ←SRC1[47:32]-SRC2[47:32]
DEST[63:48] ←SRC1[63:48]-SRC2[63:48]
DEST[79:64] ←SRC1[79:64]-SRC2[79:64]
DEST[95:80] ←SRC1[95:80]-SRC2[95:80]
DEST[111:96] ←SRC1[111:96]-SRC2[111:96]
DEST[127:112] ←SRC1[127:112]-SRC2[127:112]
DEST[MAXVL-1:128] ←0

PSUBW (128-bit Legacy SSE version)
DEST[15:0] ←DEST[15:0]-SRC[15:0]
DEST[31:16] ←DEST[31:16]-SRC[31:16]
DEST[47:32] ←DEST[47:32]-SRC[47:32]
DEST[63:48] ←DEST[63:48]-SRC[63:48]
DEST[79:64] ←DEST[79:64]-SRC[79:64]
DEST[95:80] ←DEST[95:80]-SRC[95:80]
DEST[111:96] ←DEST[111:96]-SRC[111:96]
DEST[127:112] ←DEST[127:112]-SRC[127:112]
DEST[MAXVL-1:128] (Unmodified)

VPSUBD (VEX.256 encoded version)
DEST[31:0] ←SRC1[31:0]-SRC2[31:0]
DEST[63:32] ←SRC1[63:32]-SRC2[63:32]
DEST[95:64] ←SRC1[95:64]-SRC2[95:64]
DEST[127:96] ←SRC1[127:96]-SRC2[127:96]
DEST[159:128] ←SRC1[159:128]-SRC2[159:128]
DEST[191:160] ←SRC1[191:160]-SRC2[191:160]
DEST[223:192] ←SRC1[223:192]-SRC2[223:192]
DEST[255:224] ←SRC1[255:224]-SRC2[255:224]
DEST[MAXVL-1:256] ←0

VPSUBD (VEX.128 encoded version)
DEST[31:0] ←SRC1[31:0]-SRC2[31:0]
DEST[63:32] ←SRC1[63:32]-SRC2[63:32]
DEST[95:64] ←SRC1[95:64]-SRC2[95:64]
DEST[127:96] ←SRC1[127:96]-SRC2[127:96]
DEST[MAXVL-1:128] ←0

PSUBD (128-bit Legacy SSE version)
DEST[31:0] ←DEST[31:0]-SRC[31:0]
DEST[63:32] ←DEST[63:32]-SRC[63:32]

DEST[95:64] ←DEST[95:64]-SRC[95:64]
DEST[127:96] ←DEST[127:96]-SRC[127:96]
DEST[MAXVL-1:128] (Unmodified)

Intel C/C++ Compiler Intrinsic Equivalents

VPSUBB __m512i _mm512_sub_epi8(__m512i a, __m512i b);
VPSUBB __m512i _mm512_mask_sub_epi8(__m512i s, __mmask64 k, __m512i a, __m512i b);
VPSUBB __m512i _mm512_maskz_sub_epi8(__mmask64 k, __m512i a, __m512i b);
VPSUBB __m256i _mm256_mask_sub_epi8(__m256i s, __mmask32 k, __m256i a, __m256i b);
VPSUBB __m256i _mm256_maskz_sub_epi8(__mmask32 k, __m256i a, __m256i b);
VPSUBB __m128i _mm_mask_sub_epi8(__m128i s, __mmask16 k, __m128i a, __m128i b);
VPSUBB __m128i _mm_maskz_sub_epi8(__mmask16 k, __m128i a, __m128i b);
VPSUBW __m512i _mm512_sub_epi16(__m512i a, __m512i b);
VPSUBW __m512i _mm512_mask_sub_epi16(__m512i s, __mmask32 k, __m512i a, __m512i b);
VPSUBW __m512i _mm512_maskz_sub_epi16(__mmask32 k, __m512i a, __m512i b);
VPSUBW __m256i _mm256_mask_sub_epi16(__m256i s, __mmask16 k, __m256i a, __m256i b);
VPSUBW __m256i _mm256_maskz_sub_epi16(__mmask16 k, __m256i a, __m256i b);
VPSUBW __m128i _mm_mask_sub_epi16(__m128i s, __mmask8 k, __m128i a, __m128i b);
VPSUBW __m128i _mm_maskz_sub_epi16(__mmask8 k, __m128i a, __m128i b);
VPSUBD __m512i _mm512_sub_epi32(__m512i a, __m512i b);
VPSUBD __m512i _mm512_mask_sub_epi32(__m512i s, __mmask16 k, __m512i a, __m512i b);
VPSUBD __m512i _mm512_maskz_sub_epi32(__mmask16 k, __m512i a, __m512i b);
VPSUBD __m256i _mm256_mask_sub_epi32(__m256i s, __mmask8 k, __m256i a, __m256i b);
VPSUBD __m256i _mm256_maskz_sub_epi32(__mmask8 k, __m256i a, __m256i b);
VPSUBD __m128i _mm_mask_sub_epi32(__m128i s, __mmask8 k, __m128i a, __m128i b);
VPSUBD __m128i _mm_maskz_sub_epi32(__mmask8 k, __m128i a, __m128i b);
PSUBB:__m64 _mm_sub_pi8(__m64 m1, __m64 m2)
(V)PSUBB:__m128i _mm_sub_epi8 (__m128i a, __m128i b)
VPSUBB:__m256i _mm256_sub_epi8 (__m256i a, __m256i b)
PSUBW:__m64 _mm_sub_pi16(__m64 m1, __m64 m2)
(V)PSUBW:__m128i _mm_sub_epi16 (__m128i a, __m128i b)
VPSUBW:__m256i _mm256_sub_epi16 (__m256i a, __m256i b)
PSUBD:__m64 _mm_sub_pi32(__m64 m1, __m64 m2)
(V)PSUBD:__m128i _mm_sub_epi32 (__m128i a, __m128i b)
VPSUBD:__m256i _mm256_sub_epi32 (__m256i a, __m256i b)

Flags Affected

None.

Numeric Exceptions

None.

Other Exceptions

Non-EVEX-encoded instruction, see Exceptions Type 4.

EVEX-encoded VPSUBD, see Exceptions Type E4.

EVEX-encoded VPSUBB/W, see Exceptions Type E4.nb.

PSUBQ—Subtract Packed Quadword Integers

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F FB /r[1] PSUBQ *mm1, mm2/m64*	A	V/V	SSE2	Subtract quadword integer in *mm1* from *mm2 /m64*.
66 0F FB /r PSUBQ *xmm1, xmm2/m128*	A	V/V	SSE2	Subtract packed quadword integers in *xmm1* from *xmm2 /m128*.
VEX.NDS.128.66.0F.WIG FB/r VPSUBQ *xmm1, xmm2, xmm3/m128*	B	V/V	AVX	Subtract packed quadword integers in *xmm3/m128* from *xmm2*.
VEX.NDS.256.66.0F.WIG FB /r VPSUBQ *ymm1, ymm2, ymm3/m256*	B	V/V	AVX2	Subtract packed quadword integers in *ymm3/m256* from *ymm2*.
EVEX.NDS.128.66.0F.W1 FB /r VPSUBQ xmm1 {k1}{z}, xmm2, xmm3/m128/m64bcst	C	V/V	AVX512VL AVX512F	Subtract packed quadword integers in xmm3/m128/m64bcst from xmm2 and store in xmm1 using writemask k1.
EVEX.NDS.256.66.0F.W1 FB /r VPSUBQ ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst	C	V/V	AVX512VL AVX512F	Subtract packed quadword integers in ymm3/m256/m64bcst from ymm2 and store in ymm1 using writemask k1.
EVEX.NDS.512.66.0F.W1 FB/r VPSUBQ zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst	C	V/V	AVX512F	Subtract packed quadword integers in zmm3/m512/m64bcst from zmm2 and store in zmm1 using writemask k1.

NOTES:

1. See note in Section 2.4, "AVX and SSE Instruction Exception Specification" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A* and Section 22.25.3, "Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	NA	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
B	NA	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA
C	Full Vector	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA

Description

Subtracts the second operand (source operand) from the first operand (destination operand) and stores the result in the destination operand. When packed quadword operands are used, a SIMD subtract is performed. When a quadword result is too large to be represented in 64 bits (overflow), the result is wrapped around and the low 64 bits are written to the destination element (that is, the carry is ignored).

Note that the (V)PSUBQ instruction can operate on either unsigned or signed (two's complement notation) integers; however, it does not set bits in the EFLAGS register to indicate overflow and/or a carry. To prevent undetected overflow conditions, software must control the ranges of the values upon which it operates.

In 64-bit mode and not encoded with VEX/EVEX, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

Legacy SSE version 64-bit operand: The source operand can be a quadword integer stored in an MMX technology register or a 64-bit memory location.

128-bit Legacy SSE version: The second source operand is an XMM register or a 128-bit memory location. The first source operand and destination operands are XMM registers. Bits (MAXVL-1:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: The second source operand is an XMM register or a 128-bit memory location. The first source operand and destination operands are XMM registers. Bits (MAXVL-1:128) of the destination YMM register are zeroed.

VEX.256 encoded versions: The second source operand is an YMM register or an 256-bit memory location. The first source operand and destination operands are YMM registers. Bits (MAXVL-1:256) of the corresponding ZMM register are zeroed.

EVEX encoded VPSUBQ: The second source operand is a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 32/64-bit memory location. The first source operand and destination operands are ZMM/YMM/XMM registers. The destination is conditionally updated with writemask k1.

Operation

PSUBQ (with 64-Bit operands)

```
DEST[63:0] ← DEST[63:0] – SRC[63:0];
```

PSUBQ (with 128-Bit operands)

```
DEST[63:0] ← DEST[63:0] – SRC[63:0];
DEST[127:64] ← DEST[127:64] – SRC[127:64];
```

VPSUBQ (VEX.128 encoded version)

```
DEST[63:0] ← SRC1[63:0]-SRC2[63:0]
DEST[127:64] ← SRC1[127:64]-SRC2[127:64]
DEST[MAXVL-1:128] ← 0
```

VPSUBQ (VEX.256 encoded version)

```
DEST[63:0] ← SRC1[63:0]-SRC2[63:0]
DEST[127:64] ← SRC1[127:64]-SRC2[127:64]
DEST[191:128] ← SRC1[191:128]-SRC2[191:128]
DEST[255:192] ← SRC1[255:192]-SRC2[255:192]
DEST[MAXVL-1:256] ← 0
```

VPSUBQ (EVEX encoded versions)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j ← 0 TO KL-1
    i ← j * 64
    IF k1[j] OR *no writemask* THEN
            IF (EVEX.b = 1) AND (SRC2 *is memory*)
                THEN DEST[i+63:i] ← SRC1[i+63:i] - SRC2[63:0]
                ELSE DEST[i+63:i] ← SRC1[i+63:i] - SRC2[i+63:i]
            FI;
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE *zeroing-masking*                  ; zeroing-masking
                    DEST[i+63:i] ← 0
            FI
    FI;
ENDFOR;
DEST[MAXVL-1:VL] ← 0
```

Intel C/C++ Compiler Intrinsic Equivalents

VPSUBQ __m512i _mm512_sub_epi64(__m512i a, __m512i b);
VPSUBQ __m512i _mm512_mask_sub_epi64(__m512i s, __mmask8 k, __m512i a, __m512i b);
VPSUBQ __m512i _mm512_maskz_sub_epi64(__mmask8 k, __m512i a, __m512i b);
VPSUBQ __m256i _mm256_mask_sub_epi64(__m256i s, __mmask8 k, __m256i a, __m256i b);
VPSUBQ __m256i _mm256_maskz_sub_epi64(__mmask8 k, __m256i a, __m256i b);
VPSUBQ __m128i _mm_mask_sub_epi64(__m128i s, __mmask8 k, __m128i a, __m128i b);
VPSUBQ __m128i _mm_maskz_sub_epi64(__mmask8 k, __m128i a, __m128i b);
PSUBQ:__m64 _mm_sub_si64(__m64 m1, __m64 m2)
(V)PSUBQ:__m128i _mm_sub_epi64(__m128i m1, __m128i m2)
VPSUBQ:__m256i _mm256_sub_epi64(__m256i m1, __m256i m2)

Flags Affected

None.

Numeric Exceptions

None.

Other Exceptions

Non-EVEX-encoded instruction, see Exceptions Type 4.

EVEX-encoded VPSUBQ, see Exceptions Type E4.

PSUBSB/PSUBSW—Subtract Packed Signed Integers with Signed Saturation

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F E8 /r[1] PSUBSB *mm, mm/m64*	A	V/V	MMX	Subtract signed packed bytes in *mm/m64* from signed packed bytes in *mm* and saturate results.
66 0F E8 /r PSUBSB *xmm1, xmm2/m128*	A	V/V	SSE2	Subtract packed signed byte integers in *xmm2/m128* from packed signed byte integers in *xmm1* and saturate results.
NP 0F E9 /r[1] PSUBSW *mm, mm/m64*	A	V/V	MMX	Subtract signed packed words in *mm/m64* from signed packed words in *mm* and saturate results.
66 0F E9 /r PSUBSW *xmm1, xmm2/m128*	A	V/V	SSE2	Subtract packed signed word integers in *xmm2/m128* from packed signed word integers in *xmm1* and saturate results.
VEX.NDS.128.66.0F.WIG E8 /r VPSUBSB *xmm1, xmm2, xmm3/m128*	B	V/V	AVX	Subtract packed signed byte integers in *xmm3/m128* from packed signed byte integers in *xmm2* and saturate results.
VEX.NDS.128.66.0F.WIG E9 /r VPSUBSW *xmm1, xmm2, xmm3/m128*	B	V/V	AVX	Subtract packed signed word integers in *xmm3/m128* from packed signed word integers in *xmm2* and saturate results.
VEX.NDS.256.66.0F.WIG E8 /r VPSUBSB *ymm1, ymm2, ymm3/m256*	B	V/V	AVX2	Subtract packed signed byte integers in *ymm3/m256* from packed signed byte integers in *ymm2* and saturate results.
VEX.NDS.256.66.0F.WIG E9 /r VPSUBSW *ymm1, ymm2, ymm3/m256*	B	V/V	AVX2	Subtract packed signed word integers in *ymm3/m256* from packed signed word integers in *ymm2* and saturate results.
EVEX.NDS.128.66.0F.WIG E8 /r VPSUBSB xmm1 {k1}{z}, xmm2, xmm3/m128	C	V/V	AVX512VL AVX512BW	Subtract packed signed byte integers in xmm3/m128 from packed signed byte integers in xmm2 and saturate results and store in xmm1 using writemask k1.
EVEX.NDS.256.66.0F.WIG E8 /r VPSUBSB ymm1 {k1}{z}, ymm2, ymm3/m256	C	V/V	AVX512VL AVX512BW	Subtract packed signed byte integers in ymm3/m256 from packed signed byte integers in ymm2 and saturate results and store in ymm1 using writemask k1.
EVEX.NDS.512.66.0F.WIG E8 /r VPSUBSB zmm1 {k1}{z}, zmm2, zmm3/m512	C	V/V	AVX512BW	Subtract packed signed byte integers in zmm3/m512 from packed signed byte integers in zmm2 and saturate results and store in zmm1 using writemask k1.
EVEX.NDS.128.66.0F.WIG E9 /r VPSUBSW xmm1 {k1}{z}, xmm2, xmm3/m128	C	V/V	AVX512VL AVX512BW	Subtract packed signed word integers in xmm3/m128 from packed signed word integers in xmm2 and saturate results and store in xmm1 using writemask k1.
EVEX.NDS.256.66.0F.WIG E9 /r VPSUBSW ymm1 {k1}{z}, ymm2, ymm3/m256	C	V/V	AVX512VL AVX512BW	Subtract packed signed word integers in ymm3/m256 from packed signed word integers in ymm2 and saturate results and store in ymm1 using writemask k1.

EVEX.NDS.512.66.0F.WIG E9 /r VPSUBSW zmm1 {k1}{z}, zmm2, zmm3/m512	C	V/V	AVX512BW	Subtract packed signed word integers in zmm3/m512 from packed signed word integers in zmm2 and saturate results and store in zmm1 using writemask k1.

NOTES:

1. See note in Section 2.4, "AVX and SSE Instruction Exception Specification" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A* and Section 22.25.3, "Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	NA	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
B	NA	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA
C	Full Vector Mem	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA

Description

Performs a SIMD subtract of the packed signed integers of the source operand (second operand) from the packed signed integers of the destination operand (first operand), and stores the packed integer results in the destination operand. See Figure 9-4 in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*, for an illustration of a SIMD operation. Overflow is handled with signed saturation, as described in the following paragraphs.

The (V)PSUBSB instruction subtracts packed signed byte integers. When an individual byte result is beyond the range of a signed byte integer (that is, greater than 7FH or less than 80H), the saturated value of 7FH or 80H, respectively, is written to the destination operand.

The (V)PSUBSW instruction subtracts packed signed word integers. When an individual word result is beyond the range of a signed word integer (that is, greater than 7FFFH or less than 8000H), the saturated value of 7FFFH or 8000H, respectively, is written to the destination operand.

In 64-bit mode and not encoded with VEX/EVEX, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

Legacy SSE version 64-bit operand: The destination operand must be an MMX technology register and the source operand can be either an MMX technology register or a 64-bit memory location.

128-bit Legacy SSE version: The second source operand is an XMM register or a 128-bit memory location. The first source operand and destination operands are XMM registers. Bits (MAXVL-1:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: The second source operand is an XMM register or a 128-bit memory location. The first source operand and destination operands are XMM registers. Bits (MAXVL-1:128) of the destination YMM register are zeroed.

VEX.256 encoded versions: The second source operand is an YMM register or an 256-bit memory location. The first source operand and destination operands are YMM registers. Bits (MAXVL-1:256) of the corresponding ZMM register are zeroed.

EVEX encoded version: The second source operand is an ZMM/YMM/XMM register or an 512/256/128-bit memory location. The first source operand and destination operands are ZMM/YMM/XMM registers. The destination is conditionally updated with writemask k1.

Operation

PSUBSB (with 64-bit operands)

```
DEST[7:0] ← SaturateToSignedByte (DEST[7:0] – SRC (7:0]);
(* Repeat subtract operation for 2nd through 7th bytes *)
DEST[63:56] ← SaturateToSignedByte (DEST[63:56] – SRC[63:56] );
```

PSUBSW (with 64-bit operands)

```
    DEST[15:0] ← SaturateToSignedWord (DEST[15:0] – SRC[15:0] );
    (* Repeat subtract operation for 2nd and 7th words *)
    DEST[63:48] ← SaturateToSignedWord (DEST[63:48] – SRC[63:48] );
```

VPSUBSB (EVEX encoded versions)

```
(KL, VL) = (16, 128), (32, 256), (64, 512)
FOR j ← 0 TO KL-1
    i ← j * 8;
    IF k1[j] OR *no writemask*
        THEN DEST[i+7:i] ← SaturateToSignedByte (SRC1[i+7:i] - SRC2[i+7:i])
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+7:i] remains unchanged*
                ELSE *zeroing-masking*              ; zeroing-masking
                    DEST[i+7:i] ← 0;
            FI
    FI;
ENDFOR;
DEST[MAXVL-1:VL] ← 0
```

VPSUBSW (EVEX encoded versions)

```
(KL, VL) = (8, 128), (16, 256), (32, 512)
FOR j ← 0 TO KL-1
    i ← j * 16
    IF k1[j] OR *no writemask*
        THEN DEST[i+15:i] ← SaturateToSignedWord (SRC1[i+15:i] - SRC2[i+15:i])
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+15:i] remains unchanged*
                ELSE *zeroing-masking*              ; zeroing-masking
                    DEST[i+15:i] ← 0;
            FI
    FI;
ENDFOR;
DEST[MAXVL-1:VL] ← 0;
```

VPSUBSB (VEX.256 encoded version)

```
DEST[7:0] ← SaturateToSignedByte (SRC1[7:0] - SRC2[7:0]);
(* Repeat subtract operation for 2nd through 31th bytes *)
DEST[255:248] ← SaturateToSignedByte (SRC1[255:248] - SRC2[255:248]);
DEST[MAXVL-1:256] ←0;
```

VPSUBSB (VEX.128 encoded version)

```
DEST[7:0] ← SaturateToSignedByte (SRC1[7:0] - SRC2[7:0]);
(* Repeat subtract operation for 2nd through 14th bytes *)
DEST[127:120] ← SaturateToSignedByte (SRC1[127:120] - SRC2[127:120]);
DEST[MAXVL-1:128] ← 0;
```

PSUBSB (128-bit Legacy SSE Version)

```
DEST[7:0] ← SaturateToSignedByte (DEST[7:0] - SRC[7:0]);
(* Repeat subtract operation for 2nd through 14th bytes *)
DEST[127:120] ← SaturateToSignedByte (DEST[127:120] - SRC[127:120]);
DEST[MAXVL-1:128] (Unmodified);
```

VPSUBSW (VEX.256 encoded version)
DEST[15:0] ← SaturateToSignedWord (SRC1[15:0] - SRC2[15:0]);
(* Repeat subtract operation for 2nd through 15th words *)
DEST[255:240] ← SaturateToSignedWord (SRC1[255:240] - SRC2[255:240]);
DEST[MAXVL-1:256] ← 0;

VPSUBSW (VEX.128 encoded version)
DEST[15:0] ← SaturateToSignedWord (SRC1[15:0] - SRC2[15:0]);
(* Repeat subtract operation for 2nd through 7th words *)
DEST[127:112] ← SaturateToSignedWord (SRC1[127:112] - SRC2[127:112]);
DEST[MAXVL-1:128] ← 0;

PSUBSW (128-bit Legacy SSE Version)
DEST[15:0] ← SaturateToSignedWord (DEST[15:0] - SRC[15:0]);
(* Repeat subtract operation for 2nd through 7th words *)
DEST[127:112] ← SaturateToSignedWord (DEST[127:112] - SRC[127:112]);
DEST[MAXVL-1:128] (Unmodified);

Intel C/C++ Compiler Intrinsic Equivalents

VPSUBSB __m512i _mm512_subs_epi8(__m512i a, __m512i b);
VPSUBSB __m512i _mm512_mask_subs_epi8(__m512i s, __mmask64 k, __m512i a, __m512i b);
VPSUBSB __m512i _mm512_maskz_subs_epi8(__mmask64 k, __m512i a, __m512i b);
VPSUBSB __m256i _mm256_mask_subs_epi8(__m256i s, __mmask32 k, __m256i a, __m256i b);
VPSUBSB __m256i _mm256_maskz_subs_epi8(__mmask32 k, __m256i a, __m256i b);
VPSUBSB __m128i _mm_mask_subs_epi8(__m128i s, __mmask16 k, __m128i a, __m128i b);
VPSUBSB __m128i _mm_maskz_subs_epi8(__mmask16 k, __m128i a, __m128i b);
VPSUBSW __m512i _mm512_subs_epi16(__m512i a, __m512i b);
VPSUBSW __m512i _mm512_mask_subs_epi16(__m512i s, __mmask32 k, __m512i a, __m512i b);
VPSUBSW __m512i _mm512_maskz_subs_epi16(__mmask32 k, __m512i a, __m512i b);
VPSUBSW __m256i _mm256_mask_subs_epi16(__m256i s, __mmask16 k, __m256i a, __m256i b);
VPSUBSW __m256i _mm256_maskz_subs_epi16(__mmask16 k, __m256i a, __m256i b);
VPSUBSW __m128i _mm_mask_subs_epi16(__m128i s, __mmask8 k, __m128i a, __m128i b);
VPSUBSW __m128i _mm_maskz_subs_epi16(__mmask8 k, __m128i a, __m128i b);
PSUBSB:__m64 _mm_subs_pi8(__m64 m1, __m64 m2)
(V)PSUBSB:__m128i _mm_subs_epi8(__m128i m1, __m128i m2)
VPSUBSB:__m256i _mm256_subs_epi8(__m256i m1, __m256i m2)
PSUBSW:__m64 _mm_subs_pi16(__m64 m1, __m64 m2)
(V)PSUBSW:__m128i _mm_subs_epi16(__m128i m1, __m128i m2)
VPSUBSW:__m256i _mm256_subs_epi16(__m256i m1, __m256i m2)

Flags Affected

None.

Numeric Exceptions

None.

Other Exceptions

Non-EVEX-encoded instruction, see Exceptions Type 4.
EVEX-encoded instruction, see Exceptions Type E4.nb.

PSUBUSB/PSUBUSW—Subtract Packed Unsigned Integers with Unsigned Saturation

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F D8 /r[1] PSUBUSB *mm, mm/m64*	A	V/V	MMX	Subtract unsigned packed bytes in *mm/m64* from unsigned packed bytes in *mm* and saturate result.
66 0F D8 /r PSUBUSB *xmm1, xmm2/m128*	A	V/V	SSE2	Subtract packed unsigned byte integers in *xmm2/m128* from packed unsigned byte integers in xmm1 and saturate result.
NP 0F D9 /r[1] PSUBUSW *mm, mm/m64*	A	V/V	MMX	Subtract unsigned packed words in *mm/m64* from unsigned packed words in *mm* and saturate result.
66 0F D9 /r PSUBUSW *xmm1, xmm2/m128*	A	V/V	SSE2	Subtract packed unsigned word integers in *xmm2/m128* from packed unsigned word integers in *xmm1* and saturate result.
VEX.NDS.128.66.0F.WIG D8 /r VPSUBUSB *xmm1, xmm2, xmm3/m128*	B	V/V	AVX	Subtract packed unsigned byte integers in *xmm3/m128* from packed unsigned byte integers in *xmm2* and saturate result.
VEX.NDS.128.66.0F.WIG D9 /r VPSUBUSW *xmm1, xmm2, xmm3/m128*	B	V/V	AVX	Subtract packed unsigned word integers in *xmm3/m128* from packed unsigned word integers in *xmm2* and saturate result.
VEX.NDS.256.66.0F.WIG D8 /r VPSUBUSB *ymm1, ymm2, ymm3/m256*	B	V/V	AVX2	Subtract packed unsigned byte integers in *ymm3/m256* from packed unsigned byte integers in *ymm2* and saturate result.
VEX.NDS.256.66.0F.WIG D9 /r VPSUBUSW *ymm1, ymm2, ymm3/m256*	B	V/V	AVX2	Subtract packed unsigned word integers in *ymm3/m256* from packed unsigned word integers in *ymm2* and saturate result.
EVEX.NDS.128.66.0F.WIG D8 /r VPSUBUSB xmm1 {k1}{z}, xmm2, xmm3/m128	C	V/V	AVX512VL AVX512BW	Subtract packed unsigned byte integers in xmm3/m128 from packed unsigned byte integers in xmm2, saturate results and store in xmm1 using writemask k1.
EVEX.NDS.256.66.0F.WIG D8 /r VPSUBUSB ymm1 {k1}{z}, ymm2, ymm3/m256	C	V/V	AVX512VL AVX512BW	Subtract packed unsigned byte integers in ymm3/m256 from packed unsigned byte integers in ymm2, saturate results and store in ymm1 using writemask k1.
EVEX.NDS.512.66.0F.WIG D8 /r VPSUBUSB zmm1 {k1}{z}, zmm2, zmm3/m512	C	V/V	AVX512BW	Subtract packed unsigned byte integers in zmm3/m512 from packed unsigned byte integers in zmm2, saturate results and store in zmm1 using writemask k1.
EVEX.NDS.128.66.0F.WIG D9 /r VPSUBUSW xmm1 {k1}{z}, xmm2, xmm3/m128	C	V/V	AVX512VL AVX512BW	Subtract packed unsigned word integers in xmm3/m128 from packed unsigned word integers in xmm2 and saturate results and store in xmm1 using writemask k1.
EVEX.NDS.256.66.0F.WIG D9 /r VPSUBUSW ymm1 {k1}{z}, ymm2, ymm3/m256	C	V/V	AVX512VL AVX512BW	Subtract packed unsigned word integers in ymm3/m256 from packed unsigned word integers in ymm2, saturate results and store in ymm1 using writemask k1.

EVEX.NDS.512.66.0F.WIG D9 /r VPSUBUSW zmm1 {k1}{z}, zmm2, zmm3/m512	C	V/V	AVX512BW	Subtract packed unsigned word integers in zmm3/m512 from packed unsigned word integers in zmm2, saturate results and store in zmm1 using writemask k1.

NOTES:

1. See note in Section 2.4, "AVX and SSE Instruction Exception Specification" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A* and Section 22.25.3, "Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	NA	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
B	NA	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA
C	Full Vector Mem	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA

Description

Performs a SIMD subtract of the packed unsigned integers of the source operand (second operand) from the packed unsigned integers of the destination operand (first operand), and stores the packed unsigned integer results in the destination operand. See Figure 9-4 in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*, for an illustration of a SIMD operation. Overflow is handled with unsigned saturation, as described in the following paragraphs.

These instructions can operate on either 64-bit or 128-bit operands.

The (V)PSUBUSB instruction subtracts packed unsigned byte integers. When an individual byte result is less than zero, the saturated value of 00H is written to the destination operand.

The (V)PSUBUSW instruction subtracts packed unsigned word integers. When an individual word result is less than zero, the saturated value of 0000H is written to the destination operand.

In 64-bit mode and not encoded with VEX/EVEX, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

Legacy SSE version 64-bit operand: The destination operand must be an MMX technology register and the source operand can be either an MMX technology register or a 64-bit memory location.

128-bit Legacy SSE version: The second source operand is an XMM register or a 128-bit memory location. The first source operand and destination operands are XMM registers. Bits (MAXVL-1:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: The second source operand is an XMM register or a 128-bit memory location. The first source operand and destination operands are XMM registers. Bits (MAXVL-1:128) of the destination YMM register are zeroed.

VEX.256 encoded versions: The second source operand is an YMM register or an 256-bit memory location. The first source operand and destination operands are YMM registers. Bits (MAXVL-1:256) of the corresponding ZMM register are zeroed.

EVEX encoded version: The second source operand is an ZMM/YMM/XMM register or an 512/256/128-bit memory location. The first source operand and destination operands are ZMM/YMM/XMM registers. The destination is conditionally updated with writemask k1.

Operation

PSUBUSB (with 64-bit operands)

```
DEST[7:0] ← SaturateToUnsignedByte (DEST[7:0] – SRC (7:0] );
(* Repeat add operation for 2nd through 7th bytes *)
DEST[63:56] ← SaturateToUnsignedByte (DEST[63:56] – SRC[63:56];
```

PSUBUSW (with 64-bit operands)

```
    DEST[15:0] ← SaturateToUnsignedWord (DEST[15:0] – SRC[15:0] );
    (* Repeat add operation for 2nd and 3rd words *)
    DEST[63:48] ← SaturateToUnsignedWord (DEST[63:48] – SRC[63:48] );
```

VPSUBUSB (EVEX encoded versions)

```
(KL, VL) = (16, 128), (32, 256), (64, 512)
FOR j ← 0 TO KL-1
    i ← j * 8;
    IF k1[j] OR *no writemask*
        THEN DEST[i+7:i] ← SaturateToUnsignedByte (SRC1[i+7:i] - SRC2[i+7:i])
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+7:i] remains unchanged*
                ELSE *zeroing-masking*              ; zeroing-masking
                    DEST[i+7:i] ← 0;
            FI
    FI;
ENDFOR;
DEST[MAXVL-1:VL] ← 0;
```

VPSUBUSW (EVEX encoded versions)

```
(KL, VL) = (8, 128), (16, 256), (32, 512)
FOR j ← 0 TO KL-1
    i ← j * 16;
    IF k1[j] OR *no writemask*
        THEN DEST[i+15:i] ← SaturateToUnsignedWord (SRC1[i+15:i] - SRC2[i+15:i])
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+15:i] remains unchanged*
                ELSE *zeroing-masking*              ; zeroing-masking
                    DEST[i+15:i] ← 0;
            FI
    FI;
ENDFOR;
DEST[MAXVL-1:VL] ← 0;
```

VPSUBUSB (VEX.256 encoded version)

```
DEST[7:0] ← SaturateToUnsignedByte (SRC1[7:0] - SRC2[7:0]);
(* Repeat subtract operation for 2nd through 31st bytes *)
DEST[255:148] ← SaturateToUnsignedByte (SRC1[255:248] - SRC2[255:248]);
DEST[MAXVL-1:256] ← 0;
```

VPSUBUSB (VEX.128 encoded version)

```
DEST[7:0] ← SaturateToUnsignedByte (SRC1[7:0] - SRC2[7:0]);
(* Repeat subtract operation for 2nd through 14th bytes *)
DEST[127:120] ← SaturateToUnsignedByte (SRC1[127:120] - SRC2[127:120]);
DEST[MAXVL-1:128] ← 0
```

PSUBUSB (128-bit Legacy SSE Version)

```
DEST[7:0] ← SaturateToUnsignedByte (DEST[7:0] - SRC[7:0]);
(* Repeat subtract operation for 2nd through 14th bytes *)
DEST[127:120] ← SaturateToUnsignedByte (DEST[127:120] - SRC[127:120]);
DEST[MAXVL-1:128] (Unmodified)
```

VPSUBUSW (VEX.256 encoded version)
DEST[15:0] ← SaturateToUnsignedWord (SRC1[15:0] - SRC2[15:0]);
(* Repeat subtract operation for 2nd through 15th words *)
DEST[255:240] ← SaturateToUnsignedWord (SRC1[255:240] - SRC2[255:240]);
DEST[MAXVL-1:256] ← 0;

VPSUBUSW (VEX.128 encoded version)
DEST[15:0] ← SaturateToUnsignedWord (SRC1[15:0] - SRC2[15:0]);
(* Repeat subtract operation for 2nd through 7th words *)
DEST[127:112] ← SaturateToUnsignedWord (SRC1[127:112] - SRC2[127:112]);
DEST[MAXVL-1:128] ← 0

PSUBUSW (128-bit Legacy SSE Version)
DEST[15:0] ← SaturateToUnsignedWord (DEST[15:0] - SRC[15:0]);
(* Repeat subtract operation for 2nd through 7th words *)
DEST[127:112] ← SaturateToUnsignedWord (DEST[127:112] - SRC[127:112]);
DEST[MAXVL-1:128] (Unmodified)

Intel C/C++ Compiler Intrinsic Equivalents

VPSUBUSB __m512i _mm512_subs_epu8(__m512i a, __m512i b);
VPSUBUSB __m512i _mm512_mask_subs_epu8(__m512i s, __mmask64 k, __m512i a, __m512i b);
VPSUBUSB __m512i _mm512_maskz_subs_epu8(__mmask64 k, __m512i a, __m512i b);
VPSUBUSB __m256i _mm256_mask_subs_epu8(__m256i s, __mmask32 k, __m256i a, __m256i b);
VPSUBUSB __m256i _mm256_maskz_subs_epu8(__mmask32 k, __m256i a, __m256i b);
VPSUBUSB __m128i _mm_mask_subs_epu8(__m128i s, __mmask16 k, __m128i a, __m128i b);
VPSUBUSB __m128i _mm_maskz_subs_epu8(__mmask16 k, __m128i a, __m128i b);
VPSUBUSW __m512i _mm512_subs_epu16(__m512i a, __m512i b);
VPSUBUSW __m512i _mm512_mask_subs_epu16(__m512i s, __mmask32 k, __m512i a, __m512i b);
VPSUBUSW __m512i _mm512_maskz_subs_epu16(__mmask32 k, __m512i a, __m512i b);
VPSUBUSW __m256i _mm256_mask_subs_epu16(__m256i s, __mmask16 k, __m256i a, __m256i b);
VPSUBUSW __m256i _mm256_maskz_subs_epu16(__mmask16 k, __m256i a, __m256i b);
VPSUBUSW __m128i _mm_mask_subs_epu16(__m128i s, __mmask8 k, __m128i a, __m128i b);
VPSUBUSW __m128i _mm_maskz_subs_epu16(__mmask8 k, __m128i a, __m128i b);
PSUBUSB:__m64 _mm_subs_pu8(__m64 m1, __m64 m2)
(V)PSUBUSB:__m128i _mm_subs_epu8(__m128i m1, __m128i m2)
VPSUBUSB:__m256i _mm256_subs_epu8(__m256i m1, __m256i m2)
PSUBUSW:__m64 _mm_subs_pu16(__m64 m1, __m64 m2)
(V)PSUBUSW:__m128i _mm_subs_epu16(__m128i m1, __m128i m2)
VPSUBUSW:__m256i _mm256_subs_epu16(__m256i m1, __m256i m2)

Flags Affected

None.

Numeric Exceptions

None.

Other Exceptions

Non-EVEX-encoded instruction, see Exceptions Type 4.
EVEX-encoded instruction, see Exceptions Type E4.

PTEST- Logical Compare

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 38 17 /r PTEST *xmm1, xmm2/m128*	RM	V/V	SSE4_1	Set ZF if *xmm2/m128* AND *xmm1* result is all 0s. Set CF if *xmm2/m128* AND NOT *xmm1* result is all 0s.
VEX.128.66.0F38.WIG 17 /r VPTEST *xmm1, xmm2/m128*	RM	V/V	AVX	Set ZF and CF depending on bitwise AND and ANDN of sources.
VEX.256.66.0F38.WIG 17 /r VPTEST *ymm1, ymm2/m256*	RM	V/V	AVX	Set ZF and CF depending on bitwise AND and ANDN of sources.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r)	ModRM:r/m (r)	NA	NA

Description

PTEST and VPTEST set the ZF flag if all bits in the result are 0 of the bitwise AND of the first source operand (first operand) and the second source operand (second operand). VPTEST sets the CF flag if all bits in the result are 0 of the bitwise AND of the second source operand (second operand) and the logical NOT of the destination operand.

The first source register is specified by the ModR/M *reg* field.

128-bit versions: The first source register is an XMM register. The second source register can be an XMM register or a 128-bit memory location. The destination register is not modified.

VEX.256 encoded version: The first source register is a YMM register. The second source register can be a YMM register or a 256-bit memory location. The destination register is not modified.

Note: In VEX-encoded versions, VEX.vvvv is reserved and must be 1111b, otherwise instructions will #UD.

Operation

(V)PTEST (128-bit version)

```
IF (SRC[127:0] BITWISE AND DEST[127:0] = 0)
    THEN ZF ← 1;
    ELSE ZF ← 0;
IF (SRC[127:0] BITWISE AND NOT DEST[127:0] = 0)
    THEN CF ← 1;
    ELSE CF ← 0;
DEST (unmodified)
AF ← OF ← PF ← SF ← 0;
```

VPTEST (VEX.256 encoded version)

```
IF (SRC[255:0] BITWISE AND DEST[255:0] = 0) THEN ZF ← 1;
    ELSE ZF ← 0;
IF (SRC[255:0] BITWISE AND NOT DEST[255:0] = 0) THEN CF ← 1;
    ELSE CF ← 0;
DEST (unmodified)
AF ← OF ← PF ← SF ← 0;
```

Intel C/C++ Compiler Intrinsic Equivalent

PTEST

int _mm_testz_si128 (__m128i s1, __m128i s2);

int _mm_testc_si128 (__m128i s1, __m128i s2);

int _mm_testnzc_si128 (__m128i s1, __m128i s2);

VPTEST

int _mm256_testz_si256 (__m256i s1, __m256i s2);

int _mm256_testc_si256 (__m256i s1, __m256i s2);

int _mm256_testnzc_si256 (__m256i s1, __m256i s2);

int _mm_testz_si128 (__m128i s1, __m128i s2);

int _mm_testc_si128 (__m128i s1, __m128i s2);

int _mm_testnzc_si128 (__m128i s1, __m128i s2);

Flags Affected

The 0F, AF, PF, SF flags are cleared and the ZF, CF flags are set according to the operation.

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Exceptions Type 4; additionally

#UD	If VEX.vvvv ≠ 1111B.

PTWRITE - Write Data to a Processor Trace Packet

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
F3 REX.W 0F AE /4 PTWRITE *r64/m64*	RM	V/N.E		Reads the data from r64/m64 to encod into a PTW packet if dependencies are met (see details below).
F3 0F AE /4 PTWRITE *r32/m32*	RM	V/V		Reads the data from r32/m32 to encode into a PTW packet if dependencies are met (see details below).

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:rm (r)	NA	NA	NA

Description

This instruction reads data in the source operand and sends it to the Intel Processor Trace hardware to be encoded in a PTW packet if TriggerEn, ContextEn, FilterEn, and PTWEn are all set to 1. For more details on these values, see *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3C*, Section 35.2.3, "Power Event Tracing". The size of data is 64-bit if using REX.W in 64-bit mode, otherwise 32-bits of data are copied from the source operand.

Note: The instruction will #UD if prefix 66H is used.

Operation

```
IF (IA32_RTIT_STATUS.TriggerEn & IA32_RTIT_STATUS.ContextEn & IA32_RTIT_STATUS.FilterEn & IA32_RTIT_CTL.PTWEn) = 1
    PTW.PayloadBytes ← Encoded payload size;
    PTW.IP ← IA32_RTIT_CTL.FUPonPTW
    IF IA32_RTIT_CTL.FUPonPTW = 1
        Insert FUP packet with IP of PTWRITE;
    FI;
FI;
```

Flags Affected

None.

Other Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS or GS segments.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF (fault-code)	For a page fault.
#AC(0)	If an unaligned memory reference is made while the current privilege level is 3 and alignment checking is enabled.
#UD	If CPUID.(EAX=14H, ECX=0):EBX.PTWRITE [Bit 4] = 0.
	If LOCK prefix is used.
	If 66H prefix is used.

Real-Address Mode Exceptions

#GP(0)	If any part of the operand lies outside of the effective address space from 0 to 0FFFFH.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#UD	If CPUID.(EAX=14H, ECX=0):EBX.PTWRITE [Bit 4] = 0.
	If LOCK prefix is used.
	If 66H prefix is used.

Virtual 8086 Mode Exceptions

#GP(0)	If any part of the operand lies outside of the effective address space from 0 to 0FFFFH.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF (fault-code)	For a page fault.
#AC(0)	If an unaligned memory reference is made while alignment checking is enabled.
#UD	If CPUID.(EAX=14H, ECX=0):EBX.PTWRITE [Bit 4] = 0.
	If LOCK prefix is used.
	If 66H prefix is used.

Compatibility Mode Exceptions

Same exceptions as in Protected Mode.

64-Bit Mode Exceptions

#GP(0)	If the memory address is in a non-canonical form.
#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#PF (fault-code)	For a page fault.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If CPUID.(EAX=14H, ECX=0):EBX.PTWRITE [Bit 4] = 0.
	If LOCK prefix is used.
	If 66H prefix is used.

PUNPCKHBW/PUNPCKHWD/PUNPCKHDQ/PUNPCKHQDQ— Unpack High Data

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F 68 /r[1] PUNPCKHBW *mm, mm/m64*	A	V/V	MMX	Unpack and interleave high-order bytes from *mm* and *mm/m64* into *mm*.
66 0F 68 /r PUNPCKHBW *xmm1, xmm2/m128*	A	V/V	SSE2	Unpack and interleave high-order bytes from *xmm1* and *xmm2/m128* into *xmm1*.
NP 0F 69 /r[1] PUNPCKHWD *mm, mm/m64*	A	V/V	MMX	Unpack and interleave high-order words from *mm* and *mm/m64* into *mm*.
66 0F 69 /r PUNPCKHWD *xmm1, xmm2/m128*	A	V/V	SSE2	Unpack and interleave high-order words from *xmm1* and *xmm2/m128* into *xmm1*.
NP 0F 6A /r[1] PUNPCKHDQ *mm, mm/m64*	A	V/V	MMX	Unpack and interleave high-order doublewords from *mm* and *mm/m64* into *mm*.
66 0F 6A /r PUNPCKHDQ *xmm1, xmm2/m128*	A	V/V	SSE2	Unpack and interleave high-order doublewords from *xmm1* and *xmm2/m128* into *xmm1*.
66 0F 6D /r PUNPCKHQDQ *xmm1, xmm2/m128*	A	V/V	SSE2	Unpack and interleave high-order quadwords from *xmm1* and *xmm2/m128* into *xmm1.*
VEX.NDS.128.66.0F.WIG 68/r VPUNPCKHBW *xmm1,xmm2, xmm3/m128*	B	V/V	AVX	Interleave high-order bytes from *xmm2* and *xmm3/m128* into *xmm1*.
VEX.NDS.128.66.0F.WIG 69/r VPUNPCKHWD *xmm1,xmm2, xmm3/m128*	B	V/V	AVX	Interleave high-order words from *xmm2* and *xmm3/m128* into *xmm1*.
VEX.NDS.128.66.0F.WIG 6A/r VPUNPCKHDQ *xmm1, xmm2, xmm3/m128*	B	V/V	AVX	Interleave high-order doublewords from *xmm2* and *xmm3/m128* into *xmm1*.
VEX.NDS.128.66.0F.WIG 6D/r VPUNPCKHQDQ *xmm1, xmm2, xmm3/m128*	B	V/V	AVX	Interleave high-order quadword from *xmm2* and *xmm3/m128* into *xmm1* register.
VEX.NDS.256.66.0F.WIG 68 /r VPUNPCKHBW *ymm1, ymm2, ymm3/m256*	B	V/V	AVX2	Interleave high-order bytes from *ymm2* and *ymm3/m256* into *ymm1* register.
VEX.NDS.256.66.0F.WIG 69 /r VPUNPCKHWD *ymm1, ymm2, ymm3/m256*	B	V/V	AVX2	Interleave high-order words from *ymm2* and *ymm3/m256* into *ymm1* register.
VEX.NDS.256.66.0F.WIG 6A /r VPUNPCKHDQ *ymm1, ymm2, ymm3/m256*	B	V/V	AVX2	Interleave high-order doublewords from *ymm2* and *ymm3/m256* into *ymm1* register.
VEX.NDS.256.66.0F.WIG 6D /r VPUNPCKHQDQ *ymm1, ymm2, ymm3/m256*	B	V/V	AVX2	Interleave high-order quadword from *ymm2* and *ymm3/m256* into *ymm1* register.
EVEX.NDS.128.66.0F.WIG 68 /r VPUNPCKHBW xmm1 {k1}{z}, xmm2, xmm3/m128	C	V/V	AVX512VL AVX512BW	Interleave high-order bytes from xmm2 and xmm3/m128 into xmm1 register using k1 write mask.
EVEX.NDS.128.66.0F.WIG 69 /r VPUNPCKHWD xmm1 {k1}{z}, xmm2, xmm3/m128	C	V/V	AVX512VL AVX512BW	Interleave high-order words from xmm2 and xmm3/m128 into xmm1 register using k1 write mask.
EVEX.NDS.128.66.0F.W0 6A /r VPUNPCKHDQ xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst	D	V/V	AVX512VL AVX512F	Interleave high-order doublewords from xmm2 and xmm3/m128/m32bcst into xmm1 register using k1 write mask.
EVEX.NDS.128.66.0F.W1 6D /r VPUNPCKHQDQ xmm1 {k1}{z}, xmm2, xmm3/m128/m64bcst	D	V/V	AVX512VL AVX512F	Interleave high-order quadword from xmm2 and xmm3/m128/m64bcst into xmm1 register using k1 write mask.

EVEX.NDS.256.66.0F.WIG 68 /r VPUNPCKHBW ymm1 {k1}{z}, ymm2, ymm3/m256	C	V/V	AVX512VL AVX512BW	Interleave high-order bytes from ymm2 and ymm3/m256 into ymm1 register using k1 write mask.
EVEX.NDS.256.66.0F.WIG 69 /r VPUNPCKHWD ymm1 {k1}{z}, ymm2, ymm3/m256	C	V/V	AVX512VL AVX512BW	Interleave high-order words from ymm2 and ymm3/m256 into ymm1 register using k1 write mask.
EVEX.NDS.256.66.0F.W0 6A /r VPUNPCKHDQ ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst	D	V/V	AVX512VL AVX512F	Interleave high-order doublewords from ymm2 and ymm3/m256/m32bcst into ymm1 register using k1 write mask.
EVEX.NDS.256.66.0F.W1 6D /r VPUNPCKHQDQ ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst	D	V/V	AVX512VL AVX512F	Interleave high-order quadword from ymm2 and ymm3/m256/m64bcst into ymm1 register using k1 write mask.
EVEX.NDS.512.66.0F.WIG 68/r VPUNPCKHBW zmm1 {k1}{z}, zmm2, zmm3/m512	C	V/V	AVX512BW	Interleave high-order bytes from zmm2 and zmm3/m512 into zmm1 register.
EVEX.NDS.512.66.0F.WIG 69/r VPUNPCKHWD zmm1 {k1}{z}, zmm2, zmm3/m512	C	V/V	AVX512BW	Interleave high-order words from zmm2 and zmm3/m512 into zmm1 register.
EVEX.NDS.512.66.0F.W0 6A /r VPUNPCKHDQ zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst	D	V/V	AVX512F	Interleave high-order doublewords from zmm2 and zmm3/m512/m32bcst into zmm1 register using k1 write mask.
EVEX.NDS.512.66.0F.W1 6D /r VPUNPCKHQDQ zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst	D	V/V	AVX512F	Interleave high-order quadword from zmm2 and zmm3/m512/m64bcst into zmm1 register using k1 write mask.

NOTES:

1. See note in Section 2.4, "AVX and SSE Instruction Exception Specification" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A* and Section 22.25.3, "Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	NA	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
B	NA	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA
C	Full Vector Mem	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA
D	Full Vector	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA

Description

Unpacks and interleaves the high-order data elements (bytes, words, doublewords, or quadwords) of the destination operand (first operand) and source operand (second operand) into the destination operand. Figure 4-20 shows the unpack operation for bytes in 64-bit operands. The low-order data elements are ignored.

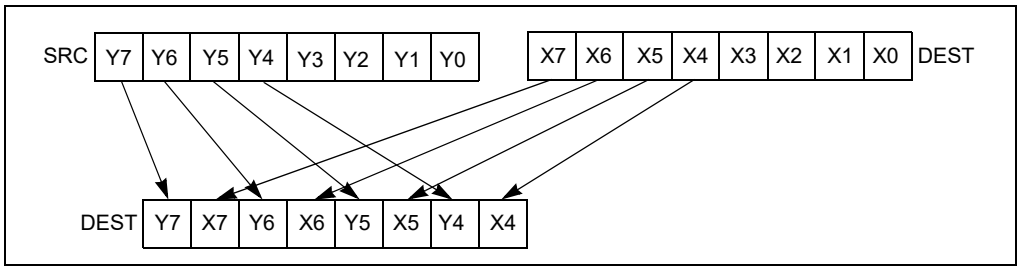


Figure 4-20. PUNPCKHBW Instruction Operation Using 64-bit Operands

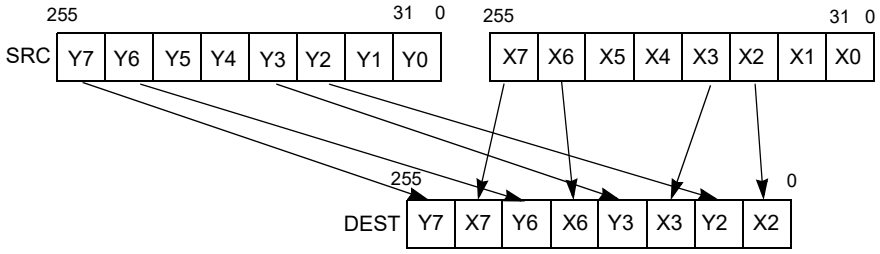


Figure 4-21. 256-bit VPUNPCKHDQ Instruction Operation

When the source data comes from a 64-bit memory operand, the full 64-bit operand is accessed from memory, but the instruction uses only the high-order 32 bits. When the source data comes from a 128-bit memory operand, an implementation may fetch only the appropriate 64 bits; however, alignment to a 16-byte boundary and normal segment checking will still be enforced.

The (V)PUNPCKHBW instruction interleaves the high-order bytes of the source and destination operands, the (V)PUNPCKHWD instruction interleaves the high-order words of the source and destination operands, the (V)PUNPCKHDQ instruction interleaves the high-order doubleword (or doublewords) of the source and destination operands, and the (V)PUNPCKHQDQ instruction interleaves the high-order quadwords of the source and destination operands.

These instructions can be used to convert bytes to words, words to doublewords, doublewords to quadwords, and quadwords to double quadwords, respectively, by placing all 0s in the source operand. Here, if the source operand contains all 0s, the result (stored in the destination operand) contains zero extensions of the high-order data elements from the original value in the destination operand. For example, with the (V)PUNPCKHBW instruction the high-order bytes are zero extended (that is, unpacked into unsigned word integers), and with the (V)PUNPCKHWD instruction, the high-order words are zero extended (unpacked into unsigned doubleword integers).

In 64-bit mode and not encoded with VEX/EVEX, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

Legacy SSE versions 64-bit operand: The source operand can be an MMX technology register or a 64-bit memory location. The destination operand is an MMX technology register.

128-bit Legacy SSE versions: The second source operand is an XMM register or a 128-bit memory location. The first source operand and destination operands are XMM registers. Bits (MAXVL-1:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded versions: The second source operand is an XMM register or a 128-bit memory location. The first source operand and destination operands are XMM registers. Bits (MAXVL-1:128) of the destination YMM register are zeroed.

VEX.256 encoded version: The second source operand is an YMM register or an 256-bit memory location. The first source operand and destination operands are YMM registers.

EVEX encoded VPUNPCKHDQ/QDQ: The second source operand is a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 32/64-bit memory location. The first source operand and destination operands are ZMM/YMM/XMM registers. The destination is conditionally updated with writemask k1.

EVEX encoded VPUNPCKHWD/BW: The second source operand is a ZMM/YMM/XMM register, a 512/256/128-bit memory location. The first source operand and destination operands are ZMM/YMM/XMM registers. The destination is conditionally updated with writemask k1.

Operation

PUNPCKHBW instruction with 64-bit operands:

```
DEST[7:0] ← DEST[39:32];
DEST[15:8] ← SRC[39:32];
DEST[23:16] ← DEST[47:40];
DEST[31:24] ← SRC[47:40];
DEST[39:32] ← DEST[55:48];
DEST[47:40] ← SRC[55:48];
DEST[55:48] ← DEST[63:56];
DEST[63:56] ← SRC[63:56];
```

PUNPCKHW instruction with 64-bit operands:

```
DEST[15:0] ← DEST[47:32];
DEST[31:16] ← SRC[47:32];
DEST[47:32] ← DEST[63:48];
DEST[63:48] ← SRC[63:48];
```

PUNPCKHDQ instruction with 64-bit operands:

```
DEST[31:0] ← DEST[63:32];
DEST[63:32] ← SRC[63:32];
```

```
INTERLEAVE_HIGH_BYTES_512b (SRC1, SRC2)
TMP_DEST[255:0] ← INTERLEAVE_HIGH_BYTES_256b(SRC1[255:0], SRC[255:0])
TMP_DEST[511:256] ← INTERLEAVE_HIGH_BYTES_256b(SRC1[511:256], SRC[511:256])

INTERLEAVE_HIGH_BYTES_256b (SRC1, SRC2)
DEST[7:0] ← SRC1[71:64]
DEST[15:8] ← SRC2[71:64]
DEST[23:16] ← SRC1[79:72]
DEST[31:24] ← SRC2[79:72]
DEST[39:32] ← SRC1[87:80]
DEST[47:40] ← SRC2[87:80]
DEST[55:48] ← SRC1[95:88]
DEST[63:56] ← SRC2[95:88]
DEST[71:64] ← SRC1[103:96]
DEST[79:72] ← SRC2[103:96]
DEST[87:80] ← SRC1[111:104]
DEST[95:88] ← SRC2[111:104]
DEST[103:96] ← SRC1[119:112]
DEST[111:104] ← SRC2[119:112]
DEST[119:112] ← SRC1[127:120]
DEST[127:120] ← SRC2[127:120]
DEST[135:128] ← SRC1[199:192]
DEST[143:136] ← SRC2[199:192]
DEST[151:144] ← SRC1[207:200]
DEST[159:152] ← SRC2[207:200]
```

```
DEST[167:160] ← SRC1[215:208]
DEST[175:168] ← SRC2[215:208]
DEST[183:176] ← SRC1[223:216]
DEST[191:184] ← SRC2[223:216]
DEST[199:192] ← SRC1[231:224]
DEST[207:200] ← SRC2[231:224]
DEST[215:208] ← SRC1[239:232]
DEST[223:216] ← SRC2[239:232]
DEST[231:224] ← SRC1[247:240]
DEST[239:232] ← SRC2[247:240]
DEST[247:240] ← SRC1[255:248]
DEST[255:248] ← SRC2[255:248]

INTERLEAVE_HIGH_BYTES (SRC1, SRC2)
DEST[7:0] ← SRC1[71:64]
DEST[15:8] ← SRC2[71:64]
DEST[23:16] ← SRC1[79:72]
DEST[31:24] ← SRC2[79:72]
DEST[39:32] ← SRC1[87:80]
DEST[47:40] ← SRC2[87:80]
DEST[55:48] ← SRC1[95:88]
DEST[63:56] ← SRC2[95:88]
DEST[71:64] ← SRC1[103:96]
DEST[79:72] ← SRC2[103:96]
DEST[87:80] ← SRC1[111:104]
DEST[95:88] ← SRC2[111:104]
DEST[103:96] ← SRC1[119:112]
DEST[111:104] ← SRC2[119:112]
DEST[119:112] ← SRC1[127:120]
DEST[127:120] ← SRC2[127:120]

INTERLEAVE_HIGH_WORDS_512b (SRC1, SRC2)
TMP_DEST[255:0] ← INTERLEAVE_HIGH_WORDS_256b(SRC1[255:0], SRC[255:0])
TMP_DEST[511:256] ← INTERLEAVE_HIGH_WORDS_256b(SRC1[511:256], SRC[511:256])

INTERLEAVE_HIGH_WORDS_256b(SRC1, SRC2)
DEST[15:0] ← SRC1[79:64]
DEST[31:16] ← SRC2[79:64]
DEST[47:32] ← SRC1[95:80]
DEST[63:48] ← SRC2[95:80]
DEST[79:64] ← SRC1[111:96]
DEST[95:80] ← SRC2[111:96]
DEST[111:96] ← SRC1[127:112]
DEST[127:112] ← SRC2[127:112]
DEST[143:128] ← SRC1[207:192]
DEST[159:144] ← SRC2[207:192]
DEST[175:160] ← SRC1[223:208]
DEST[191:176] ← SRC2[223:208]
DEST[207:192] ← SRC1[239:224]
DEST[223:208] ← SRC2[239:224]
DEST[239:224] ← SRC1[255:240]
DEST[255:240] ← SRC2[255:240]

INTERLEAVE_HIGH_WORDS (SRC1, SRC2)
```

DEST[15:0] ← SRC1[79:64]
DEST[31:16] ← SRC2[79:64]
DEST[47:32] ← SRC1[95:80]
DEST[63:48] ← SRC2[95:80]
DEST[79:64] ← SRC1[111:96]
DEST[95:80] ← SRC2[111:96]
DEST[111:96] ← SRC1[127:112]
DEST[127:112] ← SRC2[127:112]

INTERLEAVE_HIGH_DWORDS_512b (SRC1, SRC2)
TMP_DEST[255:0] ← INTERLEAVE_HIGH_DWORDS_256b(SRC1[255:0], SRC2[255:0])
TMP_DEST[511:256] ← INTERLEAVE_HIGH_DWORDS_256b(SRC1[511:256], SRC2[511:256])

INTERLEAVE_HIGH_DWORDS_256b(SRC1, SRC2)
DEST[31:0] ← SRC1[95:64]
DEST[63:32] ← SRC2[95:64]
DEST[95:64] ← SRC1[127:96]
DEST[127:96] ← SRC2[127:96]
DEST[159:128] ← SRC1[223:192]
DEST[191:160] ← SRC2[223:192]
DEST[223:192] ← SRC1[255:224]
DEST[255:224] ← SRC2[255:224]

INTERLEAVE_HIGH_DWORDS(SRC1, SRC2)
DEST[31:0] ← SRC1[95:64]
DEST[63:32] ← SRC2[95:64]
DEST[95:64] ← SRC1[127:96]
DEST[127:96] ← SRC2[127:96]

INTERLEAVE_HIGH_QWORDS_512b (SRC1, SRC2)
TMP_DEST[255:0] ← INTERLEAVE_HIGH_QWORDS_256b(SRC1[255:0], SRC2[255:0])
TMP_DEST[511:256] ← INTERLEAVE_HIGH_QWORDS_256b(SRC1[511:256], SRC2[511:256])

INTERLEAVE_HIGH_QWORDS_256b(SRC1, SRC2)
DEST[63:0] ← SRC1[127:64]
DEST[127:64] ← SRC2[127:64]
DEST[191:128] ← SRC1[255:192]
DEST[255:192] ← SRC2[255:192]

INTERLEAVE_HIGH_QWORDS(SRC1, SRC2)
DEST[63:0] ← SRC1[127:64]
DEST[127:64] ← SRC2[127:64]

PUNPCKHBW (128-bit Legacy SSE Version)
DEST[127:0] ←INTERLEAVE_HIGH_BYTES(DEST, SRC)
DEST[255:127] (Unmodified)

VPUNPCKHBW (VEX.128 encoded version)
DEST[127:0] ←INTERLEAVE_HIGH_BYTES(SRC1, SRC2)
DEST[MAXVL-1:127] ←0

VPUNPCKHBW (VEX.256 encoded version)
DEST[255:0] ←INTERLEAVE_HIGH_BYTES_256b(SRC1, SRC2)
DEST[MAXVL-1:256] ←0

VPUNPCKHBW (EVEX encoded versions)

```
(KL, VL) = (16, 128), (32, 256), (64, 512)
IF VL = 128
    TMP_DEST[VL-1:0] ← INTERLEAVE_HIGH_BYTES(SRC1[VL-1:0], SRC2[VL-1:0])
FI;
IF VL = 256
    TMP_DEST[VL-1:0] ← INTERLEAVE_HIGH_BYTES_256b(SRC1[VL-1:0], SRC2[VL-1:0])
FI;
IF VL = 512
    TMP_DEST[VL-1:0] ← INTERLEAVE_HIGH_BYTES_512b(SRC1[VL-1:0], SRC2[VL-1:0])
FI;

FOR j ← 0 TO KL-1
    i ← j * 8
    IF k1[j] OR *no writemask*
        THEN DEST[i+7:i] ← TMP_DEST[i+7:i]
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+7:i] remains unchanged*
                ELSE *zeroing-masking*              ; zeroing-masking
                    DEST[i+7:i] ← 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0
```

PUNPCKHWD (128-bit Legacy SSE Version)

```
DEST[127:0] ←INTERLEAVE_HIGH_WORDS(DEST, SRC)
DEST[255:127] (Unmodified)
```

VPUNPCKHWD (VEX.128 encoded version)

```
DEST[127:0] ←INTERLEAVE_HIGH_WORDS(SRC1, SRC2)
DEST[MAXVL-1:127] ←0
```

VPUNPCKHWD (VEX.256 encoded version)

```
DEST[255:0] ←INTERLEAVE_HIGH_WORDS_256b(SRC1, SRC2)
DEST[MAXVL-1:256] ←0
```

VPUNPCKHWD (EVEX encoded versions)

```
(KL, VL) = (8, 128), (16, 256), (32, 512)
IF VL = 128
    TMP_DEST[VL-1:0] ← INTERLEAVE_HIGH_WORDS(SRC1[VL-1:0], SRC2[VL-1:0])
FI;
IF VL = 256
    TMP_DEST[VL-1:0] ← INTERLEAVE_HIGH_WORDS_256b(SRC1[VL-1:0], SRC2[VL-1:0])
FI;
IF VL = 512
    TMP_DEST[VL-1:0] ← INTERLEAVE_HIGH_WORDS_512b(SRC1[VL-1:0], SRC2[VL-1:0])
FI;

FOR j ← 0 TO KL-1
    i ← j * 16
    IF k1[j] OR *no writemask*
```

```
        THEN DEST[i+15:i] ← TMP_DEST[i+15:i]
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+15:i] remains unchanged*
                ELSE *zeroing-masking*              ; zeroing-masking
                    DEST[i+15:i] ← 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0
```

PUNPCKHDQ (128-bit Legacy SSE Version)

```
DEST[127:0] ←INTERLEAVE_HIGH_DWORDS(DEST, SRC)
DEST[255:127] (Unmodified)
```

VPUNPCKHDQ (VEX.128 encoded version)

```
DEST[127:0] ←INTERLEAVE_HIGH_DWORDS(SRC1, SRC2)
DEST[MAXVL-1:127] ←0
```

VPUNPCKHDQ (VEX.256 encoded version)

```
DEST[255:0] ←INTERLEAVE_HIGH_DWORDS_256b(SRC1, SRC2)
DEST[MAXVL-1:256] ←0
```

VPUNPCKHDQ (EVEX.512 encoded version)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j ← 0 TO KL-1
    i ← j * 32
    IF (EVEX.b = 1) AND (SRC2 *is memory*)
        THEN TMP_SRC2[i+31:i] ← SRC2[31:0]
        ELSE TMP_SRC2[i+31:i] ← SRC2[i+31:i]
    FI;
ENDFOR;
IF VL = 128
    TMP_DEST[VL-1:0] ← INTERLEAVE_HIGH_DWORDS(SRC1[VL-1:0], TMP_SRC2[VL-1:0])
FI;
IF VL = 256
    TMP_DEST[VL-1:0] ← INTERLEAVE_HIGH_DWORDS_256b(SRC1[VL-1:0], TMP_SRC2[VL-1:0])
FI;
IF VL = 512
    TMP_DEST[VL-1:0] ← INTERLEAVE_HIGH_DWORDS_512b(SRC1[VL-1:0], TMP_SRC2[VL-1:0])
FI;

FOR j ← 0 TO KL-1
    i ← j * 32
    IF k1[j] OR *no writemask*
        THEN DEST[i+31:i] ← TMP_DEST[i+31:i]
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE *zeroing-masking*              ; zeroing-masking
                    DEST[i+31:i] ← 0
            FI
    FI;
ENDFOR
```

```
DEST[MAXVL-1:VL] ← 0

PUNPCKHQDQ (128-bit Legacy SSE Version)
DEST[127:0] ←INTERLEAVE_HIGH_QWORDS(DEST, SRC)
DEST[MAXVL-1:128] (Unmodified)

VPUNPCKHQDQ (VEX.128 encoded version)
DEST[127:0] ←INTERLEAVE_HIGH_QWORDS(SRC1, SRC2)
DEST[MAXVL-1:128] ←0

VPUNPCKHQDQ (VEX.256 encoded version)
DEST[255:0] ←INTERLEAVE_HIGH_QWORDS_256b(SRC1, SRC2)
DEST[MAXVL-1:256] ←0

VPUNPCKHQDQ (EVEX encoded versions)
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j ← 0 TO KL-1
    i ← j * 64
    IF (EVEX.b = 1) AND (SRC2 *is memory*)
        THEN TMP_SRC2[i+63:i] ← SRC2[63:0]
        ELSE TMP_SRC2[i+63:i] ← SRC2[i+63:i]
    FI;
ENDFOR;
IF VL = 128
    TMP_DEST[VL-1:0] ← INTERLEAVE_HIGH_QWORDS(SRC1[VL-1:0], TMP_SRC2[VL-1:0])
FI;
IF VL = 256
    TMP_DEST[VL-1:0] ← INTERLEAVE_HIGH_QWORDS_256b(SRC1[VL-1:0], TMP_SRC2[VL-1:0])
FI;
IF VL = 512
    TMP_DEST[VL-1:0] ← INTERLEAVE_HIGH_QWORDS_512b(SRC1[VL-1:0], TMP_SRC2[VL-1:0])
FI;

FOR j ← 0 TO KL-1
    i ← j * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+63:i] ← TMP_DEST[i+63:i]
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE *zeroing-masking*              ; zeroing-masking
                    DEST[i+63:i] ← 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0
```

Intel C/C++ Compiler Intrinsic Equivalents

VPUNPCKHBW __m512i _mm512_unpackhi_epi8(__m512i a, __m512i b);
VPUNPCKHBW __m512i _mm512_mask_unpackhi_epi8(__m512i s, __mmask64 k, __m512i a, __m512i b);
VPUNPCKHBW __m512i _mm512_maskz_unpackhi_epi8(__mmask64 k, __m512i a, __m512i b);
VPUNPCKHBW __m256i _mm256_mask_unpackhi_epi8(__m256i s, __mmask32 k, __m256i a, __m256i b);
VPUNPCKHBW __m256i _mm256_maskz_unpackhi_epi8(__mmask32 k, __m256i a, __m256i b);
VPUNPCKHBW __m128i _mm_mask_unpackhi_epi8(v s, __mmask16 k, __m128i a, __m128i b);

VPUNPCKHBW __m128i _mm_maskz_unpackhi_epi8(__mmask16 k, __m128i a, __m128i b);
VPUNPCKHWD __m512i _mm512_unpackhi_epi16(__m512i a, __m512i b);
VPUNPCKHWD __m512i _mm512_mask_unpackhi_epi16(__m512i s, __mmask32 k, __m512i a, __m512i b);
VPUNPCKHWD __m512i _mm512_maskz_unpackhi_epi16(__mmask32 k, __m512i a, __m512i b);
VPUNPCKHWD __m256i _mm256_mask_unpackhi_epi16(__m256i s, __mmask16 k, __m256i a, __m256i b);
VPUNPCKHWD __m256i _mm256_maskz_unpackhi_epi16(__mmask16 k, __m256i a, __m256i b);
VPUNPCKHWD __m128i _mm_mask_unpackhi_epi16(v s, __mmask8 k, __m128i a, __m128i b);
VPUNPCKHWD __m128i _mm_maskz_unpackhi_epi16(__mmask8 k, __m128i a, __m128i b);
VPUNPCKHDQ __m512i _mm512_unpackhi_epi32(__m512i a, __m512i b);
VPUNPCKHDQ __m512i _mm512_mask_unpackhi_epi32(__m512i s, __mmask16 k, __m512i a, __m512i b);
VPUNPCKHDQ __m512i _mm512_maskz_unpackhi_epi32(__mmask16 k, __m512i a, __m512i b);
VPUNPCKHDQ __m256i _mm256_mask_unpackhi_epi32(__m512i s, __mmask8 k, __m512i a, __m512i b);
VPUNPCKHDQ __m256i _mm256_maskz_unpackhi_epi32(__mmask8 k, __m512i a, __m512i b);
VPUNPCKHDQ __m128i _mm_mask_unpackhi_epi32(__m512i s, __mmask8 k, __m512i a, __m512i b);
VPUNPCKHDQ __m128i _mm_maskz_unpackhi_epi32(__mmask8 k, __m512i a, __m512i b);
VPUNPCKHQDQ __m512i _mm512_unpackhi_epi64(__m512i a, __m512i b);
VPUNPCKHQDQ __m512i _mm512_mask_unpackhi_epi64(__m512i s, __mmask8 k, __m512i a, __m512i b);
VPUNPCKHQDQ __m512i _mm512_maskz_unpackhi_epi64(__mmask8 k, __m512i a, __m512i b);
VPUNPCKHQDQ __m256i _mm256_mask_unpackhi_epi64(__m512i s, __mmask8 k, __m512i a, __m512i b);
VPUNPCKHQDQ __m256i _mm256_maskz_unpackhi_epi64(__mmask8 k, __m512i a, __m512i b);
VPUNPCKHQDQ __m128i _mm_mask_unpackhi_epi64(__m512i s, __mmask8 k, __m512i a, __m512i b);
VPUNPCKHQDQ __m128i _mm_maskz_unpackhi_epi64(__mmask8 k, __m512i a, __m512i b);
PUNPCKHBW:__m64 _mm_unpackhi_pi8(__m64 m1, __m64 m2)
(V)PUNPCKHBW:__m128i _mm_unpackhi_epi8(__m128i m1, __m128i m2)
VPUNPCKHBW:__m256i _mm256_unpackhi_epi8(__m256i m1, __m256i m2)
PUNPCKHWD:__m64 _mm_unpackhi_pi16(__m64 m1,__m64 m2)
(V)PUNPCKHWD:__m128i _mm_unpackhi_epi16(__m128i m1,__m128i m2)
VPUNPCKHWD:__m256i _mm256_unpackhi_epi16(__m256i m1,__m256i m2)
PUNPCKHDQ:__m64 _mm_unpackhi_pi32(__m64 m1, __m64 m2)
(V)PUNPCKHDQ:__m128i _mm_unpackhi_epi32(__m128i m1, __m128i m2)
VPUNPCKHDQ:__m256i _mm256_unpackhi_epi32(__m256i m1, __m256i m2)
(V)PUNPCKHQDQ:__m128i _mm_unpackhi_epi64 (__m128i a, __m128i b)
VPUNPCKHQDQ:__m256i _mm256_unpackhi_epi64 (__m256i a, __m256i b)

Flags Affected

None.

Numeric Exceptions

None.

Other Exceptions

Non-EVEX-encoded instruction, see Exceptions Type 4.
EVEX-encoded VPUNPCKHQDQ/QDQ, see Exceptions Type E4NF.
EVEX-encoded VPUNPCKHBW/WD, see Exceptions Type E4NF.nb.

PUNPCKLBW/PUNPCKLWD/PUNPCKLDQ/PUNPCKLQDQ—Unpack Low Data

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F 60 /r[1] PUNPCKLBW *mm, mm/m32*	A	V/V	MMX	Interleave low-order bytes from *mm* and *mm/m32* into *mm*.
66 0F 60 /r PUNPCKLBW *xmm1, xmm2/m128*	A	V/V	SSE2	Interleave low-order bytes from *xmm1* and *xmm2/m128* into *xmm1*.
NP 0F 61 /r[1] PUNPCKLWD *mm, mm/m32*	A	V/V	MMX	Interleave low-order words from *mm* and *mm/m32* into *mm*.
66 0F 61 /r PUNPCKLWD *xmm1, xmm2/m128*	A	V/V	SSE2	Interleave low-order words from *xmm1* and *xmm2/m128* into *xmm1*.
NP 0F 62 /r[1] PUNPCKLDQ *mm, mm/m32*	A	V/V	MMX	Interleave low-order doublewords from *mm* and *mm/m32* into *mm*.
66 0F 62 /r PUNPCKLDQ *xmm1, xmm2/m128*	A	V/V	SSE2	Interleave low-order doublewords from *xmm1* and *xmm2/m128* into xmm1.
66 0F 6C /r PUNPCKLQDQ *xmm1, xmm2/m128*	A	V/V	SSE2	Interleave low-order quadword from *xmm1* and *xmm2/m128* into *xmm1* register.
VEX.NDS.128.66.0F.WIG 60/r VPUNPCKLBW *xmm1,xmm2, xmm3/m128*	B	V/V	AVX	Interleave low-order bytes from *xmm2* and *xmm3/m128* into *xmm1*.
VEX.NDS.128.66.0F.WIG 61/r VPUNPCKLWD *xmm1,xmm2, xmm3/m128*	B	V/V	AVX	Interleave low-order words from *xmm2* and *xmm3/m128* into *xmm1*.
VEX.NDS.128.66.0F.WIG 62/r VPUNPCKLDQ *xmm1, xmm2, xmm3/m128*	B	V/V	AVX	Interleave low-order doublewords from *xmm2* and *xmm3/m128* into *xmm1*.
VEX.NDS.128.66.0F.WIG 6C/r VPUNPCKLQDQ *xmm1, xmm2, xmm3/m128*	B	V/V	AVX	Interleave low-order quadword from *xmm2* and *xmm3/m128* into *xmm1* register.
VEX.NDS.256.66.0F.WIG 60 /r VPUNPCKLBW *ymm1, ymm2, ymm3/m256*	B	V/V	AVX2	Interleave low-order bytes from *ymm2* and *ymm3/m256* into *ymm1* register.
VEX.NDS.256.66.0F.WIG 61 /r VPUNPCKLWD *ymm1, ymm2, ymm3/m256*	B	V/V	AVX2	Interleave low-order words from *ymm2* and *ymm3/m256* into *ymm1* register.
VEX.NDS.256.66.0F.WIG 62 /r VPUNPCKLDQ *ymm1, ymm2, ymm3/m256*	B	V/V	AVX2	Interleave low-order doublewords from *ymm2* and *ymm3/m256* into *ymm1* register.
VEX.NDS.256.66.0F.WIG 6C /r VPUNPCKLQDQ *ymm1, ymm2, ymm3/m256*	B	V/V	AVX2	Interleave low-order quadword from *ymm2* and *ymm3/m256* into *ymm1* register.
EVEX.NDS.128.66.0F.WIG 60 /r VPUNPCKLBW xmm1 {k1}{z}, xmm2, xmm3/m128	C	V/V	AVX512VL AVX512BW	Interleave low-order bytes from xmm2 and xmm3/m128 into xmm1 register subject to write mask k1.
EVEX.NDS.128.66.0F.WIG 61 /r VPUNPCKLWD xmm1 {k1}{z}, xmm2, xmm3/m128	C	V/V	AVX512VL AVX512BW	Interleave low-order words from xmm2 and xmm3/m128 into xmm1 register subject to write mask k1.
EVEX.NDS.128.66.0F.W0 62 /r VPUNPCKLDQ xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst	D	V/V	AVX512VL AVX512F	Interleave low-order doublewords from xmm2 and xmm3/m128/m32bcst into xmm1 register subject to write mask k1.
EVEX.NDS.128.66.0F.W1 6C /r VPUNPCKLQDQ xmm1 {k1}{z}, xmm2, xmm3/m128/m64bcst	D	V/V	AVX512VL AVX512F	Interleave low-order quadword from zmm2 and zmm3/m512/m64bcst into zmm1 register subject to write mask k1.

EVEX.NDS.256.66.0F.WIG 60 /r VPUNPCKLBW ymm1 {k1}{z}, ymm2, ymm3/m256	C	V/V	AVX512VL AVX512BW	Interleave low-order bytes from ymm2 and ymm3/m256 into ymm1 register subject to write mask k1.
EVEX.NDS.256.66.0F.WIG 61 /r VPUNPCKLWD ymm1 {k1}{z}, ymm2, ymm3/m256	C	V/V	AVX512VL AVX512BW	Interleave low-order words from ymm2 and ymm3/m256 into ymm1 register subject to write mask k1.
EVEX.NDS.256.66.0F.W0 62 /r VPUNPCKLDQ ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst	D	V/V	AVX512VL AVX512F	Interleave low-order doublewords from ymm2 and ymm3/m256/m32bcst into ymm1 register subject to write mask k1.
EVEX.NDS.256.66.0F.W1 6C /r VPUNPCKLQDQ ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst	D	V/V	AVX512VL AVX512F	Interleave low-order quadword from ymm2 and ymm3/m256/m64bcst into ymm1 register subject to write mask k1.
EVEX.NDS.512.66.0F.WIG 60/r VPUNPCKLBW zmm1 {k1}{z}, zmm2, zmm3/m512	C	V/V	AVX512BW	Interleave low-order bytes from zmm2 and zmm3/m512 into zmm1 register subject to write mask k1.
EVEX.NDS.512.66.0F.WIG 61/r VPUNPCKLWD zmm1 {k1}{z}, zmm2, zmm3/m512	C	V/V	AVX512BW	Interleave low-order words from zmm2 and zmm3/m512 into zmm1 register subject to write mask k1.
EVEX.NDS.512.66.0F.W0 62 /r VPUNPCKLDQ zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst	D	V/V	AVX512F	Interleave low-order doublewords from zmm2 and zmm3/m512/m32bcst into zmm1 register subject to write mask k1.
EVEX.NDS.512.66.0F.W1 6C /r VPUNPCKLQDQ zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst	D	V/V	AVX512F	Interleave low-order quadword from zmm2 and zmm3/m512/m64bcst into zmm1 register subject to write mask k1.

NOTES:

1. See note in Section 2.4, "AVX and SSE Instruction Exception Specification" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A* and Section 22.25.3, "Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	NA	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
B	NA	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA
C	Full Vector Mem	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA
D	Full Vector	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA

Description

Unpacks and interleaves the low-order data elements (bytes, words, doublewords, and quadwords) of the destination operand (first operand) and source operand (second operand) into the destination operand. (Figure 4-22 shows the unpack operation for bytes in 64-bit operands.). The high-order data elements are ignored.

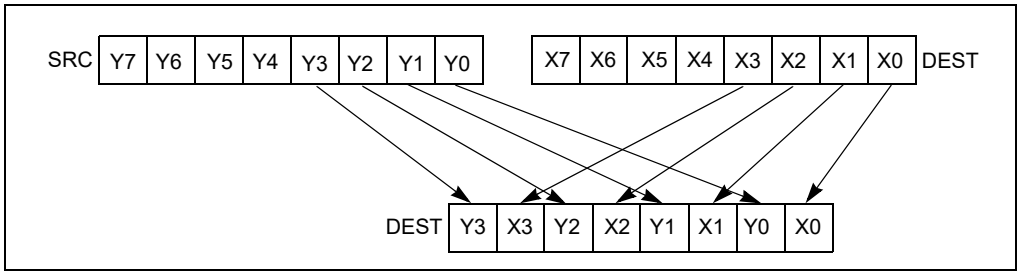


Figure 4-22. PUNPCKLBW Instruction Operation Using 64-bit Operands

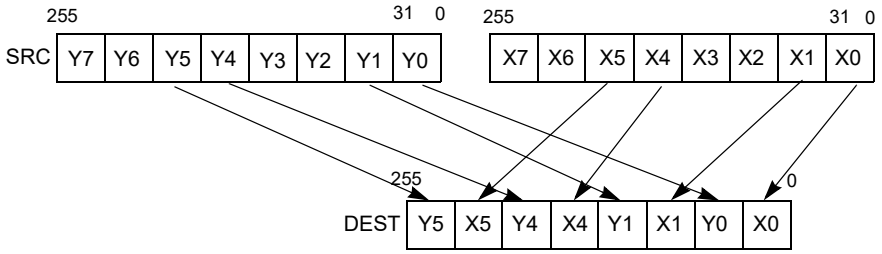


Figure 4-23. 256-bit VPUNPCKLDQ Instruction Operation

When the source data comes from a 128-bit memory operand, an implementation may fetch only the appropriate 64 bits; however, alignment to a 16-byte boundary and normal segment checking will still be enforced.

The (V)PUNPCKLBW instruction interleaves the low-order bytes of the source and destination operands, the (V)PUNPCKLWD instruction interleaves the low-order words of the source and destination operands, the (V)PUNPCKLDQ instruction interleaves the low-order doubleword (or doublewords) of the source and destination operands, and the (V)PUNPCKLQDQ instruction interleaves the low-order quadwords of the source and destination operands.

These instructions can be used to convert bytes to words, words to doublewords, doublewords to quadwords, and quadwords to double quadwords, respectively, by placing all 0s in the source operand. Here, if the source operand contains all 0s, the result (stored in the destination operand) contains zero extensions of the high-order data elements from the original value in the destination operand. For example, with the (V)PUNPCKLBW instruction the high-order bytes are zero extended (that is, unpacked into unsigned word integers), and with the (V)PUNPCKLWD instruction, the high-order words are zero extended (unpacked into unsigned doubleword integers).

In 64-bit mode and not encoded with VEX/EVEX, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

Legacy SSE versions 64-bit operand: The source operand can be an MMX technology register or a 32-bit memory location. The destination operand is an MMX technology register.

128-bit Legacy SSE versions: The second source operand is an XMM register or a 128-bit memory location. The first source operand and destination operands are XMM registers. Bits (MAXVL-1:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded versions: The second source operand is an XMM register or a 128-bit memory location. The first source operand and destination operands are XMM registers. Bits (MAXVL-1:128) of the destination YMM register are zeroed.

VEX.256 encoded version: The second source operand is an YMM register or an 256-bit memory location. The first source operand and destination operands are YMM registers. Bits (MAXVL-1:256) of the corresponding ZMM register are zeroed.

EVEX encoded VPUNPCKLDQ/QDQ: The second source operand is a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 32/64-bit memory location. The first source operand and destination operands are ZMM/YMM/XMM registers. The destination is conditionally updated with writemask k1.

EVEX encoded VPUNPCKLWD/BW: The second source operand is a ZMM/YMM/XMM register, a 512/256/128-bit memory location. The first source operand and destination operands are ZMM/YMM/XMM registers. The destination is conditionally updated with writemask k1.

Operation

PUNPCKLBW instruction with 64-bit operands:

```
DEST[63:56] ← SRC[31:24];
DEST[55:48] ← DEST[31:24];
DEST[47:40] ← SRC[23:16];
DEST[39:32] ← DEST[23:16];
DEST[31:24] ← SRC[15:8];
DEST[23:16] ← DEST[15:8];
DEST[15:8] ← SRC[7:0];
DEST[7:0] ← DEST[7:0];
```

PUNPCKLWD instruction with 64-bit operands:

```
DEST[63:48] ← SRC[31:16];
DEST[47:32] ← DEST[31:16];
DEST[31:16] ← SRC[15:0];
DEST[15:0] ← DEST[15:0];
```

PUNPCKLDQ instruction with 64-bit operands:

```
DEST[63:32] ← SRC[31:0];
DEST[31:0] ← DEST[31:0];
```

```
INTERLEAVE_BYTES_512b (SRC1, SRC2)
TMP_DEST[255:0] ← INTERLEAVE_BYTES_256b(SRC1[255:0], SRC[255:0])
TMP_DEST[511:256] ← INTERLEAVE_BYTES_256b(SRC1[511:256], SRC[511:256])

INTERLEAVE_BYTES_256b (SRC1, SRC2)
DEST[7:0] ← SRC1[7:0]
DEST[15:8] ← SRC2[7:0]
DEST[23:16] ← SRC1[15:8]
DEST[31:24] ← SRC2[15:8]
DEST[39:32] ← SRC1[23:16]
DEST[47:40] ← SRC2[23:16]
DEST[55:48] ← SRC1[31:24]
DEST[63:56] ← SRC2[31:24]
DEST[71:64] ← SRC1[39:32]
DEST[79:72] ← SRC2[39:32]
DEST[87:80] ← SRC1[47:40]
DEST[95:88] ← SRC2[47:40]
DEST[103:96] ← SRC1[55:48]
DEST[111:104] ← SRC2[55:48]
DEST[119:112] ← SRC1[63:56]
DEST[127:120] ← SRC2[63:56]
DEST[135:128] ← SRC1[135:128]
DEST[143:136] ← SRC2[135:128]
DEST[151:144] ← SRC1[143:136]
DEST[159:152] ← SRC2[143:136]
DEST[167:160] ← SRC1[151:144]
```

```
DEST[175:168] ← SRC2[151:144]
DEST[183:176] ← SRC1[159:152]
DEST[191:184] ← SRC2[159:152]
DEST[199:192] ← SRC1[167:160]
DEST[207:200] ← SRC2[167:160]
DEST[215:208] ← SRC1[175:168]
DEST[223:216] ← SRC2[175:168]
DEST[231:224] ← SRC1[183:176]
DEST[239:232] ← SRC2[183:176]
DEST[247:240] ← SRC1[191:184]
DEST[255:248] ← SRC2[191:184]

INTERLEAVE_BYTES (SRC1, SRC2)
DEST[7:0] ← SRC1[7:0]
DEST[15:8] ← SRC2[7:0]
DEST[23:16] ← SRC2[15:8]
DEST[31:24] ← SRC2[15:8]
DEST[39:32] ← SRC1[23:16]
DEST[47:40] ← SRC2[23:16]
DEST[55:48] ← SRC1[31:24]
DEST[63:56] ← SRC2[31:24]
DEST[71:64] ← SRC1[39:32]
DEST[79:72] ← SRC2[39:32]
DEST[87:80] ← SRC1[47:40]
DEST[95:88] ← SRC2[47:40]
DEST[103:96] ← SRC1[55:48]
DEST[111:104] ← SRC2[55:48]
DEST[119:112] ← SRC1[63:56]
DEST[127:120] ← SRC2[63:56]

INTERLEAVE_WORDS_512b (SRC1, SRC2)
TMP_DEST[255:0] ← INTERLEAVE_WORDS_256b(SRC1[255:0], SRC[255:0])
TMP_DEST[511:256] ← INTERLEAVE_WORDS_256b(SRC1[511:256], SRC[511:256])

INTERLEAVE_WORDS_256b(SRC1, SRC2)
DEST[15:0] ← SRC1[15:0]
DEST[31:16] ← SRC2[15:0]
DEST[47:32] ← SRC1[31:16]
DEST[63:48] ← SRC2[31:16]
DEST[79:64] ← SRC1[47:32]
DEST[95:80] ← SRC2[47:32]
DEST[111:96] ← SRC1[63:48]
DEST[127:112] ← SRC2[63:48]
DEST[143:128] ← SRC1[143:128]
DEST[159:144] ← SRC2[143:128]
DEST[175:160] ← SRC1[159:144]
DEST[191:176] ← SRC2[159:144]
DEST[207:192] ← SRC1[175:160]
DEST[223:208] ← SRC2[175:160]
DEST[239:224] ← SRC1[191:176]
DEST[255:240] ← SRC2[191:176]

INTERLEAVE_WORDS (SRC1, SRC2)
DEST[15:0] ← SRC1[15:0]
```

DEST[31:16] ← SRC2[15:0]
DEST[47:32] ← SRC1[31:16]
DEST[63:48] ← SRC2[31:16]
DEST[79:64] ← SRC1[47:32]
DEST[95:80] ← SRC2[47:32]
DEST[111:96] ← SRC1[63:48]
DEST[127:112] ← SRC2[63:48]

INTERLEAVE_DWORDS_512b (SRC1, SRC2)
TMP_DEST[255:0] ← INTERLEAVE_DWORDS_256b(SRC1[255:0], SRC2[255:0])
TMP_DEST[511:256] ← INTERLEAVE_DWORDS_256b(SRC1[511:256], SRC2[511:256])

INTERLEAVE_DWORDS_256b(SRC1, SRC2)
DEST[31:0] ← SRC1[31:0]
DEST[63:32] ← SRC2[31:0]
DEST[95:64] ← SRC1[63:32]
DEST[127:96] ← SRC2[63:32]
DEST[159:128] ← SRC1[159:128]
DEST[191:160] ← SRC2[159:128]
DEST[223:192] ← SRC1[191:160]
DEST[255:224] ← SRC2[191:160]

INTERLEAVE_DWORDS(SRC1, SRC2)
DEST[31:0] ← SRC1[31:0]
DEST[63:32] ← SRC2[31:0]
DEST[95:64] ← SRC1[63:32]
DEST[127:96] ← SRC2[63:32]
INTERLEAVE_QWORDS_512b (SRC1, SRC2)
TMP_DEST[255:0] ← INTERLEAVE_QWORDS_256b(SRC1[255:0], SRC2[255:0])
TMP_DEST[511:256] ← INTERLEAVE_QWORDS_256b(SRC1[511:256], SRC2[511:256])

INTERLEAVE_QWORDS_256b(SRC1, SRC2)
DEST[63:0] ← SRC1[63:0]
DEST[127:64] ← SRC2[63:0]
DEST[191:128] ← SRC1[191:128]
DEST[255:192] ← SRC2[191:128]

INTERLEAVE_QWORDS(SRC1, SRC2)
DEST[63:0] ← SRC1[63:0]
DEST[127:64] ← SRC2[63:0]

PUNPCKLBW
DEST[127:0] ←INTERLEAVE_BYTES(DEST, SRC)
DEST[255:127] (Unmodified)

VPUNPCKLBW (VEX.128 encoded instruction)
DEST[127:0] ←INTERLEAVE_BYTES(SRC1, SRC2)
DEST[MAXVL-1:127] ←0

VPUNPCKLBW (VEX.256 encoded instruction)
DEST[255:0] ←INTERLEAVE_BYTES_256b(SRC1, SRC2)
DEST[MAXVL-1:256] ←0

VPUNPCKLBW (EVEX.512 encoded instruction)

```
(KL, VL) = (16, 128), (32, 256), (64, 512)
IF VL = 128
    TMP_DEST[VL-1:0] ← INTERLEAVE_BYTES(SRC1[VL-1:0], SRC2[VL-1:0])
FI;
IF VL = 256
    TMP_DEST[VL-1:0] ← INTERLEAVE_BYTES_256b(SRC1[VL-1:0], SRC2[VL-1:0])
FI;
IF VL = 512
    TMP_DEST[VL-1:0] ← INTERLEAVE_BYTES_512b(SRC1[VL-1:0], SRC2[VL-1:0])
FI;

FOR j ← 0 TO KL-1
    i ← j * 8
    IF k1[j] OR *no writemask*
        THEN DEST[i+7:i] ← TMP_DEST[i+7:i]
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+7:i] remains unchanged*
                ELSE *zeroing-masking*              ; zeroing-masking
                    DEST[i+7:i] ← 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0
DEST[511:0] ← INTERLEAVE_BYTES_512b(SRC1, SRC2)
```

PUNPCKLWD

```
DEST[127:0] ←INTERLEAVE_WORDS(DEST, SRC)
DEST[255:127] (Unmodified)
```

VPUNPCKLWD (VEX.128 encoded instruction)

```
DEST[127:0] ←INTERLEAVE_WORDS(SRC1, SRC2)
DEST[MAXVL-1:127] ←0
```

VPUNPCKLWD (VEX.256 encoded instruction)

```
DEST[255:0] ←INTERLEAVE_WORDS_256b(SRC1, SRC2)
DEST[MAXVL-1:256] ←0
```

VPUNPCKLWD (EVEX.512 encoded instruction)

```
(KL, VL) = (8, 128), (16, 256), (32, 512)
IF VL = 128
    TMP_DEST[VL-1:0] ← INTERLEAVE_WORDS(SRC1[VL-1:0], SRC2[VL-1:0])
FI;
IF VL = 256
    TMP_DEST[VL-1:0] ← INTERLEAVE_WORDS_256b(SRC1[VL-1:0], SRC2[VL-1:0])
FI;
IF VL = 512
    TMP_DEST[VL-1:0] ← INTERLEAVE_WORDS_512b(SRC1[VL-1:0], SRC2[VL-1:0])
FI;

FOR j ← 0 TO KL-1
    i ← j * 16
    IF k1[j] OR *no writemask*
```

```
            THEN DEST[i+15:i] ← TMP_DEST[i+15:i]
            ELSE
                IF *merging-masking*                    ; merging-masking
                    THEN *DEST[i+15:i] remains unchanged*
                    ELSE *zeroing-masking*              ; zeroing-masking
                        DEST[i+15:i] ← 0
                FI
        FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0
DEST[511:0] ← INTERLEAVE_WORDS_512b(SRC1, SRC2)
```

PUNPCKLDQ

```
DEST[127:0] ←INTERLEAVE_DWORDS(DEST, SRC)
DEST[MAXVL-1:128] (Unmodified)
```

VPUNPCKLDQ (VEX.128 encoded instruction)

```
DEST[127:0] ←INTERLEAVE_DWORDS(SRC1, SRC2)
DEST[MAXVL-1:128] ←0
```

VPUNPCKLDQ (VEX.256 encoded instruction)

```
DEST[255:0] ←INTERLEAVE_DWORDS_256b(SRC1, SRC2)
DEST[MAXVL-1:256] ←0
```

VPUNPCKLDQ (EVEX encoded instructions)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j ← 0 TO KL-1
    i ← j * 32
    IF (EVEX.b = 1) AND (SRC2 *is memory*)
        THEN TMP_SRC2[i+31:i] ← SRC2[31:0]
        ELSE TMP_SRC2[i+31:i] ← SRC2[i+31:i]
    FI;
ENDFOR;
IF VL = 128
    TMP_DEST[VL-1:0] ← INTERLEAVE_DWORDS(SRC1[VL-1:0], TMP_SRC2[VL-1:0])
FI;
IF VL = 256
    TMP_DEST[VL-1:0] ← INTERLEAVE_DWORDS_256b(SRC1[VL-1:0], TMP_SRC2[VL-1:0])
FI;
IF VL = 512
    TMP_DEST[VL-1:0] ← INTERLEAVE_DWORDS_512b(SRC1[VL-1:0], TMP_SRC2[VL-1:0])
FI;

FOR j ← 0 TO KL-1
    i ← j * 32
    IF k1[j] OR *no writemask*
        THEN DEST[i+31:i] ← TMP_DEST[i+31:i]
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE *zeroing-masking*              ; zeroing-masking
                    DEST[i+31:i] ← 0
            FI
    FI;
```

```
ENDFOR
DEST511:0] ←INTERLEAVE_DWORDS_512b(SRC1, SRC2)
DEST[MAXVL-1:VL] ← 0
```

PUNPCKLQDQ

```
DEST[127:0] ←INTERLEAVE_QWORDS(DEST, SRC)
DEST[MAXVL-1:128] (Unmodified)
```

VPUNPCKLQDQ (VEX.128 encoded instruction)

```
DEST[127:0] ←INTERLEAVE_QWORDS(SRC1, SRC2)
DEST[MAXVL-1:128] ←0
```

VPUNPCKLQDQ (VEX.256 encoded instruction)

```
DEST[255:0] ←INTERLEAVE_QWORDS_256b(SRC1, SRC2)
DEST[MAXVL-1:256] ←0
```

VPUNPCKLQDQ (EVEX encoded instructions)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j ← 0 TO KL-1
    i ← j * 64
    IF (EVEX.b = 1) AND (SRC2 *is memory*)
        THEN TMP_SRC2[i+63:i] ← SRC2[63:0]
        ELSE TMP_SRC2[i+63:i] ← SRC2[i+63:i]
    FI;
ENDFOR;
IF VL = 128
    TMP_DEST[VL-1:0] ← INTERLEAVE_QWORDS(SRC1[VL-1:0], TMP_SRC2[VL-1:0])
FI;
IF VL = 256
    TMP_DEST[VL-1:0] ← INTERLEAVE_QWORDS_256b(SRC1[VL-1:0], TMP_SRC2[VL-1:0])
FI;
IF VL = 512
    TMP_DEST[VL-1:0] ← INTERLEAVE_QWORDS_512b(SRC1[VL-1:0], TMP_SRC2[VL-1:0])
FI;

FOR j ← 0 TO KL-1
    i ← j * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+63:i] ← TMP_DEST[i+63:i]
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE *zeroing-masking*              ; zeroing-masking
                    DEST[i+63:i] ← 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0
```

Intel C/C++ Compiler Intrinsic Equivalents

```
VPUNPCKLBW __m512i _mm512_unpacklo_epi8(__m512i a, __m512i b);
VPUNPCKLBW __m512i _mm512_mask_unpacklo_epi8(__m512i s, __mmask64 k, __m512i a, __m512i b);
VPUNPCKLBW __m512i _mm512_maskz_unpacklo_epi8( __mmask64 k, __m512i a, __m512i b);
VPUNPCKLBW __m256i _mm256_mask_unpacklo_epi8(__m256i s, __mmask32 k, __m256i a, __m256i b);
```

VPUNPCKLBW __m256i _mm256_maskz_unpacklo_epi8(__mmask32 k, __m256i a, __m256i b);
VPUNPCKLBW __m128i _mm_mask_unpacklo_epi8(v s, __mmask16 k, __m128i a, __m128i b);
VPUNPCKLBW __m128i _mm_maskz_unpacklo_epi8(__mmask16 k, __m128i a, __m128i b);
VPUNPCKLWD __m512i _mm512_unpacklo_epi16(__m512i a, __m512i b);
VPUNPCKLWD __m512i _mm512_mask_unpacklo_epi16(__m512i s, __mmask32 k, __m512i a, __m512i b);
VPUNPCKLWD __m512i _mm512_maskz_unpacklo_epi16(__mmask32 k, __m512i a, __m512i b);
VPUNPCKLWD __m256i _mm256_mask_unpacklo_epi16(__m256i s, __mmask16 k, __m256i a, __m256i b);
VPUNPCKLWD __m256i _mm256_maskz_unpacklo_epi16(__mmask16 k, __m256i a, __m256i b);
VPUNPCKLWD __m128i _mm_mask_unpacklo_epi16(v s, __mmask8 k, __m128i a, __m128i b);
VPUNPCKLWD __m128i _mm_maskz_unpacklo_epi16(__mmask8 k, __m128i a, __m128i b);
VPUNPCKLDQ __m512i _mm512_unpacklo_epi32(__m512i a, __m512i b);
VPUNPCKLDQ __m512i _mm512_mask_unpacklo_epi32(__m512i s, __mmask16 k, __m512i a, __m512i b);
VPUNPCKLDQ __m512i _mm512_maskz_unpacklo_epi32(__mmask16 k, __m512i a, __m512i b);
VPUNPCKLDQ __m256i _mm256_mask_unpacklo_epi32(__m256i s, __mmask8 k, __m256i a, __m256i b);
VPUNPCKLDQ __m256i _mm256_maskz_unpacklo_epi32(__mmask8 k, __m256i a, __m256i b);
VPUNPCKLDQ __m128i _mm_mask_unpacklo_epi32(v s, __mmask8 k, __m128i a, __m128i b);
VPUNPCKLDQ __m128i _mm_maskz_unpacklo_epi32(__mmask8 k, __m128i a, __m128i b);
VPUNPCKLQDQ __m512i _mm512_unpacklo_epi64(__m512i a, __m512i b);
VPUNPCKLQDQ __m512i _mm512_mask_unpacklo_epi64(__m512i s, __mmask8 k, __m512i a, __m512i b);
VPUNPCKLQDQ __m512i _mm512_maskz_unpacklo_epi64(__mmask8 k, __m512i a, __m512i b);
VPUNPCKLQDQ __m256i _mm256_mask_unpacklo_epi64(__m256i s, __mmask8 k, __m256i a, __m256i b);
VPUNPCKLQDQ __m256i _mm256_maskz_unpacklo_epi64(__mmask8 k, __m256i a, __m256i b);
VPUNPCKLQDQ __m128i _mm_mask_unpacklo_epi64(__m128i s, __mmask8 k, __m128i a, __m128i b);
VPUNPCKLQDQ __m128i _mm_maskz_unpacklo_epi64(__mmask8 k, __m128i a, __m128i b);
PUNPCKLBW:__m64 _mm_unpacklo_pi8 (__m64 m1, __m64 m2)
(V)PUNPCKLBW:__m128i _mm_unpacklo_epi8 (__m128i m1, __m128i m2)
VPUNPCKLBW:__m256i _mm256_unpacklo_epi8 (__m256i m1, __m256i m2)
PUNPCKLWD:__m64 _mm_unpacklo_pi16 (__m64 m1, __m64 m2)
(V)PUNPCKLWD:__m128i _mm_unpacklo_epi16 (__m128i m1, __m128i m2)
VPUNPCKLWD:__m256i _mm256_unpacklo_epi16 (__m256i m1, __m256i m2)
PUNPCKLDQ:__m64 _mm_unpacklo_pi32 (__m64 m1, __m64 m2)
(V)PUNPCKLDQ:__m128i _mm_unpacklo_epi32 (__m128i m1, __m128i m2)
VPUNPCKLDQ:__m256i _mm256_unpacklo_epi32 (__m256i m1, __m256i m2)
(V)PUNPCKLQDQ:__m128i _mm_unpacklo_epi64 (__m128i m1, __m128i m2)
VPUNPCKLQDQ:__m256i _mm256_unpacklo_epi64 (__m256i m1, __m256i m2)

Flags Affected

None.

Numeric Exceptions

None.

Other Exceptions

Non-EVEX-encoded instruction, see Exceptions Type 4.
EVEX-encoded VPUNPCKLDQ/QDQ, see Exceptions Type E4NF.
EVEX-encoded VPUNPCKLBW/WD, see Exceptions Type E4NF.nb.

PUSH—Push Word, Doubleword or Quadword Onto the Stack

Opcode*	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
FF /6	PUSH *r/m16*	M	Valid	Valid	Push *r/m16.*
FF /6	PUSH *r/m32*	M	N.E.	Valid	Push *r/m32.*
FF /6	PUSH *r/m64*	M	Valid	N.E.	Push *r/m64.*
50+*rw*	PUSH *r16*	O	Valid	Valid	Push *r16.*
50+*rd*	PUSH *r32*	O	N.E.	Valid	Push *r32.*
50+*rd*	PUSH *r64*	O	Valid	N.E.	Push *r64.*
6A ib	PUSH *imm8*	I	Valid	Valid	Push *imm8.*
68 iw	PUSH *imm16*	I	Valid	Valid	Push *imm16.*
68 id	PUSH *imm32*	I	Valid	Valid	Push *imm32.*
0E	PUSH CS	ZO	Invalid	Valid	Push CS.
16	PUSH SS	ZO	Invalid	Valid	Push SS.
1E	PUSH DS	ZO	Invalid	Valid	Push DS.
06	PUSH ES	ZO	Invalid	Valid	Push ES.
0F A0	PUSH FS	ZO	Valid	Valid	Push FS.
0F A8	PUSH GS	ZO	Valid	Valid	Push GS.

NOTES:

* See IA-32 Architecture Compatibility section below.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
M	ModRM:r/m (r)	NA	NA	NA
O	opcode + rd (r)	NA	NA	NA
I	imm8/16/32	NA	NA	NA
ZO	NA	NA	NA	NA

Description

Decrements the stack pointer and then stores the source operand on the top of the stack. Address and operand sizes are determined and used as follows:

- Address size. The D flag in the current code-segment descriptor determines the default address size; it may be overridden by an instruction prefix (67H).

  The address size is used only when referencing a source operand in memory.

- Operand size. The D flag in the current code-segment descriptor determines the default operand size; it may be overridden by instruction prefixes (66H or REX.W).

  The operand size (16, 32, or 64 bits) determines the amount by which the stack pointer is decremented (2, 4 or 8).

  If the source operand is an immediate of size less than the operand size, a sign-extended value is pushed on the stack. If the source operand is a segment register (16 bits) and the operand size is 64-bits, a zero-extended value is pushed on the stack; if the operand size is 32-bits, either a zero-extended value is pushed on the stack or the segment selector is written on the stack using a 16-bit move. For the last case, all recent Core and Atom processors perform a 16-bit move, leaving the upper portion of the stack location unmodified.

- Stack-address size. Outside of 64-bit mode, the B flag in the current stack-segment descriptor determines the size of the stack pointer (16 or 32 bits); in 64-bit mode, the size of the stack pointer is always 64 bits.

The stack-address size determines the width of the stack pointer when writing to the stack in memory and when decrementing the stack pointer. (As stated above, the amount by which the stack pointer is decremented is determined by the operand size.)

If the operand size is less than the stack-address size, the PUSH instruction may result in a misaligned stack pointer (a stack pointer that is not aligned on a doubleword or quadword boundary).

The PUSH ESP instruction pushes the value of the ESP register as it existed before the instruction was executed. If a PUSH instruction uses a memory operand in which the ESP register is used for computing the operand address, the address of the operand is computed before the ESP register is decremented.

If the ESP or SP register is 1 when the PUSH instruction is executed in real-address mode, a stack-fault exception (#SS) is generated (because the limit of the stack segment is violated). Its delivery encounters a second stack-fault exception (for the same reason), causing generation of a double-fault exception (#DF). Delivery of the double-fault exception encounters a third stack-fault exception, and the logical processor enters shutdown mode. See the discussion of the double-fault exception in Chapter 6 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*.

IA-32 Architecture Compatibility

For IA-32 processors from the Intel 286 on, the PUSH ESP instruction pushes the value of the ESP register as it existed before the instruction was executed. (This is also true for Intel 64 architecture, real-address and virtual-8086 modes of IA-32 architecture.) For the Intel® 8086 processor, the PUSH SP instruction pushes the new value of the SP register (that is the value after it has been decremented by 2).

Operation

```
(* See Description section for possible sign-extension or zero-extension of source operand and for *)
(* a case in which the size of the memory store may be smaller than the instruction's operand size *)
IF StackAddrSize = 64
    THEN
        IF OperandSize = 64
            THEN
                RSP ← RSP – 8;
                Memory[SS:RSP] ← SRC;           (* push quadword *)
        ELSE IF OperandSize = 32
            THEN
                RSP ← RSP – 4;
                Memory[SS:RSP] ← SRC;           (* push dword *)
            ELSE (* OperandSize = 16 *)
                RSP ← RSP – 2;
                Memory[SS:RSP] ← SRC;           (* push word *)
        FI;
ELSE IF StackAddrSize = 32
    THEN
        IF OperandSize = 64
            THEN
                ESP ← ESP – 8;
                Memory[SS:ESP] ← SRC;           (* push quadword *)
        ELSE IF OperandSize = 32
            THEN
                ESP ← ESP – 4;
                Memory[SS:ESP] ← SRC;           (* push dword *)
            ELSE (* OperandSize = 16 *)
                ESP ← ESP – 2;
                Memory[SS:ESP] ← SRC;           (* push word *)
        FI;
    ELSE (* StackAddrSize = 16 *)
```

```
        IF OperandSize = 32
            THEN
                SP ← SP – 4;
                Memory[SS:SP] ← SRC;            (* push dword *)
            ELSE (* OperandSize = 16 *)
                SP ← SP – 2;
                Memory[SS:SP] ← SRC;            (* push word *)
        FI;
FI;
```

Flags Affected

None.

Protected Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
	If the DS, ES, FS, or GS register is used to access memory and it contains a NULL segment selector.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If the LOCK prefix is used.

Real-Address Mode Exceptions

#GP	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS	If a memory operand effective address is outside the SS segment limit.
	If the new value of the SP or ESP register is outside the stack segment limit.
#UD	If the LOCK prefix is used.

Virtual-8086 Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.
#UD	If the LOCK prefix is used.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#GP(0)	If the memory address is in a non-canonical form.
#SS(0)	If the stack address is in a non-canonical form.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If the LOCK prefix is used.
	If the PUSH is of CS, SS, DS, or ES.

PUSHA/PUSHAD—Push All General-Purpose Registers

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
60	PUSHA	ZO	Invalid	Valid	Push AX, CX, DX, BX, original SP, BP, SI, and DI.
60	PUSHAD	ZO	Invalid	Valid	Push EAX, ECX, EDX, EBX, original ESP, EBP, ESI, and EDI.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
ZO	NA	NA	NA	NA

Description

Pushes the contents of the general-purpose registers onto the stack. The registers are stored on the stack in the following order: EAX, ECX, EDX, EBX, ESP (original value), EBP, ESI, and EDI (if the current operand-size attribute is 32) and AX, CX, DX, BX, SP (original value), BP, SI, and DI (if the operand-size attribute is 16). These instructions perform the reverse operation of the POPA/POPAD instructions. The value pushed for the ESP or SP register is its value before prior to pushing the first register (see the "Operation" section below).

The PUSHA (push all) and PUSHAD (push all double) mnemonics reference the same opcode. The PUSHA instruction is intended for use when the operand-size attribute is 16 and the PUSHAD instruction for when the operand-size attribute is 32. Some assemblers may force the operand size to 16 when PUSHA is used and to 32 when PUSHAD is used. Others may treat these mnemonics as synonyms (PUSHA/PUSHAD) and use the current setting of the operand-size attribute to determine the size of values to be pushed from the stack, regardless of the mnemonic used.

In the real-address mode, if the ESP or SP register is 1, 3, or 5 when PUSHA/PUSHAD executes: an #SS exception is generated but not delivered (the stack error reported prevents #SS delivery). Next, the processor generates a #DF exception and enters a shutdown state as described in the #DF discussion in Chapter 6 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*.

This instruction executes as described in compatibility mode and legacy mode. It is not valid in 64-bit mode.

Operation

```
IF 64-bit Mode
    THEN #UD
FI;

IF OperandSize = 32 (* PUSHAD instruction *)
    THEN
        Temp ← (ESP);
        Push(EAX);
        Push(ECX);
        Push(EDX);
        Push(EBX);
        Push(Temp);
        Push(EBP);
        Push(ESI);
        Push(EDI);
    ELSE (* OperandSize = 16, PUSHA instruction *)
        Temp ← (SP);
        Push(AX);
        Push(CX);
        Push(DX);
```

```
        Push(BX);
        Push(Temp);
        Push(BP);
        Push(SI);
        Push(DI);
FI;
```

Flags Affected

None.

Protected Mode Exceptions

#SS(0)	If the starting or ending stack address is outside the stack segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If an unaligned memory reference is made while the current privilege level is 3 and alignment checking is enabled.
#UD	If the LOCK prefix is used.

Real-Address Mode Exceptions

#GP	If the ESP or SP register contains 7, 9, 11, 13, or 15.
#UD	If the LOCK prefix is used.

Virtual-8086 Mode Exceptions

#GP(0)	If the ESP or SP register contains 7, 9, 11, 13, or 15.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If an unaligned memory reference is made while alignment checking is enabled.
#UD	If the LOCK prefix is used.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#UD	If in 64-bit mode.

PUSHF/PUSHFD/PUSHFQ—Push EFLAGS Register onto the Stack

Opcode*	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
9C	PUSHF	ZO	Valid	Valid	Push lower 16 bits of EFLAGS.
9C	PUSHFD	ZO	N.E.	Valid	Push EFLAGS.
9C	PUSHFQ	ZO	Valid	N.E.	Push RFLAGS.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
ZO	NA	NA	NA	NA

Description

Decrements the stack pointer by 4 (if the current operand-size attribute is 32) and pushes the entire contents of the EFLAGS register onto the stack, or decrements the stack pointer by 2 (if the operand-size attribute is 16) and pushes the lower 16 bits of the EFLAGS register (that is, the FLAGS register) onto the stack. These instructions reverse the operation of the POPF/POPFD instructions.

When copying the entire EFLAGS register to the stack, the VM and RF flags (bits 16 and 17) are not copied; instead, the values for these flags are cleared in the EFLAGS image stored on the stack. See Chapter 3 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*, for more information about the EFLAGS register.

The PUSHF (push flags) and PUSHFD (push flags double) mnemonics reference the same opcode. The PUSHF instruction is intended for use when the operand-size attribute is 16 and the PUSHFD instruction for when the operand-size attribute is 32. Some assemblers may force the operand size to 16 when PUSHF is used and to 32 when PUSHFD is used. Others may treat these mnemonics as synonyms (PUSHF/PUSHFD) and use the current setting of the operand-size attribute to determine the size of values to be pushed from the stack, regardless of the mnemonic used.

In 64-bit mode, the instruction's default operation is to decrement the stack pointer (RSP) by 8 and pushes RFLAGS on the stack. 16-bit operation is supported using the operand size override prefix 66H. 32-bit operand size cannot be encoded in this mode. When copying RFLAGS to the stack, the VM and RF flags (bits 16 and 17) are not copied; instead, values for these flags are cleared in the RFLAGS image stored on the stack.

When operating in virtual-8086 mode (EFLAGS.VM = 1) without the virtual-8086 mode extensions (CR4.VME = 0), the PUSHF/PUSHFD instructions can be used only if IOPL = 3; otherwise, a general-protection exception (#GP) occurs. If the virtual-8086 mode extensions are enabled (CR4.VME = 1), PUSHF (but not PUSHFD) can be executed in virtual-8086 mode with IOPL < 3.

(The protected-mode virtual-interrupt feature — enabled by setting CR4.PVI — affects the CLI and STI instructions in the same manner as the virtual-8086 mode extensions. PUSHF, however, is not affected by CR4.PVI.)

In the real-address mode, if the ESP or SP register is 1 when PUSHF/PUSHFD instruction executes: an #SS exception is generated but not delivered (the stack error reported prevents #SS delivery). Next, the processor generates a #DF exception and enters a shutdown state as described in the #DF discussion in Chapter 6 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*.

Operation

```
IF (PE = 0) or (PE = 1 and ((VM = 0) or (VM = 1 and IOPL = 3)))
(* Real-Address Mode, Protected mode, or Virtual-8086 mode with IOPL equal to 3 *)
    THEN
        IF OperandSize = 32
            THEN
                push (EFLAGS AND 00FCFFFFH);
                (* VM and RF bits are cleared in image stored on the stack *)
            ELSE
                push (EFLAGS); (* Lower 16 bits only *)
```

```
        FI;

    ELSE IF 64-bit MODE (* In 64-bit Mode *)
        IF OperandSize = 64
            THEN
                push (RFLAGS AND 00000000_00FCFFFFH);
                (* VM and RF bits are cleared in image stored on the stack; *)
            ELSE
                push (EFLAGS); (* Lower 16 bits only *)
        FI;

    ELSE (* In Virtual-8086 Mode with IOPL less than 3 *)
        IF (CR4.VME = 0) OR (OperandSize = 32)
            THEN #GP(0); (* Trap to virtual-8086 monitor *)
            ELSE
                tempFLAGS = EFLAGS[15:0];
                tempFLAGS[9] = tempFLAGS[19];       (* VIF replaces IF *)
                tempFlags[13:12] = 3;  (* IOPL is set to 3 in image stored on the stack *)
                push (tempFLAGS);
        FI;
FI;
```

Flags Affected

None.

Protected Mode Exceptions

#SS(0)	If the new value of the ESP register is outside the stack segment boundary.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If an unaligned memory reference is made while CPL = 3 and alignment checking is enabled.
#UD	If the LOCK prefix is used.

Real-Address Mode Exceptions

#UD	If the LOCK prefix is used.

Virtual-8086 Mode Exceptions

#GP(0)	If the I/O privilege level is less than 3.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If an unaligned memory reference is made while alignment checking is enabled.
#UD	If the LOCK prefix is used.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#GP(0)	If the memory address is in a non-canonical form.
#SS(0)	If the stack address is in a non-canonical form.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If an unaligned memory reference is made while CPL = 3 and alignment checking is enabled.
#UD	If the LOCK prefix is used.

PXOR—Logical Exclusive OR

Opcode*/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F EF /r[1] PXOR *mm, mm/m64*	A	V/V	MMX	Bitwise XOR of *mm/m64* and *mm*.
66 0F EF /r PXOR *xmm1, xmm2/m128*	A	V/V	SSE2	Bitwise XOR of *xmm2/m128* and *xmm1*.
VEX.NDS.128.66.0F.WIG EF /r VPXOR *xmm1, xmm2, xmm3/m128*	B	V/V	AVX	Bitwise XOR of *xmm3/m128* and *xmm2*.
VEX.NDS.256.66.0F.WIG EF /r VPXOR *ymm1, ymm2, ymm3/m256*	B	V/V	AVX2	Bitwise XOR of *ymm3/m256* and *ymm2*.
EVEX.NDS.128.66.0F.W0 EF /r VPXORD xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst	C	V/V	AVX512VL AVX512F	Bitwise XOR of packed doubleword integers in xmm2 and xmm3/m128 using writemask k1.
EVEX.NDS.256.66.0F.W0 EF /r VPXORD ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst	C	V/V	AVX512VL AVX512F	Bitwise XOR of packed doubleword integers in ymm2 and ymm3/m256 using writemask k1.
EVEX.NDS.512.66.0F.W0 EF /r VPXORD zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst	C	V/V	AVX512F	Bitwise XOR of packed doubleword integers in zmm2 and zmm3/m512/m32bcst using writemask k1.
EVEX.NDS.128.66.0F.W1 EF /r VPXORQ xmm1 {k1}{z}, xmm2, xmm3/m128/m64bcst	C	V/V	AVX512VL AVX512F	Bitwise XOR of packed quadword integers in xmm2 and xmm3/m128 using writemask k1.
EVEX.NDS.256.66.0F.W1 EF /r VPXORQ ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst	C	V/V	AVX512VL AVX512F	Bitwise XOR of packed quadword integers in ymm2 and ymm3/m256 using writemask k1.
EVEX.NDS.512.66.0F.W1 EF /r VPXORQ zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst	C	V/V	AVX512F	Bitwise XOR of packed quadword integers in zmm2 and zmm3/m512/m64bcst using writemask k1.

NOTES:

1. See note in Section 2.4, "AVX and SSE Instruction Exception Specification" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A* and Section 22.25.3, "Exception Conditions of Legacy SIMD Instructions Operating on MMX Registers" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	NA	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
B	NA	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA
C	Full Vector	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA

Description

Performs a bitwise logical exclusive-OR (XOR) operation on the source operand (second operand) and the destination operand (first operand) and stores the result in the destination operand. Each bit of the result is 1 if the corresponding bits of the two operands are different; each bit is 0 if the corresponding bits of the operands are the same.

In 64-bit mode and not encoded with VEX/EVEX, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

Legacy SSE instructions 64-bit operand: The source operand can be an MMX technology register or a 64-bit memory location. The destination operand is an MMX technology register.

128-bit Legacy SSE version: The second source operand is an XMM register or a 128-bit memory location. The first source operand and destination operands are XMM registers. Bits (MAXVL-1:128) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: The second source operand is an XMM register or a 128-bit memory location. The first source operand and destination operands are XMM registers. Bits (MAXVL-1:128) of the destination YMM register are zeroed.

VEX.256 encoded version: The first source operand is a YMM register. The second source operand is a YMM register or a 256-bit memory location. The destination operand is a YMM register. The upper bits (MAXVL-1:256) of the corresponding register destination are zeroed.

EVEX encoded versions: The first source operand is a ZMM/YMM/XMM register. The second source operand can be a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 32/64-bit memory location. The destination operand is a ZMM/YMM/XMM register conditionally updated with writemask k1.

Operation

PXOR (64-bit operand)

```
DEST ← DEST XOR SRC
```

PXOR (128-bit Legacy SSE version)

```
DEST ← DEST XOR SRC
DEST[MAXVL-1:128] (Unmodified)
```

VPXOR (VEX.128 encoded version)

```
DEST ← SRC1 XOR SRC2
DEST[MAXVL-1:128] ← 0
```

VPXOR (VEX.256 encoded version)

```
DEST ← SRC1 XOR SRC2
DEST[MAXVL-1:256] ← 0
```

VPXORD (EVEX encoded versions)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j ← 0 TO KL-1
    i ← j * 32
    IF k1[j] OR *no writemask* THEN
            IF (EVEX.b = 1) AND (SRC2 *is memory*)
                THEN DEST[i+31:i] ← SRC1[i+31:i] BITWISE XOR SRC2[31:0]
                ELSE DEST[i+31:i] ← SRC1[i+31:i] BITWISE XOR SRC2[i+31:i]
            FI;
    ELSE
        IF *merging-masking*                    ; merging-masking
            THEN *DEST[31:0] remains unchanged*
            ELSE                                ; zeroing-masking
                DEST[31:0] ← 0
        FI;
    FI;
ENDFOR;
DEST[MAXVL-1:VL] ← 0
```

VPXORQ (EVEX encoded versions)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j ← 0 TO KL-1
    i ← j * 64
    IF k1[j] OR *no writemask* THEN
                IF (EVEX.b = 1) AND (SRC2 *is memory*)
                    THEN DEST[i+63:i] ← SRC1[i+63:i] BITWISE XOR SRC2[63:0]
                    ELSE DEST[i+63:i] ← SRC1[i+63:i] BITWISE XOR SRC2[i+63:i]
                FI;
    ELSE
        IF *merging-masking*                    ; merging-masking
            THEN *DEST[63:0] remains unchanged*
            ELSE                                ; zeroing-masking
                DEST[63:0] ← 0
        FI;
    FI;
ENDFOR;
DEST[MAXVL-1:VL] ← 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VPXORD __m512i _mm512_xor_epi32(__m512i a, __m512i b)
VPXORD __m512i _mm512_mask_xor_epi32(__m512i s, __mmask16 m, __m512i a, __m512i b)
VPXORD __m512i _mm512_maskz_xor_epi32(__mmask16 m, __m512i a, __m512i b)
VPXORD __m256i _mm256_xor_epi32(__m256i a, __m256i b)
VPXORD __m256i _mm256_mask_xor_epi32(__m256i s, __mmask8 m, __m256i a, __m256i b)
VPXORD __m256i _mm256_maskz_xor_epi32(__mmask8 m, __m256i a, __m256i b)
VPXORD __m128i _mm_xor_epi32(__m128i a, __m128i b)
VPXORD __m128i _mm_mask_xor_epi32(__m128i s, __mmask8 m, __m128i a, __m128i b)
VPXORD __m128i _mm_maskz_xor_epi32(__mmask16 m, __m128i a, __m128i b)
VPXORQ __m512i _mm512_xor_epi64(__m512i a, __m512i b);
VPXORQ __m512i _mm512_mask_xor_epi64(__m512i s, __mmask8 m, __m512i a, __m512i b);
VPXORQ __m512i _mm512_maskz_xor_epi64(__mmask8 m, __m512i a, __m512i b);
VPXORQ __m256i _mm256_xor_epi64(__m256i a, __m256i b);
VPXORQ __m256i _mm256_mask_xor_epi64(__m256i s, __mmask8 m, __m256i a, __m256i b);
VPXORQ __m256i _mm256_maskz_xor_epi64(__mmask8 m, __m256i a, __m256i b);
VPXORQ __m128i _mm_xor_epi64(__m128i a, __m128i b);
VPXORQ __m128i _mm_mask_xor_epi64(__m128i s, __mmask8 m, __m128i a, __m128i b);
VPXORQ __m128i _mm_maskz_xor_epi64(__mmask8 m, __m128i a, __m128i b);
PXOR:__m64 _mm_xor_si64 (__m64 m1, __m64 m2)
(V)PXOR:__m128i _mm_xor_si128 (__m128i a, __m128i b)
VPXOR:__m256i _mm256_xor_si256 (__m256i a, __m256i b)

Flags Affected

None.

Numeric Exceptions

None.

Other Exceptions

Non-EVEX-encoded instruction, see Exceptions Type 4.
EVEX-encoded instruction, see Exceptions Type E4.

RCL/RCR/ROL/ROR—Rotate

Opcode**	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
D0 /2	RCL *r/m8*, 1	M1	Valid	Valid	Rotate 9 bits (CF, *r/m8*) left once.
REX + D0 /2	RCL *r/m8**, 1	M1	Valid	N.E.	Rotate 9 bits (CF, *r/m8*) left once.
D2 /2	RCL *r/m8*, CL	MC	Valid	Valid	Rotate 9 bits (CF, *r/m8*) left CL times.
REX + D2 /2	RCL *r/m8**, CL	MC	Valid	N.E.	Rotate 9 bits (CF, *r/m8*) left CL times.
C0 /2 *ib*	RCL *r/m8, imm8*	MI	Valid	Valid	Rotate 9 bits (CF, *r/m8*) left *imm8* times.
REX + C0 /2 *ib*	RCL *r/m8*, imm8*	MI	Valid	N.E.	Rotate 9 bits (CF, *r/m8*) left *imm8* times.
D1 /2	RCL *r/m16*, 1	M1	Valid	Valid	Rotate 17 bits (CF, *r/m16*) left once.
D3 /2	RCL *r/m16*, CL	MC	Valid	Valid	Rotate 17 bits (CF, *r/m16*) left CL times.
C1 /2 *ib*	RCL *r/m16, imm8*	MI	Valid	Valid	Rotate 17 bits (CF, *r/m16*) left *imm8* times.
D1 /2	RCL *r/m32*, 1	M1	Valid	Valid	Rotate 33 bits (CF, *r/m32*) left once.
REX.W + D1 /2	RCL *r/m64*, 1	M1	Valid	N.E.	Rotate 65 bits (CF, *r/m64*) left once. Uses a 6 bit count.
D3 /2	RCL *r/m32*, CL	MC	Valid	Valid	Rotate 33 bits (CF, *r/m32*) left CL times.
REX.W + D3 /2	RCL *r/m64*, CL	MC	Valid	N.E.	Rotate 65 bits (CF, *r/m64*) left CL times. Uses a 6 bit count.
C1 /2 *ib*	RCL *r/m32, imm8*	MI	Valid	Valid	Rotate 33 bits (CF, *r/m32*) left *imm8* times.
REX.W + C1 /2 *ib*	RCL *r/m64, imm8*	MI	Valid	N.E.	Rotate 65 bits (CF, *r/m64*) left *imm8* times. Uses a 6 bit count.
D0 /3	RCR *r/m8*, 1	M1	Valid	Valid	Rotate 9 bits (CF, *r/m8*) right once.
REX + D0 /3	RCR *r/m8**, 1	M1	Valid	N.E.	Rotate 9 bits (CF, *r/m8*) right once.
D2 /3	RCR *r/m8*, CL	MC	Valid	Valid	Rotate 9 bits (CF, *r/m8*) right CL times.
REX + D2 /3	RCR *r/m8**, CL	MC	Valid	N.E.	Rotate 9 bits (CF, *r/m8*) right CL times.
C0 /3 *ib*	RCR *r/m8, imm8*	MI	Valid	Valid	Rotate 9 bits (CF, *r/m8*) right *imm8* times.
REX + C0 /3 *ib*	RCR *r/m8*, imm8*	MI	Valid	N.E.	Rotate 9 bits (CF, *r/m8*) right *imm8* times.
D1 /3	RCR *r/m16*, 1	M1	Valid	Valid	Rotate 17 bits (CF, *r/m16*) right once.
D3 /3	RCR *r/m16*, CL	MC	Valid	Valid	Rotate 17 bits (CF, *r/m16*) right CL times.
C1 /3 *ib*	RCR *r/m16, imm8*	MI	Valid	Valid	Rotate 17 bits (CF, *r/m16*) right *imm8* times.
D1 /3	RCR *r/m32*, 1	M1	Valid	Valid	Rotate 33 bits (CF, *r/m32*) right once. Uses a 6 bit count.
REX.W + D1 /3	RCR *r/m64*, 1	M1	Valid	N.E.	Rotate 65 bits (CF, *r/m64*) right once. Uses a 6 bit count.
D3 /3	RCR *r/m32*, CL	MC	Valid	Valid	Rotate 33 bits (CF, *r/m32*) right CL times.
REX.W + D3 /3	RCR *r/m64*, CL	MC	Valid	N.E.	Rotate 65 bits (CF, *r/m64*) right CL times. Uses a 6 bit count.
C1 /3 *ib*	RCR *r/m32, imm8*	MI	Valid	Valid	Rotate 33 bits (CF, *r/m32*) right *imm8* times.
REX.W + C1 /3 *ib*	RCR *r/m64, imm8*	MI	Valid	N.E.	Rotate 65 bits (CF, *r/m64*) right *imm8* times. Uses a 6 bit count.
D0 /0	ROL *r/m8*, 1	M1	Valid	Valid	Rotate 8 bits *r/m8* left once.
REX + D0 /0	ROL *r/m8**, 1	M1	Valid	N.E.	Rotate 8 bits *r/m8* left once
D2 /0	ROL *r/m8*, CL	MC	Valid	Valid	Rotate 8 bits *r/m8* left CL times.
REX + D2 /0	ROL *r/m8**, CL	MC	Valid	N.E.	Rotate 8 bits *r/m8* left CL times.
C0 /0 *ib*	ROL *r/m8, imm8*	MI	Valid	Valid	Rotate 8 bits *r/m8* left *imm8* times.

Opcode**	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
REX + C0 /0 *ib*	ROL *r/m8*, imm8*	MI	Valid	N.E.	Rotate 8 bits *r/m8* left *imm8* times.
D1 /0	ROL *r/m16*, 1	M1	Valid	Valid	Rotate 16 bits *r/m16* left once.
D3 /0	ROL *r/m16*, CL	MC	Valid	Valid	Rotate 16 bits *r/m16* left CL times.
C1 /0 *ib*	ROL *r/m16, imm8*	MI	Valid	Valid	Rotate 16 bits *r/m16* left *imm8* times.
D1 /0	ROL *r/m32*, 1	M1	Valid	Valid	Rotate 32 bits *r/m32* left once.
REX.W + D1 /0	ROL *r/m64*, 1	M1	Valid	N.E.	Rotate 64 bits *r/m64* left once. Uses a 6 bit count.
D3 /0	ROL *r/m32*, CL	MC	Valid	Valid	Rotate 32 bits *r/m32* left CL times.
REX.W + D3 /0	ROL *r/m64*, CL	MC	Valid	N.E.	Rotate 64 bits *r/m64* left CL times. Uses a 6 bit count.
C1 /0 *ib*	ROL *r/m32, imm8*	MI	Valid	Valid	Rotate 32 bits *r/m32* left *imm8* times.
REX.W + C1 /0 *ib*	ROL *r/m64, imm8*	MI	Valid	N.E.	Rotate 64 bits *r/m64* left *imm8* times. Uses a 6 bit count.
D0 /1	ROR *r/m8*, 1	M1	Valid	Valid	Rotate 8 bits *r/m8* right once.
REX + D0 /1	ROR *r/m8**, 1	M1	Valid	N.E.	Rotate 8 bits *r/m8* right once.
D2 /1	ROR *r/m8*, CL	MC	Valid	Valid	Rotate 8 bits *r/m8* right CL times.
REX + D2 /1	ROR *r/m8**, CL	MC	Valid	N.E.	Rotate 8 bits *r/m8* right CL times.
C0 /1 *ib*	ROR *r/m8, imm8*	MI	Valid	Valid	Rotate 8 bits *r/m16* right *imm8* times.
REX + C0 /1 *ib*	ROR *r/m8*, imm8*	MI	Valid	N.E.	Rotate 8 bits *r/m16* right *imm8* times.
D1 /1	ROR *r/m16*, 1	M1	Valid	Valid	Rotate 16 bits *r/m16* right once.
D3 /1	ROR *r/m16*, CL	MC	Valid	Valid	Rotate 16 bits *r/m16* right CL times.
C1 /1 *ib*	ROR *r/m16, imm8*	MI	Valid	Valid	Rotate 16 bits *r/m16* right *imm8* times.
D1 /1	ROR *r/m32*, 1	M1	Valid	Valid	Rotate 32 bits *r/m32* right once.
REX.W + D1 /1	ROR *r/m64*, 1	M1	Valid	N.E.	Rotate 64 bits *r/m64* right once. Uses a 6 bit count.
D3 /1	ROR *r/m32*, CL	MC	Valid	Valid	Rotate 32 bits *r/m32* right CL times.
REX.W + D3 /1	ROR *r/m64*, CL	MC	Valid	N.E.	Rotate 64 bits *r/m64* right CL times. Uses a 6 bit count.
C1 /1 *ib*	ROR *r/m32, imm8*	MI	Valid	Valid	Rotate 32 bits *r/m32* right *imm8* times.
REX.W + C1 /1 *ib*	ROR *r/m64, imm8*	MI	Valid	N.E.	Rotate 64 bits *r/m64* right *imm8* times. Uses a 6 bit count.

NOTES:

* In 64-bit mode, r/m8 can not be encoded to access the following byte registers if a REX prefix is used: AH, BH, CH, DH.

** See IA-32 Architecture Compatibility section below.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
M1	ModRM:r/m (w)	1	NA	NA
MC	ModRM:r/m (w)	CL	NA	NA
MI	ModRM:r/m (w)	imm8	NA	NA

Description

Shifts (rotates) the bits of the first operand (destination operand) the number of bit positions specified in the second operand (count operand) and stores the result in the destination operand. The destination operand can be a register or a memory location; the count operand is an unsigned integer that can be an immediate or a value in the CL register. The count is masked to 5 bits (or 6 bits if in 64-bit mode and REX.W = 1).

The rotate left (ROL) and rotate through carry left (RCL) instructions shift all the bits toward more-significant bit positions, except for the most-significant bit, which is rotated to the least-significant bit location. The rotate right (ROR) and rotate through carry right (RCR) instructions shift all the bits toward less significant bit positions, except for the least-significant bit, which is rotated to the most-significant bit location.

The RCL and RCR instructions include the CF flag in the rotation. The RCL instruction shifts the CF flag into the least-significant bit and shifts the most-significant bit into the CF flag. The RCR instruction shifts the CF flag into the most-significant bit and shifts the least-significant bit into the CF flag. For the ROL and ROR instructions, the original value of the CF flag is not a part of the result, but the CF flag receives a copy of the bit that was shifted from one end to the other.

The OF flag is defined only for the 1-bit rotates; it is undefined in all other cases (except RCL and RCR instructions only: a zero-bit rotate does nothing, that is affects no flags). For left rotates, the OF flag is set to the exclusive OR of the CF bit (after the rotate) and the most-significant bit of the result. For right rotates, the OF flag is set to the exclusive OR of the two most-significant bits of the result.

In 64-bit mode, using a REX prefix in the form of REX.R permits access to additional registers (R8-R15). Use of REX.W promotes the first operand to 64 bits and causes the count operand to become a 6-bit counter.

IA-32 Architecture Compatibility

The 8086 does not mask the rotation count. However, all other IA-32 processors (starting with the Intel 286 processor) do mask the rotation count to 5 bits, resulting in a maximum count of 31. This masking is done in all operating modes (including the virtual-8086 mode) to reduce the maximum execution time of the instructions.

Operation

```
(* RCL and RCR instructions *)
SIZE ← OperandSize;
CASE (determine count) OF
    SIZE ← 8:     tempCOUNT ← (COUNT AND 1FH) MOD 9;
    SIZE ← 16:    tempCOUNT ← (COUNT AND 1FH) MOD 17;
    SIZE ← 32:    tempCOUNT ← COUNT AND 1FH;
    SIZE ← 64:    tempCOUNT ← COUNT AND 3FH;
ESAC;

(* RCL instruction operation *)
WHILE (tempCOUNT ≠ 0)
    DO
        tempCF ← MSB(DEST);
        DEST ← (DEST * 2) + CF;
        CF ← tempCF;
        tempCOUNT ← tempCOUNT – 1;
    OD;
ELIHW;
IF (COUNT & COUNTMASK) = 1
    THEN OF ← MSB(DEST) XOR CF;
    ELSE OF is undefined;
FI;
```

```
(* RCR instruction operation *)
IF (COUNT & COUNTMASK) = 1
    THEN OF ← MSB(DEST) XOR CF;
    ELSE OF is undefined;
FI;
WHILE (tempCOUNT ≠ 0)
    DO
        tempCF ← LSB(SRC);
        DEST ← (DEST / 2) + (CF * 2^SIZE);
        CF ← tempCF;
        tempCOUNT ← tempCOUNT – 1;
    OD;

(* ROL and ROR instructions *)
IF OperandSize = 64
    THEN COUNTMASK = 3FH;
    ELSE COUNTMASK = 1FH;
FI;

(* ROL instruction operation *)
tempCOUNT ← (COUNT & COUNTMASK) MOD SIZE

WHILE (tempCOUNT ≠ 0)
    DO
        tempCF ← MSB(DEST);
        DEST ← (DEST * 2) + tempCF;
        tempCOUNT ← tempCOUNT – 1;
    OD;
ELIHW;
IF (COUNT & COUNTMASK) ≠ 0
    THEN CF ← LSB(DEST);
FI;
IF (COUNT & COUNTMASK) = 1
    THEN OF ← MSB(DEST) XOR CF;
    ELSE OF is undefined;
FI;

(* ROR instruction operation *)
tempCOUNT ← (COUNT & COUNTMASK) MOD SIZE
WHILE (tempCOUNT ≠ 0)
    DO
        tempCF ← LSB(SRC);
        DEST ← (DEST / 2) + (tempCF * 2^SIZE);
        tempCOUNT ← tempCOUNT – 1;
    OD;
ELIHW;
IF (COUNT & COUNTMASK) ≠ 0
    THEN CF ← MSB(DEST);
FI;
IF (COUNT & COUNTMASK) = 1
    THEN OF ← MSB(DEST) XOR MSB – 1(DEST);
    ELSE OF is undefined;
FI;
```

Flags Affected

If the masked count is 0, the flags are not affected. If the masked count is 1, then the OF flag is affected, otherwise (masked count is greater than 1) the OF flag is undefined. The CF flag is affected when the masked count is non-zero. The SF, ZF, AF, and PF flags are always unaffected.

Protected Mode Exceptions

#GP(0)	If the source operand is located in a non-writable segment.
	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
	If the DS, ES, FS, or GS register contains a NULL segment selector.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If the LOCK prefix is used.

Real-Address Mode Exceptions

#GP	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS	If a memory operand effective address is outside the SS segment limit.
#UD	If the LOCK prefix is used.

Virtual-8086 Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.
#UD	If the LOCK prefix is used.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the source operand is located in a nonwritable segment.
	If the memory address is in a non-canonical form.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If the LOCK prefix is used.

RCPPS—Compute Reciprocals of Packed Single-Precision Floating-Point Values

Opcode*/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F 53 /r RCPPS *xmm1, xmm2/m128*	RM	V/V	SSE	Computes the approximate reciprocals of the packed single-precision floating-point values in *xmm2/m128* and stores the results in *xmm1*.
VEX.128.0F.WIG 53 /r VRCPPS *xmm1, xmm2/m128*	RM	V/V	AVX	Computes the approximate reciprocals of packed single-precision values in *xmm2/mem* and stores the results in *xmm1*.
VEX.256.0F.WIG 53 /r VRCPPS *ymm1, ymm2/m256*	RM	V/V	AVX	Computes the approximate reciprocals of packed single-precision values in *ymm2/mem* and stores the results in *ymm1*.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

Description

Performs a SIMD computation of the approximate reciprocals of the four packed single-precision floating-point values in the source operand (second operand) stores the packed single-precision floating-point results in the destination operand. The source operand can be an XMM register or a 128-bit memory location. The destination operand is an XMM register. See Figure 10-5 in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*, for an illustration of a SIMD single-precision floating-point operation.

The relative error for this approximation is:

$$|\text{Relative Error}| \leq 1.5 * 2^{-12}$$

The RCPPS instruction is not affected by the rounding control bits in the MXCSR register. When a source value is a 0.0, an ∞ of the sign of the source value is returned. A denormal source value is treated as a 0.0 (of the same sign). Tiny results (see Section 4.9.1.5, "Numeric Underflow Exception (#U)" in *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*) are always flushed to 0.0, with the sign of the operand. (Input values greater than or equal to $|1.11111111110100000000000B*2^{125}|$ are guaranteed to not produce tiny results; input values less than or equal to $|1.00000000000110000000001B*2^{126}|$ are guaranteed to produce tiny results, which are in turn flushed to 0.0; and input values in between this range may or may not produce tiny results, depending on the implementation.) When a source value is an SNaN or QNaN, the SNaN is converted to a QNaN or the source QNaN is returned.

In 64-bit mode, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

128-bit Legacy SSE version: The second source can be an XMM register or an 128-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (MAXVL-1:128) of the corresponding YMM register destination are unmodified.

VEX.128 encoded version: the first source operand is an XMM register or 128-bit memory location. The destination operand is an XMM register. The upper bits (MAXVL-1:128) of the corresponding YMM register destination are zeroed.

VEX.256 encoded version: The first source operand is a YMM register. The second source operand can be a YMM register or a 256-bit memory location. The destination operand is a YMM register.

Note: In VEX-encoded versions, VEX.vvvv is reserved and must be 1111b, otherwise instructions will #UD.

Operation

RCPPS (128-bit Legacy SSE version)

```
DEST[31:0] ← APPROXIMATE(1/SRC[31:0])
DEST[63:32] ← APPROXIMATE(1/SRC[63:32])
DEST[95:64] ← APPROXIMATE(1/SRC[95:64])
DEST[127:96] ← APPROXIMATE(1/SRC[127:96])
DEST[MAXVL-1:128] (Unmodified)
```

VRCPPS (VEX.128 encoded version)

```
DEST[31:0] ← APPROXIMATE(1/SRC[31:0])
DEST[63:32] ← APPROXIMATE(1/SRC[63:32])
DEST[95:64] ← APPROXIMATE(1/SRC[95:64])
DEST[127:96] ← APPROXIMATE(1/SRC[127:96])
DEST[MAXVL-1:128] ← 0
```

VRCPPS (VEX.256 encoded version)

```
DEST[31:0] ← APPROXIMATE(1/SRC[31:0])
DEST[63:32] ← APPROXIMATE(1/SRC[63:32])
DEST[95:64] ← APPROXIMATE(1/SRC[95:64])
DEST[127:96] ← APPROXIMATE(1/SRC[127:96])
DEST[159:128] ← APPROXIMATE(1/SRC[159:128])
DEST[191:160] ← APPROXIMATE(1/SRC[191:160])
DEST[223:192] ← APPROXIMATE(1/SRC[223:192])
DEST[255:224] ← APPROXIMATE(1/SRC[255:224])
```

Intel C/C++ Compiler Intrinsic Equivalent

RCCPS: __m128 _mm_rcp_ps(__m128 a)

RCPPS: __m256 _mm256_rcp_ps (__m256 a);

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Exceptions Type 4; additionally

#UD	If VEX.vvvv ≠ 1111B.

RCPSS—Compute Reciprocal of Scalar Single-Precision Floating-Point Values

Opcode*/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
F3 0F 53 /r RCPSS *xmm1*, xmm2/m32	RM	V/V	SSE	Computes the approximate reciprocal of the scalar single-precision floating-point value in *xmm2/m32* and stores the result in *xmm1*.
VEX.NDS.LIG.F3.0F.WIG 53 /r VRCPSS *xmm1, xmm2, xmm3/m32*	RVM	V/V	AVX	Computes the approximate reciprocal of the scalar single-precision floating-point value in *xmm3/m32* and stores the result in *xmm1*. Also, upper single precision floating-point values (bits[127:32]) from *xmm2* are copied to *xmm1*[127:32].

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (w)	ModRM:r/m (r)	NA	NA
RVM	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA

Description

Computes of an approximate reciprocal of the low single-precision floating-point value in the source operand (second operand) and stores the single-precision floating-point result in the destination operand. The source operand can be an XMM register or a 32-bit memory location. The destination operand is an XMM register. The three high-order doublewords of the destination operand remain unchanged. See Figure 10-6 in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*, for an illustration of a scalar single-precision floating-point operation.

The relative error for this approximation is:

$$|\text{Relative Error}| \leq 1.5 * 2^{-12}$$

The RCPSS instruction is not affected by the rounding control bits in the MXCSR register. When a source value is a 0.0, an ∞ of the sign of the source value is returned. A denormal source value is treated as a 0.0 (of the same sign). Tiny results (see Section 4.9.1.5, "Numeric Underflow Exception (#U)" in *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*) are always flushed to 0.0, with the sign of the operand. (Input values greater than or equal to $|1.11111111110100000000000B*2^{125}|$ are guaranteed to not produce tiny results; input values less than or equal to $|1.00000000000110000000001B*2^{126}|$ are guaranteed to produce tiny results, which are in turn flushed to 0.0; and input values in between this range may or may not produce tiny results, depending on the implementation.) When a source value is an SNaN or QNaN, the SNaN is converted to a QNaN or the source QNaN is returned.

In 64-bit mode, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

128-bit Legacy SSE version: The first source operand and the destination operand are the same. Bits (MAXVL-1:32) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: Bits (MAXVL-1:128) of the destination YMM register are zeroed.

Operation

RCPSS (128-bit Legacy SSE version)

```
DEST[31:0] ← APPROXIMATE(1/SRC[31:0])
DEST[MAXVL-1:32] (Unmodified)
```

VRCPSS (VEX.128 encoded version)
DEST[31:0] ← APPROXIMATE(1/SRC2[31:0])
DEST[127:32] ← SRC1[127:32]
DEST[MAXVL-1:128] ← 0

Intel C/C++ Compiler Intrinsic Equivalent

RCPSS: __m128 _mm_rcp_ss(__m128 a)

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Exceptions Type 5.

RDFSBASE/RDGSBASE—Read FS/GS Segment Base

Opcode/ Instruction	Op/ En	64/32-bit Mode	CPUID Feature Flag	Description
F3 0F AE /0 RDFSBASE r32	M	V/I	FSGSBASE	Load the 32-bit destination register with the FS base address.
F3 REX.W 0F AE /0 RDFSBASE r64	M	V/I	FSGSBASE	Load the 64-bit destination register with the FS base address.
F3 0F AE /1 RDGSBASE *r32*	M	V/I	FSGSBASE	Load the 32-bit destination register with the GS base address.
F3 REX.W 0F AE /1 RDGSBASE *r64*	M	V/I	FSGSBASE	Load the 64-bit destination register with the GS base address.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
M	ModRM:r/m (w)	NA	NA	NA

Description

Loads the general-purpose register indicated by the modR/M:r/m field with the FS or GS segment base address.

The destination operand may be either a 32-bit or a 64-bit general-purpose register. The REX.W prefix indicates the operand size is 64 bits. If no REX.W prefix is used, the operand size is 32 bits; the upper 32 bits of the source base address (for FS or GS) are ignored and upper 32 bits of the destination register are cleared.

This instruction is supported only in 64-bit mode.

Operation

```
DEST ← FS/GS segment base address;
```

Flags Affected

None

C/C++ Compiler Intrinsic Equivalent

RDFSBASE: unsigned int _readfsbase_u32(void);

RDFSBASE: unsigned __int64 _readfsbase_u64(void);

RDGSBASE: unsigned int _readgsbase_u32(void);

RDGSBASE: unsigned __int64 _readgsbase_u64(void);

Protected Mode Exceptions

#UD The RDFSBASE and RDGSBASE instructions are not recognized in protected mode.

Real-Address Mode Exceptions

#UD The RDFSBASE and RDGSBASE instructions are not recognized in real-address mode.

Virtual-8086 Mode Exceptions

#UD The RDFSBASE and RDGSBASE instructions are not recognized in virtual-8086 mode.

Compatibility Mode Exceptions

#UD The RDFSBASE and RDGSBASE instructions are not recognized in compatibility mode.

64-Bit Mode Exceptions

#UD	If the LOCK prefix is used. If CR4.FSGSBASE[bit 16] = 0. If CPUID.07H.0H:EBX.FSGSBASE[bit 0] = 0.

RDMSR—Read from Model Specific Register

Opcode*	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
0F 32	RDMSR	ZO	Valid	Valid	Read MSR specified by ECX into EDX:EAX.

NOTES:

* See IA-32 Architecture Compatibility section below.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
ZO	NA	NA	NA	NA

Description

Reads the contents of a 64-bit model specific register (MSR) specified in the ECX register into registers EDX:EAX. (On processors that support the Intel 64 architecture, the high-order 32 bits of RCX are ignored.) The EDX register is loaded with the high-order 32 bits of the MSR and the EAX register is loaded with the low-order 32 bits. (On processors that support the Intel 64 architecture, the high-order 32 bits of each of RAX and RDX are cleared.) If fewer than 64 bits are implemented in the MSR being read, the values returned to EDX:EAX in unimplemented bit locations are undefined.

This instruction must be executed at privilege level 0 or in real-address mode; otherwise, a general protection exception #GP(0) will be generated. Specifying a reserved or unimplemented MSR address in ECX will also cause a general protection exception.

The MSRs control functions for testability, execution tracing, performance-monitoring, and machine check errors. Chapter 2, "Model-Specific Registers (MSRs)" of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 4*, lists all the MSRs that can be read with this instruction and their addresses. Note that each processor family has its own set of MSRs.

The CPUID instruction should be used to determine whether MSRs are supported (CPUID.01H:EDX[5] = 1) before using this instruction.

IA-32 Architecture Compatibility

The MSRs and the ability to read them with the RDMSR instruction were introduced into the IA-32 Architecture with the Pentium processor. Execution of this instruction by an IA-32 processor earlier than the Pentium processor results in an invalid opcode exception #UD.

See "Changes to Instruction Behavior in VMX Non-Root Operation" in Chapter 25 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3C*, for more information about the behavior of this instruction in VMX non-root operation.

Operation

```
EDX:EAX ← MSR[ECX];
```

Flags Affected

None.

Protected Mode Exceptions

#GP(0)	If the current privilege level is not 0.
	If the value in ECX specifies a reserved or unimplemented MSR address.
#UD	If the LOCK prefix is used.

Real-Address Mode Exceptions

#GP	If the value in ECX specifies a reserved or unimplemented MSR address.
#UD	If the LOCK prefix is used.

Virtual-8086 Mode Exceptions

#GP(0)	The RDMSR instruction is not recognized in virtual-8086 mode.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

Same exceptions as in protected mode.

RDPID—Read Processor ID

Opcode/ Instruction	Op/ En	64/32-bit Mode	CPUID Feature Flag	Description
F3 0F C7 /7 RDPID r32	R	N.E./V	RDPID	Read IA32_TSC_AUX into r32.
F3 0F C7 /7 RDPID r64	R	V/N.E.	RDPID	Read IA32_TSC_AUX into r64.

Instruction Operand Encoding[1]

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
R	ModRM:r/m (w)	NA	NA	NA

Description

Reads the value of the IA32_TSC_AUX MSR (address C0000103H) into the destination register. The value of CS.D and operand-size prefixes (66H and REX.W) do not affect the behavior of the RDPID instruction.

Operation

DEST ← IA32_TSC_AUX

Flags Affected

None.

Protected Mode Exceptions

#UD	If the LOCK prefix is used.
	If CPUID.7H.0:ECX.RDPID[bit 22] = 0.

Real-Address Mode Exceptions

Same exceptions as in protected mode.

Virtual-8086 Mode Exceptions

Same exceptions as in protected mode.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

Same exceptions as in protected mode.

1. ModRM.MOD = 011B required

RDPKRU—Read Protection Key Rights for User Pages

Opcode*	Instruction	Op/ En	64/32bit Mode Support	CPUID Feature Flag	Description
NP 0F 01 EE	RDPKRU	ZO	V/V	OSPKE	Reads PKRU into EAX.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
ZO	NA	NA	NA	NA

Description

Reads the value of PKRU into EAX and clears EDX. ECX must be 0 when RDPKRU is executed; otherwise, a general-protection exception (#GP) occurs.

RDPKRU can be executed only if CR4.PKE = 1; otherwise, an invalid-opcode exception (#UD) occurs. Software can discover the value of CR4.PKE by examining CPUID.(EAX=07H,ECX=0H):ECX.OSPKE [bit 4].

On processors that support the Intel 64 Architecture, the high-order 32-bits of RCX are ignored and the high-order 32-bits of RDX and RAX are cleared.

Operation

```
IF (ECX = 0)
    THEN
        EAX ← PKRU;
        EDX ← 0;
    ELSE #GP(0);
FI;
```

Flags Affected

None.

C/C++ Compiler Intrinsic Equivalent

RDPKRU: uint32_t _rdpkru_u32(void);

Protected Mode Exceptions

#GP(0)	If ECX ≠ 0
#UD	If the LOCK prefix is used.
	If CR4.PKE = 0.

Real-Address Mode Exceptions

Same exceptions as in protected mode.

Virtual-8086 Mode Exceptions

Same exceptions as in protected mode.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

Same exceptions as in protected mode.

RDPMC—Read Performance-Monitoring Counters

Opcode*	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
0F 33	RDPMC	ZO	Valid	Valid	Read performance-monitoring counter specified by ECX into EDX:EAX.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
ZO	NA	NA	NA	NA

Description

The EAX register is loaded with the low-order 32 bits. The EDX register is loaded with the supported high-order bits of the counter. The number of high-order bits loaded into EDX is implementation specific on processors that do no support architectural performance monitoring. The width of fixed-function and general-purpose performance counters on processors supporting architectural performance monitoring are reported by CPUID 0AH leaf. See below for the treatment of the EDX register for "fast" reads.

The ECX register specifies the counter type (if the processor supports architectural performance monitoring) and counter index. Counter type is specified in ECX[30] to select one of two type of performance counters. If the processor does not support architectural performance monitoring, ECX[30:0] specifies the counter index; otherwise ECX[29:0] specifies the index relative to the base of each counter type. ECX[31] selects "fast" read mode if supported. The two counter types are:

- General-purpose or special-purpose performance counters are specified with ECX[30] = 0: The number of general-purpose performance counters on processor supporting architectural performance monitoring are reported by CPUID 0AH leaf. The number of general-purpose counters is model specific if the processor does not support architectural performance monitoring, see Chapter 18, "Performance Monitoring" of *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3B*. Special-purpose counters are available only in selected processor members, see Table 4-16.
- Fixed-function performance counters are specified with ECX[30] = 1. The number fixed-function performance counters is enumerated by CPUID 0AH leaf. See Chapter 18, "Performance Monitoring" of *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3B*. This counter type is selected if ECX[30] is set.

The width of fixed-function performance counters and general-purpose performance counters on processor supporting architectural performance monitoring are reported by CPUID 0AH leaf. The width of general-purpose performance counters are 40-bits for processors that do not support architectural performance monitoring counters. The width of special-purpose performance counters are implementation specific.

Table 4-16 lists valid indices of the general-purpose and special-purpose performance counters according to the DisplayFamily_DisplayModel values of CPUID encoding for each processor family (see CPUID instruction in Chapter 3, "Instruction Set Reference, A-L" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A*).

Table 4-16. Valid General and Special Purpose Performance Counter Index Range for RDPMC

Processor Family	DisplayFamily_DisplayModel/ Other Signatures	Valid PMC Index Range	General-purpose Counters
P6	06H_01H, 06H_03H, 06H_05H, 06H_06H, 06H_07H, 06H_08H, 06H_0AH, 06H_0BH	0, 1	0, 1
Processors Based on Intel NetBurst microarchitecture (No L3)	0FH_00H, 0FH_01H, 0FH_02H, 0FH_03H, 0FH_04H, 0FH_06H	≥ 0 and ≤ 17	≥ 0 and ≤ 17
Pentium M processors	06H_09H, 06H_0DH	0, 1	0, 1
Processors Based on Intel NetBurst microarchitecture (No L3)	0FH_03H, 0FH_04H) and (L3 is present)	≥ 0 and ≤ 25	≥ 0 and ≤ 17

Table 4-16. Valid General and Special Purpose Performance Counter Index Range for RDPMC (Contd.)

Processor Family	DisplayFamily_DisplayModel/ Other Signatures	Valid PMC Index Range	General-purpose Counters
Intel® Core™ Solo and Intel® Core™ Duo processors, Dual-core Intel® Xeon® processor LV	06H_0EH	0, 1	0, 1
Intel® Core™2 Duo processor, Intel Xeon processor 3000, 5100, 5300, 7300 Series - general-purpose PMC	06H_0FH	0, 1	0, 1
Intel® Core™2 Duo processor family, Intel Xeon processor 3100, 3300, 5200, 5400 series - general-purpose PMC	06H_17H	0, 1	0, 1
Intel Xeon processors 7400 series	(06H_1DH)	≥ 0 and ≤ 9	0, 1
45 nm and 32 nm Intel® Atom™ processors	06H_1CH, 06_26H, 06_27H, 06_35H, 06_36H	0, 1	0, 1
Intel® Atom™ processors based on Silvermont or Airmont microarchitectures	06H_37H, 06_4AH, 06_4DH, 06_5AH, 06_5DH, 06_4CH	0, 1	0, 1
Next Generation Intel® Atom™ processors based on Goldmont microarchitecture	06H_5CH, 06_5FH	0-3	0-3
Intel® processors based on the Nehalem, Westmere microarchitectures	06H_1AH, 06H_1EH, 06H_1FH, 06_25H, 06_2CH, 06H_2EH, 06_2FH	0-3	0-3
Intel® processors based on the Sandy Bridge, Ivy Bridge microarchitecture	06H_2AH, 06H_2DH, 06H_3AH, 06H_3EH	0-3 (0-7 if HyperThreading is off)	0-3 (0-7 if HyperThreading is off)
Intel® processors based on the Haswell, Broadwell, SkyLake microarchitectures	06H_3CH, 06H_45H, 06H_46H, 06H_3FH, 06_3DH, 06_47H, 4FH, 06_56H, 06_4EH, 06_5EH	0-3 (0-7 if HyperThreading is off)	0-3 (0-7 if HyperThreading is off)

Processors based on Intel NetBurst microarchitecture support "fast" (32-bit) and "slow" (40-bit) reads on the first 18 performance counters. Selected this option using ECX[31]. If bit 31 is set, RDPMC reads only the low 32 bits of the selected performance counter. If bit 31 is clear, all 40 bits are read. A 32-bit result is returned in EAX and EDX is set to 0. A 32-bit read executes faster on these processors than a full 40-bit read.

On processors based on Intel NetBurst microarchitecture with L3, performance counters with indices 18-25 are 32-bit counters. EDX is cleared after executing RDPMC for these counters.

In Intel Core 2 processor family, Intel Xeon processor 3000, 5100, 5300 and 7400 series, the fixed-function performance counters are 40-bits wide; they can be accessed by RDMPC with ECX between from 4000_0000H and 4000_0002H.

On Intel Xeon processor 7400 series, there are eight 32-bit special-purpose counters addressable with indices 2-9, ECX[30]=0.

When in protected or virtual 8086 mode, the performance-monitoring counters enabled (PCE) flag in register CR4 restricts the use of the RDPMC instruction as follows. When the PCE flag is set, the RDPMC instruction can be executed at any privilege level; when the flag is clear, the instruction can only be executed at privilege level 0. (When in real-address mode, the RDPMC instruction is always enabled.)

The performance-monitoring counters can also be read with the RDMSR instruction, when executing at privilege level 0.

The performance-monitoring counters are event counters that can be programmed to count events such as the number of instructions decoded, number of interrupts received, or number of cache loads. Chapter 19, "Performance Monitoring Events," in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3B*, lists the events that can be counted for various processors in the Intel 64 and IA-32 architecture families.

The RDPMC instruction is not a serializing instruction; that is, it does not imply that all the events caused by the preceding instructions have been completed or that events caused by subsequent instructions have not begun. If

an exact event count is desired, software must insert a serializing instruction (such as the CPUID instruction) before and/or after the RDPMC instruction.

Performing back-to-back fast reads are not guaranteed to be monotonic. To guarantee monotonicity on back-to-back reads, a serializing instruction must be placed between the two RDPMC instructions.

The RDPMC instruction can execute in 16-bit addressing mode or virtual-8086 mode; however, the full contents of the ECX register are used to select the counter, and the event count is stored in the full EAX and EDX registers. The RDPMC instruction was introduced into the IA-32 Architecture in the Pentium Pro processor and the Pentium processor with MMX technology. The earlier Pentium processors have performance-monitoring counters, but they must be read with the RDMSR instruction.

Operation

```
(* Intel processors that support architectural performance monitoring *)

Most significant counter bit (MSCB) = 47

IF ((CR4.PCE = 1) or (CPL = 0) or (CR0.PE = 0))
    THEN IF (ECX[30] = 1 and ECX[29:0] in valid fixed-counter range)
        EAX ← IA32_FIXED_CTR(ECX)[30:0];
        EDX ← IA32_FIXED_CTR(ECX)[MSCB:32];
    ELSE IF (ECX[30] = 0 and ECX[29:0] in valid general-purpose counter range)
        EAX ← PMC(ECX[30:0])[31:0];
        EDX ← PMC(ECX[30:0])[MSCB:32];
    ELSE (* ECX is not valid or CR4.PCE is 0 and CPL is 1, 2, or 3 and CR0.PE is 1 *)
        #GP(0);
FI;

(* Intel Core 2 Duo processor family and Intel Xeon processor 3000, 5100, 5300, 7400 series*)

Most significant counter bit (MSCB) = 39

IF ((CR4.PCE = 1) or (CPL = 0) or (CR0.PE = 0))
    THEN IF (ECX[30] = 1 and ECX[29:0] in valid fixed-counter range)
        EAX ← IA32_FIXED_CTR(ECX)[30:0];
        EDX ← IA32_FIXED_CTR(ECX)[MSCB:32];
    ELSE IF (ECX[30] = 0 and ECX[29:0] in valid general-purpose counter range)
        EAX ← PMC(ECX[30:0])[31:0];
        EDX ← PMC(ECX[30:0])[MSCB:32];
    ELSE IF (ECX[30] = 0 and ECX[29:0] in valid special-purpose counter range)
        EAX ← PMC(ECX[30:0])[31:0]; (* 32-bit read *)
    ELSE (* ECX is not valid or CR4.PCE is 0 and CPL is 1, 2, or 3 and CR0.PE is 1 *)
        #GP(0);
FI;

(* P6 family processors and Pentium processor with MMX technology *)

IF (ECX = 0 or 1) and ((CR4.PCE = 1) or (CPL = 0) or (CR0.PE = 0))
    THEN
        EAX ← PMC(ECX)[31:0];
        EDX ← PMC(ECX)[39:32];
    ELSE (* ECX is not 0 or 1 or CR4.PCE is 0 and CPL is 1, 2, or 3 and CR0.PE is 1 *)
        #GP(0);
FI;
(* Processors based on Intel NetBurst microarchitecture *)
IF ((CR4.PCE = 1) or (CPL = 0) or (CR0.PE = 0))
    THEN IF (ECX[30:0] = 0:17)
        THEN IF ECX[31] = 0
```

```
            THEN
                EAX ← PMC(ECX[30:0])[31:0]; (* 40-bit read *)
                EDX ← PMC(ECX[30:0])[39:32];
        ELSE (* ECX[31] = 1*)
            THEN
                EAX ← PMC(ECX[30:0])[31:0]; (* 32-bit read *)
                EDX ← 0;
        FI;
    ELSE IF (*64-bit Intel processor based on Intel NetBurst microarchitecture with L3 *)
        THEN IF (ECX[30:0] = 18:25 )
            EAX ← PMC(ECX[30:0])[31:0]; (* 32-bit read *)
            EDX ← 0;
        FI;
    ELSE (* Invalid PMC index in ECX[30:0], see Table 4-19. *)
        GP(0);
    FI;
ELSE (* CR4.PCE = 0 and (CPL = 1, 2, or 3) and CR0.PE = 1 *)
    #GP(0);
FI;
```

Flags Affected

None.

Protected Mode Exceptions

#GP(0)	If the current privilege level is not 0 and the PCE flag in the CR4 register is clear.
	If an invalid performance counter index is specified (see Table 4-16).
#UD	If the LOCK prefix is used.

Real-Address Mode Exceptions

#GP	If an invalid performance counter index is specified (see Table 4-16).
#UD	If the LOCK prefix is used.

Virtual-8086 Mode Exceptions

#GP(0)	If the PCE flag in the CR4 register is clear.
	If an invalid performance counter index is specified (see Table 4-16).
#UD	If the LOCK prefix is used.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#GP(0)	If the current privilege level is not 0 and the PCE flag in the CR4 register is clear.
	If an invalid performance counter index is specified (see Table 4-16).
#UD	If the LOCK prefix is used.

RDRAND—Read Random Number

Opcode*/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
0F C7 /6 RDRAND r16	M	V/V	RDRAND	Read a 16-bit random number and store in the destination register.
0F C7 /6 RDRAND r32	M	V/V	RDRAND	Read a 32-bit random number and store in the destination register.
REX.W + 0F C7 /6 RDRAND r64	M	V/I	RDRAND	Read a 64-bit random number and store in the destination register.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
M	ModRM:r/m (w)	NA	NA	NA

Description

Loads a hardware generated random value and store it in the destination register. The size of the random value is determined by the destination register size and operating mode. The Carry Flag indicates whether a random value is available at the time the instruction is executed. CF=1 indicates that the data in the destination is valid. Otherwise CF=0 and the data in the destination operand will be returned as zeros for the specified width. All other flags are forced to 0 in either situation. Software must check the state of CF=1 for determining if a valid random value has been returned, otherwise it is expected to loop and retry execution of RDRAND (see *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*, Section 7.3.17, "Random Number Generator Instructions").

This instruction is available at all privilege levels.

In 64-bit mode, the instruction's default operation size is 32 bits. Using a REX prefix in the form of REX.B permits access to additional registers (R8-R15). Using a REX prefix in the form of REX.W promotes operation to 64 bit operands. See the summary chart at the beginning of this section for encoding data and limits.

Operation

```
IF HW_RND_GEN.ready = 1
    THEN
        CASE of
            osize is 64: DEST[63:0] ← HW_RND_GEN.data;
            osize is 32: DEST[31:0] ← HW_RND_GEN.data;
            osize is 16: DEST[15:0] ← HW_RND_GEN.data;
        ESAC
        CF ← 1;
    ELSE
        CASE of
            osize is 64: DEST[63:0] ← 0;
            osize is 32: DEST[31:0] ← 0;
            osize is 16: DEST[15:0] ← 0;
        ESAC
        CF ← 0;
FI
OF, SF, ZF, AF, PF ← 0;
```

Flags Affected

The CF flag is set according to the result (see the "Operation" section above). The OF, SF, ZF, AF, and PF flags are set to 0.

Intel C/C++ Compiler Intrinsic Equivalent

RDRAND: int _rdrand16_step(unsigned short *);

RDRAND: int _rdrand32_step(unsigned int *);

RDRAND: int _rdrand64_step(unsigned __int64 *);

Protected Mode Exceptions

#UD	If the LOCK prefix is used. If the F2H or F3H prefix is used. If CPUID.01H:ECX.RDRAND[bit 30] = 0.

Real-Address Mode Exceptions

Same exceptions as in protected mode.

Virtual-8086 Mode Exceptions

Same exceptions as in protected mode.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

Same exceptions as in protected mode.

RDSEED—Read Random SEED

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
0F C7 /7 RDSEED r16	M	V/V	RDSEED	Read a 16-bit NIST SP800-90B & C compliant random value and store in the destination register.
0F C7 /7 RDSEED r32	M	V/V	RDSEED	Read a 32-bit NIST SP800-90B & C compliant random value and store in the destination register.
REX.W + 0F C7 /7 RDSEED r64	M	V/I	RDSEED	Read a 64-bit NIST SP800-90B & C compliant random value and store in the destination register.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
M	ModRM:r/m (w)	NA	NA	NA

Description

Loads a hardware generated random value and store it in the destination register. The random value is generated from an Enhanced NRBG (Non Deterministic Random Bit Generator) that is compliant to NIST SP800-90B and NIST SP800-90C in the XOR construction mode. The size of the random value is determined by the destination register size and operating mode. The Carry Flag indicates whether a random value is available at the time the instruction is executed. CF=1 indicates that the data in the destination is valid. Otherwise CF=0 and the data in the destination operand will be returned as zeros for the specified width. All other flags are forced to 0 in either situation. Software must check the state of CF=1 for determining if a valid random seed value has been returned, otherwise it is expected to loop and retry execution of RDSEED (see Section 1.2).

The RDSEED instruction is available at all privilege levels. The RDSEED instruction executes normally either inside or outside a transaction region.

In 64-bit mode, the instruction's default operation size is 32 bits. Using a REX prefix in the form of REX.B permits access to additional registers (R8-R15). Using a REX prefix in the form of REX.W promotes operation to 64 bit operands. See the summary chart at the beginning of this section for encoding data and limits.

Operation

```
IF HW_NRND_GEN.ready = 1
    THEN
        CASE of
            osize is 64: DEST[63:0] ← HW_NRND_GEN.data;
            osize is 32: DEST[31:0] ← HW_NRND_GEN.data;
            osize is 16: DEST[15:0] ← HW_NRND_GEN.data;
        ESAC;
        CF ← 1;
    ELSE
        CASE of
            osize is 64: DEST[63:0] ← 0;
            osize is 32: DEST[31:0] ← 0;
            osize is 16: DEST[15:0] ← 0;
        ESAC;
        CF ← 0;
FI;

OF, SF, ZF, AF, PF ← 0;
```

Flags Affected

The CF flag is set according to the result (see the "Operation" section above). The OF, SF, ZF, AF, and PF flags are set to 0.

C/C++ Compiler Intrinsic Equivalent

RDSEED int _rdseed16_step(unsigned short *);

RDSEED int _rdseed32_step(unsigned int *);

RDSEED int _rdseed64_step(unsigned __int64 *);

Protected Mode Exceptions

#UD	If the LOCK prefix is used.
	If the F2H or F3H prefix is used.
	If CPUID.(EAX=07H, ECX=0H):EBX.RDSEED[bit 18] = 0.

Real-Address Mode Exceptions

#UD	If the LOCK prefix is used.
	If the F2H or F3H prefix is used.
	If CPUID.(EAX=07H, ECX=0H):EBX.RDSEED[bit 18] = 0.

Virtual-8086 Mode Exceptions

#UD	If the LOCK prefix is used.
	If the F2H or F3H prefix is used.
	If CPUID.(EAX=07H, ECX=0H):EBX.RDSEED[bit 18] = 0.

Compatibility Mode Exceptions

#UD	If the LOCK prefix is used.
	If the F2H or F3H prefix is used.
	If CPUID.(EAX=07H, ECX=0H):EBX.RDSEED[bit 18] = 0.

64-Bit Mode Exceptions

#UD	If the LOCK prefix is used.
	If the F2H or F3H prefix is used.
	If CPUID.(EAX=07H, ECX=0H):EBX.RDSEED[bit 18] = 0.

RDTSC—Read Time-Stamp Counter

Opcode*	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
0F 31	RDTSC	ZO	Valid	Valid	Read time-stamp counter into EDX:EAX.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
ZO	NA	NA	NA	NA

Description

Reads the current value of the processor's time-stamp counter (a 64-bit MSR) into the EDX:EAX registers. The EDX register is loaded with the high-order 32 bits of the MSR and the EAX register is loaded with the low-order 32 bits. (On processors that support the Intel 64 architecture, the high-order 32 bits of each of RAX and RDX are cleared.)

The processor monotonically increments the time-stamp counter MSR every clock cycle and resets it to 0 whenever the processor is reset. See "Time Stamp Counter" in Chapter 17 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3B*, for specific details of the time stamp counter behavior.

The time stamp disable (TSD) flag in register CR4 restricts the use of the RDTSC instruction as follows. When the flag is clear, the RDTSC instruction can be executed at any privilege level; when the flag is set, the instruction can only be executed at privilege level 0.

The time-stamp counter can also be read with the RDMSR instruction, when executing at privilege level 0.

The RDTSC instruction is not a serializing instruction. It does not necessarily wait until all previous instructions have been executed before reading the counter. Similarly, subsequent instructions may begin execution before the read operation is performed. If software requires RDTSC to be executed only after all previous instructions have completed locally, it can either use RDTSCP (if the processor supports that instruction) or execute the sequence LFENCE;RDTSC.

This instruction was introduced by the Pentium processor.

See "Changes to Instruction Behavior in VMX Non-Root Operation" in Chapter 25 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3C*, for more information about the behavior of this instruction in VMX non-root operation.

Operation

```
IF (CR4.TSD = 0) or (CPL = 0) or (CR0.PE = 0)
    THEN EDX:EAX ← TimeStampCounter;
    ELSE (* CR4.TSD = 1 and (CPL = 1, 2, or 3) and CR0.PE = 1 *)
        #GP(0);
FI;
```

Flags Affected

None.

Protected Mode Exceptions

#GP(0)	If the TSD flag in register CR4 is set and the CPL is greater than 0.
#UD	If the LOCK prefix is used.

Real-Address Mode Exceptions

#UD	If the LOCK prefix is used.

Virtual-8086 Mode Exceptions

#GP(0)	If the TSD flag in register CR4 is set.
#UD	If the LOCK prefix is used.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

Same exceptions as in protected mode.

RDTSCP—Read Time-Stamp Counter and Processor ID

Opcode*	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
0F 01 F9	RDTSCP	ZO	Valid	Valid	Read 64-bit time-stamp counter and IA32_TSC_AUX value into EDX:EAX and ECX.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
ZO	NA	NA	NA	NA

Description

Reads the current value of the processor's time-stamp counter (a 64-bit MSR) into the EDX:EAX registers and also reads the value of the IA32_TSC_AUX MSR (address C0000103H) into the ECX register. The EDX register is loaded with the high-order 32 bits of the IA32_TSC MSR; the EAX register is loaded with the low-order 32 bits of the IA32_TSC MSR; and the ECX register is loaded with the low-order 32-bits of IA32_TSC_AUX MSR. On processors that support the Intel 64 architecture, the high-order 32 bits of each of RAX, RDX, and RCX are cleared.

The processor monotonically increments the time-stamp counter MSR every clock cycle and resets it to 0 whenever the processor is reset. See "Time Stamp Counter" in Chapter 17 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3B*, for specific details of the time stamp counter behavior.

The time stamp disable (TSD) flag in register CR4 restricts the use of the RDTSCP instruction as follows. When the flag is clear, the RDTSCP instruction can be executed at any privilege level; when the flag is set, the instruction can only be executed at privilege level 0.

The RDTSCP instruction waits until all previous instructions have been executed before reading the counter. However, subsequent instructions may begin execution before the read operation is performed.

See "Changes to Instruction Behavior in VMX Non-Root Operation" in Chapter 25 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3C*, for more information about the behavior of this instruction in VMX non-root operation.

Operation

```
IF (CR4.TSD = 0) or (CPL = 0) or (CR0.PE = 0)
    THEN
        EDX:EAX ← TimeStampCounter;
        ECX ← IA32_TSC_AUX[31:0];
    ELSE (* CR4.TSD = 1 and (CPL = 1, 2, or 3) and CR0.PE = 1 *)
        #GP(0);
FI;
```

Flags Affected

None.

Protected Mode Exceptions

#GP(0)	If the TSD flag in register CR4 is set and the CPL is greater than 0.
#UD	If the LOCK prefix is used.
	If CPUID.80000001H:EDX.RDTSCP[bit 27] = 0.

Real-Address Mode Exceptions

#UD	If the LOCK prefix is used.
	If CPUID.80000001H:EDX.RDTSCP[bit 27] = 0.

Virtual-8086 Mode Exceptions

#GP(0)	If the TSD flag in register CR4 is set.
#UD	If the LOCK prefix is used.
	If CPUID.80000001H:EDX.RDTSCP[bit 27] = 0.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

Same exceptions as in protected mode.

REP/REPE/REPZ/REPNE/REPNZ—Repeat String Operation Prefix

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
F3 6C	REP INS *m8*, DX	ZO	Valid	Valid	Input (E)CX bytes from port DX into ES:[(E)DI].
F3 6C	REP INS *m8*, DX	ZO	Valid	N.E.	Input RCX bytes from port DX into [RDI].
F3 6D	REP INS *m16*, DX	ZO	Valid	Valid	Input (E)CX words from port DX into ES:[(E)DI.]
F3 6D	REP INS *m32*, DX	ZO	Valid	Valid	Input (E)CX doublewords from port DX into ES:[(E)DI].
F3 6D	REP INS *r/m32*, DX	ZO	Valid	N.E.	Input RCX default size from port DX into [RDI].
F3 A4	REP MOVS *m8, m8*	ZO	Valid	Valid	Move (E)CX bytes from DS:[(E)SI] to ES:[(E)DI].
F3 REX.W A4	REP MOVS *m8, m8*	ZO	Valid	N.E.	Move RCX bytes from [RSI] to [RDI].
F3 A5	REP MOVS *m16, m16*	ZO	Valid	Valid	Move (E)CX words from DS:[(E)SI] to ES:[(E)DI].
F3 A5	REP MOVS *m32, m32*	ZO	Valid	Valid	Move (E)CX doublewords from DS:[(E)SI] to ES:[(E)DI].
F3 REX.W A5	REP MOVS *m64, m64*	ZO	Valid	N.E.	Move RCX quadwords from [RSI] to [RDI].
F3 6E	REP OUTS DX, *r/m8*	ZO	Valid	Valid	Output (E)CX bytes from DS:[(E)SI] to port DX.
F3 REX.W 6E	REP OUTS DX, *r/m8**	ZO	Valid	N.E.	Output RCX bytes from [RSI] to port DX.
F3 6F	REP OUTS DX, *r/m16*	ZO	Valid	Valid	Output (E)CX words from DS:[(E)SI] to port DX.
F3 6F	REP OUTS DX, *r/m32*	ZO	Valid	Valid	Output (E)CX doublewords from DS:[(E)SI] to port DX.
F3 REX.W 6F	REP OUTS DX, *r/m32*	ZO	Valid	N.E.	Output RCX default size from [RSI] to port DX.
F3 AC	REP LODS AL	ZO	Valid	Valid	Load (E)CX bytes from DS:[(E)SI] to AL.
F3 REX.W AC	REP LODS AL	ZO	Valid	N.E.	Load RCX bytes from [RSI] to AL.
F3 AD	REP LODS AX	ZO	Valid	Valid	Load (E)CX words from DS:[(E)SI] to AX.
F3 AD	REP LODS EAX	ZO	Valid	Valid	Load (E)CX doublewords from DS:[(E)SI] to EAX.
F3 REX.W AD	REP LODS RAX	ZO	Valid	N.E.	Load RCX quadwords from [RSI] to RAX.
F3 AA	REP STOS *m8*	ZO	Valid	Valid	Fill (E)CX bytes at ES:[(E)DI] with AL.
F3 REX.W AA	REP STOS *m8*	ZO	Valid	N.E.	Fill RCX bytes at [RDI] with AL.
F3 AB	REP STOS *m16*	ZO	Valid	Valid	Fill (E)CX words at ES:[(E)DI] with AX.
F3 AB	REP STOS *m32*	ZO	Valid	Valid	Fill (E)CX doublewords at ES:[(E)DI] with EAX.
F3 REX.W AB	REP STOS *m64*	ZO	Valid	N.E.	Fill RCX quadwords at [RDI] with RAX.
F3 A6	REPE CMPS *m8, m8*	ZO	Valid	Valid	Find nonmatching bytes in ES:[(E)DI] and DS:[(E)SI].
F3 REX.W A6	REPE CMPS *m8, m8*	ZO	Valid	N.E.	Find non-matching bytes in [RDI] and [RSI].
F3 A7	REPE CMPS *m16, m16*	ZO	Valid	Valid	Find nonmatching words in ES:[(E)DI] and DS:[(E)SI].
F3 A7	REPE CMPS *m32, m32*	ZO	Valid	Valid	Find nonmatching doublewords in ES:[(E)DI] and DS:[(E)SI].
F3 REX.W A7	REPE CMPS *m64, m64*	ZO	Valid	N.E.	Find non-matching quadwords in [RDI] and [RSI].
F3 AE	REPE SCAS *m8*	ZO	Valid	Valid	Find non-AL byte starting at ES:[(E)DI].
F3 REX.W AE	REPE SCAS *m8*	ZO	Valid	N.E.	Find non-AL byte starting at [RDI].
F3 AF	REPE SCAS *m16*	ZO	Valid	Valid	Find non-AX word starting at ES:[(E)DI].
F3 AF	REPE SCAS *m32*	ZO	Valid	Valid	Find non-EAX doubleword starting at ES:[(E)DI].

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
F3 REX.W AF	REPE SCAS *m64*	ZO	Valid	N.E.	Find non-RAX quadword starting at [RDI].
F2 A6	REPNE CMPS *m8, m8*	ZO	Valid	Valid	Find matching bytes in ES:[(E)DI] and DS:[(E)SI].
F2 REX.W A6	REPNE CMPS *m8, m8*	ZO	Valid	N.E.	Find matching bytes in [RDI] and [RSI].
F2 A7	REPNE CMPS *m16, m16*	ZO	Valid	Valid	Find matching words in ES:[(E)DI] and DS:[(E)SI].
F2 A7	REPNE CMPS *m32, m32*	ZO	Valid	Valid	Find matching doublewords in ES:[(E)DI] and DS:[(E)SI].
F2 REX.W A7	REPNE CMPS *m64, m64*	ZO	Valid	N.E.	Find matching doublewords in [RDI] and [RSI].
F2 AE	REPNE SCAS *m8*	ZO	Valid	Valid	Find AL, starting at ES:[(E)DI].
F2 REX.W AE	REPNE SCAS *m8*	ZO	Valid	N.E.	Find AL, starting at [RDI].
F2 AF	REPNE SCAS *m16*	ZO	Valid	Valid	Find AX, starting at ES:[(E)DI].
F2 AF	REPNE SCAS *m32*	ZO	Valid	Valid	Find EAX, starting at ES:[(E)DI].
F2 REX.W AF	REPNE SCAS *m64*	ZO	Valid	N.E.	Find RAX, starting at [RDI].

NOTES:

* In 64-bit mode, r/m8 can not be encoded to access the following byte registers if a REX prefix is used: AH, BH, CH, DH.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
ZO	NA	NA	NA	NA

Description

Repeats a string instruction the number of times specified in the count register or until the indicated condition of the ZF flag is no longer met. The REP (repeat), REPE (repeat while equal), REPNE (repeat while not equal), REPZ (repeat while zero), and REPNZ (repeat while not zero) mnemonics are prefixes that can be added to one of the string instructions. The REP prefix can be added to the INS, OUTS, MOVS, LODS, and STOS instructions, and the REPE, REPNE, REPZ, and REPNZ prefixes can be added to the CMPS and SCAS instructions. (The REPZ and REPNZ prefixes are synonymous forms of the REPE and REPNE prefixes, respectively.) The F3H prefix is defined for the following instructions and undefined for the rest:

- F3H as REP/REPE/REPZ for string and input/output instruction.
- F3H is a mandatory prefix for POPCNT, LZCNT, and ADOX.

The REP prefixes apply only to one string instruction at a time. To repeat a block of instructions, use the LOOP instruction or another looping construct. All of these repeat prefixes cause the associated instruction to be repeated until the count in register is decremented to 0. See Table 4-17.

Table 4-17. Repeat Prefixes

Repeat Prefix	Termination Condition 1*	Termination Condition 2
REP	RCX or (E)CX = 0	None
REPE/REPZ	RCX or (E)CX = 0	ZF = 0
REPNE/REPNZ	RCX or (E)CX = 0	ZF = 1

NOTES:

* Count register is CX, ECX or RCX by default, depending on attributes of the operating modes.

The REPE, REPNE, REPZ, and REPNZ prefixes also check the state of the ZF flag after each iteration and terminate the repeat loop if the ZF flag is not in the specified state. When both termination conditions are tested, the cause of a repeat termination can be determined either by testing the count register with a JECXZ instruction or by testing the ZF flag (with a JZ, JNZ, or JNE instruction).

When the REPE/REPZ and REPNE/REPNZ prefixes are used, the ZF flag does not require initialization because both the CMPS and SCAS instructions affect the ZF flag according to the results of the comparisons they make.

A repeating string operation can be suspended by an exception or interrupt. When this happens, the state of the registers is preserved to allow the string operation to be resumed upon a return from the exception or interrupt handler. The source and destination registers point to the next string elements to be operated on, the EIP register points to the string instruction, and the ECX register has the value it held following the last successful iteration of the instruction. This mechanism allows long string operations to proceed without affecting the interrupt response time of the system.

When a fault occurs during the execution of a CMPS or SCAS instruction that is prefixed with REPE or REPNE, the EFLAGS value is restored to the state prior to the execution of the instruction. Since the SCAS and CMPS instructions do not use EFLAGS as an input, the processor can resume the instruction after the page fault handler.

Use the REP INS and REP OUTS instructions with caution. Not all I/O ports can handle the rate at which these instructions execute. Note that a REP STOS instruction is the fastest way to initialize a large block of memory.

In 64-bit mode, the operand size of the count register is associated with the address size attribute. Thus the default count register is RCX; REX.W has no effect on the address size and the count register. In 64-bit mode, if 67H is used to override address size attribute, the count register is ECX and any implicit source/destination operand will use the corresponding 32-bit index register. See the summary chart at the beginning of this section for encoding data and limits.

REP INS may read from the I/O port without writing to the memory location if an exception or VM exit occurs due to the write (e.g. #PF). If this would be problematic, for example because the I/O port read has side-effects, software should ensure the write to the memory location does not cause an exception or VM exit.

Operation

```
IF AddressSize = 16
   THEN
      Use CX for CountReg;
      Implicit Source/Dest operand for memory use of SI/DI;
   ELSE IF AddressSize = 64
      THEN Use RCX for CountReg;
      Implicit Source/Dest operand for memory use of RSI/RDI;
   ELSE
      Use ECX for CountReg;
      Implicit Source/Dest operand for memory use of ESI/EDI;
FI;
WHILE CountReg ≠ 0
    DO
         Service pending interrupts (if any);
         Execute associated string instruction;
         CountReg ← (CountReg – 1);
         IF CountReg = 0
               THEN exit WHILE loop; FI;
         IF (Repeat prefix is REPZ or REPE) and (ZF = 0)
         or (Repeat prefix is REPNZ or REPNE) and (ZF = 1)
               THEN exit WHILE loop; FI;
    OD;
```

Flags Affected

None; however, the CMPS and SCAS instructions do set the status flags in the EFLAGS register.

Exceptions (All Operating Modes)

Exceptions may be generated by an instruction associated with the prefix.

64-Bit Mode Exceptions

#GP(0)	If the memory address is in a non-canonical form.

RET—Return from Procedure

Opcode*	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
C3	RET	ZO	Valid	Valid	Near return to calling procedure.
CB	RET	ZO	Valid	Valid	Far return to calling procedure.
C2 *iw*	RET *imm16*	I	Valid	Valid	Near return to calling procedure and pop *imm16* bytes from stack.
CA *iw*	RET *imm16*	I	Valid	Valid	Far return to calling procedure and pop *imm16* bytes from stack.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
ZO	NA	NA	NA	NA
I	imm16	NA	NA	NA

Description

Transfers program control to a return address located on the top of the stack. The address is usually placed on the stack by a CALL instruction, and the return is made to the instruction that follows the CALL instruction.

The optional source operand specifies the number of stack bytes to be released after the return address is popped; the default is none. This operand can be used to release parameters from the stack that were passed to the called procedure and are no longer needed. It must be used when the CALL instruction used to switch to a new procedure uses a call gate with a non-zero word count to access the new procedure. Here, the source operand for the RET instruction must specify the same number of bytes as is specified in the word count field of the call gate.

The RET instruction can be used to execute three different types of returns:

- **Near return** — A return to a calling procedure within the current code segment (the segment currently pointed to by the CS register), sometimes referred to as an intrasegment return.
- **Far return** — A return to a calling procedure located in a different segment than the current code segment, sometimes referred to as an intersegment return.
- **Inter-privilege-level far return** — A far return to a different privilege level than that of the currently executing program or procedure.

The inter-privilege-level return type can only be executed in protected mode. See the section titled "Calling Procedures Using Call and RET" in Chapter 6 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*, for detailed information on near, far, and inter-privilege-level returns.

When executing a near return, the processor pops the return instruction pointer (offset) from the top of the stack into the EIP register and begins program execution at the new instruction pointer. The CS register is unchanged.

When executing a far return, the processor pops the return instruction pointer from the top of the stack into the EIP register, then pops the segment selector from the top of the stack into the CS register. The processor then begins program execution in the new code segment at the new instruction pointer.

The mechanics of an inter-privilege-level far return are similar to an intersegment return, except that the processor examines the privilege levels and access rights of the code and stack segments being returned to determine if the control transfer is allowed to be made. The DS, ES, FS, and GS segment registers are cleared by the RET instruction during an inter-privilege-level return if they refer to segments that are not allowed to be accessed at the new privilege level. Since a stack switch also occurs on an inter-privilege level return, the ESP and SS registers are loaded from the stack.

If parameters are passed to the called procedure during an inter-privilege level call, the optional source operand must be used with the RET instruction to release the parameters on the return. Here, the parameters are released both from the called procedure's stack and the calling procedure's stack (that is, the stack being returned to).

In 64-bit mode, the default operation size of this instruction is the stack-address size, i.e. 64 bits. This applies to near returns, not far returns; the default operation size of far returns is 32 bits.

Operation

```
(* Near return *)
IF instruction = near return
    THEN;
        IF OperandSize = 32
            THEN
                IF top 4 bytes of stack not within stack limits
                    THEN #SS(0); FI;
                EIP ← Pop();
            ELSE
                IF OperandSize = 64
                    THEN
                        IF top 8 bytes of stack not within stack limits
                            THEN #SS(0); FI;
                        RIP ← Pop();
                    ELSE (* OperandSize = 16 *)
                        IF top 2 bytes of stack not within stack limits
                            THEN #SS(0); FI;
                        tempEIP ← Pop();
                        tempEIP ← tempEIP AND 0000FFFFH;
                        IF tempEIP not within code segment limits
                            THEN #GP(0); FI;
                        EIP ← tempEIP;
                FI;
        FI;

    IF instruction has immediate operand
        THEN (* Release parameters from stack *)
            IF StackAddressSize = 32
                THEN
                    ESP ← ESP + SRC;
                ELSE
                    IF StackAddressSize = 64
                        THEN
                            RSP ← RSP + SRC;
                        ELSE (* StackAddressSize = 16 *)
                            SP ← SP + SRC;
                    FI;
            FI;
    FI;
FI;

(* Real-address mode or virtual-8086 mode *)
IF ((PE = 0) or (PE = 1 AND VM = 1)) and instruction = far return
    THEN
        IF OperandSize = 32
            THEN
                IF top 8 bytes of stack not within stack limits
                    THEN #SS(0); FI;
                EIP ← Pop();
                CS ← Pop(); (* 32-bit pop, high-order 16 bits discarded *)
            ELSE (* OperandSize = 16 *)
                IF top 4 bytes of stack not within stack limits
                    THEN #SS(0); FI;
```

```
                    tempEIP ← Pop();
                    tempEIP ← tempEIP AND 0000FFFFH;
                    IF tempEIP not within code segment limits
                        THEN #GP(0); FI;
                    EIP ← tempEIP;
                    CS ← Pop(); (* 16-bit pop *)
            FI;
        IF instruction has immediate operand
            THEN (* Release parameters from stack *)
                SP ← SP + (SRC AND FFFFH);
        FI;
FI;

(* Protected mode, not virtual-8086 mode *)
IF (PE = 1 and VM = 0 and IA32_EFER.LMA = 0) and instruction = far return
    THEN
        IF OperandSize = 32
            THEN
                IF second doubleword on stack is not within stack limits
                    THEN #SS(0); FI;
            ELSE (* OperandSize = 16 *)
                IF second word on stack is not within stack limits
                    THEN #SS(0); FI;
        FI;
    IF return code segment selector is NULL
        THEN #GP(0); FI;
    IF return code segment selector addresses descriptor beyond descriptor table limit
        THEN #GP(selector); FI;
    Obtain descriptor to which return code segment selector points from descriptor table;
    IF return code segment descriptor is not a code segment
        THEN #GP(selector); FI;
    IF return code segment selector RPL < CPL
        THEN #GP(selector); FI;
    IF return code segment descriptor is conforming
    and return code segment DPL > return code segment selector RPL
        THEN #GP(selector); FI;
    IF return code segment descriptor is non-conforming and return code
    segment DPL ≠ return code segment selector RPL
        THEN #GP(selector); FI;
    IF return code segment descriptor is not present
        THEN #NP(selector); FI:
    IF return code segment selector RPL > CPL
        THEN GOTO RETURN-TO-OUTER-PRIVILEGE-LEVEL;
        ELSE GOTO RETURN-TO-SAME-PRIVILEGE-LEVEL;
    FI;
FI;

RETURN-TO-SAME-PRIVILEGE-LEVEL:
    IF the return instruction pointer is not within the return code segment limit
        THEN #GP(0); FI;
    IF OperandSize = 32
        THEN
            EIP ← Pop();
            CS ← Pop(); (* 32-bit pop, high-order 16 bits discarded *)
```

```
        ELSE (* OperandSize = 16 *)
            EIP ← Pop();
            EIP ← EIP AND 0000FFFFH;
            CS ← Pop(); (* 16-bit pop *)
    FI;
    IF instruction has immediate operand
        THEN (* Release parameters from stack *)
            IF StackAddressSize = 32
                THEN
                    ESP ← ESP + SRC;
                ELSE (* StackAddressSize = 16 *)
                    SP ← SP + SRC;
            FI;
    FI;

RETURN-TO-OUTER-PRIVILEGE-LEVEL:
    IF top (16 + SRC) bytes of stack are not within stack limits (OperandSize = 32)
    or top (8 + SRC) bytes of stack are not within stack limits (OperandSize = 16)
            THEN #SS(0); FI;
    Read return segment selector;
    IF stack segment selector is NULL
        THEN #GP(0); FI;
    IF return stack segment selector index is not within its descriptor table limits
        THEN #GP(selector); FI;
    Read segment descriptor pointed to by return segment selector;
    IF stack segment selector RPL ≠ RPL of the return code segment selector
    or stack segment is not a writable data segment
    or stack segment descriptor DPL ≠ RPL of the return code segment selector
            THEN #GP(selector); FI;
    IF stack segment not present
        THEN #SS(StackSegmentSelector); FI;
    IF the return instruction pointer is not within the return code segment limit
        THEN #GP(0); FI;
    CPL ← ReturnCodeSegmentSelector(RPL);
    IF OperandSize = 32
        THEN
            EIP ← Pop();
            CS ← Pop(); (* 32-bit pop, high-order 16 bits discarded; segment descriptor loaded *)
            CS(RPL) ← CPL;
            IF instruction has immediate operand
                THEN (* Release parameters from called procedure's stack *)
                    IF StackAddressSize = 32
                        THEN
                            ESP ← ESP + SRC;
                        ELSE (* StackAddressSize = 16 *)
                            SP ← SP + SRC;
                    FI;
            FI;
            tempESP ← Pop();
            tempSS ← Pop(); (* 32-bit pop, high-order 16 bits discarded; seg. descriptor loaded *)
            ESP ← tempESP;
            SS ← tempSS;
        ELSE (* OperandSize = 16 *)
            EIP ← Pop();
```

```
                EIP ← EIP AND 0000FFFFH;
                CS ← Pop(); (* 16-bit pop; segment descriptor loaded *)
                CS(RPL) ← CPL;
                IF instruction has immediate operand
                    THEN (* Release parameters from called procedure's stack *)
                        IF StackAddressSize = 32
                            THEN
                                ESP ← ESP + SRC;
                            ELSE (* StackAddressSize = 16 *)
                                SP ← SP + SRC;
                        FI;
                FI;
                tempESP ← Pop();
                tempSS ← Pop(); (* 16-bit pop; segment descriptor loaded *)
                ESP ← tempESP;
                SS ← tempSS;
        FI;

        FOR each SegReg in (ES, FS, GS, and DS)
            DO
                tempDesc ← descriptor cache for SegReg (* hidden part of segment register *)
                IF (SegmentSelector == NULL) OR (tempDesc(DPL) < CPL AND tempDesc(Type) is (data or non-conforming code)))
                    THEN (* Segment register invalid *)
                        SegmentSelector ← 0; (*Segment selector becomes null*)
                FI;
            OD;

        IF instruction has immediate operand
            THEN (* Release parameters from calling procedure's stack *)
                IF StackAddressSize = 32
                    THEN
                        ESP ← ESP + SRC;
                    ELSE (* StackAddressSize = 16 *)
                        SP ← SP + SRC;
                FI;
        FI;

    (* IA-32e Mode *)
        IF (PE = 1 and VM = 0 and IA32_EFER.LMA = 1) and instruction = far return
            THEN
                IF OperandSize = 32
                    THEN
                        IF second doubleword on stack is not within stack limits
                            THEN #SS(0); FI;
                        IF first or second doubleword on stack is not in canonical space
                            THEN #SS(0); FI;
                    ELSE
                        IF OperandSize = 16
                            THEN
                                IF second word on stack is not within stack limits
                                    THEN #SS(0); FI;
                                IF first or second word on stack is not in canonical space
                                    THEN #SS(0); FI;
                            ELSE (* OperandSize = 64 *)
```

```
                                IF first or second quadword on stack is not in canonical space
                                    THEN #SS(0); FI;
                            FI
                    FI;
            IF return code segment selector is NULL
                    THEN GP(0); FI;
            IF return code segment selector addresses descriptor beyond descriptor table limit
                    THEN GP(selector); FI;
            IF return code segment selector addresses descriptor in non-canonical space
                    THEN GP(selector); FI;
            Obtain descriptor to which return code segment selector points from descriptor table;
            IF return code segment descriptor is not a code segment
                    THEN #GP(selector); FI;
            IF return code segment descriptor has L-bit = 1 and D-bit = 1
                    THEN #GP(selector); FI;
            IF return code segment selector RPL < CPL
                    THEN #GP(selector); FI;
            IF return code segment descriptor is conforming
            and return code segment DPL > return code segment selector RPL
                    THEN #GP(selector); FI;
            IF return code segment descriptor is non-conforming
            and return code segment DPL ≠ return code segment selector RPL
                    THEN #GP(selector); FI;
            IF return code segment descriptor is not present
                    THEN #NP(selector); FI:
            IF return code segment selector RPL > CPL
                    THEN GOTO IA-32E-MODE-RETURN-TO-OUTER-PRIVILEGE-LEVEL;
                    ELSE GOTO IA-32E-MODE-RETURN-TO-SAME-PRIVILEGE-LEVEL;
            FI;
    FI;

IA-32E-MODE-RETURN-TO-SAME-PRIVILEGE-LEVEL:
IF the return instruction pointer is not within the return code segment limit
    THEN #GP(0); FI;
IF the return instruction pointer is not within canonical address space
    THEN #GP(0); FI;
IF OperandSize = 32
    THEN
        EIP ← Pop();
        CS ← Pop(); (* 32-bit pop, high-order 16 bits discarded *)
    ELSE
        IF OperandSize = 16
            THEN
                EIP ← Pop();
                EIP ← EIP AND 0000FFFFH;
                CS ← Pop(); (* 16-bit pop *)
            ELSE (* OperandSize = 64 *)
                RIP ← Pop();
                CS ← Pop(); (* 64-bit pop, high-order 48 bits discarded *)
        FI;
FI;
IF instruction has immediate operand
    THEN (* Release parameters from stack *)
        IF StackAddressSize = 32
```

```
                THEN
                    ESP ← ESP + SRC;
                ELSE
                    IF StackAddressSize = 16
                        THEN
                            SP ← SP + SRC;
                        ELSE (* StackAddressSize = 64 *)
                            RSP ← RSP + SRC;
                    FI;
            FI;
FI;

IA-32E-MODE-RETURN-TO-OUTER-PRIVILEGE-LEVEL:
IF top (16 + SRC) bytes of stack are not within stack limits (OperandSize = 32)
or top (8 + SRC) bytes of stack are not within stack limits (OperandSize = 16)
    THEN #SS(0); FI;
IF top (16 + SRC) bytes of stack are not in canonical address space (OperandSize = 32)
or top (8 + SRC) bytes of stack are not in canonical address space (OperandSize = 16)
or top (32 + SRC) bytes of stack are not in canonical address space (OperandSize = 64)
    THEN #SS(0); FI;
Read return stack segment selector;
IF stack segment selector is NULL
    THEN
        IF new CS descriptor L-bit = 0
            THEN #GP(selector);
        IF stack segment selector RPL = 3
            THEN #GP(selector);
FI;
IF return stack segment descriptor is not within descriptor table limits
        THEN #GP(selector); FI;
IF return stack segment descriptor is in non-canonical address space
        THEN #GP(selector); FI;
Read segment descriptor pointed to by return segment selector;
IF stack segment selector RPL ≠ RPL of the return code segment selector
or stack segment is not a writable data segment
or stack segment descriptor DPL ≠ RPL of the return code segment selector
    THEN #GP(selector); FI;
IF stack segment not present
    THEN #SS(StackSegmentSelector); FI;
IF the return instruction pointer is not within the return code segment limit
    THEN #GP(0); FI:
IF the return instruction pointer is not within canonical address space
    THEN #GP(0); FI;
CPL ← ReturnCodeSegmentSelector(RPL);
IF OperandSize = 32
    THEN
        EIP ← Pop();
        CS ← Pop(); (* 32-bit pop, high-order 16 bits discarded, segment descriptor loaded *)
        CS(RPL) ← CPL;
        IF instruction has immediate operand
            THEN (* Release parameters from called procedure's stack *)
                IF StackAddressSize = 32
                    THEN
                        ESP ← ESP + SRC;
```

```
                        ELSE
                            IF StackAddressSize = 16
                                THEN
                                    SP ← SP + SRC;
                                ELSE (* StackAddressSize = 64 *)
                                    RSP ← RSP + SRC;
                            FI;
                    FI;
            FI;
            tempESP ← Pop();
            tempSS ← Pop(); (* 32-bit pop, high-order 16 bits discarded, segment descriptor loaded *)
            ESP ← tempESP;
            SS ← tempSS;
        ELSE
            IF OperandSize = 16
                THEN
                    EIP ← Pop();
                    EIP ← EIP AND 0000FFFFH;
                    CS ← Pop(); (* 16-bit pop; segment descriptor loaded *)
                    CS(RPL) ← CPL;
                    IF instruction has immediate operand
                        THEN (* Release parameters from called procedure's stack *)
                            IF StackAddressSize = 32
                                THEN
                                    ESP ← ESP + SRC;
                                ELSE
                                    IF StackAddressSize = 16
                                        THEN
                                            SP ← SP + SRC;
                                        ELSE (* StackAddressSize = 64 *)
                                            RSP ← RSP + SRC;
                                    FI;
                            FI;
                    FI;
                    tempESP ← Pop();
                    tempSS ← Pop(); (* 16-bit pop; segment descriptor loaded *)
                    ESP ← tempESP;
                    SS ← tempSS;
                ELSE (* OperandSize = 64 *)
                    RIP ← Pop();
                    CS ← Pop(); (* 64-bit pop; high-order 48 bits discarded; seg. descriptor loaded *)
                    CS(RPL) ← CPL;
                    IF instruction has immediate operand
                        THEN (* Release parameters from called procedure's stack *)
                            RSP ← RSP + SRC;
                    FI;
                    tempESP ← Pop();
                    tempSS ← Pop(); (* 64-bit pop; high-order 48 bits discarded; seg. desc. loaded *)
                    ESP ← tempESP;
                    SS ← tempSS;
            FI;
FI;

FOR each of segment register (ES, FS, GS, and DS)
```

```
    DO
        IF segment register points to data or non-conforming code segment
        and CPL > segment descriptor DPL; (* DPL in hidden part of segment register *)
            THEN SegmentSelector ← 0; (* SegmentSelector invalid *)
        FI;
    OD;

IF instruction has immediate operand
    THEN (* Release parameters from calling procedure's stack *)
        IF StackAddressSize = 32
            THEN
                ESP ← ESP + SRC;
            ELSE
                IF StackAddressSize = 16
                    THEN
                        SP ← SP + SRC;
                    ELSE (* StackAddressSize = 64 *)
                        RSP ← RSP + SRC;
                FI;
        FI;
FI;
```

Flags Affected

None.

Protected Mode Exceptions

#GP(0)	If the return code or stack segment selector is NULL.
	If the return instruction pointer is not within the return code segment limit
#GP(selector)	If the RPL of the return code segment selector is less then the CPL.
	If the return code or stack segment selector index is not within its descriptor table limits.
	If the return code segment descriptor does not indicate a code segment.
	If the return code segment is non-conforming and the segment selector's DPL is not equal to the RPL of the code segment's segment selector
	If the return code segment is conforming and the segment selector's DPL greater than the RPL of the code segment's segment selector
	If the stack segment is not a writable data segment.
	If the stack segment selector RPL is not equal to the RPL of the return code segment selector.
	If the stack segment descriptor DPL is not equal to the RPL of the return code segment selector.
#SS(0)	If the top bytes of stack are not within stack limits.
	If the return stack segment is not present.
#NP(selector)	If the return code segment is not present.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If an unaligned memory access occurs when the CPL is 3 and alignment checking is enabled.

Real-Address Mode Exceptions

#GP	If the return instruction pointer is not within the return code segment limit
#SS	If the top bytes of stack are not within stack limits.

Virtual-8086 Mode Exceptions

#GP(0)	If the return instruction pointer is not within the return code segment limit
#SS(0)	If the top bytes of stack are not within stack limits.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If an unaligned memory access occurs when alignment checking is enabled.

Compatibility Mode Exceptions

Same as 64-bit mode exceptions.

64-Bit Mode Exceptions

#GP(0)	If the return instruction pointer is non-canonical.
	If the return instruction pointer is not within the return code segment limit.
	If the stack segment selector is NULL going back to compatibility mode.
	If the stack segment selector is NULL going back to CPL3 64-bit mode.
	If a NULL stack segment selector RPL is not equal to CPL going back to non-CPL3 64-bit mode.
	If the return code segment selector is NULL.
#GP(selector)	If the proposed segment descriptor for a code segment does not indicate it is a code segment.
	If the proposed new code segment descriptor has both the D-bit and L-bit set.
	If the DPL for a nonconforming-code segment is not equal to the RPL of the code segment selector.
	If CPL is greater than the RPL of the code segment selector.
	If the DPL of a conforming-code segment is greater than the return code segment selector RPL.
	If a segment selector index is outside its descriptor table limits.
	If a segment descriptor memory address is non-canonical.
	If the stack segment is not a writable data segment.
	If the stack segment descriptor DPL is not equal to the RPL of the return code segment selector.
	If the stack segment selector RPL is not equal to the RPL of the return code segment selector.
#SS(0)	If an attempt to pop a value off the stack violates the SS limit.
	If an attempt to pop a value off the stack causes a non-canonical address to be referenced.
#NP(selector)	If the return code or stack segment is not present.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

RORX — Rotate Right Logical Without Affecting Flags

Opcode/ Instruction	Op/ En	64/32 -bit Mode	CPUID Feature Flag	Description
VEX.LZ.F2.0F3A.W0 F0 /r ib RORX r32, *r/m32, imm8*	RMI	V/V	BMI2	Rotate 32-bit *r/m32* right *imm8* times without affecting arithmetic flags.
VEX.LZ.F2.0F3A.W1 F0 /r ib RORX r64, *r/m64, imm8*	RMI	V/N.E.	BMI2	Rotate 64-bit *r/m64* right *imm8* times without affecting arithmetic flags.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RMI	ModRM:reg (w)	ModRM:r/m (r)	Imm8	NA

Description

Rotates the bits of second operand right by the count value specified in imm8 without affecting arithmetic flags. The RORX instruction does not read or write the arithmetic flags.

This instruction is not supported in real mode and virtual-8086 mode. The operand size is always 32 bits if not in 64-bit mode. In 64-bit mode operand size 64 requires VEX.W1. VEX.W1 is ignored in non-64-bit modes. An attempt to execute this instruction with VEX.L not equal to 0 will cause #UD.

Operation

```
IF (OperandSize = 32)
    y ← imm8 AND 1FH;
    DEST ← (SRC >> y) | (SRC << (32-y));
ELSEIF (OperandSize = 64 )
    y ← imm8 AND 3FH;
    DEST ← (SRC >> y) | (SRC << (64-y));
ENDIF
```

Flags Affected

None

Intel C/C++ Compiler Intrinsic Equivalent

Auto-generated from high-level language.

SIMD Floating-Point Exceptions

None

Other Exceptions

See Section 2.5.1, "Exception Conditions for VEX-Encoded GPR Instructions", Table 2-29; additionally

#UD	If VEX.W = 1.

ROUNDPD — Round Packed Double Precision Floating-Point Values

Opcode*/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 3A 09 /r ib ROUNDPD *xmm1, xmm2/m128, imm8*	RMI	V/V	SSE4_1	Round packed double precision floating-point values in *xmm2/m128* and place the result in *xmm1*. The rounding mode is determined by *imm8*.
VEX.128.66.0F3A.WIG 09 /r ib VROUNDPD *xmm1, xmm2/m128, imm8*	RMI	V/V	AVX	Round packed double-precision floating-point values in *xmm2/m128* and place the result in *xmm1*. The rounding mode is determined by *imm8*.
VEX.256.66.0F3A.WIG 09 /r ib VROUNDPD *ymm1, ymm2/m256, imm8*	RMI	V/V	AVX	Round packed double-precision floating-point values in *ymm2/m256* and place the result in *ymm1*. The rounding mode is determined by *imm8*.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RMI	ModRM:reg (w)	ModRM:r/m (r)	imm8	NA

Description

Round the 2 double-precision floating-point values in the source operand (second operand) using the rounding mode specified in the immediate operand (third operand) and place the results in the destination operand (first operand). The rounding process rounds each input floating-point value to an integer value and returns the integer result as a double-precision floating-point value.

The immediate operand specifies control fields for the rounding operation, three bit fields are defined and shown in Figure 4-24. Bit 3 of the immediate byte controls processor behavior for a precision exception, bit 2 selects the source of rounding mode control. Bits 1:0 specify a non-sticky rounding-mode value (Table 4-18 lists the encoded values for rounding-mode field).

The Precision Floating-Point Exception is signaled according to the immediate operand. If any source operand is an SNaN then it will be converted to a QNaN. If DAZ is set to `1 then denormals will be converted to zero before rounding.

128-bit Legacy SSE version: The second source can be an XMM register or 128-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (MAXVL-1:128) of the corresponding YMM register destination are unmodified.

VEX.128 encoded version: the source operand second source operand or a 128-bit memory location. The destination operand is an XMM register. The upper bits (MAXVL-1:128) of the corresponding YMM register destination are zeroed.

VEX.256 encoded version: The source operand is a YMM register or a 256-bit memory location. The destination operand is a YMM register.

Note: In VEX-encoded versions, VEX.vvvv is reserved and must be 1111b, otherwise instructions will #UD.

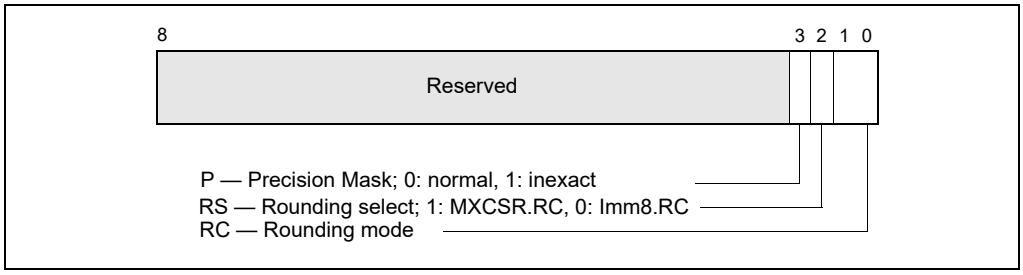


Figure 4-24. Bit Control Fields of Immediate Byte for ROUNDxx Instruction

Table 4-18. Rounding Modes and Encoding of Rounding Control (RC) Field

Rounding Mode	RC Field Setting	Description
Round to nearest (even)	00B	Rounded result is the closest to the infinitely precise result. If two values are equally close, the result is the even value (i.e., the integer value with the least-significant bit of zero).
Round down (toward −∞)	01B	Rounded result is closest to but no greater than the infinitely precise result.
Round up (toward +∞)	10B	Rounded result is closest to but no less than the infinitely precise result.
Round toward zero (Truncate)	11B	Rounded result is closest to but no greater in absolute value than the infinitely precise result.

Operation

```
IF (imm[2] = '1)
    THEN    // rounding mode is determined by MXCSR.RC
        DEST[63:0] ← ConvertDPFPToInteger_M(SRC[63:0]);
        DEST[127:64] ← ConvertDPFPToInteger_M(SRC[127:64]);
    ELSE    // rounding mode is determined by IMM8.RC
        DEST[63:0] ← ConvertDPFPToInteger_Imm(SRC[63:0]);
        DEST[127:64] ← ConvertDPFPToInteger_Imm(SRC[127:64]);
FI
```

ROUNDPD (128-bit Legacy SSE version)

```
DEST[63:0] ← RoundToInteger(SRC[63:0]], ROUND_CONTROL)
DEST[127:64] ← RoundToInteger(SRC[127:64]], ROUND_CONTROL)
DEST[MAXVL-1:128] (Unmodified)
```

VROUNDPD (VEX.128 encoded version)

```
DEST[63:0] ← RoundToInteger(SRC[63:0]], ROUND_CONTROL)
DEST[127:64] ← RoundToInteger(SRC[127:64]], ROUND_CONTROL)
DEST[MAXVL-1:128] ← 0
```

VROUNDPD (VEX.256 encoded version)

```
DEST[63:0] ← RoundToInteger(SRC[63:0], ROUND_CONTROL)
DEST[127:64] ← RoundToInteger(SRC[127:64]], ROUND_CONTROL)
DEST[191:128] ← RoundToInteger(SRC[191:128]], ROUND_CONTROL)
DEST[255:192] ← RoundToInteger(SRC[255:192] ], ROUND_CONTROL)
```

Intel C/C++ Compiler Intrinsic Equivalent

__m128 _mm_round_pd(__m128d s1, int iRoundMode);

__m128 _mm_floor_pd(__m128d s1);

__m128 _mm_ceil_pd(__m128d s1)

__m256 _mm256_round_pd(__m256d s1, int iRoundMode);

__m256 _mm256_floor_pd(__m256d s1);

__m256 _mm256_ceil_pd(__m256d s1)

SIMD Floating-Point Exceptions

Invalid (signaled only if SRC = SNaN)

Precision (signaled only if imm[3] = '0; if imm[3] = '1, then the Precision Mask in the MXSCSR is ignored and precision exception is not signaled.)

Note that Denormal is not signaled by ROUNDPD.

Other Exceptions

See Exceptions Type 2; additionally

#UD	If VEX.vvvv ≠ 1111B.

ROUNDPS — Round Packed Single Precision Floating-Point Values

Opcode*/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 3A 08 /r ib ROUNDPS *xmm1, xmm2/m128, imm8*	RMI	V/V	SSE4_1	Round packed single precision floating-point values in *xmm2/m128* and place the result in *xmm1*. The rounding mode is determined by *imm8*.
VEX.128.66.0F3A.WIG 08 /r ib VROUNDPS *xmm1, xmm2/m128, imm8*	RMI	V/V	AVX	Round packed single-precision floating-point values in *xmm2/m128* and place the result in xmm1. The rounding mode is determined by *imm8*.
VEX.256.66.0F3A.WIG 08 /r ib VROUNDPS *ymm1, ymm2/m256, imm8*	RMI	V/V	AVX	Round packed single-precision floating-point values in *ymm2/m256* and place the result in ymm1. The rounding mode is determined by *imm8*.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RMI	ModRM:reg (w)	ModRM:r/m (r)	imm8	NA

Description

Round the 4 single-precision floating-point values in the source operand (second operand) using the rounding mode specified in the immediate operand (third operand) and place the results in the destination operand (first operand). The rounding process rounds each input floating-point value to an integer value and returns the integer result as a single-precision floating-point value.

The immediate operand specifies control fields for the rounding operation, three bit fields are defined and shown in Figure 4-24. Bit 3 of the immediate byte controls processor behavior for a precision exception, bit 2 selects the source of rounding mode control. Bits 1:0 specify a non-sticky rounding-mode value (Table 4-18 lists the encoded values for rounding-mode field).

The Precision Floating-Point Exception is signaled according to the immediate operand. If any source operand is an SNaN then it will be converted to a QNaN. If DAZ is set to `1 then denormals will be converted to zero before rounding.

128-bit Legacy SSE version: The second source can be an XMM register or 128-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (MAXVL-1:128) of the corresponding YMM register destination are unmodified.

VEX.128 encoded version: the source operand second source operand or a 128-bit memory location. The destination operand is an XMM register. The upper bits (MAXVL-1:128) of the corresponding YMM register destination are zeroed.

VEX.256 encoded version: The source operand is a YMM register or a 256-bit memory location. The destination operand is a YMM register.

Note: In VEX-encoded versions, VEX.vvvv is reserved and must be 1111b otherwise instructions will #UD.

Operation

```
IF (imm[2] = '1)
    THEN    // rounding mode is determined by MXCSR.RC
        DEST[31:0] ← ConvertSPFPToInteger_M(SRC[31:0]);
        DEST[63:32] ← ConvertSPFPToInteger_M(SRC[63:32]);
        DEST[95:64] ← ConvertSPFPToInteger_M(SRC[95:64]);
        DEST[127:96] ← ConvertSPFPToInteger_M(SRC[127:96]);
    ELSE    // rounding mode is determined by IMM8.RC
        DEST[31:0] ← ConvertSPFPToInteger_Imm(SRC[31:0]);
        DEST[63:32] ← ConvertSPFPToInteger_Imm(SRC[63:32]);
        DEST[95:64] ← ConvertSPFPToInteger_Imm(SRC[95:64]);
        DEST[127:96] ← ConvertSPFPToInteger_Imm(SRC[127:96]);
FI;
```

ROUNDPS(128-bit Legacy SSE version)

```
DEST[31:0] ← RoundToInteger(SRC[31:0], ROUND_CONTROL)
DEST[63:32] ← RoundToInteger(SRC[63:32], ROUND_CONTROL)
DEST[95:64] ← RoundToInteger(SRC[95:64]], ROUND_CONTROL)
DEST[127:96] ← RoundToInteger(SRC[127:96]], ROUND_CONTROL)
DEST[MAXVL-1:128] (Unmodified)
```

VROUNDPS (VEX.128 encoded version)

```
DEST[31:0] ← RoundToInteger(SRC[31:0], ROUND_CONTROL)
DEST[63:32] ← RoundToInteger(SRC[63:32], ROUND_CONTROL)
DEST[95:64] ← RoundToInteger(SRC[95:64]], ROUND_CONTROL)
DEST[127:96] ← RoundToInteger(SRC[127:96]], ROUND_CONTROL)
DEST[MAXVL-1:128] ← 0
```

VROUNDPS (VEX.256 encoded version)

```
DEST[31:0] ← RoundToInteger(SRC[31:0], ROUND_CONTROL)
DEST[63:32] ← RoundToInteger(SRC[63:32], ROUND_CONTROL)
DEST[95:64] ← RoundToInteger(SRC[95:64]], ROUND_CONTROL)
DEST[127:96] ← RoundToInteger(SRC[127:96]], ROUND_CONTROL)
DEST[159:128] ← RoundToInteger(SRC[159:128]], ROUND_CONTROL)
DEST[191:160] ← RoundToInteger(SRC[191:160]], ROUND_CONTROL)
DEST[223:192] ← RoundToInteger(SRC[223:192] ], ROUND_CONTROL)
DEST[255:224] ← RoundToInteger(SRC[255:224] ], ROUND_CONTROL)
```

Intel C/C++ Compiler Intrinsic Equivalent

__m128 _mm_round_ps(__m128 s1, int iRoundMode);

__m128 _mm_floor_ps(__m128 s1);

__m128 _mm_ceil_ps(__m128 s1)

__m256 _mm256_round_ps(__m256 s1, int iRoundMode);

__m256 _mm256_floor_ps(__m256 s1);

__m256 _mm256_ceil_ps(__m256 s1)

SIMD Floating-Point Exceptions

Invalid (signaled only if SRC = SNaN)

Precision (signaled only if imm[3] = ‘0; if imm[3] = ‘1, then the Precision Mask in the MXSCSR is ignored and precision exception is not signaled.)

Note that Denormal is not signaled by ROUNDPS.

Other Exceptions

See Exceptions Type 2; additionally

#UD	If VEX.vvvv ≠ 1111B.

ROUNDSD — Round Scalar Double Precision Floating-Point Values

Opcode*/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 3A 0B /r ib ROUNDSD *xmm1, xmm2/m64, imm8*	RMI	V/V	SSE4_1	Round the low packed double precision floating-point value in *xmm2/m64* and place the result in *xmm1*. The rounding mode is determined by *imm8*.
VEX.NDS.LIG.66.0F3A.WIG 0B /r ib VROUNDSD *xmm1, xmm2, xmm3/m64, imm8*	RVMI	V/V	AVX	Round the low packed double precision floating-point value in *xmm3/m64* and place the result in *xmm1*. The rounding mode is determined by *imm8*. Upper packed double precision floating-point value (bits[127:64]) from *xmm2* is copied to *xmm1*[127:64].

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RMI	ModRM:reg (w)	ModRM:r/m (r)	imm8	NA
RVMI	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	imm8

Description

Round the DP FP value in the lower qword of the source operand (second operand) using the rounding mode specified in the immediate operand (third operand) and place the result in the destination operand (first operand). The rounding process rounds a double-precision floating-point input to an integer value and returns the integer result as a double precision floating-point value in the lowest position. The upper double precision floating-point value in the destination is retained.

The immediate operand specifies control fields for the rounding operation, three bit fields are defined and shown in Figure 4-24. Bit 3 of the immediate byte controls processor behavior for a precision exception, bit 2 selects the source of rounding mode control. Bits 1:0 specify a non-sticky rounding-mode value (Table 4-18 lists the encoded values for rounding-mode field).

The Precision Floating-Point Exception is signaled according to the immediate operand. If any source operand is an SNaN then it will be converted to a QNaN. If DAZ is set to `1 then denormals will be converted to zero before rounding.

128-bit Legacy SSE version: The first source operand and the destination operand are the same. Bits (MAXVL-1:64) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: Bits (MAXVL-1:128) of the destination YMM register are zeroed.

Operation

```
IF (imm[2] = '1)
    THEN    // rounding mode is determined by MXCSR.RC
        DEST[63:0] ← ConvertDPFPToInteger_M(SRC[63:0]);
    ELSE    // rounding mode is determined by IMM8.RC
        DEST[63:0] ← ConvertDPFPToInteger_Imm(SRC[63:0]);
FI;
DEST[127:63] remains unchanged ;
```

ROUNDSD (128-bit Legacy SSE version)

```
DEST[63:0] ← RoundToInteger(SRC[63:0], ROUND_CONTROL)
DEST[MAXVL-1:64] (Unmodified)
```

VROUNDSD (VEX.128 encoded version)
DEST[63:0] ← RoundToInteger(SRC2[63:0], ROUND_CONTROL)
DEST[127:64] ← SRC1[127:64]
DEST[MAXVL-1:128] ← 0

Intel C/C++ Compiler Intrinsic Equivalent

ROUNDSD: __m128d mm_round_sd(__m128d dst, __m128d s1, int iRoundMode);
__m128d mm_floor_sd(__m128d dst, __m128d s1);
__m128d mm_ceil_sd(__m128d dst, __m128d s1);

SIMD Floating-Point Exceptions

Invalid (signaled only if SRC = SNaN)

Precision (signaled only if imm[3] = '0; if imm[3] = '1, then the Precision Mask in the MXSCSR is ignored and precision exception is not signaled.)

Note that Denormal is not signaled by ROUNDSD.

Other Exceptions

See Exceptions Type 3.

ROUNDSS — Round Scalar Single Precision Floating-Point Values

Opcode*/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 3A 0A /r ib ROUNDSS *xmm1, xmm2/m32, imm8*	RMI	V/V	SSE4_1	Round the low packed single precision floating-point value in *xmm2/m32* and place the result in *xmm1*. The rounding mode is determined by *imm8*.
VEX.NDS.LIG.66.0F3A.WIG 0A /r ib VROUNDSS *xmm1, xmm2, xmm3/m32, imm8*	RVMI	V/V	AVX	Round the low packed single precision floating-point value in *xmm3/m32* and place the result in *xmm1*. The rounding mode is determined by *imm8*. Also, upper packed single precision floating-point values (bits[127:32]) from *xmm2* are copied to *xmm1*[127:32].

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RMI	ModRM:reg (w)	ModRM:r/m (r)	imm8	NA
RVMI	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	imm8

Description

Round the single-precision floating-point value in the lowest dword of the source operand (second operand) using the rounding mode specified in the immediate operand (third operand) and place the result in the destination operand (first operand). The rounding process rounds a single-precision floating-point input to an integer value and returns the result as a single-precision floating-point value in the lowest position. The upper three single-precision floating-point values in the destination are retained.

The immediate operand specifies control fields for the rounding operation, three bit fields are defined and shown in Figure 4-24. Bit 3 of the immediate byte controls processor behavior for a precision exception, bit 2 selects the source of rounding mode control. Bits 1:0 specify a non-sticky rounding-mode value (Table 4-18 lists the encoded values for rounding-mode field).

The Precision Floating-Point Exception is signaled according to the immediate operand. If any source operand is an SNaN then it will be converted to a QNaN. If DAZ is set to '1 then denormals will be converted to zero before rounding.

128-bit Legacy SSE version: The first source operand and the destination operand are the same. Bits (MAXVL-1:32) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: Bits (MAXVL-1:128) of the destination YMM register are zeroed.

Operation

```
IF (imm[2] = '1)
    THEN    // rounding mode is determined by MXCSR.RC
        DEST[31:0] ← ConvertSPFPToInteger_M(SRC[31:0]);
    ELSE    // rounding mode is determined by IMM8.RC
        DEST[31:0] ← ConvertSPFPToInteger_Imm(SRC[31:0]);
FI;
DEST[127:32] remains unchanged ;
```

ROUNDSS (128-bit Legacy SSE version)

```
DEST[31:0] ← RoundToInteger(SRC[31:0], ROUND_CONTROL)
DEST[MAXVL-1:32] (Unmodified)
```

VROUNDSS (VEX.128 encoded version)

```
DEST[31:0] ← RoundToInteger(SRC2[31:0], ROUND_CONTROL)
DEST[127:32] ← SRC1[127:32]
DEST[MAXVL-1:128] ← 0
```

Intel C/C++ Compiler Intrinsic Equivalent

ROUNDSS: __m128 mm_round_ss(__m128 dst, __m128 s1, int iRoundMode);
__m128 mm_floor_ss(__m128 dst, __m128 s1);
__m128 mm_ceil_ss(__m128 dst, __m128 s1);

SIMD Floating-Point Exceptions

Invalid (signaled only if SRC = SNaN)

Precision (signaled only if imm[3] = '0; if imm[3] = '1, then the Precision Mask in the MXSCSR is ignored and precision exception is not signaled.)

Note that Denormal is not signaled by ROUNDSS.

Other Exceptions

See Exceptions Type 3.

RSM—Resume from System Management Mode

Opcode*	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
0F AA	RSM	ZO	Valid	Valid	Resume operation of interrupted program.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
ZO	NA	NA	NA	NA

Description

Returns program control from system management mode (SMM) to the application program or operating-system procedure that was interrupted when the processor received an SMM interrupt. The processor's state is restored from the dump created upon entering SMM. If the processor detects invalid state information during state restoration, it enters the shutdown state. The following invalid information can cause a shutdown:

- Any reserved bit of CR4 is set to 1.
- Any illegal combination of bits in CR0, such as (PG=1 and PE=0) or (NW=1 and CD=0).
- (Intel Pentium and Intel486™ processors only.) The value stored in the state dump base field is not a 32-KByte aligned address.

The contents of the model-specific registers are not affected by a return from SMM.

The SMM state map used by RSM supports resuming processor context for non-64-bit modes and 64-bit mode.

See Chapter 34, "System Management Mode," in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3C*, for more information about SMM and the behavior of the RSM instruction.

Operation

```
ReturnFromSMM;
IF (IA-32e mode supported) or (CPUID DisplayFamily_DisplayModel = 06H_0CH )
    THEN
        ProcessorState ← Restore(SMMDump(IA-32e SMM STATE MAP));
    Else
        ProcessorState ← Restore(SMMDump(Non-32-Bit-Mode SMM STATE MAP));
FI
```

Flags Affected

All.

Protected Mode Exceptions

#UD	If an attempt is made to execute this instruction when the processor is not in SMM. If the LOCK prefix is used.

Real-Address Mode Exceptions

Same exceptions as in protected mode.

Virtual-8086 Mode Exceptions

Same exceptions as in protected mode.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

Same exceptions as in protected mode.

RSQRTPS—Compute Reciprocals of Square Roots of Packed Single-Precision Floating-Point Values

Opcode*/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F 52 /r RSQRTPS *xmm1, xmm2/m128*	RM	V/V	SSE	Computes the approximate reciprocals of the square roots of the packed single-precision floating-point values in *xmm2/m128* and stores the results in *xmm1*.
VEX.128.0F.WIG 52 /r VRSQRTPS *xmm1, xmm2/m128*	RM	V/V	AVX	Computes the approximate reciprocals of the square roots of packed single-precision values in *xmm2/mem* and stores the results in *xmm1*.
VEX.256.0F.WIG 52 /r VRSQRTPS *ymm1, ymm2/m256*	RM	V/V	AVX	Computes the approximate reciprocals of the square roots of packed single-precision values in *ymm2/mem* and stores the results in *ymm1*.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

Description

Performs a SIMD computation of the approximate reciprocals of the square roots of the four packed single-precision floating-point values in the source operand (second operand) and stores the packed single-precision floating-point results in the destination operand. The source operand can be an XMM register or a 128-bit memory location. The destination operand is an XMM register. See Figure 10-5 in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*, for an illustration of a SIMD single-precision floating-point operation.

The relative error for this approximation is:

$$|\text{Relative Error}| \leq 1.5 * 2^{-12}$$

The RSQRTPS instruction is not affected by the rounding control bits in the MXCSR register. When a source value is a 0.0, an ∞ of the sign of the source value is returned. A denormal source value is treated as a 0.0 (of the same sign). When a source value is a negative value (other than –0.0), a floating-point indefinite is returned. When a source value is an SNaN or QNaN, the SNaN is converted to a QNaN or the source QNaN is returned.

In 64-bit mode, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

128-bit Legacy SSE version: The second source can be an XMM register or an 128-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (MAXVL-1:128) of the corresponding YMM register destination are unmodified.

VEX.128 encoded version: the first source operand is an XMM register or 128-bit memory location. The destination operand is an XMM register. The upper bits (MAXVL-1:128) of the corresponding YMM register destination are zeroed.

VEX.256 encoded version: The first source operand is a YMM register. The second source operand can be a YMM register or a 256-bit memory location. The destination operand is a YMM register.

Note: In VEX-encoded versions, VEX.vvvv is reserved and must be 1111b, otherwise instructions will #UD.

Operation

RSQRTPS (128-bit Legacy SSE version)

```
DEST[31:0] ← APPROXIMATE(1/SQRT(SRC[31:0]))
DEST[63:32] ← APPROXIMATE(1/SQRT(SRC1[63:32]))
DEST[95:64] ← APPROXIMATE(1/SQRT(SRC1[95:64]))
DEST[127:96] ← APPROXIMATE(1/SQRT(SRC2[127:96]))
DEST[MAXVL-1:128] (Unmodified)
```

VRSQRTPS (VEX.128 encoded version)

```
DEST[31:0] ← APPROXIMATE(1/SQRT(SRC[31:0]))
DEST[63:32] ← APPROXIMATE(1/SQRT(SRC1[63:32]))
DEST[95:64] ← APPROXIMATE(1/SQRT(SRC1[95:64]))
DEST[127:96] ← APPROXIMATE(1/SQRT(SRC2[127:96]))
DEST[MAXVL-1:128] ← 0
```

VRSQRTPS (VEX.256 encoded version)

```
DEST[31:0] ← APPROXIMATE(1/SQRT(SRC[31:0]))
DEST[63:32] ← APPROXIMATE(1/SQRT(SRC1[63:32]))
DEST[95:64] ← APPROXIMATE(1/SQRT(SRC1[95:64]))
DEST[127:96] ← APPROXIMATE(1/SQRT(SRC2[127:96]))
DEST[159:128] ← APPROXIMATE(1/SQRT(SRC2[159:128]))
DEST[191:160] ← APPROXIMATE(1/SQRT(SRC2[191:160]))
DEST[223:192] ← APPROXIMATE(1/SQRT(SRC2[223:192]))
DEST[255:224] ← APPROXIMATE(1/SQRT(SRC2[255:224]))
```

Intel C/C++ Compiler Intrinsic Equivalent

RSQRTPS: __m128 _mm_rsqrt_ps(__m128 a)

RSQRTPS: __m256 _mm256_rsqrt_ps (__m256 a);

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Exceptions Type 4; additionally

#UD	If VEX.vvvv ≠ 1111B.

RSQRTSS—Compute Reciprocal of Square Root of Scalar Single-Precision Floating-Point Value

Opcode*/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
F3 0F 52 /r RSQRTSS *xmm1*, *xmm2/m32*	RM	V/V	SSE	Computes the approximate reciprocal of the square root of the low single-precision floating-point value in *xmm2/m32* and stores the results in *xmm1*.
VEX.NDS.LIG.F3.0F.WIG 52 /r VRSQRTSS *xmm1, xmm2, xmm3/m32*	RVM	V/V	AVX	Computes the approximate reciprocal of the square root of the low single precision floating-point value in *xmm3/m32* and stores the results in *xmm1*. Also, upper single precision floating-point values (bits[127:32]) from *xmm2* are copied to *xmm1*[127:32].

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (w)	ModRM:r/m (r)	NA	NA
RVM	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA

Description

Computes an approximate reciprocal of the square root of the low single-precision floating-point value in the source operand (second operand) stores the single-precision floating-point result in the destination operand. The source operand can be an XMM register or a 32-bit memory location. The destination operand is an XMM register. The three high-order doublewords of the destination operand remain unchanged. See Figure 10-6 in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*, for an illustration of a scalar single-precision floating-point operation.

The relative error for this approximation is:

$$|\text{Relative Error}| \leq 1.5 * 2^{-12}$$

The RSQRTSS instruction is not affected by the rounding control bits in the MXCSR register. When a source value is a 0.0, an ∞ of the sign of the source value is returned. A denormal source value is treated as a 0.0 (of the same sign). When a source value is a negative value (other than –0.0), a floating-point indefinite is returned. When a source value is an SNaN or QNaN, the SNaN is converted to a QNaN or the source QNaN is returned.

In 64-bit mode, using a REX prefix in the form of REX.R permits this instruction to access additional registers (XMM8-XMM15).

128-bit Legacy SSE version: The first source operand and the destination operand are the same. Bits (MAXVL-1:32) of the corresponding YMM destination register remain unchanged.

VEX.128 encoded version: Bits (MAXVL-1:128) of the destination YMM register are zeroed.

Operation

RSQRTSS (128-bit Legacy SSE version)

```
DEST[31:0] ← APPROXIMATE(1/SQRT(SRC2[31:0]))
DEST[MAXVL-1:32] (Unmodified)
```

VRSQRTSS (VEX.128 encoded version)

```
DEST[31:0] ← APPROXIMATE(1/SQRT(SRC2[31:0]))
DEST[127:32] ← SRC1[127:32]
DEST[MAXVL-1:128] ← 0
```

Intel C/C++ Compiler Intrinsic Equivalent

RSQRTSS: __m128 _mm_rsqrt_ss(__m128 a)

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Exceptions Type 5.

SAHF—Store AH into Flags

Opcode*	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
9E	SAHF	ZO	Invalid*	Valid	Loads SF, ZF, AF, PF, and CF from AH into EFLAGS register.

NOTES:
* Valid in specific steppings. See Description section.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
ZO	NA	NA	NA	NA

Description

Loads the SF, ZF, AF, PF, and CF flags of the EFLAGS register with values from the corresponding bits in the AH register (bits 7, 6, 4, 2, and 0, respectively). Bits 1, 3, and 5 of register AH are ignored; the corresponding reserved bits (1, 3, and 5) in the EFLAGS register remain as shown in the "Operation" section below.

This instruction executes as described above in compatibility mode and legacy mode. It is valid in 64-bit mode only if CPUID.80000001H:ECX.LAHF-SAHF[bit 0] = 1.

Operation

```
IF IA-64 Mode
    THEN
        IF CPUID.80000001H.ECX[0] = 1;
            THEN
                RFLAGS(SF:ZF:0:AF:0:PF:1:CF) ← AH;
            ELSE
                #UD;
        FI
    ELSE
        EFLAGS(SF:ZF:0:AF:0:PF:1:CF) ← AH;
FI;
```

Flags Affected

The SF, ZF, AF, PF, and CF flags are loaded with values from the AH register. Bits 1, 3, and 5 of the EFLAGS register are unaffected, with the values remaining 1, 0, and 0, respectively.

Protected Mode Exceptions

None.

Real-Address Mode Exceptions

None.

Virtual-8086 Mode Exceptions

None.

Compatibility Mode Exceptions

None.

64-Bit Mode Exceptions

#UD	If CPUID.80000001H.ECX[0] = 0. If the LOCK prefix is used.

SAL/SAR/SHL/SHR—Shift

Opcode***	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
D0 /4	SAL *r/m8*, 1	M1	Valid	Valid	Multiply *r/m8* by 2, once.
REX + D0 /4	SAL *r/m8***, 1	M1	Valid	N.E.	Multiply *r/m8* by 2, once.
D2 /4	SAL *r/m8*, CL	MC	Valid	Valid	Multiply *r/m8* by 2, CL times.
REX + D2 /4	SAL *r/m8***, CL	MC	Valid	N.E.	Multiply *r/m8* by 2, CL times.
C0 /4 *ib*	SAL *r/m8, imm8*	MI	Valid	Valid	Multiply *r/m8* by 2, *imm8* times.
REX + C0 /4 *ib*	SAL *r/m8***, *imm8*	MI	Valid	N.E.	Multiply *r/m8* by 2, *imm8* times.
D1 /4	SAL *r/m16*, 1	M1	Valid	Valid	Multiply *r/m16* by 2, once.
D3 /4	SAL *r/m16*, CL	MC	Valid	Valid	Multiply *r/m16* by 2, CL times.
C1 /4 *ib*	SAL *r/m16, imm8*	MI	Valid	Valid	Multiply *r/m16* by 2, *imm8* times.
D1 /4	SAL *r/m32*, 1	M1	Valid	Valid	Multiply *r/m32* by 2, once.
REX.W + D1 /4	SAL *r/m64*, 1	M1	Valid	N.E.	Multiply *r/m64* by 2, once.
D3 /4	SAL *r/m32*, CL	MC	Valid	Valid	Multiply *r/m32* by 2, CL times.
REX.W + D3 /4	SAL *r/m64*, CL	MC	Valid	N.E.	Multiply *r/m64* by 2, CL times.
C1 /4 *ib*	SAL *r/m32, imm8*	MI	Valid	Valid	Multiply *r/m32* by 2, *imm8* times.
REX.W + C1 /4 *ib*	SAL *r/m64, imm8*	MI	Valid	N.E.	Multiply *r/m64* by 2, *imm8* times.
D0 /7	SAR *r/m8*, 1	M1	Valid	Valid	Signed divide* *r/m8* by 2, once.
REX + D0 /7	SAR *r/m8***, 1	M1	Valid	N.E.	Signed divide* *r/m8* by 2, once.
D2 /7	SAR *r/m8*, CL	MC	Valid	Valid	Signed divide* *r/m8* by 2, CL times.
REX + D2 /7	SAR *r/m8***, CL	MC	Valid	N.E.	Signed divide* *r/m8* by 2, CL times.
C0 /7 *ib*	SAR *r/m8, imm8*	MI	Valid	Valid	Signed divide* *r/m8* by 2, *imm8* time.
REX + C0 /7 *ib*	SAR *r/m8***, *imm8*	MI	Valid	N.E.	Signed divide* *r/m8* by 2, *imm8* times.
D1 /7	SAR *r/m16*,1	M1	Valid	Valid	Signed divide* *r/m16* by 2, once.
D3 /7	SAR *r/m16*, CL	MC	Valid	Valid	Signed divide* *r/m16* by 2, CL times.
C1 /7 *ib*	SAR *r/m16, imm8*	MI	Valid	Valid	Signed divide* *r/m16* by 2, *imm8* times.
D1 /7	SAR *r/m32*, 1	M1	Valid	Valid	Signed divide* *r/m32* by 2, once.
REX.W + D1 /7	SAR *r/m64*, 1	M1	Valid	N.E.	Signed divide* *r/m64* by 2, once.
D3 /7	SAR *r/m32*, CL	MC	Valid	Valid	Signed divide* *r/m32* by 2, CL times.
REX.W + D3 /7	SAR *r/m64*, CL	MC	Valid	N.E.	Signed divide* *r/m64* by 2, CL times.
C1 /7 *ib*	SAR *r/m32, imm8*	MI	Valid	Valid	Signed divide* *r/m32* by 2, *imm8* times.
REX.W + C1 /7 *ib*	SAR *r/m64, imm8*	MI	Valid	N.E.	Signed divide* *r/m64* by 2, *imm8* times
D0 /4	SHL *r/m8*, 1	M1	Valid	Valid	Multiply *r/m8* by 2, once.
REX + D0 /4	SHL *r/m8***, 1	M1	Valid	N.E.	Multiply *r/m8* by 2, once.
D2 /4	SHL *r/m8*, CL	MC	Valid	Valid	Multiply *r/m8* by 2, CL times.
REX + D2 /4	SHL *r/m8***, CL	MC	Valid	N.E.	Multiply *r/m8* by 2, CL times.
C0 /4 *ib*	SHL *r/m8, imm8*	MI	Valid	Valid	Multiply *r/m8* by 2, *imm8* times.
REX + C0 /4 *ib*	SHL *r/m8***, *imm8*	MI	Valid	N.E.	Multiply *r/m8* by 2, *imm8* times.
D1 /4	SHL *r/m16*,1	M1	Valid	Valid	Multiply *r/m16* by 2, once.
D3 /4	SHL *r/m16*, CL	MC	Valid	Valid	Multiply *r/m16* by 2, CL times.
C1 /4 *ib*	SHL *r/m16, imm8*	MI	Valid	Valid	Multiply *r/m16* by 2, *imm8* times.
D1 /4	SHL *r/m32*,1	M1	Valid	Valid	Multiply *r/m32* by 2, once.

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
REX.W + D1 /4	SHL *r/m64*,1	M1	Valid	N.E.	Multiply *r/m64* by 2, once.
D3 /4	SHL *r/m32*, CL	MC	Valid	Valid	Multiply *r/m32* by 2, CL times.
REX.W + D3 /4	SHL *r/m64*, CL	MC	Valid	N.E.	Multiply *r/m64* by 2, CL times.
C1 /4 *ib*	SHL *r/m32, imm8*	MI	Valid	Valid	Multiply *r/m32* by 2, *imm8* times.
REX.W + C1 /4 *ib*	SHL *r/m64, imm8*	MI	Valid	N.E.	Multiply *r/m64* by 2, *imm8* times.
D0 /5	SHR *r/m8*,1	M1	Valid	Valid	Unsigned divide *r/m8* by 2, once.
REX + D0 /5	SHR *r/m8***, 1	M1	Valid	N.E.	Unsigned divide *r/m8* by 2, once.
D2 /5	SHR *r/m8*, CL	MC	Valid	Valid	Unsigned divide *r/m8* by 2, CL times.
REX + D2 /5	SHR *r/m8***, CL	MC	Valid	N.E.	Unsigned divide *r/m8* by 2, CL times.
C0 /5 *ib*	SHR *r/m8, imm8*	MI	Valid	Valid	Unsigned divide *r/m8* by 2, *imm8* times.
REX + C0 /5 *ib*	SHR *r/m8***, *imm8*	MI	Valid	N.E.	Unsigned divide *r/m8* by 2, *imm8* times.
D1 /5	SHR *r/m16*, 1	M1	Valid	Valid	Unsigned divide *r/m16* by 2, once.
D3 /5	SHR *r/m16*, CL	MC	Valid	Valid	Unsigned divide *r/m16* by 2, CL times
C1 /5 *ib*	SHR *r/m16, imm8*	MI	Valid	Valid	Unsigned divide *r/m16* by 2, *imm8* times.
D1 /5	SHR *r/m32*, 1	M1	Valid	Valid	Unsigned divide *r/m32* by 2, once.
REX.W + D1 /5	SHR *r/m64*, 1	M1	Valid	N.E.	Unsigned divide *r/m64* by 2, once.
D3 /5	SHR *r/m32*, CL	MC	Valid	Valid	Unsigned divide *r/m32* by 2, CL times.
REX.W + D3 /5	SHR *r/m64*, CL	MC	Valid	N.E.	Unsigned divide *r/m64* by 2, CL times.
C1 /5 *ib*	SHR *r/m32, imm8*	MI	Valid	Valid	Unsigned divide *r/m32* by 2, *imm8* times.
REX.W + C1 /5 *ib*	SHR *r/m64, imm8*	MI	Valid	N.E.	Unsigned divide *r/m64* by 2, *imm8* times.

NOTES:

* Not the same form of division as IDIV; rounding is toward negative infinity.

** In 64-bit mode, r/m8 can not be encoded to access the following byte registers if a REX prefix is used: AH, BH, CH, DH.

***See IA-32 Architecture Compatibility section below.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
M1	ModRM:r/m (r, w)	1	NA	NA
MC	ModRM:r/m (r, w)	CL	NA	NA
MI	ModRM:r/m (r, w)	imm8	NA	NA

Description

Shifts the bits in the first operand (destination operand) to the left or right by the number of bits specified in the second operand (count operand). Bits shifted beyond the destination operand boundary are first shifted into the CF flag, then discarded. At the end of the shift operation, the CF flag contains the last bit shifted out of the destination operand.

The destination operand can be a register or a memory location. The count operand can be an immediate value or the CL register. The count is masked to 5 bits (or 6 bits if in 64-bit mode and REX.W is used). The count range is limited to 0 to 31 (or 63 if 64-bit mode and REX.W is used). A special opcode encoding is provided for a count of 1.

The shift arithmetic left (SAL) and shift logical left (SHL) instructions perform the same operation; they shift the bits in the destination operand to the left (toward more significant bit locations). For each shift count, the most significant bit of the destination operand is shifted into the CF flag, and the least significant bit is cleared (see Figure 7-7 in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*).

The shift arithmetic right (SAR) and shift logical right (SHR) instructions shift the bits of the destination operand to the right (toward less significant bit locations). For each shift count, the least significant bit of the destination operand is shifted into the CF flag, and the most significant bit is either set or cleared depending on the instruction type. The SHR instruction clears the most significant bit (see Figure 7-8 in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*); the SAR instruction sets or clears the most significant bit to correspond to the sign (most significant bit) of the original value in the destination operand. In effect, the SAR instruction fills the empty bit position's shifted value with the sign of the unshifted value (see Figure 7-9 in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*).

The SAR and SHR instructions can be used to perform signed or unsigned division, respectively, of the destination operand by powers of 2. For example, using the SAR instruction to shift a signed integer 1 bit to the right divides the value by 2.

Using the SAR instruction to perform a division operation does not produce the same result as the IDIV instruction. The quotient from the IDIV instruction is rounded toward zero, whereas the "quotient" of the SAR instruction is rounded toward negative infinity. This difference is apparent only for negative numbers. For example, when the IDIV instruction is used to divide -9 by 4, the result is -2 with a remainder of -1. If the SAR instruction is used to shift -9 right by two bits, the result is -3 and the "remainder" is +3; however, the SAR instruction stores only the most significant bit of the remainder (in the CF flag).

The OF flag is affected only on 1-bit shifts. For left shifts, the OF flag is set to 0 if the most-significant bit of the result is the same as the CF flag (that is, the top two bits of the original operand were the same); otherwise, it is set to 1. For the SAR instruction, the OF flag is cleared for all 1-bit shifts. For the SHR instruction, the OF flag is set to the most-significant bit of the original operand.

In 64-bit mode, the instruction's default operation size is 32 bits and the mask width for CL is 5 bits. Using a REX prefix in the form of REX.R permits access to additional registers (R8-R15). Using a REX prefix in the form of REX.W promotes operation to 64-bits and sets the mask width for CL to 6 bits. See the summary chart at the beginning of this section for encoding data and limits.

IA-32 Architecture Compatibility

The 8086 does not mask the shift count. However, all other IA-32 processors (starting with the Intel 286 processor) do mask the shift count to 5 bits, resulting in a maximum count of 31. This masking is done in all operating modes (including the virtual-8086 mode) to reduce the maximum execution time of the instructions.

Operation

```
IF 64-Bit Mode and using REX.W
    THEN
        countMASK ← 3FH;
    ELSE
        countMASK ← 1FH;
FI

tempCOUNT ← (COUNT AND countMASK);
tempDEST ← DEST;
WHILE (tempCOUNT ≠ 0)
DO
    IF instruction is SAL or SHL
        THEN
            CF ← MSB(DEST);
        ELSE (* Instruction is SAR or SHR *)
            CF ← LSB(DEST);
    FI;
    IF instruction is SAL or SHL
        THEN
            DEST ← DEST * 2;
        ELSE
            IF instruction is SAR
```

```
                THEN
                    DEST ← DEST / 2; (* Signed divide, rounding toward negative infinity *)
                ELSE (* Instruction is SHR *)
                    DEST ← DEST / 2 ; (* Unsigned divide *)
            FI;
    FI;
    tempCOUNT ← tempCOUNT – 1;
OD;

(* Determine overflow for the various instructions *)
IF (COUNT and countMASK) = 1
    THEN
        IF instruction is SAL or SHL
            THEN
                OF ← MSB(DEST) XOR CF;
            ELSE
                IF instruction is SAR
                    THEN
                        OF ← 0;
                    ELSE (* Instruction is SHR *)
                        OF ← MSB(tempDEST);
                FI;
        FI;
    ELSE IF (COUNT AND countMASK) = 0
        THEN
            All flags unchanged;
        ELSE (* COUNT not 1 or 0 *)
            OF ← undefined;
    FI;
FI;
```

Flags Affected

The CF flag contains the value of the last bit shifted out of the destination operand; it is undefined for SHL and SHR instructions where the count is greater than or equal to the size (in bits) of the destination operand. The OF flag is affected only for 1-bit shifts (see "Description" above); otherwise, it is undefined. The SF, ZF, and PF flags are set according to the result. If the count is 0, the flags are not affected. For a non-zero count, the AF flag is undefined.

Protected Mode Exceptions

#GP(0)	If the destination is located in a non-writable segment.
	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
	If the DS, ES, FS, or GS register contains a NULL segment selector.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If the LOCK prefix is used.

Real-Address Mode Exceptions

#GP	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS	If a memory operand effective address is outside the SS segment limit.
#UD	If the LOCK prefix is used.

Virtual-8086 Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.
#UD	If the LOCK prefix is used.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If the LOCK prefix is used.

SARX/SHLX/SHRX — Shift Without Affecting Flags

Opcode/ Instruction	Op/ En	64/32 -bit Mode	CPUID Feature Flag	Description
VEX.NDS.LZ.F3.0F38.W0 F7 /r SARX *r32a, r/m32, r32b*	RMV	V/V	BMI2	Shift *r/m32* arithmetically right with count specified in *r32b*.
VEX.NDS.LZ.66.0F38.W0 F7 /r SHLX *r32a, r/m32, r32b*	RMV	V/V	BMI2	Shift *r/m32* logically left with count specified in *r32b*.
VEX.NDS.LZ.F2.0F38.W0 F7 /r SHRX *r32a, r/m32, r32b*	RMV	V/V	BMI2	Shift *r/m32* logically right with count specified in *r32b*.
VEX.NDS.LZ.F3.0F38.W1 F7 /r SARX *r64a, r/m64, r64b*	RMV	V/N.E.	BMI2	Shift *r/m64* arithmetically right with count specified in *r64b*.
VEX.NDS.LZ.66.0F38.W1 F7 /r SHLX *r64a, r/m64, r64b*	RMV	V/N.E.	BMI2	Shift *r/m64* logically left with count specified in *r64b*.
VEX.NDS.LZ.F2.0F38.W1 F7 /r SHRX *r64a, r/m64, r64b*	RMV	V/N.E.	BMI2	Shift *r/m64* logically right with count specified in *r64b*.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RMV	ModRM:reg (w)	ModRM:r/m (r)	VEX.vvvv (r)	NA

Description

Shifts the bits of the first source operand (the second operand) to the left or right by a COUNT value specified in the second source operand (the third operand). The result is written to the destination operand (the first operand).

The shift arithmetic right (SARX) and shift logical right (SHRX) instructions shift the bits of the destination operand to the right (toward less significant bit locations), SARX keeps and propagates the most significant bit (sign bit) while shifting.

The logical shift left (SHLX) shifts the bits of the destination operand to the left (toward more significant bit locations).

This instruction is not supported in real mode and virtual-8086 mode. The operand size is always 32 bits if not in 64-bit mode. In 64-bit mode operand size 64 requires VEX.W1. VEX.W1 is ignored in non-64-bit modes. An attempt to execute this instruction with VEX.L not equal to 0 will cause #UD.

If the value specified in the first source operand exceeds OperandSize -1, the COUNT value is masked.

SARX,SHRX, and SHLX instructions do not update flags.

Operation

```
TEMP ← SRC1;
IF VEX.W1 and CS.L = 1
THEN
    countMASK ←3FH;
ELSE
    countMASK ←1FH;
FI
COUNT ← (SRC2 AND countMASK)

DEST[OperandSize -1] = TEMP[OperandSize -1];
DO WHILE (COUNT ≠ 0)
    IF instruction is SHLX
        THEN
            DEST[] ← DEST *2;
```

```
        ELSE IF instruction is SHRX
            THEN
                DEST[] ← DEST /2; //unsigned divide
        ELSE            // SARX
                DEST[] ← DEST /2; // signed divide, round toward negative infinity
    FI;
    COUNT ← COUNT - 1;
OD
```

Flags Affected

None.

Intel C/C++ Compiler Intrinsic Equivalent

Auto-generated from high-level language.

SIMD Floating-Point Exceptions

None

Other Exceptions

See Section 2.5.1, "Exception Conditions for VEX-Encoded GPR Instructions", Table 2-29; additionally

#UD	If VEX.W = 1.

SBB—Integer Subtraction with Borrow

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
1C *ib*	SBB AL, *imm8*	I	Valid	Valid	Subtract with borrow *imm8* from AL.
1D *iw*	SBB AX, *imm16*	I	Valid	Valid	Subtract with borrow *imm16* from AX.
1D *id*	SBB EAX, *imm32*	I	Valid	Valid	Subtract with borrow *imm32* from EAX.
REX.W + 1D *id*	SBB RAX, *imm32*	I	Valid	N.E.	Subtract with borrow sign-extended *imm.32 to 64-bits* from RAX.
80 /3 *ib*	SBB *r/m8, imm8*	MI	Valid	Valid	Subtract with borrow *imm8* from *r/m8.*
REX + 80 /3 *ib*	SBB *r/m8*, imm8*	MI	Valid	N.E.	Subtract with borrow *imm8* from *r/m8.*
81 /3 *iw*	SBB *r/m16, imm16*	MI	Valid	Valid	Subtract with borrow *imm16* from *r/m16.*
81 /3 *id*	SBB *r/m32, imm32*	MI	Valid	Valid	Subtract with borrow *imm32* from *r/m32.*
REX.W + 81 /3 *id*	SBB *r/m64, imm32*	MI	Valid	N.E.	Subtract with borrow sign-extended *imm32 to 64-bits* from *r/m64.*
83 /3 *ib*	SBB *r/m16, imm8*	MI	Valid	Valid	Subtract with borrow sign-extended *imm8* from *r/m16.*
83 /3 *ib*	SBB *r/m32, imm8*	MI	Valid	Valid	Subtract with borrow sign-extended *imm8* from *r/m32.*
REX.W + 83 /3 *ib*	SBB *r/m64, imm8*	MI	Valid	N.E.	Subtract with borrow sign-extended *imm8* from *r/m64.*
18 /*r*	SBB *r/m8, r8*	MR	Valid	Valid	Subtract with borrow *r8* from *r/m8.*
REX + 18 /*r*	SBB *r/m8*, r8*	MR	Valid	N.E.	Subtract with borrow *r8* from *r/m8.*
19 /*r*	SBB *r/m16, r16*	MR	Valid	Valid	Subtract with borrow *r16* from *r/m16.*
19 /*r*	SBB *r/m32, r32*	MR	Valid	Valid	Subtract with borrow *r32* from *r/m32.*
REX.W + 19 /*r*	SBB *r/m64, r64*	MR	Valid	N.E.	Subtract with borrow *r64* from *r/m64.*
1A /*r*	SBB *r8, r/m8*	RM	Valid	Valid	Subtract with borrow *r/m8* from *r8.*
REX + 1A /*r*	SBB *r8*, r/m8**	RM	Valid	N.E.	Subtract with borrow *r/m8* from *r8.*
1B /*r*	SBB *r16, r/m16*	RM	Valid	Valid	Subtract with borrow *r/m16* from *r16.*
1B /*r*	SBB *r32, r/m32*	RM	Valid	Valid	Subtract with borrow *r/m32* from *r32.*
REX.W + 1B /*r*	SBB *r64, r/m64*	RM	Valid	N.E.	Subtract with borrow *r/m64* from *r64.*

NOTES:

* In 64-bit mode, r/m8 can not be encoded to access the following byte registers if a REX prefix is used: AH, BH, CH, DH.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
I	AL/AX/EAX/RAX	imm8/16/32	NA	NA
MI	ModRM:r/m (w)	imm8/16/32	NA	NA
MR	ModRM:r/m (w)	ModRM:reg (r)	NA	NA
RM	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

Description

Adds the source operand (second operand) and the carry (CF) flag, and subtracts the result from the destination operand (first operand). The result of the subtraction is stored in the destination operand. The destination operand can be a register or a memory location; the source operand can be an immediate, a register, or a memory location. (However, two memory operands cannot be used in one instruction.) The state of the CF flag represents a borrow from a previous subtraction.

When an immediate value is used as an operand, it is sign-extended to the length of the destination operand format.

The SBB instruction does not distinguish between signed or unsigned operands. Instead, the processor evaluates the result for both data types and sets the OF and CF flags to indicate a borrow in the signed or unsigned result, respectively. The SF flag indicates the sign of the signed result.

The SBB instruction is usually executed as part of a multibyte or multiword subtraction in which a SUB instruction is followed by a SBB instruction.

This instruction can be used with a LOCK prefix to allow the instruction to be executed atomically.

In 64-bit mode, the instruction's default operation size is 32 bits. Using a REX prefix in the form of REX.R permits access to additional registers (R8-R15). Using a REX prefix in the form of REX.W promotes operation to 64 bits. See the summary chart at the beginning of this section for encoding data and limits.

Operation

```
DEST ← (DEST – (SRC + CF));
```

Intel C/C++ Compiler Intrinsic Equivalent

SBB: extern unsigned char _subborrow_u8(unsigned char c_in, unsigned char src1, unsigned char src2, unsigned char *diff_out);

SBB: extern unsigned char _subborrow_u16(unsigned char c_in, unsigned short src1, unsigned short src2, unsigned short *diff_out);

SBB: extern unsigned char _subborrow_u32(unsigned char c_in, unsigned int src1, unsigned char int, unsigned int *diff_out);

SBB: extern unsigned char _subborrow_u64(unsigned char c_in, unsigned __int64 src1, unsigned __int64 src2, unsigned __int64 *diff_out);

Flags Affected

The OF, SF, ZF, AF, PF, and CF flags are set according to the result.

Protected Mode Exceptions

#GP(0)	If the destination is located in a non-writable segment.
	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
	If the DS, ES, FS, or GS register contains a NULL segment selector.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If the LOCK prefix is used but the destination is not a memory operand.

Real-Address Mode Exceptions

#GP	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS	If a memory operand effective address is outside the SS segment limit.
#UD	If the LOCK prefix is used but the destination is not a memory operand.

Virtual-8086 Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.
#UD	If the LOCK prefix is used but the destination is not a memory operand.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If the LOCK prefix is used but the destination is not a memory operand.

SCAS/SCASB/SCASW/SCASD—Scan String

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
AE	SCAS *m8*	ZO	Valid	Valid	Compare AL with byte at ES:(E)DI or RDI, then set status flags.*
AF	SCAS *m16*	ZO	Valid	Valid	Compare AX with word at ES:(E)DI or RDI, then set status flags.*
AF	SCAS *m32*	ZO	Valid	Valid	Compare EAX with doubleword at ES(E)DI or RDI then set status flags.*
REX.W + AF	SCAS *m64*	ZO	Valid	N.E.	Compare RAX with quadword at RDI or EDI then set status flags.
AE	SCASB	ZO	Valid	Valid	Compare AL with byte at ES:(E)DI or RDI then set status flags.*
AF	SCASW	ZO	Valid	Valid	Compare AX with word at ES:(E)DI or RDI then set status flags.*
AF	SCASD	ZO	Valid	Valid	Compare EAX with doubleword at ES:(E)DI or RDI then set status flags.*
REX.W + AF	SCASQ	ZO	Valid	N.E.	Compare RAX with quadword at RDI or EDI then set status flags.

NOTES:

* In 64-bit mode, only 64-bit (RDI) and 32-bit (EDI) address sizes are supported. In non-64-bit mode, only 32-bit (EDI) and 16-bit (DI) address sizes are supported.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
ZO	NA	NA	NA	NA

Description

In non-64-bit modes and in default 64-bit mode: this instruction compares a byte, word, doubleword or quadword specified using a memory operand with the value in AL, AX, or EAX. It then sets status flags in EFLAGS recording the results. The memory operand address is read from ES:(E)DI register (depending on the address-size attribute of the instruction and the current operational mode). Note that ES cannot be overridden with a segment override prefix.

At the assembly-code level, two forms of this instruction are allowed. The explicit-operand form and the no-operands form. The explicit-operand form (specified using the SCAS mnemonic) allows a memory operand to be specified explicitly. The memory operand must be a symbol that indicates the size and location of the operand value. The register operand is then automatically selected to match the size of the memory operand (AL register for byte comparisons, AX for word comparisons, EAX for doubleword comparisons). The explicit-operand form is provided to allow documentation. Note that the documentation provided by this form can be misleading. That is, the memory operand symbol must specify the correct type (size) of the operand (byte, word, or doubleword) but it does not have to specify the correct location. The location is always specified by ES:(E)DI.

The no-operands form of the instruction uses a short form of SCAS. Again, ES:(E)DI is assumed to be the memory operand and AL, AX, or EAX is assumed to be the register operand. The size of operands is selected by the mnemonic: SCASB (byte comparison), SCASW (word comparison), or SCASD (doubleword comparison).

After the comparison, the (E)DI register is incremented or decremented automatically according to the setting of the DF flag in the EFLAGS register. If the DF flag is 0, the (E)DI register is incremented; if the DF flag is 1, the (E)DI register is decremented. The register is incremented or decremented by 1 for byte operations, by 2 for word operations, and by 4 for doubleword operations.

SCAS, SCASB, SCASW, SCASD, and SCASQ can be preceded by the REP prefix for block comparisons of ECX bytes, words, doublewords, or quadwords. Often, however, these instructions will be used in a LOOP construct that takes

some action based on the setting of status flags. See "REP/REPE/REPZ /REPNE/REPNZ—Repeat String Operation Prefix" in this chapter for a description of the REP prefix.

In 64-bit mode, the instruction's default address size is 64-bits, 32-bit address size is supported using the prefix 67H. Using a REX prefix in the form of REX.W promotes operation on doubleword operand to 64 bits. The 64-bit no-operand mnemonic is SCASQ. Address of the memory operand is specified in either RDI or EDI, and AL/AX/EAX/RAX may be used as the register operand. After a comparison, the destination register is incremented or decremented by the current operand size (depending on the value of the DF flag). See the summary chart at the beginning of this section for encoding data and limits.

Operation

Non-64-bit Mode:

```
IF (Byte comparison)
    THEN
        temp ← AL – SRC;
        SetStatusFlags(temp);
            THEN IF DF = 0
                THEN (E)DI ← (E)DI + 1;
                ELSE (E)DI ← (E)DI – 1; FI;
    ELSE IF (Word comparison)
        THEN
            temp ← AX – SRC;
            SetStatusFlags(temp);
            IF DF = 0
                THEN (E)DI ← (E)DI + 2;
                ELSE (E)DI ← (E)DI – 2; FI;
        FI;
    ELSE IF (Doubleword comparison)
        THEN
            temp ← EAX – SRC;
            SetStatusFlags(temp);
            IF DF = 0
                THEN (E)DI ← (E)DI + 4;
                ELSE (E)DI ← (E)DI – 4; FI;
        FI;
FI;
```

64-bit Mode:

```
IF (Byte cmparison)
    THEN
        temp ← AL – SRC;
        SetStatusFlags(temp);
            THEN IF DF = 0
                THEN (R|E)DI ← (R|E)DI + 1;
                ELSE (R|E)DI ← (R|E)DI – 1; FI;
    ELSE IF (Word comparison)
        THEN
            temp ← AX – SRC;
            SetStatusFlags(temp);
            IF DF = 0
                THEN (R|E)DI ← (R|E)DI + 2;
                ELSE (R|E)DI ← (R|E)DI – 2; FI;
        FI;
```

```
    ELSE IF (Doubleword comparison)
        THEN
            temp ← EAX – SRC;
            SetStatusFlags(temp);
            IF DF = 0
                THEN (R|E)DI ← (R|E)DI + 4;
                ELSE (R|E)DI ← (R|E)DI – 4; FI;
        FI;
    ELSE IF (Quadword comparison using REX.W )
        THEN
            temp ← RAX – SRC;
            SetStatusFlags(temp);
            IF DF = 0
                THEN (R|E)DI ← (R|E)DI + 8;
                ELSE (R|E)DI ← (R|E)DI – 8;
            FI;
    FI;
F
```

Flags Affected

The OF, SF, ZF, AF, PF, and CF flags are set according to the temporary result of the comparison.

Protected Mode Exceptions

#GP(0)	If a memory operand effective address is outside the limit of the ES segment.
	If the ES register contains a NULL segment selector.
	If an illegal memory operand effective address in the ES segment is given.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If the LOCK prefix is used.

Real-Address Mode Exceptions

#GP	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS	If a memory operand effective address is outside the SS segment limit.
#UD	If the LOCK prefix is used.

Virtual-8086 Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.
#UD	If the LOCK prefix is used.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#GP(0)	If the memory address is in a non-canonical form.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If the LOCK prefix is used.

SETcc—Set Byte on Condition

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
0F 97	SETA *r/m8*	M	Valid	Valid	Set byte if above (CF=0 and ZF=0).
REX + 0F 97	SETA *r/m8**	M	Valid	N.E.	Set byte if above (CF=0 and ZF=0).
0F 93	SETAE *r/m8*	M	Valid	Valid	Set byte if above or equal (CF=0).
REX + 0F 93	SETAE *r/m8**	M	Valid	N.E.	Set byte if above or equal (CF=0).
0F 92	SETB *r/m8*	M	Valid	Valid	Set byte if below (CF=1).
REX + 0F 92	SETB *r/m8**	M	Valid	N.E.	Set byte if below (CF=1).
0F 96	SETBE *r/m8*	M	Valid	Valid	Set byte if below or equal (CF=1 or ZF=1).
REX + 0F 96	SETBE *r/m8**	M	Valid	N.E.	Set byte if below or equal (CF=1 or ZF=1).
0F 92	SETC *r/m8*	M	Valid	Valid	Set byte if carry (CF=1).
REX + 0F 92	SETC *r/m8**	M	Valid	N.E.	Set byte if carry (CF=1).
0F 94	SETE *r/m8*	M	Valid	Valid	Set byte if equal (ZF=1).
REX + 0F 94	SETE *r/m8**	M	Valid	N.E.	Set byte if equal (ZF=1).
0F 9F	SETG *r/m8*	M	Valid	Valid	Set byte if greater (ZF=0 and SF=OF).
REX + 0F 9F	SETG *r/m8**	M	Valid	N.E.	Set byte if greater (ZF=0 and SF=OF).
0F 9D	SETGE *r/m8*	M	Valid	Valid	Set byte if greater or equal (SF=OF).
REX + 0F 9D	SETGE *r/m8**	M	Valid	N.E.	Set byte if greater or equal (SF=OF).
0F 9C	SETL *r/m8*	M	Valid	Valid	Set byte if less (SF≠ OF).
REX + 0F 9C	SETL *r/m8**	M	Valid	N.E.	Set byte if less (SF≠ OF).
0F 9E	SETLE *r/m8*	M	Valid	Valid	Set byte if less or equal (ZF=1 or SF≠ OF).
REX + 0F 9E	SETLE *r/m8**	M	Valid	N.E.	Set byte if less or equal (ZF=1 or SF≠ OF).
0F 96	SETNA *r/m8*	M	Valid	Valid	Set byte if not above (CF=1 or ZF=1).
REX + 0F 96	SETNA *r/m8**	M	Valid	N.E.	Set byte if not above (CF=1 or ZF=1).
0F 92	SETNAE *r/m8*	M	Valid	Valid	Set byte if not above or equal (CF=1).
REX + 0F 92	SETNAE *r/m8**	M	Valid	N.E.	Set byte if not above or equal (CF=1).
0F 93	SETNB *r/m8*	M	Valid	Valid	Set byte if not below (CF=0).
REX + 0F 93	SETNB *r/m8**	M	Valid	N.E.	Set byte if not below (CF=0).
0F 97	SETNBE *r/m8*	M	Valid	Valid	Set byte if not below or equal (CF=0 and ZF=0).
REX + 0F 97	SETNBE *r/m8**	M	Valid	N.E.	Set byte if not below or equal (CF=0 and ZF=0).
0F 93	SETNC *r/m8*	M	Valid	Valid	Set byte if not carry (CF=0).
REX + 0F 93	SETNC *r/m8**	M	Valid	N.E.	Set byte if not carry (CF=0).
0F 95	SETNE *r/m8*	M	Valid	Valid	Set byte if not equal (ZF=0).
REX + 0F 95	SETNE *r/m8**	M	Valid	N.E.	Set byte if not equal (ZF=0).
0F 9E	SETNG *r/m8*	M	Valid	Valid	Set byte if not greater (ZF=1 or SF≠ OF)
REX + 0F 9E	SETNG *r/m8**	M	Valid	N.E.	Set byte if not greater (ZF=1 or SF≠ OF).
0F 9C	SETNGE *r/m8*	M	Valid	Valid	Set byte if not greater or equal (SF≠ OF).
REX + 0F 9C	SETNGE *r/m8**	M	Valid	N.E.	Set byte if not greater or equal (SF≠ OF).
0F 9D	SETNL *r/m8*	M	Valid	Valid	Set byte if not less (SF=OF).
REX + 0F 9D	SETNL *r/m8**	M	Valid	N.E.	Set byte if not less (SF=OF).
0F 9F	SETNLE *r/m8*	M	Valid	Valid	Set byte if not less or equal (ZF=0 and SF=OF).

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
REX + 0F 9F	SETNLE *r/m8**	M	Valid	N.E.	Set byte if not less or equal (ZF=0 and SF=OF).
0F 91	SETNO *r/m8*	M	Valid	Valid	Set byte if not overflow (OF=0).
REX + 0F 91	SETNO *r/m8**	M	Valid	N.E.	Set byte if not overflow (OF=0).
0F 9B	SETNP *r/m8*	M	Valid	Valid	Set byte if not parity (PF=0).
REX + 0F 9B	SETNP *r/m8**	M	Valid	N.E.	Set byte if not parity (PF=0).
0F 99	SETNS *r/m8*	M	Valid	Valid	Set byte if not sign (SF=0).
REX + 0F 99	SETNS *r/m8**	M	Valid	N.E.	Set byte if not sign (SF=0).
0F 95	SETNZ *r/m8*	M	Valid	Valid	Set byte if not zero (ZF=0).
REX + 0F 95	SETNZ *r/m8**	M	Valid	N.E.	Set byte if not zero (ZF=0).
0F 90	SETO *r/m8*	M	Valid	Valid	Set byte if overflow (OF=1)
REX + 0F 90	SETO *r/m8**	M	Valid	N.E.	Set byte if overflow (OF=1).
0F 9A	SETP *r/m8*	M	Valid	Valid	Set byte if parity (PF=1).
REX + 0F 9A	SETP *r/m8**	M	Valid	N.E.	Set byte if parity (PF=1).
0F 9A	SETPE *r/m8*	M	Valid	Valid	Set byte if parity even (PF=1).
REX + 0F 9A	SETPE *r/m8**	M	Valid	N.E.	Set byte if parity even (PF=1).
0F 9B	SETPO *r/m8*	M	Valid	Valid	Set byte if parity odd (PF=0).
REX + 0F 9B	SETPO *r/m8**	M	Valid	N.E.	Set byte if parity odd (PF=0).
0F 98	SETS *r/m8*	M	Valid	Valid	Set byte if sign (SF=1).
REX + 0F 98	SETS *r/m8**	M	Valid	N.E.	Set byte if sign (SF=1).
0F 94	SETZ *r/m8*	M	Valid	Valid	Set byte if zero (ZF=1).
REX + 0F 94	SETZ *r/m8**	M	Valid	N.E.	Set byte if zero (ZF=1).

NOTES:

* In 64-bit mode, r/m8 can not be encoded to access the following byte registers if a REX prefix is used: AH, BH, CH, DH.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
M	ModRM:r/m (r)	NA	NA	NA

Description

Sets the destination operand to 0 or 1 depending on the settings of the status flags (CF, SF, OF, ZF, and PF) in the EFLAGS register. The destination operand points to a byte register or a byte in memory. The condition code suffix (*cc*) indicates the condition being tested for.

The terms "above" and "below" are associated with the CF flag and refer to the relationship between two unsigned integer values. The terms "greater" and "less" are associated with the SF and OF flags and refer to the relationship between two signed integer values.

Many of the SET*cc* instruction opcodes have alternate mnemonics. For example, SETG (set byte if greater) and SETNLE (set if not less or equal) have the same opcode and test for the same condition: ZF equals 0 and SF equals OF. These alternate mnemonics are provided to make code more intelligible. Appendix B, "EFLAGS Condition Codes," in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*, shows the alternate mnemonics for various test conditions.

Some languages represent a logical one as an integer with all bits set. This representation can be obtained by choosing the logically opposite condition for the SET*cc* instruction, then decrementing the result. For example, to test for overflow, use the SETNO instruction, then decrement the result.

In IA-64 mode, the operand size is fixed at 8 bits. Use of REX prefix enable uniform addressing to additional byte registers. Otherwise, this instruction's operation is the same as in legacy mode and compatibility mode.

Operation

```
IF condition
    THEN DEST ← 1;
    ELSE DEST ← 0;
FI;
```

Flags Affected

None.

Protected Mode Exceptions

#GP(0)	If the destination is located in a non-writable segment.
	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
	If the DS, ES, FS, or GS register contains a NULL segment selector.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#UD	If the LOCK prefix is used.

Real-Address Mode Exceptions

#GP	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS	If a memory operand effective address is outside the SS segment limit.
#UD	If the LOCK prefix is used.

Virtual-8086 Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#UD	If the LOCK prefix is used.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form.
#PF(fault-code)	If a page fault occurs.
#UD	If the LOCK prefix is used.

SFENCE—Store Fence

Opcode*	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
NP 0F AE F8	SFENCE	ZO	Valid	Valid	Serializes store operations.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
ZO	NA	NA	NA	NA

Description

Performs a serializing operation on all store-to-memory instructions that were issued prior the SFENCE instruction. This serializing operation guarantees that every store instruction that precedes the SFENCE instruction in program order becomes globally visible before any store instruction that follows the SFENCE instruction. The SFENCE instruction is ordered with respect to store instructions, other SFENCE instructions, any LFENCE and MFENCE instructions, and any serializing instructions (such as the CPUID instruction). It is not ordered with respect to load instructions.

Weakly ordered memory types can be used to achieve higher processor performance through such techniques as out-of-order issue, write-combining, and write-collapsing. The degree to which a consumer of data recognizes or knows that the data is weakly ordered varies among applications and may be unknown to the producer of this data. The SFENCE instruction provides a performance-efficient way of ensuring store ordering between routines that produce weakly-ordered results and routines that consume this data.

This instruction’s operation is the same in non-64-bit modes and 64-bit mode.

Specification of the instruction's opcode above indicates a ModR/M byte of F8. For this instruction, the processor ignores the r/m field of the ModR/M byte. Thus, SFENCE is encoded by any opcode of the form 0F AE Fx, where x is in the range 8-F.

Operation

```
Wait_On_Following_Stores_Until(preceding_stores_globally_visible);
```

Intel C/C++ Compiler Intrinsic Equivalent

void _mm_sfence(void)

Exceptions (All Operating Modes)

#UD	If CPUID.01H:EDX.SSE[bit 25] = 0.
	If the LOCK prefix is used.

SGDT—Store Global Descriptor Table Register

Opcode*	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
0F 01 /0	SGDT *m*	M	Valid	Valid	Store GDTR to *m*.

NOTES:

* See IA-32 Architecture Compatibility section below.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
M	ModRM:r/m (w)	NA	NA	NA

Description

Stores the content of the global descriptor table register (GDTR) in the destination operand. The destination operand specifies a memory location.

In legacy or compatibility mode, the destination operand is a 6-byte memory location. If the operand-size attribute is 16 or 32 bits, the 16-bit limit field of the register is stored in the low 2 bytes of the memory location and the 32-bit base address is stored in the high 4 bytes.

In 64-bit mode, the operand size is fixed at 8+2 bytes. The instruction stores an 8-byte base and a 2-byte limit.

SGDT is useful only by operating-system software. However, it can be used in application programs without causing an exception to be generated if CR4.UMIP = 0. See "LGDT/LIDT—Load Global/Interrupt Descriptor Table Register" in Chapter 3, *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A*, for information on loading the GDTR and IDTR.

IA-32 Architecture Compatibility

The 16-bit form of the SGDT is compatible with the Intel 286 processor if the upper 8 bits are not referenced. The Intel 286 processor fills these bits with 1s; processor generations later than the Intel 286 processor fill these bits with 0s.

Operation

```
IF instruction is SGDT
        IF OperandSize =16 or OperandSize = 32 (* Legacy or Compatibility Mode *)
            THEN
                DEST[0:15] ← GDTR(Limit);
                DEST[16:47] ← GDTR(Base); (* Full 32-bit base address stored *)
                FI;
            ELSE (* 64-bit Mode *)
                DEST[0:15] ← GDTR(Limit);
                DEST[16:79] ← GDTR(Base); (* Full 64-bit base address stored *)
        FI;
FI;
```

Flags Affected

None.

Protected Mode Exceptions

#UD	If the LOCK prefix is used.
#GP(0)	If the destination is located in a non-writable segment.
	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
	If the DS, ES, FS, or GS register is used to access memory and it contains a NULL segment selector.
	If CR4.UMIP = 1 and CPL > 0.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while CPL = 3.

Real-Address Mode Exceptions

#UD	If the LOCK prefix is used.
#GP	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS	If a memory operand effective address is outside the SS segment limit.

Virtual-8086 Mode Exceptions

#UD	If the LOCK prefix is used.
#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
	If CR4.UMIP = 1.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#UD	If the LOCK prefix is used.
#GP(0)	If the memory address is in a non-canonical form.
	If CR4.UMIP = 1 and CPL > 0.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while CPL = 3.

SHA1RNDS4—Perform Four Rounds of SHA1 Operation

Opcode/ Instruction	Op/En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F 3A CC /r ib SHA1RNDS4 xmm1, xmm2/m128, imm8	RMI	V/V	SHA	Performs four rounds of SHA1 operation operating on SHA1 state (A,B,C,D) from xmm1, with a pre-computed sum of the next 4 round message dwords and state variable E from xmm2/m128. The immediate byte controls logic functions and round constants.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3
RMI	ModRM:reg (r, w)	ModRM:r/m (r)	Imm8

Description

The SHA1RNDS4 instruction performs four rounds of SHA1 operation using an initial SHA1 state (A,B,C,D) from the first operand (which is a source operand and the destination operand) and some pre-computed sum of the next 4 round message dwords, and state variable E from the second operand (a source operand). The updated SHA1 state (A,B,C,D) after four rounds of processing is stored in the destination operand.

Operation

SHA1RNDS4

The function f() and Constant K are dependent on the value of the immediate.

```
IF ( imm8[1:0] = 0 )
    THEN f() ← f0(), K ← K0;
ELSE IF ( imm8[1:0] = 1 )
    THEN f() ← f1(), K ← K1;
ELSE IF ( imm8[1:0] = 2 )
    THEN f() ← f2(), K ← K2;
ELSE IF ( imm8[1:0] = 3 )
    THEN f() ← f3(), K ← K3;
FI;

A ← SRC1[127:96];
B ← SRC1[95:64];
C ← SRC1[63:32];
D ← SRC1[31:0];
W0E ← SRC2[127:96];
W1 ← SRC2[95:64];
W2 ← SRC2[63:32];
W3 ← SRC2[31:0];

Round i = 0 operation:
A_1 ← f (B, C, D) + (A ROL 5) +W0E +K;
B_1 ← A;
C_1 ← B ROL 30;
D_1 ← C;
E_1 ← D;

FOR i = 1 to 3
    A_(i +1) ← f (B_i, C_i, D_i) + (A_i ROL 5) +Wi+ E_i +K;
    B_(i +1) ← A_i;
```

```
    C_(i +1) ← B_i ROL 30;
    D_(i +1) ← C_i;
    E_(i +1) ← D_i;
ENDFOR

DEST[127:96] ← A_4;
DEST[95:64] ← B_4;
DEST[63:32] ← C_4;
DEST[31:0] ← D_4;
```

Intel C/C++ Compiler Intrinsic Equivalent

SHA1RNDS4: __m128i _mm_sha1rnds4_epu32(__m128i, __m128i, const int);

Flags Affected

None

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type 4.

SHA1NEXTE—Calculate SHA1 State Variable E after Four Rounds

Opcode/ Instruction	Op/En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F 38 C8 /r SHA1NEXTE xmm1, xmm2/m128	RM	V/V	SHA	Calculates SHA1 state variable E after four rounds of operation from the current SHA1 state variable A in xmm1. The calculated value of the SHA1 state variable E is added to the scheduled dwords in xmm2/m128, and stored with some of the scheduled dwords in xmm1.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA

Description

The SHA1NEXTE calculates the SHA1 state variable E after four rounds of operation from the current SHA1 state variable A in the destination operand. The calculated value of the SHA1 state variable E is added to the source operand, which contains the scheduled dwords.

Operation

SHA1NEXTE

TMP ← (SRC1[127:96] ROL 30);

DEST[127:96] ← SRC2[127:96] + TMP;
DEST[95:64] ← SRC2[95:64];
DEST[63:32] ← SRC2[63:32];
DEST[31:0] ← SRC2[31:0];

Intel C/C++ Compiler Intrinsic Equivalent

SHA1NEXTE: __m128i _mm_sha1nexte_epu32(__m128i, __m128i);

Flags Affected

None

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type 4.

SHA1MSG1—Perform an Intermediate Calculation for the Next Four SHA1 Message Dwords

Opcode/ Instruction	Op/En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F 38 C9 /r SHA1MSG1 xmm1, xmm2/m128	RM	V/V	SHA	Performs an intermediate calculation for the next four SHA1 message dwords using previous message dwords from xmm1 and xmm2/m128, storing the result in xmm1.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA

Description

The SHA1MSG1 instruction is one of two SHA1 message scheduling instructions. The instruction performs an intermediate calculation for the next four SHA1 message dwords.

Operation

SHA1MSG1

```
W0 ← SRC1[127:96] ;
W1 ← SRC1[95:64] ;
W2 ← SRC1[63: 32] ;
W3 ← SRC1[31: 0] ;
W4 ← SRC2[127:96] ;
W5 ← SRC2[95:64] ;

DEST[127:96] ← W2 XOR W0;
DEST[95:64] ← W3 XOR W1;
DEST[63:32] ← W4 XOR W2;
DEST[31:0] ← W5 XOR W3;
```

Intel C/C++ Compiler Intrinsic Equivalent

SHA1MSG1: __m128i _mm_sha1msg1_epu32(__m128i, __m128i);

Flags Affected

None

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type 4.

SHA1MSG2—Perform a Final Calculation for the Next Four SHA1 Message Dwords

Opcode/ Instruction	Op/En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F 38 CA /r SHA1MSG2 xmm1, xmm2/m128	RM	V/V	SHA	Performs the final calculation for the next four SHA1 message dwords using intermediate results from xmm1 and the previous message dwords from xmm2/m128, storing the result in xmm1.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA

Description

The SHA1MSG2 instruction is one of two SHA1 message scheduling instructions. The instruction performs the final calculation to derive the next four SHA1 message dwords.

Operation

SHA1MSG2

```
W13 ← SRC2[95:64] ;
W14 ← SRC2[63: 32] ;
W15 ← SRC2[31: 0] ;
W16 ← (SRC1[127:96] XOR W13 ) ROL 1;
W17 ← (SRC1[95:64] XOR W14) ROL 1;
W18 ← (SRC1[63: 32] XOR W15) ROL 1;
W19 ← (SRC1[31: 0] XOR W16) ROL 1;

DEST[127:96] ← W16;
DEST[95:64] ← W17;
DEST[63:32] ← W18;
DEST[31:0] ← W19;
```

Intel C/C++ Compiler Intrinsic Equivalent

SHA1MSG2: __m128i _mm_sha1msg2_epu32(__m128i, __m128i);

Flags Affected

None

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type 4.

SHA256RNDS2—Perform Two Rounds of SHA256 Operation

Opcode/ Instruction	Op/En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F 38 CB /r SHA256RNDS2 xmm1, xmm2/m128, <XMM0>	RM0	V/V	SHA	Perform 2 rounds of SHA256 operation using an initial SHA256 state (C,D,G,H) from xmm1, an initial SHA256 state (A,B,E,F) from xmm2/m128, and a pre-computed sum of the next 2 round message dwords and the corresponding round constants from the implicit operand XMM0, storing the updated SHA256 state (A,B,E,F) result in xmm1.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3
RMI	ModRM:reg (r, w)	ModRM:r/m (r)	Implicit XMM0 (r)

Description

The SHA256RNDS2 instruction performs 2 rounds of SHA256 operation using an initial SHA256 state (C,D,G,H) from the first operand, an initial SHA256 state (A,B,E,F) from the second operand, and a pre-computed sum of the next 2 round message dwords and the corresponding round constants from the implicit operand xmm0. Note that only the two lower dwords of XMM0 are used by the instruction.

The updated SHA256 state (A,B,E,F) is written to the first operand, and the second operand can be used as the updated state (C,D,G,H) in later rounds.

Operation

SHA256RNDS2

A_0 ← SRC2[127:96];
B_0 ← SRC2[95:64];
C_0 ← SRC1[127:96];
D_0 ← SRC1[95:64];
E_0 ← SRC2[63:32];
F_0 ← SRC2[31:0];
G_0 ← SRC1[63:32];
H_0 ← SRC1[31:0];
WK_0 ← XMM0[31: 0];
WK_1 ← XMM0[63: 32];

FOR i = 0 to 1
    A_(i +1) ← Ch (E_i, F_i, G_i) +Σ_1(E_i) +WK_i+ H_i + Maj(A_i , B_i, C_i) +Σ_0(A_i);
    B_(i +1) ← A_i;
    C_(i +1) ← B_i ;
    D_(i +1) ← C_i;
    E_(i +1) ← Ch (E_i, F_i, G_i) +Σ_1(E_i) +WK_i+ H_i + D_i;
    F_(i +1) ← E_i ;
    G_(i +1) ← F_i;
    H_(i +1) ← G_i;
ENDFOR

DEST[127:96] ← A_2;
DEST[95:64] ← B_2;
DEST[63:32] ← E_2;
DEST[31:0] ← F_2;

Intel C/C++ Compiler Intrinsic Equivalent

SHA256RNDS2: __m128i _mm_sha256rnds2_epu32(__m128i, __m128i, __m128i);

Flags Affected

None

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type 4.

SHA256MSG1—Perform an Intermediate Calculation for the Next Four SHA256 Message Dwords

Opcode/ Instruction	Op/En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F 38 CC /r SHA256MSG1 xmm1, xmm2/m128	RM	V/V	SHA	Performs an intermediate calculation for the next four SHA256 message dwords using previous message dwords from xmm1 and xmm2/m128, storing the result in xmm1.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA

Description

The SHA256MSG1 instruction is one of two SHA256 message scheduling instructions. The instruction performs an intermediate calculation for the next four SHA256 message dwords.

Operation

SHA256MSG1

W4 ← SRC2[31: 0] ;
W3 ← SRC1[127:96] ;
W2 ← SRC1[95:64] ;
W1 ← SRC1[63: 32] ;
W0 ← SRC1[31: 0] ;

DEST[127:96] ← W3 + σ_0(W4);
DEST[95:64] ← W2 + σ_0(W3);
DEST[63:32] ← W1 + σ_0(W2);
DEST[31:0] ← W0 + σ_0(W1);

Intel C/C++ Compiler Intrinsic Equivalent

SHA256MSG1: __m128i _mm_sha256msg1_epu32(__m128i, __m128i);

Flags Affected

None

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type 4.

SHA256MSG2—Perform a Final Calculation for the Next Four SHA256 Message Dwords

Opcode/ Instruction	Op/En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F 38 CD /r SHA256MSG2 xmm1, xmm2/m128	RM	V/V	SHA	Performs the final calculation for the next four SHA256 message dwords using previous message dwords from xmm1 and xmm2/m128, storing the result in xmm1.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA

Description

The SHA256MSG2 instruction is one of two SHA2 message scheduling instructions. The instruction performs the final calculation for the next four SHA256 message dwords.

Operation

SHA256MSG2

W14 ← SRC2[95:64] ;
W15 ← SRC2[127:96] ;
W16 ← SRC1[31: 0] + σ_1(W14) ;
W17 ← SRC1[63: 32] + σ_1(W15) ;
W18 ← SRC1[95: 64] + σ_1(W16) ;
W19 ← SRC1[127: 96] + σ_1(W17) ;

DEST[127:96] ← W19 ;
DEST[95:64] ← W18 ;
DEST[63:32] ← W17 ;
DEST[31:0] ← W16;

Intel C/C++ Compiler Intrinsic Equivalent

SHA256MSG2 : __m128i _mm_sha256msg2_epu32(__m128i, __m128i);

Flags Affected

None

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type 4.

SHLD—Double Precision Shift Left

Opcode*	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
0F A4 /r ib	SHLD *r/m16, r16, imm8*	MRI	Valid	Valid	Shift *r/m16* to left *imm8* places while shifting bits from *r16* in from the right.
0F A5 /r	SHLD *r/m16, r16*, CL	MRC	Valid	Valid	Shift *r/m16* to left CL places while shifting bits from *r16* in from the right.
0F A4 /r ib	SHLD *r/m32, r32, imm8*	MRI	Valid	Valid	Shift *r/m32* to left *imm8* places while shifting bits from *r32* in from the right.
REX.W + 0F A4 /r ib	SHLD *r/m64, r64, imm8*	MRI	Valid	N.E.	Shift *r/m64* to left *imm8* places while shifting bits from *r64* in from the right.
0F A5 /r	SHLD *r/m32, r32*, CL	MRC	Valid	Valid	Shift *r/m32* to left CL places while shifting bits from *r32* in from the right.
REX.W + 0F A5 /r	SHLD *r/m64, r64*, CL	MRC	Valid	N.E.	Shift *r/m64* to left CL places while shifting bits from *r64* in from the right.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
MRI	ModRM:r/m (w)	ModRM:reg (r)	imm8	NA
MRC	ModRM:r/m (w)	ModRM:reg (r)	CL	NA

Description

The SHLD instruction is used for multi-precision shifts of 64 bits or more.

The instruction shifts the first operand (destination operand) to the left the number of bits specified by the third operand (count operand). The second operand (source operand) provides bits to shift in from the right (starting with bit 0 of the destination operand).

The destination operand can be a register or a memory location; the source operand is a register. The count operand is an unsigned integer that can be stored in an immediate byte or in the CL register. If the count operand is CL, the shift count is the logical AND of CL and a count mask. In non-64-bit modes and default 64-bit mode; only bits 0 through 4 of the count are used. This masks the count to a value between 0 and 31. If a count is greater than the operand size, the result is undefined.

If the count is 1 or greater, the CF flag is filled with the last bit shifted out of the destination operand. For a 1-bit shift, the OF flag is set if a sign change occurred; otherwise, it is cleared. If the count operand is 0, flags are not affected.

In 64-bit mode, the instruction's default operation size is 32 bits. Using a REX prefix in the form of REX.R permits access to additional registers (R8-R15). Using a REX prefix in the form of REX.W promotes operation to 64 bits (upgrading the count mask to 6 bits). See the summary chart at the beginning of this section for encoding data and limits.

Operation

```
IF (In 64-Bit Mode and REX.W = 1)
    THEN COUNT ← COUNT MOD 64;
    ELSE COUNT ← COUNT MOD 32;
FI
SIZE ← OperandSize;
IF COUNT = 0
    THEN
        No operation;
    ELSE
```

```
        IF COUNT > SIZE
            THEN (* Bad parameters *)
                DEST is undefined;
                CF, OF, SF, ZF, AF, PF are undefined;
            ELSE (* Perform the shift *)
                CF ← BIT[DEST, SIZE – COUNT];
                (* Last bit shifted out on exit *)
                FOR i ← SIZE – 1 DOWN TO COUNT
                    DO
                        Bit(DEST, i) ← Bit(DEST, i – COUNT);
                    OD;
                FOR i ← COUNT – 1 DOWN TO 0
                    DO
                        BIT[DEST, i] ← BIT[SRC, i – COUNT + SIZE];
                    OD;
        FI;
FI;
```

Flags Affected

If the count is 1 or greater, the CF flag is filled with the last bit shifted out of the destination operand and the SF, ZF, and PF flags are set according to the value of the result. For a 1-bit shift, the OF flag is set if a sign change occurred; otherwise, it is cleared. For shifts greater than 1 bit, the OF flag is undefined. If a shift occurs, the AF flag is undefined. If the count operand is 0, the flags are not affected. If the count is greater than the operand size, the flags are undefined.

Protected Mode Exceptions

#GP(0)	If the destination is located in a non-writable segment.
	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
	If the DS, ES, FS, or GS register contains a NULL segment selector.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If the LOCK prefix is used.

Real-Address Mode Exceptions

#GP	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS	If a memory operand effective address is outside the SS segment limit.
#UD	If the LOCK prefix is used.

Virtual-8086 Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.
#UD	If the LOCK prefix is used.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If the LOCK prefix is used.

SHRD—Double Precision Shift Right

Opcode*	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
0F AC /r ib	SHRD *r/m16, r16, imm8*	MRI	Valid	Valid	Shift *r/m16* to right *imm8* places while shifting bits from *r16* in from the left.
0F AD /r	SHRD *r/m16, r16*, CL	MRC	Valid	Valid	Shift *r/m16* to right CL places while shifting bits from *r16* in from the left.
0F AC /r ib	SHRD *r/m32, r32, imm8*	MRI	Valid	Valid	Shift *r/m32* to right *imm8* places while shifting bits from *r32* in from the left.
REX.W + 0F AC /r ib	SHRD *r/m64, r64, imm8*	MRI	Valid	N.E.	Shift *r/m64* to right *imm8* places while shifting bits from *r64* in from the left.
0F AD /r	SHRD *r/m32, r32*, CL	MRC	Valid	Valid	Shift *r/m32* to right CL places while shifting bits from *r32* in from the left.
REX.W + 0F AD /r	SHRD *r/m64, r64*, CL	MRC	Valid	N.E.	Shift *r/m64* to right CL places while shifting bits from *r64* in from the left.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
MRI	ModRM:r/m (w)	ModRM:reg (r)	imm8	NA
MRC	ModRM:r/m (w)	ModRM:reg (r)	CL	NA

Description

The SHRD instruction is useful for multi-precision shifts of 64 bits or more.

The instruction shifts the first operand (destination operand) to the right the number of bits specified by the third operand (count operand). The second operand (source operand) provides bits to shift in from the left (starting with the most significant bit of the destination operand).

The destination operand can be a register or a memory location; the source operand is a register. The count operand is an unsigned integer that can be stored in an immediate byte or the CL register. If the count operand is CL, the shift count is the logical AND of CL and a count mask. In non-64-bit modes and default 64-bit mode, the width of the count mask is 5 bits. Only bits 0 through 4 of the count register are used (masking the count to a value between 0 and 31). If the count is greater than the operand size, the result is undefined.

If the count is 1 or greater, the CF flag is filled with the last bit shifted out of the destination operand. For a 1-bit shift, the OF flag is set if a sign change occurred; otherwise, it is cleared. If the count operand is 0, flags are not affected.

In 64-bit mode, the instruction's default operation size is 32 bits. Using a REX prefix in the form of REX.R permits access to additional registers (R8-R15). Using a REX prefix in the form of REX.W promotes operation to 64 bits (upgrading the count mask to 6 bits). See the summary chart at the beginning of this section for encoding data and limits.

Operation

```
IF (In 64-Bit Mode and REX.W = 1)
    THEN COUNT ← COUNT MOD 64;
    ELSE COUNT ← COUNT MOD 32;
FI
SIZE ← OperandSize;
IF COUNT = 0
    THEN
        No operation;
    ELSE
```

```
        IF COUNT > SIZE
            THEN (* Bad parameters *)
                DEST is undefined;
                CF, OF, SF, ZF, AF, PF are undefined;
            ELSE (* Perform the shift *)
                CF ← BIT[DEST, COUNT – 1]; (* Last bit shifted out on exit *)
                FOR i ← 0 TO SIZE – 1 – COUNT
                    DO
                        BIT[DEST, i] ← BIT[DEST, i + COUNT];
                    OD;
                FOR i ← SIZE – COUNT TO SIZE – 1
                    DO
                        BIT[DEST,i] ← BIT[SRC, i + COUNT – SIZE];
                    OD;
        FI;
FI;
```

Flags Affected

If the count is 1 or greater, the CF flag is filled with the last bit shifted out of the destination operand and the SF, ZF, and PF flags are set according to the value of the result. For a 1-bit shift, the OF flag is set if a sign change occurred; otherwise, it is cleared. For shifts greater than 1 bit, the OF flag is undefined. If a shift occurs, the AF flag is undefined. If the count operand is 0, the flags are not affected. If the count is greater than the operand size, the flags are undefined.

Protected Mode Exceptions

#GP(0)	If the destination is located in a non-writable segment.
	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
	If the DS, ES, FS, or GS register contains a NULL segment selector.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If the LOCK prefix is used.

Real-Address Mode Exceptions

#GP	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS	If a memory operand effective address is outside the SS segment limit.
#UD	If the LOCK prefix is used.

Virtual-8086 Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.
#UD	If the LOCK prefix is used.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If the LOCK prefix is used.

SHUFPD—Packed Interleave Shuffle of Pairs of Double-Precision Floating-Point Values

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F C6 /r ib SHUFPD xmm1, xmm2/m128, imm8	A	V/V	SSE2	Shuffle two pairs of double-precision floating-point values from xmm1 and xmm2/m128 using imm8 to select from each pair, interleaved result is stored in xmm1.
VEX.NDS.128.66.0F.WIG C6 /r ib VSHUFPD xmm1, xmm2, xmm3/m128, imm8	B	V/V	AVX	Shuffle two pairs of double-precision floating-point values from xmm2 and xmm3/m128 using imm8 to select from each pair, interleaved result is stored in xmm1.
VEX.NDS.256.66.0F.WIG C6 /r ib VSHUFPD ymm1, ymm2, ymm3/m256, imm8	B	V/V	AVX	Shuffle four pairs of double-precision floating-point values from ymm2 and ymm3/m256 using imm8 to select from each pair, interleaved result is stored in xmm1.
EVEX.NDS.128.66.0F.W1 C6 /r ib VSHUFPD xmm1{k1}{z}, xmm2, xmm3/m128/m64bcst, imm8	C	V/V	AVX512VL AVX512F	Shuffle two paris of double-precision floating-point values from xmm2 and xmm3/m128/m64bcst using imm8 to select from each pair. store interleaved results in xmm1 subject to writemask k1.
EVEX.NDS.256.66.0F.W1 C6 /r ib VSHUFPD ymm1{k1}{z}, ymm2, ymm3/m256/m64bcst, imm8	C	V/V	AVX512VL AVX512F	Shuffle four paris of double-precision floating-point values from ymm2 and ymm3/m256/m64bcst using imm8 to select from each pair. store interleaved results in ymm1 subject to writemask k1.
EVEX.NDS.512.66.0F.W1 C6 /r ib VSHUFPD zmm1{k1}{z}, zmm2, zmm3/m512/m64bcst, imm8	C	V/V	AVX512F	Shuffle eight paris of double-precision floating-point values from zmm2 and zmm3/m512/m64bcst using imm8 to select from each pair. store interleaved results in zmm1 subject to writemask k1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	NA	ModRM:reg (r, w)	ModRM:r/m (r)	Imm8	NA
B	NA	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	Imm8
C	Full Vector	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	Imm8

Description

Selects a double-precision floating-point value of an input pair using a bit control and move to a designated element of the destination operand. The low-to-high order of double-precision element of the destination operand is interleaved between the first source operand and the second source operand at the granularity of input pair of 128 bits. Each bit in the imm8 byte, starting from bit 0, is the select control of the corresponding element of the destination to received the shuffled result of an input pair.

EVEX encoded versions: The first source operand is a ZMM/YMM/XMM register. The second source operand can be a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 64-bit memory location The destination operand is a ZMM/YMM/XMM register updated according to the writemask. The select controls are the lower 8/4/2 bits of the imm8 byte.

VEX.256 encoded version: The first source operand is a YMM register. The second source operand can be a YMM register or a 256-bit memory location. The destination operand is a YMM register. The select controls are the bit 3:0 of the imm8 byte, imm8[7:4] are ignored.

VEX.128 encoded version: The first source operand is a XMM register. The second source operand can be a XMM register or a 128-bit memory location. The destination operand is a XMM register. The upper bits (MAXVL-1:128) of the corresponding ZMM register destination are zeroed. The select controls are the bit 1:0 of the imm8 byte, imm8[7:2) are ignored.

128-bit Legacy SSE version: The second source can be an XMM register or an 128-bit memory location. The destination operand and the first source operand is the same and is an XMM register. The upper bits (MAXVL-1:128) of the corresponding ZMM register destination are unmodified. The select controls are the bit 1:0 of the imm8 byte, imm8[7:2) are ignored.

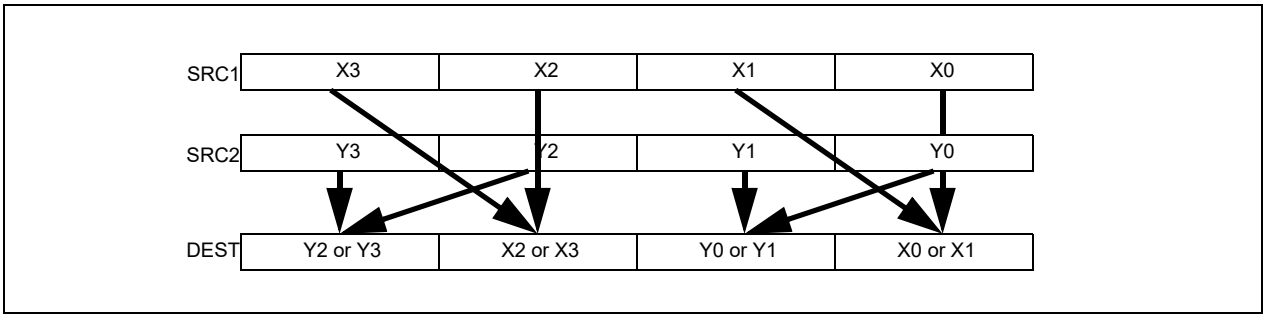


Figure 4-25. 256-bit VSHUFPD Operation of Four Pairs of DP FP Values

Operation

VSHUFPD (EVEX encoded versions when SRC2 is a vector register)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
IF IMM0[0] = 0
    THEN TMP_DEST[63:0] ← SRC1[63:0]
    ELSE TMP_DEST[63:0] ← SRC1[127:64] FI;
IF IMM0[1] = 0
    THEN TMP_DEST[127:64] ← SRC2[63:0]
    ELSE TMP_DEST[127:64] ← SRC2[127:64] FI;
IF VL >= 256
    IF IMM0[2] = 0
        THEN TMP_DEST[191:128] ← SRC1[191:128]
        ELSE TMP_DEST[191:128] ← SRC1[255:192] FI;
    IF IMM0[3] = 0
        THEN TMP_DEST[255:192] ← SRC2[191:128]
        ELSE TMP_DEST[255:192] ← SRC2[255:192] FI;
FI;
IF VL >= 512
    IF IMM0[4] = 0
        THEN TMP_DEST[319:256] ← SRC1[319:256]
        ELSE TMP_DEST[319:256] ← SRC1[383:320] FI;
    IF IMM0[5] = 0
        THEN TMP_DEST[383:320] ← SRC2[319:256]
        ELSE TMP_DEST[383:320] ← SRC2[383:320] FI;
    IF IMM0[6] = 0
        THEN TMP_DEST[447:384] ← SRC1[447:384]
        ELSE TMP_DEST[447:384] ← SRC1[511:448] FI;
    IF IMM0[7] = 0
        THEN TMP_DEST[511:448] ← SRC2[447:384]
        ELSE TMP_DEST[511:448] ← SRC2[511:448] FI;
FI;
FOR j ← 0 TO KL-1
    i ← j * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+63:i] ← TMP_DEST[i+63:i]
        ELSE
```

```
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE *zeroing-masking*              ; zeroing-masking
                    DEST[i+63:i] ← 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0
```

VSHUFPD (EVEX encoded versions when SRC2 is memory)

(KL, VL) = (2, 128), (4, 256), (8, 512)

```
FOR j ← 0 TO KL-1
    i ← j * 64
    IF (EVEX.b = 1)
        THEN TMP_SRC2[i+63:i] ← SRC2[63:0]
        ELSE TMP_SRC2[i+63:i] ← SRC2[i+63:i]
    FI;
ENDFOR;
IF IMM0[0] = 0
    THEN TMP_DEST[63:0] ← SRC1[63:0]
    ELSE TMP_DEST[63:0] ← SRC1[127:64] FI;
IF IMM0[1] = 0
    THEN TMP_DEST[127:64] ← TMP_SRC2[63:0]
    ELSE TMP_DEST[127:64] ← TMP_SRC2[127:64] FI;
IF VL >= 256
    IF IMM0[2] = 0
        THEN TMP_DEST[191:128] ← SRC1[191:128]
        ELSE TMP_DEST[191:128] ← SRC1[255:192] FI;
    IF IMM0[3] = 0
        THEN TMP_DEST[255:192] ← TMP_SRC2[191:128]
        ELSE TMP_DEST[255:192] ← TMP_SRC2[255:192] FI;
FI;
IF VL >= 512
    IF IMM0[4] = 0
        THEN TMP_DEST[319:256] ← SRC1[319:256]
        ELSE TMP_DEST[319:256] ← SRC1[383:320] FI;
    IF IMM0[5] = 0
        THEN TMP_DEST[383:320] ← TMP_SRC2[319:256]
        ELSE TMP_DEST[383:320] ← TMP_SRC2[383:320] FI;
    IF IMM0[6] = 0
        THEN TMP_DEST[447:384] ← SRC1[447:384]
        ELSE TMP_DEST[447:384] ← SRC1[511:448] FI;
    IF IMM0[7] = 0
        THEN TMP_DEST[511:448] ← TMP_SRC2[447:384]
        ELSE TMP_DEST[511:448] ← TMP_SRC2[511:448] FI;
FI;
FOR j ← 0 TO KL-1
    i ← j * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+63:i] ← TMP_DEST[i+63:i]
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
```

```
                ELSE *zeroing-masking*              ; zeroing-masking
                    DEST[i+63:i] ← 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0
```

VSHUFPD (VEX.256 encoded version)

```
IF IMM0[0] = 0
    THEN DEST[63:0] ←SRC1[63:0]
    ELSE DEST[63:0] ←SRC1[127:64] FI;
IF IMM0[1] = 0
    THEN DEST[127:64] ←SRC2[63:0]
    ELSE DEST[127:64] ←SRC2[127:64] FI;
IF IMM0[2] = 0
    THEN DEST[191:128] ←SRC1[191:128]
    ELSE DEST[191:128] ←SRC1[255:192] FI;
IF IMM0[3] = 0
    THEN DEST[255:192] ←SRC2[191:128]
    ELSE DEST[255:192] ←SRC2[255:192] FI;
DEST[MAXVL-1:256] (Unmodified)
```

VSHUFPD (VEX.128 encoded version)

```
IF IMM0[0] = 0
    THEN DEST[63:0] ←SRC1[63:0]
    ELSE DEST[63:0] ←SRC1[127:64] FI;
IF IMM0[1] = 0
    THEN DEST[127:64] ←SRC2[63:0]
    ELSE DEST[127:64] ←SRC2[127:64] FI;
DEST[MAXVL-1:128] ←0
```

VSHUFPD (128-bit Legacy SSE version)

```
IF IMM0[0] = 0
    THEN DEST[63:0] ←SRC1[63:0]
    ELSE DEST[63:0] ←SRC1[127:64] FI;
IF IMM0[1] = 0
    THEN DEST[127:64] ←SRC2[63:0]
    ELSE DEST[127:64] ←SRC2[127:64] FI;
DEST[MAXVL-1:128] (Unmodified)
```

Intel C/C++ Compiler Intrinsic Equivalent

VSHUFPD __m512d _mm512_shuffle_pd(__m512d a, __m512d b, int imm);
VSHUFPD __m512d _mm512_mask_shuffle_pd(__m512d s, __mmask8 k, __m512d a, __m512d b, int imm);
VSHUFPD __m512d _mm512_maskz_shuffle_pd(__mmask8 k, __m512d a, __m512d b, int imm);
VSHUFPD __m256d _mm256_shuffle_pd (__m256d a, __m256d b, const int select);
VSHUFPD __m256d _mm256_mask_shuffle_pd(__m256d s, __mmask8 k, __m256d a, __m256d b, int imm);
VSHUFPD __m256d _mm256_maskz_shuffle_pd(__mmask8 k, __m256d a, __m256d b, int imm);
SHUFPD __m128d _mm_shuffle_pd (__m128d a, __m128d b, const int select);
VSHUFPD __m128d _mm_mask_shuffle_pd(__m128d s, __mmask8 k, __m128d a, __m128d b, int imm);
VSHUFPD __m128d _mm_maskz_shuffle_pd(__mmask8 k, __m128d a, __m128d b, int imm);

SIMD Floating-Point Exceptions

None

Other Exceptions

Non-EVEX-encoded instruction, see Exceptions Type 4.
EVEX-encoded instruction, see Exceptions Type E4NF.

SHUFPS—Packed Interleave Shuffle of Quadruplets of Single-Precision Floating-Point Values

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F C6 /r ib SHUFPS xmm1, xmm3/m128, imm8	A	V/V	SSE	Select from quadruplet of single-precision floating-point values in xmm1 and xmm2/m128 using imm8, interleaved result pairs are stored in xmm1.
VEX.NDS.128.0F.WIG C6 /r ib VSHUFPS xmm1, xmm2, xmm3/m128, imm8	B	V/V	AVX	Select from quadruplet of single-precision floating-point values in xmm1 and xmm2/m128 using imm8, interleaved result pairs are stored in xmm1.
VEX.NDS.256.0F.WIG C6 /r ib VSHUFPS ymm1, ymm2, ymm3/m256, imm8	B	V/V	AVX	Select from quadruplet of single-precision floating-point values in ymm2 and ymm3/m256 using imm8, interleaved result pairs are stored in ymm1.
EVEX.NDS.128.0F.W0 C6 /r ib VSHUFPS xmm1{k1}{z}, xmm2, xmm3/m128/m32bcst, imm8	C	V/V	AVX512VL AVX512F	Select from quadruplet of single-precision floating-point values in xmm1 and xmm2/m128 using imm8, interleaved result pairs are stored in xmm1, subject to writemask k1.
EVEX.NDS.256.0F.W0 C6 /r ib VSHUFPS ymm1{k1}{z}, ymm2, ymm3/m256/m32bcst, imm8	C	V/V	AVX512VL AVX512F	Select from quadruplet of single-precision floating-point values in ymm2 and ymm3/m256 using imm8, interleaved result pairs are stored in ymm1, subject to writemask k1.
EVEX.NDS.512.0F.W0 C6 /r ib VSHUFPS zmm1{k1}{z}, zmm2, zmm3/m512/m32bcst, imm8	C	V/V	AVX512F	Select from quadruplet of single-precision floating-point values in zmm2 and zmm3/m512 using imm8, interleaved result pairs are stored in zmm1, subject to writemask k1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	NA	ModRM:reg (r, w)	ModRM:r/m (r)	Imm8	NA
B	NA	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	Imm8
C	Full Vector	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	Imm8

Description

Selects a single-precision floating-point value of an input quadruplet using a two-bit control and move to a designated element of the destination operand. Each 64-bit element-pair of a 128-bit lane of the destination operand is interleaved between the corresponding lane of the first source operand and the second source operand at the granularity 128 bits. Each two bits in the imm8 byte, starting from bit 0, is the select control of the corresponding element of a 128-bit lane of the destination to received the shuffled result of an input quadruplet. The two lower elements of a 128-bit lane in the destination receives shuffle results from the quadruple of the first source operand. The next two elements of the destination receives shuffle results from the quadruple of the second source operand.

EVEX encoded versions: The first source operand is a ZMM/YMM/XMM register. The second source operand can be a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 32-bit memory location. The destination operand is a ZMM/YMM/XMM register updated according to the writemask. Imm8[7:0] provides 4 select controls for each applicable 128-bit lane of the destination.

VEX.256 encoded version: The first source operand is a YMM register. The second source operand can be a YMM register or a 256-bit memory location. The destination operand is a YMM register. Imm8[7:0] provides 4 select controls for the high and low 128-bit of the destination.

VEX.128 encoded version: The first source operand is a XMM register. The second source operand can be a XMM register or a 128-bit memory location. The destination operand is a XMM register. The upper bits (MAXVL-1:128) of the corresponding ZMM register destination are zeroed. Imm8[7:0] provides 4 select controls for each element of the destination.

128-bit Legacy SSE version: The source can be an XMM register or an 128-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (MAXVL-1:128) of the corresponding ZMM register destination are unmodified. Imm8[7:0] provides 4 select controls for each element of the destination.

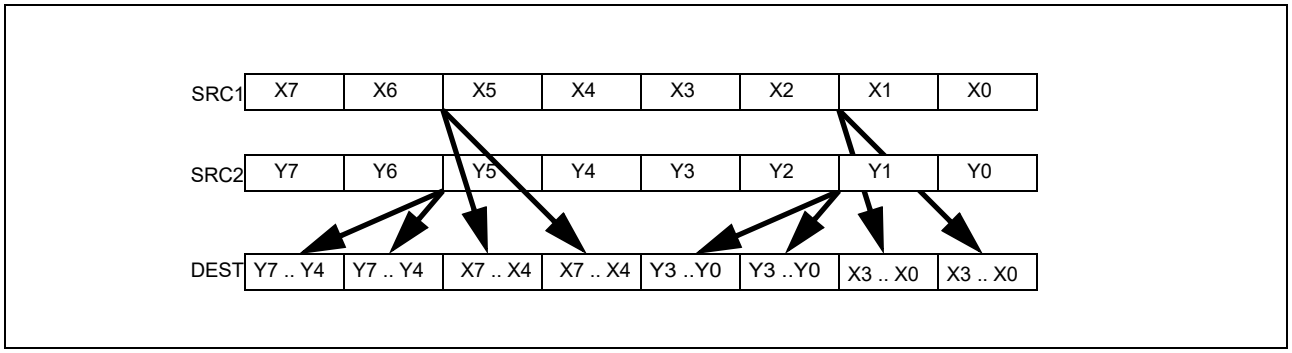


Figure 4-26. 256-bit VSHUFPS Operation of Selection from Input Quadruplet and Pair-wise Interleaved Result

Operation

```
Select4(SRC, control) {
CASE (control[1:0]) OF
    0:    TMP ←SRC[31:0];
    1:    TMP ←SRC[63:32];
    2:    TMP ←SRC[95:64];
    3:    TMP ←SRC[127:96];
ESAC;
RETURN TMP
}
```

VPSHUFPS (EVEX encoded versions when SRC2 is a vector register)
(KL, VL) = (4, 128), (8, 256), (16, 512)

```
TMP_DEST[31:0] ← Select4(SRC1[127:0], imm8[1:0]);
TMP_DEST[63:32] ← Select4(SRC1[127:0], imm8[3:2]);
TMP_DEST[95:64] ← Select4(SRC2[127:0], imm8[5:4]);
TMP_DEST[127:96] ← Select4(SRC2[127:0], imm8[7:6]);
IF VL >= 256
    TMP_DEST[159:128] ← Select4(SRC1[255:128], imm8[1:0]);
    TMP_DEST[191:160] ← Select4(SRC1[255:128], imm8[3:2]);
    TMP_DEST[223:192] ← Select4(SRC2[255:128], imm8[5:4]);
    TMP_DEST[255:224] ← Select4(SRC2[255:128], imm8[7:6]);
FI;
IF VL >= 512
    TMP_DEST[287:256] ← Select4(SRC1[383:256], imm8[1:0]);
    TMP_DEST[319:288] ← Select4(SRC1[383:256], imm8[3:2]);
    TMP_DEST[351:320] ← Select4(SRC2[383:256], imm8[5:4]);
    TMP_DEST[383:352] ← Select4(SRC2[383:256], imm8[7:6]);
    TMP_DEST[415:384] ← Select4(SRC1[511:384], imm8[1:0]);
    TMP_DEST[447:416] ← Select4(SRC1[511:384], imm8[3:2]);
    TMP_DEST[479:448] ←Select4(SRC2[511:384], imm8[5:4]);
    TMP_DEST[511:480] ← Select4(SRC2[511:384], imm8[7:6]);
FI;
FOR j ← 0 TO KL-1
```

```
    i ← j * 32
    IF k1[j] OR *no writemask*
        THEN DEST[i+31:i] ← TMP_DEST[i+31:i]
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE *zeroing-masking*              ; zeroing-masking
                    DEST[i+31:i] ← 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0
```

VPSHUFPS (EVEX encoded versions when SRC2 is memory)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j ← 0 TO KL-1
    i ← j * 32
    IF (EVEX.b = 1)
        THEN TMP_SRC2[i+31:i] ← SRC2[31:0]
        ELSE TMP_SRC2[i+31:i] ← SRC2[i+31:i]
    FI;
ENDFOR;
TMP_DEST[31:0] ← Select4(SRC1[127:0], imm8[1:0]);
TMP_DEST[63:32] ← Select4(SRC1[127:0], imm8[3:2]);
TMP_DEST[95:64] ← Select4(TMP_SRC2[127:0], imm8[5:4]);
TMP_DEST[127:96] ← Select4(TMP_SRC2[127:0], imm8[7:6]);
IF VL >= 256
    TMP_DEST[159:128] ← Select4(SRC1[255:128], imm8[1:0]);
    TMP_DEST[191:160] ← Select4(SRC1[255:128], imm8[3:2]);
    TMP_DEST[223:192] ← Select4(TMP_SRC2[255:128], imm8[5:4]);
    TMP_DEST[255:224] ← Select4(TMP_SRC2[255:128], imm8[7:6]);
FI;
IF VL >= 512
    TMP_DEST[287:256] ← Select4(SRC1[383:256], imm8[1:0]);
    TMP_DEST[319:288] ← Select4(SRC1[383:256], imm8[3:2]);
    TMP_DEST[351:320] ← Select4(TMP_SRC2[383:256], imm8[5:4]);
    TMP_DEST[383:352] ← Select4(TMP_SRC2[383:256], imm8[7:6]);
    TMP_DEST[415:384] ← Select4(SRC1[511:384], imm8[1:0]);
    TMP_DEST[447:416] ← Select4(SRC1[511:384], imm8[3:2]);
    TMP_DEST[479:448] ←Select4(TMP_SRC2[511:384], imm8[5:4]);
    TMP_DEST[511:480] ← Select4(TMP_SRC2[511:384], imm8[7:6]);
FI;
FOR j ← 0 TO KL-1
    i ← j * 32
    IF k1[j] OR *no writemask*
        THEN DEST[i+31:i] ← TMP_DEST[i+31:i]
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE *zeroing-masking*              ; zeroing-masking
                    DEST[i+31:i] ← 0
            FI
    FI;
ENDFOR
```

DEST[MAXVL-1:VL] ← 0

VSHUFPS (VEX.256 encoded version)

DEST[31:0] ←Select4(SRC1[127:0], imm8[1:0]);
DEST[63:32] ←Select4(SRC1[127:0], imm8[3:2]);
DEST[95:64] ←Select4(SRC2[127:0], imm8[5:4]);
DEST[127:96] ←Select4(SRC2[127:0], imm8[7:6]);
DEST[159:128] ←Select4(SRC1[255:128], imm8[1:0]);
DEST[191:160] ←Select4(SRC1[255:128], imm8[3:2]);
DEST[223:192] ←Select4(SRC2[255:128], imm8[5:4]);
DEST[255:224] ←Select4(SRC2[255:128], imm8[7:6]);
DEST[MAXVL-1:256] ←0

VSHUFPS (VEX.128 encoded version)

DEST[31:0] ←Select4(SRC1[127:0], imm8[1:0]);
DEST[63:32] ←Select4(SRC1[127:0], imm8[3:2]);
DEST[95:64] ←Select4(SRC2[127:0], imm8[5:4]);
DEST[127:96] ←Select4(SRC2[127:0], imm8[7:6]);
DEST[MAXVL-1:128] ←0

SHUFPS (128-bit Legacy SSE version)

DEST[31:0] ←Select4(SRC1[127:0], imm8[1:0]);
DEST[63:32] ←Select4(SRC1[127:0], imm8[3:2]);
DEST[95:64] ←Select4(SRC2[127:0], imm8[5:4]);
DEST[127:96] ←Select4(SRC2[127:0], imm8[7:6]);
DEST[MAXVL-1:128] (Unmodified)

Intel C/C++ Compiler Intrinsic Equivalent

VSHUFPS __m512 _mm512_shuffle_ps(__m512 a, __m512 b, int imm);
VSHUFPS __m512 _mm512_mask_shuffle_ps(__m512 s, __mmask16 k, __m512 a, __m512 b, int imm);
VSHUFPS __m512 _mm512_maskz_shuffle_ps(__mmask16 k, __m512 a, __m512 b, int imm);
VSHUFPS __m256 _mm256_shuffle_ps (__m256 a, __m256 b, const int select);
VSHUFPS __m256 _mm256_mask_shuffle_ps(__m256 s, __mmask8 k, __m256 a, __m256 b, int imm);
VSHUFPS __m256 _mm256_maskz_shuffle_ps(__mmask8 k, __m256 a, __m256 b, int imm);
SHUFPS __m128 _mm_shuffle_ps (__m128 a, __m128 b, const int select);
VSHUFPS __m128 _mm_mask_shuffle_ps(__m128 s, __mmask8 k, __m128 a, __m128 b, int imm);
VSHUFPS __m128 _mm_maskz_shuffle_ps(__mmask8 k, __m128 a, __m128 b, int imm);

SIMD Floating-Point Exceptions

None

Other Exceptions

Non-EVEX-encoded instruction, see Exceptions Type 4.
EVEX-encoded instruction, see Exceptions Type E4NF.

SIDT—Store Interrupt Descriptor Table Register

Opcode*	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
0F 01 /1	SIDT *m*	M	Valid	Valid	Store IDTR to *m.*

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
M	ModRM:r/m (w)	NA	NA	NA

Description

Stores the content the interrupt descriptor table register (IDTR) in the destination operand. The destination operand specifies a 6-byte memory location.

In non-64-bit modes, the 16-bit limit field of the register is stored in the low 2 bytes of the memory location and the 32-bit base address is stored in the high 4 bytes.

In 64-bit mode, the operand size fixed at 8+2 bytes. The instruction stores 8-byte base and 2-byte limit values.

SIDT is only useful in operating-system software; however, it can be used in application programs without causing an exception to be generated if CR4.UMIP = 0. See "LGDT/LIDT—Load Global/Interrupt Descriptor Table Register" in Chapter 3, *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A*, for information on loading the GDTR and IDTR.

IA-32 Architecture Compatibility

The 16-bit form of SIDT is compatible with the Intel 286 processor if the upper 8 bits are not referenced. The Intel 286 processor fills these bits with 1s; processor generations later than the Intel 286 processor fill these bits with 0s.

Operation

```
IF instruction is SIDT
    THEN
        IF OperandSize =16 or OperandSize = 32 (* Legacy or Compatibility Mode *)
            THEN
                DEST[0:15] ← IDTR(Limit);
                DEST[16:47] ← IDTR(Base); FI; (* Full 32-bit base address stored *)
            ELSE (* 64-bit Mode *)
                DEST[0:15] ← IDTR(Limit);
                DEST[16:79] ← IDTR(Base); (* Full 64-bit base address stored *)
        FI;
FI;
```

Flags Affected

None.

Protected Mode Exceptions

#GP(0)	If the destination is located in a non-writable segment.
	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
	If the DS, ES, FS, or GS register is used to access memory and it contains a NULL segment selector.
	If CR4.UMIP = 1 and CPL > 0.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while CPL = 3.
#UD	If the LOCK prefix is used.

Real-Address Mode Exceptions

#GP	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS	If a memory operand effective address is outside the SS segment limit.
#UD	If the LOCK prefix is used.

Virtual-8086 Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
	If CR4.UMIP = 1.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.
#UD	If the LOCK prefix is used.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#UD	If the LOCK prefix is used.
#GP(0)	If the memory address is in a non-canonical form.
	If CR4.UMIP = 1 and CPL > 0.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while CPL = 3.

SLDT—Store Local Descriptor Table Register

Opcode*	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
0F 00 /0	SLDT *r/m16*	M	Valid	Valid	Stores segment selector from LDTR in *r/m16*.
REX.W + 0F 00 /0	SLDT *r64/m16*	M	Valid	Valid	Stores segment selector from LDTR in *r64/m16*.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
M	ModRM:r/m (w)	NA	NA	NA

Description

Stores the segment selector from the local descriptor table register (LDTR) in the destination operand. The destination operand can be a general-purpose register or a memory location. The segment selector stored with this instruction points to the segment descriptor (located in the GDT) for the current LDT. This instruction can only be executed in protected mode.

Outside IA-32e mode, when the destination operand is a 32-bit register, the 16-bit segment selector is copied into the low-order 16 bits of the register. The high-order 16 bits of the register are cleared for the Pentium 4, Intel Xeon, and P6 family processors. They are undefined for Pentium, Intel486, and Intel386 processors. When the destination operand is a memory location, the segment selector is written to memory as a 16-bit quantity, regardless of the operand size.

In compatibility mode, when the destination operand is a 32-bit register, the 16-bit segment selector is copied into the low-order 16 bits of the register. The high-order 16 bits of the register are cleared. When the destination operand is a memory location, the segment selector is written to memory as a 16-bit quantity, regardless of the operand size.

In 64-bit mode, using a REX prefix in the form of REX.R permits access to additional registers (R8-R15). The behavior of SLDT with a 64-bit register is to zero-extend the 16-bit selector and store it in the register. If the destination is memory and operand size is 64, SLDT will write the 16-bit selector to memory as a 16-bit quantity, regardless of the operand size.

Operation

```
DEST ← LDTR(SegmentSelector);
```

Flags Affected

None.

Protected Mode Exceptions

#GP(0)	If the destination is located in a non-writable segment.
	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
	If the DS, ES, FS, or GS register is used to access memory and it contains a NULL segment selector.
	If CR4.UMIP = 1 and CPL > 0.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while CPL = 3.
#UD	If the LOCK prefix is used.

Real-Address Mode Exceptions

#UD	The SLDT instruction is not recognized in real-address mode.

Virtual-8086 Mode Exceptions

#UD	The SLDT instruction is not recognized in virtual-8086 mode.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form. If CR4.UMIP = 1 and CPL > 0.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while CPL = 3.
#UD	If the LOCK prefix is used.

SMSW—Store Machine Status Word

Opcode*	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
0F 01 /4	SMSW *r/m16*	M	Valid	Valid	Store machine status word to *r/m16*.
0F 01 /4	SMSW *r32/m16*	M	Valid	Valid	Store machine status word in low-order 16 bits of *r32/m16*; high-order 16 bits of *r32* are undefined.
REX.W + 0F 01 /4	SMSW *r64/m16*	M	Valid	Valid	Store machine status word in low-order 16 bits of *r64/m16*; high-order 16 bits of *r32* are undefined.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
M	ModRM:r/m (w)	NA	NA	NA

Description

Stores the machine status word (bits 0 through 15 of control register CR0) into the destination operand. The destination operand can be a general-purpose register or a memory location.

In non-64-bit modes, when the destination operand is a 32-bit register, the low-order 16 bits of register CR0 are copied into the low-order 16 bits of the register and the high-order 16 bits are undefined. When the destination operand is a memory location, the low-order 16 bits of register CR0 are written to memory as a 16-bit quantity, regardless of the operand size.

In 64-bit mode, the behavior of the SMSW instruction is defined by the following examples:

- SMSW r16 operand size 16, store CR0[15:0] in r16
- SMSW r32 operand size 32, zero-extend CR0[31:0], and store in r32
- SMSW r64 operand size 64, zero-extend CR0[63:0], and store in r64
- SMSW m16 operand size 16, store CR0[15:0] in m16
- SMSW m16 operand size 32, store CR0[15:0] in m16 (not m32)
- SMSW m16 operands size 64, store CR0[15:0] in m16 (not m64)

SMSW is only useful in operating-system software. However, it is not a privileged instruction and can be used in application programs if CR4.UMIP = 0. It is provided for compatibility with the Intel 286 processor. Programs and procedures intended to run on IA-32 and Intel 64 processors beginning with the Intel386 processors should use the MOV CR instruction to load the machine status word.

See "Changes to Instruction Behavior in VMX Non-Root Operation" in Chapter 25 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3C*, for more information about the behavior of this instruction in VMX non-root operation.

Operation

```
DEST ← CR0[15:0];
(* Machine status word *)
```

Flags Affected

None.

Protected Mode Exceptions

#GP(0)	If the destination is located in a non-writable segment.
	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
	If the DS, ES, FS, or GS register is used to access memory and it contains a NULL segment selector.
	If CR4.UMIP = 1 and CPL > 0.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while CPL = 3.
#UD	If the LOCK prefix is used.

Real-Address Mode Exceptions

#GP	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#UD	If the LOCK prefix is used.

Virtual-8086 Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
	If CR4.UMIP = 1.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.
#UD	If the LOCK prefix is used.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form.
	If CR4.UMIP = 1 and CPL > 0.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while CPL = 3.
#UD	If the LOCK prefix is used.

SQRTPD—Square Root of Double-Precision Floating-Point Values

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 51 /r SQRTPD xmm1, xmm2/m128	A	V/V	SSE2	Computes Square Roots of the packed double-precision floating-point values in xmm2/m128 and stores the result in xmm1.
VEX.128.66.0F.WIG 51 /r VSQRTPD xmm1, xmm2/m128	A	V/V	AVX	Computes Square Roots of the packed double-precision floating-point values in xmm2/m128 and stores the result in xmm1.
VEX.256.66.0F.WIG 51 /r VSQRTPD ymm1, ymm2/m256	A	V/V	AVX	Computes Square Roots of the packed double-precision floating-point values in ymm2/m256 and stores the result in ymm1.
EVEX.128.66.0F.W1 51 /r VSQRTPD xmm1 {k1}{z}, xmm2/m128/m32bcst	B	V/V	AVX512VL AVX512F	Computes Square Roots of the packed double-precision floating-point values in xmm2/m128/m64bcst and stores the result in xmm1 subject to writemask k1.
EVEX.256.66.0F.W1 51 /r VSQRTPD ymm1 {k1}{z}, ymm2/m256/m32bcst	B	V/V	AVX512VL AVX512F	Computes Square Roots of the packed double-precision floating-point values in ymm2/m256/m64bcst and stores the result in ymm1 subject to writemask k1.
EVEX.512.66.0F.W1 51 /r VSQRTPD zmm1 {k1}{z}, zmm2/m512/m64bcst{er}	B	V/V	AVX512F	Computes Square Roots of the packed double-precision floating-point values in zmm2/m512/m64bcst and stores the result in zmm1 subject to writemask k1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	NA	ModRM:reg (w)	ModRM:r/m (r)	NA	NA
B	Full Vector	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

Description

Performs a SIMD computation of the square roots of the two, four or eight packed double-precision floating-point values in the source operand (the second operand) stores the packed double-precision floating-point results in the destination operand (the first operand).

EVEX encoded versions: The source operand is a ZMM/YMM/XMM register, a 512/256/128-bit memory location, or a 512/256/128-bit vector broadcasted from a 64-bit memory location. The destination operand is a ZMM/YMM/XMM register updated according to the writemask.

VEX.256 encoded version: The source operand is a YMM register or a 256-bit memory location. The destination operand is a YMM register. The upper bits (MAXVL-1:256) of the corresponding ZMM register destination are zeroed.

VEX.128 encoded version: the source operand second source operand or a 128-bit memory location. The destination operand is an XMM register. The upper bits (MAXVL-1:128) of the corresponding ZMM register destination are zeroed.

128-bit Legacy SSE version: The second source can be an XMM register or 128-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (MAXVL-1:128) of the corresponding ZMM register destination are unmodified.

Note: VEX.vvvv and EVEX.vvvv are reserved and must be 1111b otherwise instructions will #UD.

Operation

VSQRTPD (EVEX encoded versions)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
IF (VL = 512) AND (EVEX.b = 1) AND (SRC *is register*)
    THEN
        SET_RM(EVEX.RC);
    ELSE
        SET_RM(MXCSR.RM);
FI;
FOR j ← 0 TO KL-1
    i ← j * 64
    IF k1[j] OR *no writemask* THEN
            IF (EVEX.b = 1) AND (SRC *is memory*)
                THEN DEST[i+63:i] ← SQRT(SRC[63:0])
                ELSE DEST[i+63:i] ← SQRT(SRC[i+63:i])
            FI;
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE                                ; zeroing-masking
                    DEST[i+63:i] ← 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0
```

VSQRTPD (VEX.256 encoded version)

```
DEST[63:0] ←SQRT(SRC[63:0])
DEST[127:64] ←SQRT(SRC[127:64])
DEST[191:128] ←SQRT(SRC[191:128])
DEST[255:192] ←SQRT(SRC[255:192])
DEST[MAXVL-1:256] ← 0
```

.

VSQRTPD (VEX.128 encoded version)

```
DEST[63:0] ←SQRT(SRC[63:0])
DEST[127:64] ←SQRT(SRC[127:64])
DEST[MAXVL-1:128] ←0
```

SQRTPD (128-bit Legacy SSE version)

```
DEST[63:0] ←SQRT(SRC[63:0])
DEST[127:64] ←SQRT(SRC[127:64])
DEST[MAXVL-1:128] (Unmodified)
```

Intel C/C++ Compiler Intrinsic Equivalent

VSQRTPD __m512d _mm512_sqrt_round_pd(__m512d a, int r);
VSQRTPD __m512d _mm512_mask_sqrt_round_pd(__m512d s, __mmask8 k, __m512d a, int r);
VSQRTPD __m512d _mm512_maskz_sqrt_round_pd(__mmask8 k, __m512d a, int r);
VSQRTPD __m256d _mm256_sqrt_pd (__m256d a);
VSQRTPD __m256d _mm256_mask_sqrt_pd(__m256d s, __mmask8 k, __m256d a, int r);
VSQRTPD __m256d _mm256_maskz_sqrt_pd(__mmask8 k, __m256d a, int r);
SQRTPD __m128d _mm_sqrt_pd (__m128d a);
VSQRTPD __m128d _mm_mask_sqrt_pd(__m128d s, __mmask8 k, __m128d a, int r);
VSQRTPD __m128d _mm_maskz_sqrt_pd(__mmask8 k, __m128d a, int r);

SIMD Floating-Point Exceptions

Invalid, Precision, Denormal

Other Exceptions

Non-EVEX-encoded instruction, see Exceptions Type 2; additionally

#UD	If VEX.vvvv != 1111B.

EVEX-encoded instruction, see Exceptions Type E2.

#UD	If EVEX.vvvv != 1111B.

SQRTPS—Square Root of Single-Precision Floating-Point Values

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F 51 /r SQRTPS xmm1, xmm2/m128	A	V/V	SSE	Computes Square Roots of the packed single-precision floating-point values in xmm2/m128 and stores the result in xmm1.
VEX.128.0F.WIG 51 /r VSQRTPS xmm1, xmm2/m128	A	V/V	AVX	Computes Square Roots of the packed single-precision floating-point values in xmm2/m128 and stores the result in xmm1.
VEX.256.0F.WIG 51/r VSQRTPS ymm1, ymm2/m256	A	V/V	AVX	Computes Square Roots of the packed single-precision floating-point values in ymm2/m256 and stores the result in ymm1.
EVEX.128.0F.W0 51 /r VSQRTPS xmm1 {k1}{z}, xmm2/m128/m32bcst	B	V/V	AVX512VL AVX512F	Computes Square Roots of the packed single-precision floating-point values in xmm2/m128/m32bcst and stores the result in xmm1 subject to writemask k1.
EVEX.256.0F.W0 51 /r VSQRTPS ymm1 {k1}{z}, ymm2/m256/m32bcst	B	V/V	AVX512VL AVX512F	Computes Square Roots of the packed single-precision floating-point values in ymm2/m256/m32bcst and stores the result in ymm1 subject to writemask k1.
EVEX.512.0F.W0 51/r VSQRTPS zmm1 {k1}{z}, zmm2/m512/m32bcst{er}	B	V/V	AVX512F	Computes Square Roots of the packed single-precision floating-point values in zmm2/m512/m32bcst and stores the result in zmm1 subject to writemask k1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	NA	ModRM:reg (w)	ModRM:r/m (r)	NA	NA
B	Full Vector	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

Description

Performs a SIMD computation of the square roots of the four, eight or sixteen packed single-precision floating-point values in the source operand (second operand) stores the packed single-precision floating-point results in the destination operand.

EVEX.512 encoded versions: The source operand is a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 32-bit memory location. The destination operand is a ZMM/YMM/XMM register updated according to the writemask.

VEX.256 encoded version: The source operand is a YMM register or a 256-bit memory location. The destination operand is a YMM register. The upper bits (MAXVL-1:256) of the corresponding ZMM register destination are zeroed.

VEX.128 encoded version: the source operand second source operand or a 128-bit memory location. The destination operand is an XMM register. The upper bits (MAXVL-1:128) of the corresponding ZMM register destination are zeroed.

128-bit Legacy SSE version: The second source can be an XMM register or 128-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (MAXVL-1:128) of the corresponding ZMM register destination are unmodified.

Note: VEX.vvvv and EVEX.vvvv are reserved and must be 1111b otherwise instructions will #UD.

Operation

VSQRTPS (EVEX encoded versions)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
IF (VL = 512) AND (EVEX.b = 1) AND (SRC *is register*)
    THEN
        SET_RM(EVEX.RC);
    ELSE
        SET_RM(MXCSR.RM);
FI;
FOR j ← 0 TO KL-1
    i ← j * 32
    IF k1[j] OR *no writemask* THEN
            IF (EVEX.b = 1) AND (SRC *is memory*)
                THEN DEST[i+31:i] ← SQRT(SRC[31:0])
                ELSE DEST[i+31:i] ← SQRT(SRC[i+31:i])
            FI;
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE                                ; zeroing-masking
                    DEST[i+31:i] ← 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0
```

VSQRTPS (VEX.256 encoded version)

```
DEST[31:0] ←SQRT(SRC[31:0])
DEST[63:32] ←SQRT(SRC[63:32])
DEST[95:64] ←SQRT(SRC[95:64])
DEST[127:96] ←SQRT(SRC[127:96])
DEST[159:128] ←SQRT(SRC[159:128])
DEST[191:160] ←SQRT(SRC[191:160])
DEST[223:192] ←SQRT(SRC[223:192])
DEST[255:224] ←SQRT(SRC[255:224])
```

VSQRTPS (VEX.128 encoded version)

```
DEST[31:0] ←SQRT(SRC[31:0])
DEST[63:32] ←SQRT(SRC[63:32])
DEST[95:64] ←SQRT(SRC[95:64])
DEST[127:96] ←SQRT(SRC[127:96])
DEST[MAXVL-1:128] ←0
```

SQRTPS (128-bit Legacy SSE version)

```
DEST[31:0] ←SQRT(SRC[31:0])
DEST[63:32] ←SQRT(SRC[63:32])
DEST[95:64] ←SQRT(SRC[95:64])
DEST[127:96] ←SQRT(SRC[127:96])
DEST[MAXVL-1:128] (Unmodified)
```

Intel C/C++ Compiler Intrinsic Equivalent

VSQRTPS __m512 _mm512_sqrt_round_ps(__m512 a, int r);
VSQRTPS __m512 _mm512_mask_sqrt_round_ps(__m512 s, __mmask16 k, __m512 a, int r);
VSQRTPS __m512 _mm512_maskz_sqrt_round_ps(__mmask16 k, __m512 a, int r);
VSQRTPS __m256 _mm256_sqrt_ps (__m256 a);
VSQRTPS __m256 _mm256_mask_sqrt_ps(__m256 s, __mmask8 k, __m256 a, int r);
VSQRTPS __m256 _mm256_maskz_sqrt_ps(__mmask8 k, __m256 a, int r);
SQRTPS __m128 _mm_sqrt_ps (__m128 a);
VSQRTPS __m128 _mm_mask_sqrt_ps(__m128 s, __mmask8 k, __m128 a, int r);
VSQRTPS __m128 _mm_maskz_sqrt_ps(__mmask8 k, __m128 a, int r);

SIMD Floating-Point Exceptions

Invalid, Precision, Denormal

Other Exceptions

Non-EVEX-encoded instruction, see Exceptions Type 2; additionally

#UD	If VEX.vvvv != 1111B.

EVEX-encoded instruction, see Exceptions Type E2.

#UD	If EVEX.vvvv != 1111B.

SQRTSD—Compute Square Root of Scalar Double-Precision Floating-Point Value

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
F2 0F 51/r SQRTSD xmm1,xmm2/m64	A	V/V	SSE2	Computes square root of the low double-precision floating-point value in xmm2/m64 and stores the results in xmm1.
VEX.NDS.LIG.F2.0F.WIG 51/r VSQRTSD xmm1,xmm2, xmm3/m64	B	V/V	AVX	Computes square root of the low double-precision floating-point value in xmm3/m64 and stores the results in xmm1. Also, upper double-precision floating-point value (bits[127:64]) from xmm2 is copied to xmm1[127:64].
EVEX.NDS.LIG.F2.0F.W1 51/r VSQRTSD xmm1 {k1}{z}, xmm2, xmm3/m64{er}	C	V/V	AVX512F	Computes square root of the low double-precision floating-point value in xmm3/m64 and stores the results in xmm1 under writemask k1. Also, upper double-precision floating-point value (bits[127:64]) from xmm2 is copied to xmm1[127:64].

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	NA	ModRM:reg (w)	ModRM:r/m (r)	NA	NA
B	NA	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA
C	Tuple1 Scalar	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA

Description

Computes the square root of the low double-precision floating-point value in the second source operand and stores the double-precision floating-point result in the destination operand. The second source operand can be an XMM register or a 64-bit memory location. The first source and destination operands are XMM registers.

128-bit Legacy SSE version: The first source operand and the destination operand are the same. The quadword at bits 127:64 of the destination operand remains unchanged. Bits (MAXVL-1:64) of the corresponding destination register remain unchanged.

VEX.128 and EVEX encoded versions: Bits 127:64 of the destination operand are copied from the corresponding bits of the first source operand. Bits (MAXVL-1:128) of the destination register are zeroed.

EVEX encoded version: The low quadword element of the destination operand is updated according to the writemask.

Software should ensure VSQRTSD is encoded with VEX.L=0. Encoding VSQRTSD with VEX.L=1 may encounter unpredictable behavior across different processor generations.

Operation

VSQRTSD (EVEX encoded version)

```
IF (EVEX.b = 1) AND (SRC2 *is register*)
    THEN
        SET_RM(EVEX.RC);
    ELSE
        SET_RM(MXCSR.RM);
FI;
IF k1[0] or *no writemask*
    THEN    DEST[63:0] ← SQRT(SRC2[63:0])
    ELSE
        IF *merging-masking*                ; merging-masking
            THEN *DEST[63:0] remains unchanged*
            ELSE                            ; zeroing-masking
                THEN DEST[63:0] ← 0
        FI;
FI;
DEST[127:64] ← SRC1[127:64]
DEST[MAXVL-1:128] ← 0
```

VSQRTSD (VEX.128 encoded version)

```
DEST[63:0] ←SQRT(SRC2[63:0])
DEST[127:64] ←SRC1[127:64]
DEST[MAXVL-1:128] ←0
```

SQRTSD (128-bit Legacy SSE version)

```
DEST[63:0] ←SQRT(SRC[63:0])
DEST[MAXVL-1:64] (Unmodified)
```

Intel C/C++ Compiler Intrinsic Equivalent

VSQRTSD __m128d _mm_sqrt_round_sd(__m128d a, __m128d b, int r);
VSQRTSD __m128d _mm_mask_sqrt_round_sd(__m128d s, __mmask8 k, __m128d a, __m128d b, int r);
VSQRTSD __m128d _mm_maskz_sqrt_round_sd(__mmask8 k, __m128d a, __m128d b, int r);
SQRTSD __m128d _mm_sqrt_sd (__m128d a, __m128d b)

SIMD Floating-Point Exceptions

Invalid, Precision, Denormal

Other Exceptions

Non-EVEX-encoded instruction, see Exceptions Type 3.
EVEX-encoded instruction, see Exceptions Type E3.

SQRTSS—Compute Square Root of Scalar Single-Precision Value

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
F3 0F 51 /r SQRTSS xmm1, xmm2/m32	A	V/V	SSE	Computes square root of the low single-precision floating-point value in xmm2/m32 and stores the results in xmm1.
VEX.NDS.LIG.F3.0F.WIG 51 /r VSQRTSS xmm1, xmm2, xmm3/m32	B	V/V	AVX	Computes square root of the low single-precision floating-point value in xmm3/m32 and stores the results in xmm1. Also, upper single-precision floating-point values (bits[127:32]) from xmm2 are copied to xmm1[127:32].
EVEX.NDS.LIG.F3.0F.W0 51 /r VSQRTSS xmm1 {k1}{z}, xmm2, xmm3/m32{er}	C	V/V	AVX512F	Computes square root of the low single-precision floating-point value in xmm3/m32 and stores the results in xmm1 under writemask k1. Also, upper single-precision floating-point values (bits[127:32]) from xmm2 are copied to xmm1[127:32].

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	NA	ModRM:reg (w)	ModRM:r/m (r)	NA	NA
B	NA	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA
C	Tuple1 Scalar	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA

Description

Computes the square root of the low single-precision floating-point value in the second source operand and stores the single-precision floating-point result in the destination operand. The second source operand can be an XMM register or a 32-bit memory location. The first source and destination operands is an XMM register.

128-bit Legacy SSE version: The first source operand and the destination operand are the same. Bits (MAXVL-1:32) of the corresponding YMM destination register remain unchanged.

VEX.128 and EVEX encoded versions: Bits 127:32 of the destination operand are copied from the corresponding bits of the first source operand. Bits (MAXVL-1:128) of the destination ZMM register are zeroed.

EVEX encoded version: The low doubleword element of the destination operand is updated according to the writemask.

Software should ensure VSQRTSS is encoded with VEX.L=0. Encoding VSQRTSS with VEX.L=1 may encounter unpredictable behavior across different processor generations.

Operation

VSQRTSS (EVEX encoded version)

```
IF (EVEX.b = 1) AND (SRC2 *is register*)
    THEN
        SET_RM(EVEX.RC);
    ELSE
        SET_RM(MXCSR.RM);
FI;
IF k1[0] or *no writemask*
    THEN    DEST[31:0] ← SQRT(SRC2[31:0])
    ELSE
        IF *merging-masking*                    ; merging-masking
            THEN *DEST[31:0] remains unchanged*
            ELSE                                ; zeroing-masking
                DEST[31:0] ← 0
        FI;
FI;
DEST[127:31] ← SRC1[127:31]
DEST[MAXVL-1:128] ← 0
```

VSQRTSS (VEX.128 encoded version)

```
DEST[31:0] ←SQRT(SRC2[31:0])
DEST[127:32] ←SRC1[127:32]
DEST[MAXVL-1:128] ←0
```

SQRTSS (128-bit Legacy SSE version)

```
DEST[31:0] ←SQRT(SRC2[31:0])
DEST[MAXVL-1:32] (Unmodified)
```

Intel C/C++ Compiler Intrinsic Equivalent

VSQRTSS __m128 _mm_sqrt_round_ss(__m128 a, __m128 b, int r);
VSQRTSS __m128 _mm_mask_sqrt_round_ss(__m128 s, __mmask8 k, __m128 a, __m128 b, int r);
VSQRTSS __m128 _mm_maskz_sqrt_round_ss(__mmask8 k, __m128 a, __m128 b, int r);
SQRTSS __m128 _mm_sqrt_ss(__m128 a)

SIMD Floating-Point Exceptions

Invalid, Precision, Denormal

Other Exceptions

Non-EVEX-encoded instruction, see Exceptions Type 3.
EVEX-encoded instruction, see Exceptions Type E3.

STAC—Set AC Flag in EFLAGS Register

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F 01 CB STAC	ZO	V/V	SMAP	Set the AC flag in the EFLAGS register.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
ZO	NA	NA	NA	NA

Description

Sets the AC flag bit in EFLAGS register. This may enable alignment checking of user-mode data accesses. This allows explicit supervisor-mode data accesses to user-mode pages even if the SMAP bit is set in the CR4 register.

This instruction's operation is the same in non-64-bit modes and 64-bit mode. Attempts to execute STAC when CPL > 0 cause #UD.

Operation

EFLAGS.AC ← 1;

Flags Affected

AC set. Other flags are unaffected.

Protected Mode Exceptions

#UD If the LOCK prefix is used.
If the CPL > 0.
If CPUID.(EAX=07H, ECX=0H):EBX.SMAP[bit 20] = 0.

Real-Address Mode Exceptions

#UD If the LOCK prefix is used.
If CPUID.(EAX=07H, ECX=0H):EBX.SMAP[bit 20] = 0.

Virtual-8086 Mode Exceptions

#UD The STAC instruction is not recognized in virtual-8086 mode.

Compatibility Mode Exceptions

#UD If the LOCK prefix is used.
If the CPL > 0.
If CPUID.(EAX=07H, ECX=0H):EBX.SMAP[bit 20] = 0.

64-Bit Mode Exceptions

#UD If the LOCK prefix is used.
If the CPL > 0.
If CPUID.(EAX=07H, ECX=0H):EBX.SMAP[bit 20] = 0.

STC—Set Carry Flag

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
F9	STC	ZO	Valid	Valid	Set CF flag.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
ZO	NA	NA	NA	NA

Description

Sets the CF flag in the EFLAGS register. Operation is the same in all modes.

Operation

CF ← 1;

Flags Affected

The CF flag is set. The OF, ZF, SF, AF, and PF flags are unaffected.

Exceptions (All Operating Modes)

#UD	If the LOCK prefix is used.

STD—Set Direction Flag

Opcode	Instruction	Op/ En	64-bit Mode	Compat/ Leg Mode	Description
FD	STD	ZO	Valid	Valid	Set DF flag.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
ZO	NA	NA	NA	NA

Description

Sets the DF flag in the EFLAGS register. When the DF flag is set to 1, string operations decrement the index registers (ESI and/or EDI). Operation is the same in all modes.

Operation

DF ← 1;

Flags Affected

The DF flag is set. The CF, OF, ZF, SF, AF, and PF flags are unaffected.

Exceptions (All Operating Modes)

#UD	If the LOCK prefix is used.

STI—Set Interrupt Flag

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
FB	STI	ZO	Valid	Valid	Set interrupt flag; external, maskable interrupts enabled at the end of the next instruction.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
ZO	NA	NA	NA	NA

Description

In most cases, STI sets the interrupt flag (IF) in the EFLAGS register. After the IF flag is set, the processor begins responding to external, maskable interrupts after the next instruction is executed. The delayed effect of this instruction is provided to allow interrupts to be enabled just before returning from a procedure (or subroutine). For instance, if an STI instruction is followed by an RET instruction, the RET instruction is allowed to execute before external interrupts are recognized[1]. If the STI instruction is followed by a CLI instruction (which clears the IF flag), the effect of the STI instruction is negated.

The IF flag and the STI and CLI instructions do not prohibit the generation of exceptions and NMI interrupts. NMI interrupts (and SMIs) may be blocked for one macroinstruction following an STI.

Operation is different in two modes defined as follows:

- **PVI mode** (protected-mode virtual interrupts): CR0.PE = 1, EFLAGS.VM = 0, CPL = 3, and CR4.PVI = 1;
- **VME mode** (virtual-8086 mode extensions): CR0.PE = 1, EFLAGS.VM = 1, and CR4.VME = 1.

If IOPL < 3, EFLAGS.VIP = 1, and either VME mode or PVI mode is active, STI sets the VIF flag in the EFLAGS register, leaving IF unaffected.

Table 4-19 indicates the action of the STI instruction depending on the processor operating mode, IOPL, CPL, and EFLAGS.VIP.

Table 4-19. Decision Table for STI Results

Mode	IOPL	EFLAGS.VIP	STI Result
Real-address	X[1]	X	IF = 1
Protected, not PVI[2]	≥ CPL	X	IF = 1
	< CPL	X	#GP fault
Protected, PVI[3]	3	X	IF = 1
	0–2	0	VIF = 1
		1	#GP fault
Virtual-8086, not VME[3]	3	X	IF = 1
	0–2	X	#GP fault

1. The STI instruction delays recognition of interrupts only if it is executed with EFLAGS.IF = 0. In a sequence of STI instructions, only the first instruction in the sequence is guaranteed to delay interrupts.

   In the following instruction sequence, interrupts may be recognized before RET executes:
   STI
   STI
   RET

Table 4-19. Decision Table for STI Results

Mode	IOPL	EFLAGS.VIP	STI Result
Virtual-8086, VME[3]	3	X	IF = 1
	0–2	0	VIF = 1
		1	#GP fault

NOTES:

1. X = This setting has no effect on instruction operation.
2. For this table, "protected mode" applies whenever CR0.PE = 1 and EFLAGS.VM = 0; it includes compatibility mode and 64-bit mode.
3. PVI mode and virtual-8086 mode each imply CPL = 3.

Operation

```
IF CR0.PE = 0  (* Executing in real-address mode *)
    THEN IF ← 1; (* Set Interrupt Flag *)
    ELSE
        IF IOPL ≥ CPL  (* CPL = 3 if EFLAGS.VM = 1 *)
            THEN IF ← 1; (* Set Interrupt Flag *)
            ELSE
                IF VME mode OR PVI mode
                    THEN
                        IF EFLAGS.VIP = 0
                            THEN VIF ← 1; (* Set Virtual Interrupt Flag *)
                            ELSE #GP(0);
                        FI;
                    ELSE #GP(0);
                FI;
        FI;
FI;
```

Flags Affected

Either the IF flag or the VIF flag is set to 1. Other flags are unaffected.

Protected Mode Exceptions

#GP(0)	If CPL is greater than IOPL and PVI mode is not active.
	If CPL is greater than IOPL and EFLAGS.VIP = 1.
#UD	If the LOCK prefix is used.

Real-Address Mode Exceptions

#UD	If the LOCK prefix is used.

Virtual-8086 Mode Exceptions

#GP(0)	If IOPL is less than 3 and VME mode is not active.
	If IOPL is less than 3 and EFLAGS.VIP = 1.
#UD	If the LOCK prefix is used.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

Same exceptions as in protected mode.

STMXCSR—Store MXCSR Register State

Opcode*/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F AE /3 STMXCSR *m32*	M	V/V	SSE	Store contents of MXCSR register to *m32*.
VEX.LZ.0F.WIG AE /3 VSTMXCSR *m32*	M	V/V	AVX	Store contents of MXCSR register to *m32*.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
M	ModRM:r/m (w)	NA	NA	NA

Description

Stores the contents of the MXCSR control and status register to the destination operand. The destination operand is a 32-bit memory location. The reserved bits in the MXCSR register are stored as 0s.

This instruction's operation is the same in non-64-bit modes and 64-bit mode.

VEX.L must be 0, otherwise instructions will #UD.

Note: In VEX-encoded versions, VEX.vvvv is reserved and must be 1111b, otherwise instructions will #UD.

Operation

m32 ← MXCSR;

Intel C/C++ Compiler Intrinsic Equivalent

_mm_getcsr(void)

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Exceptions Type 5; additionally

#UD	If VEX.L= 1,
	If VEX.vvvv ≠ 1111B.

STOS/STOSB/STOSW/STOSD/STOSQ—Store String

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
AA	STOS *m8*	NA	Valid	Valid	For legacy mode, store AL at address ES:(E)DI; For 64-bit mode store AL at address RDI or EDI.
AB	STOS *m16*	NA	Valid	Valid	For legacy mode, store AX at address ES:(E)DI; For 64-bit mode store AX at address RDI or EDI.
AB	STOS *m32*	NA	Valid	Valid	For legacy mode, store EAX at address ES:(E)DI; For 64-bit mode store EAX at address RDI or EDI.
REX.W + AB	STOS *m64*	NA	Valid	N.E.	Store RAX at address RDI or EDI.
AA	STOSB	NA	Valid	Valid	For legacy mode, store AL at address ES:(E)DI; For 64-bit mode store AL at address RDI or EDI.
AB	STOSW	NA	Valid	Valid	For legacy mode, store AX at address ES:(E)DI; For 64-bit mode store AX at address RDI or EDI.
AB	STOSD	NA	Valid	Valid	For legacy mode, store EAX at address ES:(E)DI; For 64-bit mode store EAX at address RDI or EDI.
REX.W + AB	STOSQ	NA	Valid	N.E.	Store RAX at address RDI or EDI.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
NA	NA	NA	NA	NA

Description

In non-64-bit and default 64-bit mode; stores a byte, word, or doubleword from the AL, AX, or EAX register (respectively) into the destination operand. The destination operand is a memory location, the address of which is read from either the ES:EDI or ES:DI register (depending on the address-size attribute of the instruction and the mode of operation). The ES segment cannot be overridden with a segment override prefix.

At the assembly-code level, two forms of the instruction are allowed: the "explicit-operands" form and the "no-operands" form. The explicit-operands form (specified with the STOS mnemonic) allows the destination operand to be specified explicitly. Here, the destination operand should be a symbol that indicates the size and location of the destination value. The source operand is then automatically selected to match the size of the destination operand (the AL register for byte operands, AX for word operands, EAX for doubleword operands). The explicit-operands form is provided to allow documentation; however, note that the documentation provided by this form can be misleading. That is, the destination operand symbol must specify the correct **type** (size) of the operand (byte, word, or doubleword), but it does not have to specify the correct **location**. The location is always specified by the ES:(E)DI register. These must be loaded correctly before the store string instruction is executed.

The no-operands form provides "short forms" of the byte, word, doubleword, and quadword versions of the STOS instructions. Here also ES:(E)DI is assumed to be the destination operand and AL, AX, or EAX is assumed to be the source operand. The size of the destination and source operands is selected by the mnemonic: STOSB (byte read from register AL), STOSW (word from AX), STOSD (doubleword from EAX).

After the byte, word, or doubleword is transferred from the register to the memory location, the (E)DI register is incremented or decremented according to the setting of the DF flag in the EFLAGS register. If the DF flag is 0, the register is incremented; if the DF flag is 1, the register is decremented (the register is incremented or decremented by 1 for byte operations, by 2 for word operations, by 4 for doubleword operations).

NOTE: To improve performance, more recent processors support modifications to the processor's operation during the string store operations initiated with STOS and STOSB. See Section 7.3.9.3 in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1* for additional information on fast-string operation.

In 64-bit mode, the default address size is 64 bits, 32-bit address size is supported using the prefix 67H. Using a REX prefix in the form of REX.W promotes operation on doubleword operand to 64 bits. The promoted no-operand mnemonic is STOSQ. STOSQ (and its explicit operands variant) store a quadword from the RAX register into the destination addressed by RDI or EDI. See the summary chart at the beginning of this section for encoding data and limits.

The STOS, STOSB, STOSW, STOSD, STOSQ instructions can be preceded by the REP prefix for block loads of ECX bytes, words, or doublewords. More often, however, these instructions are used within a LOOP construct because data needs to be moved into the AL, AX, or EAX register before it can be stored. See "REP/REPE/REPZ /REPNE/REPNZ—Repeat String Operation Prefix" in this chapter for a description of the REP prefix.

Operation

Non-64-bit Mode:

```
IF (Byte store)
    THEN
        DEST ← AL;
            THEN IF DF = 0
                THEN (E)DI ← (E)DI + 1;
                ELSE (E)DI ← (E)DI – 1;
            FI;
    ELSE IF (Word store)
        THEN
            DEST ← AX;
                THEN IF DF = 0
                    THEN (E)DI ← (E)DI + 2;
                    ELSE (E)DI ← (E)DI – 2;
                FI;
        FI;
    ELSE IF (Doubleword store)
        THEN
            DEST ← EAX;
                THEN IF DF = 0
                    THEN (E)DI ← (E)DI + 4;
                    ELSE (E)DI ← (E)DI – 4;
                FI;
        FI;
FI;
```

64-bit Mode:

```
IF (Byte store)
    THEN
        DEST ← AL;
            THEN IF DF = 0
                THEN (R|E)DI ← (R|E)DI + 1;
                ELSE (R|E)DI ← (R|E)DI – 1;
            FI;
    ELSE IF (Word store)
        THEN
            DEST ← AX;
```

```
                THEN IF DF = 0
                    THEN (R|E)DI ← (R|E)DI + 2;
                    ELSE (R|E)DI ← (R|E)DI – 2;
                FI;
        FI;
    ELSE IF (Doubleword store)
        THEN
            DEST ← EAX;
                THEN IF DF = 0
                    THEN (R|E)DI ← (R|E)DI + 4;
                    ELSE (R|E)DI ← (R|E)DI – 4;
                FI;
        FI;
    ELSE IF (Quadword store using REX.W )
        THEN
            DEST ← RAX;
                THEN IF DF = 0
                    THEN (R|E)DI ← (R|E)DI + 8;
                    ELSE (R|E)DI ← (R|E)DI – 8;
                FI;
        FI;
FI;
```

Flags Affected

None.

Protected Mode Exceptions

#GP(0)	If the destination is located in a non-writable segment.
	If a memory operand effective address is outside the limit of the ES segment.
	If the ES register contains a NULL segment selector.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If the LOCK prefix is used.

Real-Address Mode Exceptions

#GP	If a memory operand effective address is outside the ES segment limit.
#UD	If the LOCK prefix is used.

Virtual-8086 Mode Exceptions

#GP(0)	If a memory operand effective address is outside the ES segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.
#UD	If the LOCK prefix is used.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#GP(0)	If the memory address is in a non-canonical form.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If the LOCK prefix is used.

STR—Store Task Register

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
0F 00 /1	STR *r/m16*	M	Valid	Valid	Stores segment selector from TR in *r/m16*.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
M	ModRM:r/m (w)	NA	NA	NA

Description

Stores the segment selector from the task register (TR) in the destination operand. The destination operand can be a general-purpose register or a memory location. The segment selector stored with this instruction points to the task state segment (TSS) for the currently running task.

When the destination operand is a 32-bit register, the 16-bit segment selector is copied into the lower 16 bits of the register and the upper 16 bits of the register are cleared. When the destination operand is a memory location, the segment selector is written to memory as a 16-bit quantity, regardless of operand size.

In 64-bit mode, operation is the same. The size of the memory operand is fixed at 16 bits. In register stores, the 2-byte TR is zero extended if stored to a 64-bit register.

The STR instruction is useful only in operating-system software. It can only be executed in protected mode.

Operation

```
DEST ← TR(SegmentSelector);
```

Flags Affected

None.

Protected Mode Exceptions

#GP(0)	If the destination is a memory operand that is located in a non-writable segment or if the effective address is outside the CS, DS, ES, FS, or GS segment limit.
	If the DS, ES, FS, or GS register is used to access memory and it contains a NULL segment selector.
	If CR4.UMIP = 1 and CPL > 0.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If the LOCK prefix is used.

Real-Address Mode Exceptions

#UD	The STR instruction is not recognized in real-address mode.

Virtual-8086 Mode Exceptions

#UD	The STR instruction is not recognized in virtual-8086 mode.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#GP(0)	If the memory address is in a non-canonical form. If CR4.UMIP = 1 and CPL > 0.
#SS(0)	If the stack address is in a non-canonical form.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If the LOCK prefix is used.

SUB—Subtract

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
2C *ib*	SUB AL, i*mm8*	I	Valid	Valid	Subtract *imm8* from AL.
2D *iw*	SUB AX, i*mm16*	I	Valid	Valid	Subtract *imm16* from AX.
2D *id*	SUB EAX, i*mm32*	I	Valid	Valid	Subtract *imm32* from EAX.
REX.W + 2D *id*	SUB RAX, i*mm32*	I	Valid	N.E.	Subtract *imm32* sign-extended to 64-bits from RAX.
80 /5 *ib*	SUB *r/m8, imm8*	MI	Valid	Valid	Subtract *imm8* from *r/m8.*
REX + 80 /5 *ib*	SUB *r/m8*, imm8*	MI	Valid	N.E.	Subtract *imm8* from *r/m8.*
81 /5 *iw*	SUB *r/m16, imm16*	MI	Valid	Valid	Subtract *imm16* from *r/m16.*
81 /5 *id*	SUB *r/m32, imm32*	MI	Valid	Valid	Subtract *imm32* from *r/m32.*
REX.W + 81 /5 *id*	SUB *r/m64, imm32*	MI	Valid	N.E.	Subtract *imm32* sign-extended to 64-bits from *r/m64.*
83 /5 *ib*	SUB *r/m16, imm8*	MI	Valid	Valid	Subtract sign-extended *imm8* from *r/m16.*
83 /5 *ib*	SUB *r/m32, imm8*	MI	Valid	Valid	Subtract sign-extended *imm8* from *r/m32.*
REX.W + 83 /5 *ib*	SUB *r/m64, imm8*	MI	Valid	N.E.	Subtract sign-extended *imm8* from *r/m64.*
28 /*r*	SUB *r/m8, r8*	MR	Valid	Valid	Subtract *r8* from *r/m8.*
REX + 28 /*r*	SUB *r/m8*, r8**	MR	Valid	N.E.	Subtract *r8* from *r/m8.*
29 /*r*	SUB *r/m16, r16*	MR	Valid	Valid	Subtract *r16* from *r/m16.*
29 /*r*	SUB *r/m32, r32*	MR	Valid	Valid	Subtract *r32* from *r/m32.*
REX.W + 29 /*r*	SUB *r/m64, r64*	MR	Valid	N.E.	Subtract *r64* from *r/m64.*
2A /*r*	SUB *r8, r/m8*	RM	Valid	Valid	Subtract *r/m8* from *r8.*
REX + 2A /*r*	SUB *r8*, r/m8**	RM	Valid	N.E.	Subtract *r/m8* from *r8.*
2B /*r*	SUB *r16, r/m16*	RM	Valid	Valid	Subtract *r/m16* from *r16.*
2B /*r*	SUB *r32, r/m32*	RM	Valid	Valid	Subtract *r/m32* from *r32.*
REX.W + 2B /*r*	SUB *r64, r/m64*	RM	Valid	N.E.	Subtract *r/m64* from *r64.*

NOTES:

* In 64-bit mode, r/m8 can not be encoded to access the following byte registers if a REX prefix is used: AH, BH, CH, DH.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
I	AL/AX/EAX/RAX	imm8/26/32	NA	NA
MI	ModRM:r/m (r, w)	imm8/26/32	NA	NA
MR	ModRM:r/m (r, w)	ModRM:reg (r)	NA	NA
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA

Description

Subtracts the second operand (source operand) from the first operand (destination operand) and stores the result in the destination operand. The destination operand can be a register or a memory location; the source operand can be an immediate, register, or memory location. (However, two memory operands cannot be used in one instruction.) When an immediate value is used as an operand, it is sign-extended to the length of the destination operand format.

The SUB instruction performs integer subtraction. It evaluates the result for both signed and unsigned integer operands and sets the OF and CF flags to indicate an overflow in the signed or unsigned result, respectively. The SF flag indicates the sign of the signed result.

In 64-bit mode, the instruction's default operation size is 32 bits. Using a REX prefix in the form of REX.R permits access to additional registers (R8-R15). Using a REX prefix in the form of REX.W promotes operation to 64 bits. See the summary chart at the beginning of this section for encoding data and limits.

This instruction can be used with a LOCK prefix to allow the instruction to be executed atomically.

Operation

DEST ← (DEST – SRC);

Flags Affected

The OF, SF, ZF, AF, PF, and CF flags are set according to the result.

Protected Mode Exceptions

#GP(0)	If the destination is located in a non-writable segment.
	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
	If the DS, ES, FS, or GS register contains a NULL segment selector.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If the LOCK prefix is used but the destination is not a memory operand.

Real-Address Mode Exceptions

#GP	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS	If a memory operand effective address is outside the SS segment limit.
#UD	If the LOCK prefix is used but the destination is not a memory operand.

Virtual-8086 Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.
#UD	If the LOCK prefix is used but the destination is not a memory operand.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If the LOCK prefix is used but the destination is not a memory operand.

SUBPD—Subtract Packed Double-Precision Floating-Point Values

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 5C /r SUBPD xmm1, xmm2/m128	A	V/V	SSE2	Subtract packed double-precision floating-point values in xmm2/mem from xmm1 and store result in xmm1.
VEX.NDS.128.66.0F.WIG 5C /r VSUBPD xmm1,xmm2, xmm3/m128	B	V/V	AVX	Subtract packed double-precision floating-point values in xmm3/mem from xmm2 and store result in xmm1.
VEX.NDS.256.66.0F.WIG 5C /r VSUBPD ymm1, ymm2, ymm3/m256	B	V/V	AVX	Subtract packed double-precision floating-point values in ymm3/mem from ymm2 and store result in ymm1.
EVEX.NDS.128.66.0F.W1 5C /r VSUBPD xmm1 {k1}{z}, xmm2, xmm3/m128/m64bcst	C	V/V	AVX512VL AVX512F	Subtract packed double-precision floating-point values from xmm3/m128/m64bcst to xmm2 and store result in xmm1 with writemask k1.
EVEX.NDS.256.66.0F.W1 5C /r VSUBPD ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst	C	V/V	AVX512VL AVX512F	Subtract packed double-precision floating-point values from ymm3/m256/m64bcst to ymm2 and store result in ymm1 with writemask k1.
EVEX.NDS.512.66.0F.W1 5C /r VSUBPD zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst{er}	C	V/V	AVX512F	Subtract packed double-precision floating-point values from zmm3/m512/m64bcst to zmm2 and store result in zmm1 with writemask k1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	NA	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
B	NA	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA
C	Full Vector	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA

Description

Performs a SIMD subtract of the two, four or eight packed double-precision floating-point values of the second Source operand from the first Source operand, and stores the packed double-precision floating-point results in the destination operand.

VEX.128 and EVEX.128 encoded versions: The second source operand is an XMM register or an 128-bit memory location. The first source operand and destination operands are XMM registers. Bits (MAXVL-1:128) of the corresponding destination register are zeroed.

VEX.256 and EVEX.256 encoded versions: The second source operand is an YMM register or an 256-bit memory location. The first source operand and destination operands are YMM registers. Bits (MAXVL-1:256) of the corresponding destination register are zeroed.

EVEX.512 encoded version: The second source operand is a ZMM register, a 512-bit memory location or a 512-bit vector broadcasted from a 64-bit memory location. The first source operand and destination operands are ZMM registers. The destination operand is conditionally updated according to the writemask.

128-bit Legacy SSE version: The second source can be an XMM register or an 128-bit memory location. The destination is not distinct from the first source XMM register and the upper Bits (MAXVL-1:128) of the corresponding register destination are unmodified.

Operation

VSUBPD (EVEX encoded versions) when src2 operand is a vector register

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
IF (VL = 512) AND (EVEX.b = 1)
    THEN
        SET_RM(EVEX.RC);
    ELSE
        SET_RM(MXCSR.RM);
FI;

FOR j ← 0 TO KL-1
    i ← j * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+63:i] ← SRC1[i+63:i] - SRC2[i+63:i]
    ELSE
        IF *merging-masking*                    ; merging-masking
            THEN *DEST[63:0] remains unchanged*
            ELSE                                ; zeroing-masking
                DEST[63:0] ← 0
        FI;
    FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0
```

VSUBPD (EVEX encoded versions) when src2 operand is a memory source

```
(KL, VL) = (2, 128), (4, 256), (8, 512)

FOR j ← 0 TO KL-1
    i ← j * 64
    IF k1[j] OR *no writemask* THEN
            IF (EVEX.b = 1)
                THEN    DEST[i+63:i] ← SRC1[i+63:i] - SRC2[63:0];
                ELSE    EST[i+63:i] ← SRC1[i+63:i] - SRC2[i+63:i];
            FI;
    ELSE
        IF *merging-masking*                    ; merging-masking
            THEN *DEST[63:0] remains unchanged*
            ELSE                                ; zeroing-masking
                DEST[63:0] ← 0
        FI;
    FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0
```

VSUBPD (VEX.256 encoded version)

```
DEST[63:0] ← SRC1[63:0] - SRC2[63:0]
DEST[127:64] ← SRC1[127:64] - SRC2[127:64]
DEST[191:128] ← SRC1[191:128] - SRC2[191:128]
DEST[255:192] ← SRC1[255:192] - SRC2[255:192]
DEST[MAXVL-1:256] ← 0
```

VSUBPD (VEX.128 encoded version)
DEST[63:0] ← SRC1[63:0] - SRC2[63:0]
DEST[127:64] ← SRC1[127:64] - SRC2[127:64]
DEST[MAXVL-1:128] ← 0

SUBPD (128-bit Legacy SSE version)
DEST[63:0] ← DEST[63:0] - SRC[63:0]
DEST[127:64] ← DEST[127:64] - SRC[127:64]
DEST[MAXVL-1:128] (Unmodified)

Intel C/C++ Compiler Intrinsic Equivalent

VSUBPD __m512d _mm512_sub_pd (__m512d a, __m512d b);
VSUBPD __m512d _mm512_mask_sub_pd (__m512d s, __mmask8 k, __m512d a, __m512d b);
VSUBPD __m512d _mm512_maskz_sub_pd (__mmask8 k, __m512d a, __m512d b);
VSUBPD __m512d _mm512_sub_round_pd (__m512d a, __m512d b, int);
VSUBPD __m512d _mm512_mask_sub_round_pd (__m512d s, __mmask8 k, __m512d a, __m512d b, int);
VSUBPD __m512d _mm512_maskz_sub_round_pd (__mmask8 k, __m512d a, __m512d b, int);
VSUBPD __m256d _mm256_sub_pd (__m256d a, __m256d b);
VSUBPD __m256d _mm256_mask_sub_pd (__m256d s, __mmask8 k, __m256d a, __m256d b);
VSUBPD __m256d _mm256_maskz_sub_pd (__mmask8 k, __m256d a, __m256d b);
SUBPD __m128d _mm_sub_pd (__m128d a, __m128d b);
VSUBPD __m128d _mm_mask_sub_pd (__m128d s, __mmask8 k, __m128d a, __m128d b);
VSUBPD __m128d _mm_maskz_sub_pd (__mmask8 k, __m128d a, __m128d b);

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal

Other Exceptions

VEX-encoded instructions, see Exceptions Type 2.
EVEX-encoded instructions, see Exceptions Type E2.

SUBPS—Subtract Packed Single-Precision Floating-Point Values

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F 5C /r SUBPS xmm1, xmm2/m128	A	V/V	SSE	Subtract packed single-precision floating-point values in xmm2/mem from xmm1 and store result in xmm1.
VEX.NDS.128.0F.WIG 5C /r VSUBPS xmm1,xmm2, xmm3/m128	B	V/V	AVX	Subtract packed single-precision floating-point values in xmm3/mem from xmm2 and stores result in xmm1.
VEX.NDS.256.0F.WIG 5C /r VSUBPS ymm1, ymm2, ymm3/m256	B	V/V	AVX	Subtract packed single-precision floating-point values in ymm3/mem from ymm2 and stores result in ymm1.
EVEX.NDS.128.0F.W0 5C /r VSUBPS xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst	C	V/V	AVX512VL AVX512F	Subtract packed single-precision floating-point values from xmm3/m128/m32bcst to xmm2 and stores result in xmm1 with writemask k1.
EVEX.NDS.256.0F.W0 5C /r VSUBPS ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst	C	V/V	AVX512VL AVX512F	Subtract packed single-precision floating-point values from ymm3/m256/m32bcst to ymm2 and stores result in ymm1 with writemask k1.
EVEX.NDS.512.0F.W0 5C /r VSUBPS zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst{er}	C	V/V	AVX512F	Subtract packed single-precision floating-point values in zmm3/m512/m32bcst from zmm2 and stores result in zmm1 with writemask k1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	NA	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
B	NA	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA
C	Full Vector	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA

Description

Performs a SIMD subtract of the packed single-precision floating-point values in the second Source operand from the First Source operand, and stores the packed single-precision floating-point results in the destination operand.

VEX.128 and EVEX.128 encoded versions: The second source operand is an XMM register or an 128-bit memory location. The first source operand and destination operands are XMM registers. Bits (MAXVL-1:128) of the corresponding destination register are zeroed.

VEX.256 and EVEX.256 encoded versions: The second source operand is an YMM register or an 256-bit memory location. The first source operand and destination operands are YMM registers. Bits (MAXVL-1:256) of the corresponding destination register are zeroed.

EVEX.512 encoded version: The second source operand is a ZMM register, a 512-bit memory location or a 512-bit vector broadcasted from a 32-bit memory location. The first source operand and destination operands are ZMM registers. The destination operand is conditionally updated according to the writemask.

128-bit Legacy SSE version: The second source can be an XMM register or an 128-bit memory location. The destination is not distinct from the first source XMM register and the upper Bits (MAXVL-1:128) of the corresponding register destination are unmodified.

Operation

VSUBPS (EVEX encoded versions) when src2 operand is a vector register

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
IF (VL = 512) AND (EVEX.b = 1)
    THEN
        SET_RM(EVEX.RC);
    ELSE
        SET_RM(MXCSR.RM);
FI;

FOR j ← 0 TO KL-1
    i ← j * 32
    IF k1[j] OR *no writemask*
        THEN DEST[i+31:i] ← SRC1[i+31:i] - SRC2[i+31:i]
    ELSE
        IF *merging-masking*                    ; merging-masking
            THEN *DEST[31:0] remains unchanged*
            ELSE                                ; zeroing-masking
                DEST[31:0] ← 0
        FI;
    FI;
ENDFOR;
DEST[MAXVL-1:VL] ← 0
```

VSUBPS (EVEX encoded versions) when src2 operand is a memory source

```
(KL, VL) = (4, 128), (8, 256),(16, 512)
FOR j ← 0 TO KL-1
    i ←j * 32
    IF k1[j] OR *no writemask* THEN
            IF (EVEX.b = 1)
                THEN    DEST[i+31:i] ← SRC1[i+31:i] - SRC2[31:0];
                ELSE    DEST[i+31:i] ← SRC1[i+31:i] - SRC2[i+31:i];
            FI;

    ELSE
        IF *merging-masking*                    ; merging-masking
            THEN *DEST[31:0] remains unchanged*
            ELSE                                ; zeroing-masking
                DEST[31:0] ← 0
        FI;
    FI;
ENDFOR;
DEST[MAXVL-1:VL] ← 0
```

VSUBPS (VEX.256 encoded version)

```
DEST[31:0] ← SRC1[31:0] - SRC2[31:0]
DEST[63:32] ← SRC1[63:32] - SRC2[63:32]
DEST[95:64] ← SRC1[95:64] - SRC2[95:64]
DEST[127:96] ← SRC1[127:96] - SRC2[127:96]
DEST[159:128] ← SRC1[159:128] - SRC2[159:128]
DEST[191:160]← SRC1[191:160] - SRC2[191:160]
DEST[223:192] ← SRC1[223:192] - SRC2[223:192]
DEST[255:224] ← SRC1[255:224] - SRC2[255:224].
DEST[MAXVL-1:256] ← 0
```

VSUBPS (VEX.128 encoded version)
DEST[31:0] ← SRC1[31:0] - SRC2[31:0]
DEST[63:32] ← SRC1[63:32] - SRC2[63:32]
DEST[95:64] ← SRC1[95:64] - SRC2[95:64]
DEST[127:96] ← SRC1[127:96] - SRC2[127:96]
DEST[MAXVL-1:128] ← 0

SUBPS (128-bit Legacy SSE version)
DEST[31:0] ← SRC1[31:0] - SRC2[31:0]
DEST[63:32] ← SRC1[63:32] - SRC2[63:32]
DEST[95:64] ← SRC1[95:64] - SRC2[95:64]
DEST[127:96] ← SRC1[127:96] - SRC2[127:96]
DEST[MAXVL-1:128] (Unmodified)

Intel C/C++ Compiler Intrinsic Equivalent

VSUBPS __m512 _mm512_sub_ps (__m512 a, __m512 b);
VSUBPS __m512 _mm512_mask_sub_ps (__m512 s, __mmask16 k, __m512 a, __m512 b);
VSUBPS __m512 _mm512_maskz_sub_ps (__mmask16 k, __m512 a, __m512 b);
VSUBPS __m512 _mm512_sub_round_ps (__m512 a, __m512 b, int);
VSUBPS __m512 _mm512_mask_sub_round_ps (__m512 s, __mmask16 k, __m512 a, __m512 b, int);
VSUBPS __m512 _mm512_maskz_sub_round_ps (__mmask16 k, __m512 a, __m512 b, int);
VSUBPS __m256 _mm256_sub_ps (__m256 a, __m256 b);
VSUBPS __m256 _mm256_mask_sub_ps (__m256 s, __mmask8 k, __m256 a, __m256 b);
VSUBPS __m256 _mm256_maskz_sub_ps (__mmask16 k, __m256 a, __m256 b);
SUBPS __m128 _mm_sub_ps (__m128 a, __m128 b);
VSUBPS __m128 _mm_mask_sub_ps (__m128 s, __mmask8 k, __m128 a, __m128 b);
VSUBPS __m128 _mm_maskz_sub_ps (__mmask16 k, __m128 a, __m128 b);

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal

Other Exceptions

VEX-encoded instructions, see Exceptions Type 2.
EVEX-encoded instructions, see Exceptions Type E2.

SUBSD—Subtract Scalar Double-Precision Floating-Point Value

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
F2 0F 5C /r SUBSD xmm1, xmm2/m64	A	V/V	SSE2	Subtract the low double-precision floating-point value in xmm2/m64 from xmm1 and store the result in xmm1.
VEX.NDS.LIG.F2.0F.WIG 5C /r VSUBSD xmm1,xmm2, xmm3/m64	B	V/V	AVX	Subtract the low double-precision floating-point value in xmm3/m64 from xmm2 and store the result in xmm1.
EVEX.NDS.LIG.F2.0F.W1 5C /r VSUBSD xmm1 {k1}{z}, xmm2, xmm3/m64{er}	C	V/V	AVX512F	Subtract the low double-precision floating-point value in xmm3/m64 from xmm2 and store the result in xmm1 under writemask k1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	NA	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
B	NA	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA
C	Tuple1 Scalar	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA

Description

Subtract the low double-precision floating-point value in the second source operand from the first source operand and stores the double-precision floating-point result in the low quadword of the destination operand.

The second source operand can be an XMM register or a 64-bit memory location. The first source and destination operands are XMM registers.

128-bit Legacy SSE version: The destination and first source operand are the same. Bits (MAXVL-1:64) of the corresponding destination register remain unchanged.

VEX.128 and EVEX encoded versions: Bits (127:64) of the XMM register destination are copied from corresponding bits in the first source operand. Bits (MAXVL-1:128) of the destination register are zeroed.

EVEX encoded version: The low quadword element of the destination operand is updated according to the writemask.

Software should ensure VSUBSD is encoded with VEX.L=0. Encoding VSUBSD with VEX.L=1 may encounter unpredictable behavior across different processor generations.

Operation

VSUBSD (EVEX encoded version)

```
IF (SRC2 *is register*) AND (EVEX.b = 1)
    THEN
        SET_RM(EVEX.RC);
    ELSE
        SET_RM(MXCSR.RM);
FI;
IF k1[0] or *no writemask*
    THEN    DEST[63:0] ← SRC1[63:0] - SRC2[63:0]
    ELSE
        IF *merging-masking*                    ; merging-masking
            THEN *DEST[63:0] remains unchanged*
            ELSE                                ; zeroing-masking
                THEN DEST[63:0] ← 0
        FI;
FI;
DEST[127:64] ← SRC1[127:64]
DEST[MAXVL-1:128] ← 0
```

VSUBSD (VEX.128 encoded version)

```
DEST[63:0] ←SRC1[63:0] - SRC2[63:0]
DEST[127:64] ←SRC1[127:64]
DEST[MAXVL-1:128] ←0
```

SUBSD (128-bit Legacy SSE version)

```
DEST[63:0] ←DEST[63:0] - SRC[63:0]
DEST[MAXVL-1:64] (Unmodified)
```

Intel C/C++ Compiler Intrinsic Equivalent

VSUBSD __m128d _mm_mask_sub_sd (__m128d s, __mmask8 k, __m128d a, __m128d b);
VSUBSD __m128d _mm_maskz_sub_sd (__mmask8 k, __m128d a, __m128d b);
VSUBSD __m128d _mm_sub_round_sd (__m128d a, __m128d b, int);
VSUBSD __m128d _mm_mask_sub_round_sd (__m128d s, __mmask8 k, __m128d a, __m128d b, int);
VSUBSD __m128d _mm_maskz_sub_round_sd (__mmask8 k, __m128d a, __m128d b, int);
SUBSD __m128d _mm_sub_sd (__m128d a, __m128d b);

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal

Other Exceptions

VEX-encoded instructions, see Exceptions Type 3.
EVEX-encoded instructions, see Exceptions Type E3.

SUBSS—Subtract Scalar Single-Precision Floating-Point Value

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
F3 0F 5C /r SUBSS xmm1, xmm2/m32	A	V/V	SSE	Subtract the low single-precision floating-point value in xmm2/m32 from xmm1 and store the result in xmm1.
VEX.NDS.LIG.F3.0F.WIG 5C /r VSUBSS xmm1,xmm2, xmm3/m32	B	V/V	AVX	Subtract the low single-precision floating-point value in xmm3/m32 from xmm2 and store the result in xmm1.
EVEX.NDS.LIG.F3.0F.W0 5C /r VSUBSS xmm1 {k1}{z}, xmm2, xmm3/m32{er}	C	V/V	AVX512F	Subtract the low single-precision floating-point value in xmm3/m32 from xmm2 and store the result in xmm1 under writemask k1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	NA	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
B	NA	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA
C	Tuple1 Scalar	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA

Description

Subtract the low single-precision floating-point value from the second source operand and the first source operand and store the double-precision floating-point result in the low doubleword of the destination operand.

The second source operand can be an XMM register or a 32-bit memory location. The first source and destination operands are XMM registers.

128-bit Legacy SSE version: The destination and first source operand are the same. Bits (MAXVL-1:32) of the corresponding destination register remain unchanged.

VEX.128 and EVEX encoded versions: Bits (127:32) of the XMM register destination are copied from corresponding bits in the first source operand. Bits (MAXVL-1:128) of the destination register are zeroed.

EVEX encoded version: The low doubleword element of the destination operand is updated according to the writemask.

Software should ensure VSUBSS is encoded with VEX.L=0. Encoding VSUBSD with VEX.L=1 may encounter unpredictable behavior across different processor generations.

Operation

VSUBSS (EVEX encoded version)

```
IF (SRC2 *is register*) AND (EVEX.b = 1)
    THEN
        SET_RM(EVEX.RC);
    ELSE
        SET_RM(MXCSR.RM);
FI;
IF k1[0] or *no writemask*
    THEN    DEST[31:0] ← SRC1[31:0] - SRC2[31:0]
    ELSE
        IF *merging-masking*                ; merging-masking
            THEN *DEST[31:0] remains unchanged*
            ELSE                            ; zeroing-masking
                THEN DEST[31:0] ← 0
        FI;
FI;
DEST[127:32] ← SRC1[127:32]
DEST[MAXVL-1:128] ← 0
```

VSUBSS (VEX.128 encoded version)

```
DEST[31:0] ←SRC1[31:0] - SRC2[31:0]
DEST[127:32] ←SRC1[127:32]
DEST[MAXVL-1:128] ←0
```

SUBSS (128-bit Legacy SSE version)

```
DEST[31:0] ←DEST[31:0] - SRC[31:0]
DEST[MAXVL-1:32] (Unmodified)
```

Intel C/C++ Compiler Intrinsic Equivalent

VSUBSS __m128 _mm_mask_sub_ss (__m128 s, __mmask8 k, __m128 a, __m128 b);
VSUBSS __m128 _mm_maskz_sub_ss (__mmask8 k, __m128 a, __m128 b);
VSUBSS __m128 _mm_sub_round_ss (__m128 a, __m128 b, int);
VSUBSS __m128 _mm_mask_sub_round_ss (__m128 s, __mmask8 k, __m128 a, __m128 b, int);
VSUBSS __m128 _mm_maskz_sub_round_ss (__mmask8 k, __m128 a, __m128 b, int);
SUBSS __m128 _mm_sub_ss (__m128 a, __m128 b);

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal

Other Exceptions

VEX-encoded instructions, see Exceptions Type 3.
EVEX-encoded instructions, see Exceptions Type E3.

SWAPGS—Swap GS Base Register

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
0F 01 F8	SWAPGS	ZO	Valid	Invalid	Exchanges the current GS base register value with the value contained in MSR address C0000102H.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
ZO	NA	NA	NA	NA

Description

SWAPGS exchanges the current GS base register value with the value contained in MSR address C0000102H (IA32_KERNEL_GS_BASE). The SWAPGS instruction is a privileged instruction intended for use by system software.

When using SYSCALL to implement system calls, there is no kernel stack at the OS entry point. Neither is there a straightforward method to obtain a pointer to kernel structures from which the kernel stack pointer could be read. Thus, the kernel cannot save general purpose registers or reference memory.

By design, SWAPGS does not require any general purpose registers or memory operands. No registers need to be saved before using the instruction. SWAPGS exchanges the CPL 0 data pointer from the IA32_KERNEL_GS_BASE MSR with the GS base register. The kernel can then use the GS prefix on normal memory references to access kernel data structures. Similarly, when the OS kernel is entered using an interrupt or exception (where the kernel stack is already set up), SWAPGS can be used to quickly get a pointer to the kernel data structures.

The IA32_KERNEL_GS_BASE MSR itself is only accessible using RDMSR/WRMSR instructions. Those instructions are only accessible at privilege level 0. The WRMSR instruction ensures that the IA32_KERNEL_GS_BASE MSR contains a canonical address.

Operation

```
IF CS.L ≠ 1 (* Not in 64-Bit Mode *)
    THEN
        #UD; FI;

IF CPL ≠ 0
    THEN #GP(0); FI;

tmp ← GS.base;
GS.base ← IA32_KERNEL_GS_BASE;
IA32_KERNEL_GS_BASE ← tmp;
```

Flags Affected

None

Protected Mode Exceptions

#UD	If Mode ≠ 64-Bit.

Real-Address Mode Exceptions

#UD	If Mode ≠ 64-Bit.

Virtual-8086 Mode Exceptions

#UD	If Mode ≠ 64-Bit.

Compatibility Mode Exceptions

#UD	If Mode ≠ 64-Bit.

64-Bit Mode Exceptions

#GP(0)	If CPL ≠ 0. If the LOCK prefix is used.

SYSCALL—Fast System Call

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
0F 05	SYSCALL	ZO	Valid	Invalid	Fast call to privilege level 0 system procedures.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
ZO	NA	NA	NA	NA

Description

SYSCALL invokes an OS system-call handler at privilege level 0. It does so by loading RIP from the IA32_LSTAR MSR (after saving the address of the instruction following SYSCALL into RCX). (The WRMSR instruction ensures that the IA32_LSTAR MSR always contain a canonical address.)

SYSCALL also saves RFLAGS into R11 and then masks RFLAGS using the IA32_FMASK MSR (MSR address C0000084H); specifically, the processor clears in RFLAGS every bit corresponding to a bit that is set in the IA32_FMASK MSR.

SYSCALL loads the CS and SS selectors with values derived from bits 47:32 of the IA32_STAR MSR. However, the CS and SS descriptor caches are **not** loaded from the descriptors (in GDT or LDT) referenced by those selectors. Instead, the descriptor caches are loaded with fixed values. See the Operation section for details. It is the responsibility of OS software to ensure that the descriptors (in GDT or LDT) referenced by those selector values correspond to the fixed values loaded into the descriptor caches; the SYSCALL instruction does not ensure this correspondence.

The SYSCALL instruction does not save the stack pointer (RSP). If the OS system-call handler will change the stack pointer, it is the responsibility of software to save the previous value of the stack pointer. This might be done prior to executing SYSCALL, with software restoring the stack pointer with the instruction following SYSCALL (which will be executed after SYSRET). Alternatively, the OS system-call handler may save the stack pointer and restore it before executing SYSRET.

Operation

```
IF (CS.L ≠ 1 ) or (IA32_EFER.LMA ≠ 1) or (IA32_EFER.SCE ≠ 1)
(* Not in 64-Bit Mode or SYSCALL/SYSRET not enabled in IA32_EFER *)
    THEN #UD;
FI;

RCX ← RIP;                              (* Will contain address of next instruction *)
RIP ← IA32_LSTAR;
R11 ← RFLAGS;
RFLAGS ← RFLAGS AND NOT(IA32_FMASK);

CS.Selector ← IA32_STAR[47:32] AND FFFCH (* Operating system provides CS; RPL forced to 0 *)
(* Set rest of CS to a fixed value *)
CS.Base ← 0;                            (* Flat segment *)
CS.Limit ← FFFFFH;                      (* With 4-KByte granularity, implies a 4-GByte limit *)
CS.Type ← 11;                           (* Execute/read code, accessed *)
CS.S ← 1;
CS.DPL ← 0;
CS.P ← 1;
CS.L ← 1;                               (* Entry is to 64-bit mode *)
CS.D ← 0;                               (* Required if CS.L = 1 *)
CS.G ← 1;                               (* 4-KByte granularity *)
CPL ← 0;
```

```
SS.Selector ← IA32_STAR[47:32] + 8;          (* SS just above CS *)
(* Set rest of SS to a fixed value *)
SS.Base ← 0;                                 (* Flat segment *)
SS.Limit ← FFFFFH;                           (* With 4-KByte granularity, implies a 4-GByte limit *)
SS.Type ← 3;                                 (* Read/write data, accessed *)
SS.S ← 1;
SS.DPL ← 0;
SS.P ← 1;
SS.B ← 1;                                    (* 32-bit stack segment *)
SS.G ← 1;                                    (* 4-KByte granularity *)
```

Flags Affected

All.

Protected Mode Exceptions

#UD	The SYSCALL instruction is not recognized in protected mode.

Real-Address Mode Exceptions

#UD	The SYSCALL instruction is not recognized in real-address mode.

Virtual-8086 Mode Exceptions

#UD	The SYSCALL instruction is not recognized in virtual-8086 mode.

Compatibility Mode Exceptions

#UD	The SYSCALL instruction is not recognized in compatibility mode.

64-Bit Mode Exceptions

#UD	If IA32_EFER.SCE = 0. If the LOCK prefix is used.

SYSENTER—Fast System Call

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
0F 34	SYSENTER	ZO	Valid	Valid	Fast call to privilege level 0 system procedures.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
ZO	NA	NA	NA	NA

Description

Executes a fast call to a level 0 system procedure or routine. SYSENTER is a companion instruction to SYSEXIT. The instruction is optimized to provide the maximum performance for system calls from user code running at privilege level 3 to operating system or executive procedures running at privilege level 0.

When executed in IA-32e mode, the SYSENTER instruction transitions the logical processor to 64-bit mode; otherwise, the logical processor remains in protected mode.

Prior to executing the SYSENTER instruction, software must specify the privilege level 0 code segment and code entry point, and the privilege level 0 stack segment and stack pointer by writing values to the following MSRs:

- **IA32_SYSENTER_CS** (MSR address 174H) — The lower 16 bits of this MSR are the segment selector for the privilege level 0 code segment. This value is also used to determine the segment selector of the privilege level 0 stack segment (see the Operation section). This value cannot indicate a null selector.
- **IA32_SYSENTER_EIP** (MSR address 176H) — The value of this MSR is loaded into RIP (thus, this value references the first instruction of the selected operating procedure or routine). In protected mode, only bits 31:0 are loaded.
- **IA32_SYSENTER_ESP** (MSR address 175H) — The value of this MSR is loaded into RSP (thus, this value contains the stack pointer for the privilege level 0 stack). This value cannot represent a non-canonical address. In protected mode, only bits 31:0 are loaded.

These MSRs can be read from and written to using RDMSR/WRMSR. The WRMSR instruction ensures that the IA32_SYSENTER_EIP and IA32_SYSENTER_ESP MSRs always contain canonical addresses.

While SYSENTER loads the CS and SS selectors with values derived from the IA32_SYSENTER_CS MSR, the CS and SS descriptor caches are **not** loaded from the descriptors (in GDT or LDT) referenced by those selectors. Instead, the descriptor caches are loaded with fixed values. See the Operation section for details. It is the responsibility of OS software to ensure that the descriptors (in GDT or LDT) referenced by those selector values correspond to the fixed values loaded into the descriptor caches; the SYSENTER instruction does not ensure this correspondence.

The SYSENTER instruction can be invoked from all operating modes except real-address mode.

The SYSENTER and SYSEXIT instructions are companion instructions, but they do not constitute a call/return pair. When executing a SYSENTER instruction, the processor does not save state information for the user code (e.g., the instruction pointer), and neither the SYSENTER nor the SYSEXIT instruction supports passing parameters on the stack.

To use the SYSENTER and SYSEXIT instructions as companion instructions for transitions between privilege level 3 code and privilege level 0 operating system procedures, the following conventions must be followed:

- The segment descriptors for the privilege level 0 code and stack segments and for the privilege level 3 code and stack segments must be contiguous in a descriptor table. This convention allows the processor to compute the segment selectors from the value entered in the SYSENTER_CS_MSR MSR.
- The fast system call "stub" routines executed by user code (typically in shared libraries or DLLs) must save the required return IP and processor state information if a return to the calling procedure is required. Likewise, the operating system or executive procedures called with SYSENTER instructions must have access to and use this saved return and state information when returning to the user code.

The SYSENTER and SYSEXIT instructions were introduced into the IA-32 architecture in the Pentium II processor. The availability of these instructions on a processor is indicated with the SYSENTER/SYSEXIT present (SEP) feature flag returned to the EDX register by the CPUID instruction. An operating system that qualifies the SEP flag must also qualify the processor family and model to ensure that the SYSENTER/SYSEXIT instructions are actually present. For example:

```
IF CPUID SEP bit is set
    THEN IF (Family = 6) and (Model < 3) and (Stepping < 3)
        THEN
            SYSENTER/SYSEXIT_Not_Supported; FI;
        ELSE
            SYSENTER/SYSEXIT_Supported; FI;
FI;
```

When the CPUID instruction is executed on the Pentium Pro processor (model 1), the processor returns a the SEP flag as set, but does not support the SYSENTER/SYSEXIT instructions.

Operation

```
IF CR0.PE = 0 OR IA32_SYSENTER_CS[15:2] = 0 THEN #GP(0); FI;

RFLAGS.VM ← 0;                          (* Ensures protected mode execution *)
RFLAGS.IF ← 0;                          (* Mask interrupts *)
IF in IA-32e mode
    THEN
        RSP ← IA32_SYSENTER_ESP;
        RIP ← IA32_SYSENTER_EIP;
ELSE
        ESP ← IA32_SYSENTER_ESP[31:0];
        EIP ← IA32_SYSENTER_EIP[31:0];
FI;

CS.Selector ← IA32_SYSENTER_CS[15:0] AND FFFCH;
                                        (* Operating system provides CS; RPL forced to 0 *)
(* Set rest of CS to a fixed value *)
CS.Base ← 0;                            (* Flat segment *)
CS.Limit ← FFFFFH;                      (* With 4-KByte granularity, implies a 4-GByte limit *)
CS.Type ← 11;                           (* Execute/read code, accessed *)
CS.S ← 1;
CS.DPL ← 0;
CS.P ← 1;
IF in IA-32e mode
    THEN
        CS.L ← 1;                       (* Entry is to 64-bit mode *)
        CS.D ← 0;                       (* Required if CS.L = 1 *)
    ELSE
        CS.L ← 0;
        CS.D ← 1;                       (* 32-bit code segment*)
FI;
CS.G ← 1;                               (* 4-KByte granularity *)
CPL ← 0;

SS.Selector ← CS.Selector + 8;          (* SS just above CS *)
(* Set rest of SS to a fixed value *)
SS.Base ← 0;                            (* Flat segment *)
SS.Limit ← FFFFFH;                      (* With 4-KByte granularity, implies a 4-GByte limit *)
SS.Type ← 3;                            (* Read/write data, accessed *)
```

```
SS.S ← 1;
SS.DPL ← 0;
SS.P ← 1;
SS.B ← 1;                    (* 32-bit stack segment*)
SS.G ← 1;                    (* 4-KByte granularity *)
```

Flags Affected

VM, IF (see Operation above)

Protected Mode Exceptions

#GP(0)	If IA32_SYSENTER_CS[15:2] = 0.
#UD	If the LOCK prefix is used.

Real-Address Mode Exceptions

#GP	The SYSENTER instruction is not recognized in real-address mode.
#UD	If the LOCK prefix is used.

Virtual-8086 Mode Exceptions

Same exceptions as in protected mode.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

Same exceptions as in protected mode.

SYSEXIT—Fast Return from Fast System Call

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
0F 35	SYSEXIT	ZO	Valid	Valid	Fast return to privilege level 3 user code.
REX.W + 0F 35	SYSEXIT	ZO	Valid	Valid	Fast return to 64-bit mode privilege level 3 user code.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
ZO	NA	NA	NA	NA

Description

Executes a fast return to privilege level 3 user code. SYSEXIT is a companion instruction to the SYSENTER instruction. The instruction is optimized to provide the maximum performance for returns from system procedures executing at protections levels 0 to user procedures executing at protection level 3. It must be executed from code executing at privilege level 0.

With a 64-bit operand size, SYSEXIT remains in 64-bit mode; otherwise, it either enters compatibility mode (if the logical processor is in IA-32e mode) or remains in protected mode (if it is not).

Prior to executing SYSEXIT, software must specify the privilege level 3 code segment and code entry point, and the privilege level 3 stack segment and stack pointer by writing values into the following MSR and general-purpose registers:

- **IA32_SYSENTER_CS** (MSR address 174H) — Contains a 32-bit value that is used to determine the segment selectors for the privilege level 3 code and stack segments (see the Operation section)
- **RDX** — The canonical address in this register is loaded into RIP (thus, this value references the first instruction to be executed in the user code). If the return is not to 64-bit mode, only bits 31:0 are loaded.
- **ECX** — The canonical address in this register is loaded into RSP (thus, this value contains the stack pointer for the privilege level 3 stack). If the return is not to 64-bit mode, only bits 31:0 are loaded.

The IA32_SYSENTER_CS MSR can be read from and written to using RDMSR and WRMSR.

While SYSEXIT loads the CS and SS selectors with values derived from the IA32_SYSENTER_CS MSR, the CS and SS descriptor caches are **not** loaded from the descriptors (in GDT or LDT) referenced by those selectors. Instead, the descriptor caches are loaded with fixed values. See the Operation section for details. It is the responsibility of OS software to ensure that the descriptors (in GDT or LDT) referenced by those selector values correspond to the fixed values loaded into the descriptor caches; the SYSEXIT instruction does not ensure this correspondence.

The SYSEXIT instruction can be invoked from all operating modes except real-address mode and virtual-8086 mode.

The SYSENTER and SYSEXIT instructions were introduced into the IA-32 architecture in the Pentium II processor. The availability of these instructions on a processor is indicated with the SYSENTER/SYSEXIT present (SEP) feature flag returned to the EDX register by the CPUID instruction. An operating system that qualifies the SEP flag must also qualify the processor family and model to ensure that the SYSENTER/SYSEXIT instructions are actually present. For example:

```
IF CPUID SEP bit is set
    THEN IF (Family = 6) and (Model < 3) and (Stepping < 3)
        THEN
            SYSENTER/SYSEXIT_Not_Supported; FI;
        ELSE
            SYSENTER/SYSEXIT_Supported; FI;
FI;
```

When the CPUID instruction is executed on the Pentium Pro processor (model 1), the processor returns a the SEP flag as set, but does not support the SYSENTER/SYSEXIT instructions.

Operation

```
IF IA32_SYSENTER_CS[15:2] = 0 OR CR0.PE = 0 OR CPL ≠ 0 THEN #GP(0); FI;

IF operand size is 64-bit
    THEN    (* Return to 64-bit mode *)
        RSP ← RCX;
        RIP ← RDX;
    ELSE    (* Return to protected mode or compatibility mode *)
        RSP ← ECX;
        RIP ← EDX;
FI;

IF operand size is 64-bit                   (* Operating system provides CS; RPL forced to 3 *)
    THEN CS.Selector ← IA32_SYSENTER_CS[15:0] + 32;
    ELSE CS.Selector ← IA32_SYSENTER_CS[15:0] + 16;
FI;
CS.Selector ← CS.Selector OR 3;             (* RPL forced to 3 *)
(* Set rest of CS to a fixed value *)
CS.Base ← 0;                                (* Flat segment *)
CS.Limit ← FFFFFH;                          (* With 4-KByte granularity, implies a 4-GByte limit *)
CS.Type ← 11;                               (* Execute/read code, accessed *)
CS.S ← 1;
CS.DPL ← 3;
CS.P ← 1;
IF operand size is 64-bit
    THEN    (* return to 64-bit mode *)
        CS.L ← 1;                           (* 64-bit code segment *)
        CS.D ← 0;                           (* Required if CS.L = 1 *)
    ELSE    (* return to protected mode or compatibility mode *)
        CS.L ← 0;
        CS.D ← 1;                           (* 32-bit code segment*)
FI;
CS.G ← 1;                                   (* 4-KByte granularity *)
CPL ← 3;

SS.Selector ← CS.Selector + 8;              (* SS just above CS *)
(* Set rest of SS to a fixed value *)
SS.Base ← 0;                                (* Flat segment *)
SS.Limit ← FFFFFH;                          (* With 4-KByte granularity, implies a 4-GByte limit *)
SS.Type ← 3;                                (* Read/write data, accessed *)
SS.S ← 1;
SS.DPL ← 3;
SS.P ← 1;
SS.B ← 1;                                   (* 32-bit stack segment*)
SS.G ← 1;                                   (* 4-KByte granularity *)
```

Flags Affected

None.

Protected Mode Exceptions

#GP(0)	If IA32_SYSENTER_CS[15:2] = 0. If CPL ≠ 0.
#UD	If the LOCK prefix is used.

Real-Address Mode Exceptions

#GP	The SYSEXIT instruction is not recognized in real-address mode.
#UD	If the LOCK prefix is used.

Virtual-8086 Mode Exceptions

#GP(0)	The SYSEXIT instruction is not recognized in virtual-8086 mode.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#GP(0)	If IA32_SYSENTER_CS = 0.
	If CPL ≠ 0.
	If RCX or RDX contains a non-canonical address.
#UD	If the LOCK prefix is used.

SYSRET—Return From Fast System Call

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
0F 07	SYSRET	ZO	Valid	Invalid	Return to compatibility mode from fast system call
REX.W + 0F 07	SYSRET	ZO	Valid	Invalid	Return to 64-bit mode from fast system call

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
ZO	NA	NA	NA	NA

Description

SYSRET is a companion instruction to the SYSCALL instruction. It returns from an OS system-call handler to user code at privilege level 3. It does so by loading RIP from RCX and loading RFLAGS from R11.[1] With a 64-bit operand size, SYSRET remains in 64-bit mode; otherwise, it enters compatibility mode and only the low 32 bits of the registers are loaded.

SYSRET loads the CS and SS selectors with values derived from bits 63:48 of the IA32_STAR MSR. However, the CS and SS descriptor caches are **not** loaded from the descriptors (in GDT or LDT) referenced by those selectors. Instead, the descriptor caches are loaded with fixed values. See the Operation section for details. It is the responsibility of OS software to ensure that the descriptors (in GDT or LDT) referenced by those selector values correspond to the fixed values loaded into the descriptor caches; the SYSRET instruction does not ensure this correspondence.

The SYSRET instruction does not modify the stack pointer (ESP or RSP). For that reason, it is necessary for software to switch to the user stack. The OS may load the user stack pointer (if it was saved after SYSCALL) before executing SYSRET; alternatively, user code may load the stack pointer (if it was saved before SYSCALL) after receiving control from SYSRET.

If the OS loads the stack pointer before executing SYSRET, it must ensure that the handler of any interrupt or exception delivered between restoring the stack pointer and successful execution of SYSRET is not invoked with the user stack. It can do so using approaches such as the following:

- External interrupts. The OS can prevent an external interrupt from being delivered by clearing EFLAGS.IF before loading the user stack pointer.
- Nonmaskable interrupts (NMIs). The OS can ensure that the NMI handler is invoked with the correct stack by using the interrupt stack table (IST) mechanism for gate 2 (NMI) in the IDT (see Section 6.14.5, "Interrupt Stack Table," in *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*).
- General-protection exceptions (#GP). The SYSRET instruction generates #GP(0) if the value of RCX is not canonical. The OS can address this possibility using one or more of the following approaches:
  - — Confirming that the value of RCX is canonical before executing SYSRET.
  - — Using paging to ensure that the SYSCALL instruction will never save a non-canonical value into RCX.
  - — Using the IST mechanism for gate 13 (#GP) in the IDT.

Operation

```
IF (CS.L ≠ 1 ) or (IA32_EFER.LMA ≠ 1) or (IA32_EFER.SCE ≠ 1)
(* Not in 64-Bit Mode or SYSCALL/SYSRET not enabled in IA32_EFER *)
    THEN #UD; FI;
IF (CPL ≠ 0) OR (RCX is not canonical) THEN #GP(0); FI;
```

1. Regardless of the value of R11, the RF and VM flags are always 0 in RFLAGS after execution of SYSRET. In addition, all reserved bits in RFLAGS retain the fixed values.

```
IF (operand size is 64-bit)
    THEN (* Return to 64-Bit Mode *)
        RIP ← RCX;
    ELSE (* Return to Compatibility Mode *)
        RIP ← ECX;
FI;
RFLAGS ← (R11 & 3C7FD7H) | 2;          (* Clear RF, VM, reserved bits; set bit 2 *)

IF (operand size is 64-bit)
    THEN CS.Selector ← IA32_STAR[63:48]+16;
    ELSE CS.Selector ← IA32_STAR[63:48];
FI;
CS.Selector ← CS.Selector OR 3;        (* RPL forced to 3 *)
(* Set rest of CS to a fixed value *)
CS.Base ← 0;                           (* Flat segment *)
CS.Limit ← FFFFFH;                     (* With 4-KByte granularity, implies a 4-GByte limit *)
CS.Type ← 11;                          (* Execute/read code, accessed *)
CS.S ← 1;
CS.DPL ← 3;
CS.P ← 1;
IF (operand size is 64-bit)
    THEN (* Return to 64-Bit Mode *)
        CS.L ← 1;                      (* 64-bit code segment *)
        CS.D ← 0;                      (* Required if CS.L = 1 *)
    ELSE (* Return to Compatibility Mode *)
        CS.L ← 0;                      (* Compatibility mode *)
        CS.D ← 1;                      (* 32-bit code segment *)
FI;
CS.G ← 1;                              (* 4-KByte granularity *)
CPL ← 3;

SS.Selector ← (IA32_STAR[63:48]+8) OR 3;  (* RPL forced to 3 *)
(* Set rest of SS to a fixed value *)
SS.Base ← 0;                           (* Flat segment *)
SS.Limit ← FFFFFH;                     (* With 4-KByte granularity, implies a 4-GByte limit *)
SS.Type ← 3;                           (* Read/write data, accessed *)
SS.S ← 1;
SS.DPL ← 3;
SS.P ← 1;
SS.B ← 1;                              (* 32-bit stack segment*)
SS.G ← 1;                              (* 4-KByte granularity *)
```

Flags Affected

All.

Protected Mode Exceptions

#UD	The SYSRET instruction is not recognized in protected mode.

Real-Address Mode Exceptions

#UD	The SYSRET instruction is not recognized in real-address mode.

Virtual-8086 Mode Exceptions

#UD	The SYSRET instruction is not recognized in virtual-8086 mode.

Compatibility Mode Exceptions

#UD	The SYSRET instruction is not recognized in compatibility mode.

64-Bit Mode Exceptions

#UD	If IA32_EFER.SCE = 0.
	If the LOCK prefix is used.
#GP(0)	If CPL ≠ 0.
	If RCX contains a non-canonical address.

TEST—Logical Compare

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
A8 *ib*	TEST AL, i*mm8*	I	Valid	Valid	AND *imm8* with AL; set SF, ZF, PF according to result.
A9 *iw*	TEST AX, i*mm16*	I	Valid	Valid	AND *imm16* with AX; set SF, ZF, PF according to result.
A9 *id*	TEST EAX, i*mm32*	I	Valid	Valid	AND *imm32* with EAX; set SF, ZF, PF according to result.
REX.W + A9 *id*	TEST RAX, i*mm32*	I	Valid	N.E.	AND *imm32* sign-extended to 64-bits with RAX; set SF, ZF, PF according to result.
F6 /0 *ib*	TEST *r/m8, imm8*	MI	Valid	Valid	AND *imm8* with *r/m8*; set SF, ZF, PF according to result.
REX + F6 /0 *ib*	TEST *r/m8*, imm8*	MI	Valid	N.E.	AND *imm8* with *r/m8*; set SF, ZF, PF according to result.
F7 /0 *iw*	TEST *r/m16, imm16*	MI	Valid	Valid	AND *imm16* with *r/m16*; set SF, ZF, PF according to result.
F7 /0 *id*	TEST *r/m32, imm32*	MI	Valid	Valid	AND *imm32* with *r/m32*; set SF, ZF, PF according to result.
REX.W + F7 /0 *id*	TEST *r/m64, imm32*	MI	Valid	N.E.	AND *imm32* sign-extended to 64-bits with *r/m64*; set SF, ZF, PF according to result.
84 /*r*	TEST *r/m8, r8*	MR	Valid	Valid	AND *r8* with *r/m8*; set SF, ZF, PF according to result.
REX + 84 /*r*	TEST *r/m8*, r8**	MR	Valid	N.E.	AND *r8* with *r/m8*; set SF, ZF, PF according to result.
85 /*r*	TEST *r/m16, r16*	MR	Valid	Valid	AND *r16* with *r/m16*; set SF, ZF, PF according to result.
85 /*r*	TEST *r/m32, r32*	MR	Valid	Valid	AND *r32* with *r/m32*; set SF, ZF, PF according to result.
REX.W + 85 /*r*	TEST *r/m64, r64*	MR	Valid	N.E.	AND *r64* with *r/m64*; set SF, ZF, PF according to result.

NOTES:

* In 64-bit mode, r/m8 can not be encoded to access the following byte registers if a REX prefix is used: AH, BH, CH, DH.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
I	AL/AX/EAX/RAX	imm8/16/32	NA	NA
MI	ModRM:r/m (r)	imm8/16/32	NA	NA
MR	ModRM:r/m (r)	ModRM:reg (r)	NA	NA

Description

Computes the bit-wise logical AND of first operand (source 1 operand) and the second operand (source 2 operand) and sets the SF, ZF, and PF status flags according to the result. The result is then discarded.

In 64-bit mode, using a REX prefix in the form of REX.R permits access to additional registers (R8-R15). Using a REX prefix in the form of REX.W promotes operation to 64 bits. See the summary chart at the beginning of this section for encoding data and limits.

Operation

```
TEMP ← SRC1 AND SRC2;
SF ← MSB(TEMP);

IF TEMP = 0
    THEN ZF ← 1;
    ELSE ZF ← 0;
FI:

PF ← BitwiseXNOR(TEMP[0:7]);
CF ← 0;
OF ← 0;
(* AF is undefined *)
```

Flags Affected

The OF and CF flags are set to 0. The SF, ZF, and PF flags are set according to the result (see the "Operation" section above). The state of the AF flag is undefined.

Protected Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
	If the DS, ES, FS, or GS register contains a NULL segment selector.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If the LOCK prefix is used.

Real-Address Mode Exceptions

#GP	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS	If a memory operand effective address is outside the SS segment limit.
#UD	If the LOCK prefix is used.

Virtual-8086 Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.
#UD	If the LOCK prefix is used.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If the LOCK prefix is used.

TZCNT — Count the Number of Trailing Zero Bits

Opcode/ Instruction	Op/ En	64/32 -bit Mode	CPUID Feature Flag	Description
F3 0F BC /r TZCNT *r16, r/m16*	A	V/V	BMI1	Count the number of trailing zero bits in *r/m16*, return result in *r16*.
F3 0F BC /r TZCNT *r32, r/m32*	A	V/V	BMI1	Count the number of trailing zero bits in *r/m32*, return result in *r32*.
F3 REX.W 0F BC /r TZCNT *r64, r/m64*	A	V/N.E.	BMI1	Count the number of trailing zero bits in *r/m64*, return result in *r64*.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
A	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

Description

TZCNT counts the number of trailing least significant zero bits in source operand (second operand) and returns the result in destination operand (first operand). TZCNT is an extension of the BSF instruction. The key difference between TZCNT and BSF instruction is that TZCNT provides operand size as output when source operand is zero while in the case of BSF instruction, if source operand is zero, the content of destination operand are undefined. On processors that do not support TZCNT, the instruction byte encoding is executed as BSF.

Operation

```
temp ← 0
DEST ← 0
DO WHILE ( (temp < OperandSize) and (SRC[ temp] = 0) )

    temp ← temp +1
    DEST ← DEST+ 1
OD

IF DEST = OperandSize
    CF ← 1
ELSE
    CF ← 0
FI

IF DEST = 0
    ZF ← 1
ELSE
    ZF ← 0
FI
```

Flags Affected

ZF is set to 1 in case of zero output (least significant bit of the source is set), and to 0 otherwise, CF is set to 1 if the input was zero and cleared otherwise. OF, SF, PF and AF flags are undefined.

Intel C/C++ Compiler Intrinsic Equivalent

TZCNT: unsigned __int32 _tzcnt_u32(unsigned __int32 src);

TZCNT: unsigned __int64 _tzcnt_u64(unsigned __int64 src);

Protected Mode Exceptions

#GP(0)	For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments. If the DS, ES, FS, or GS register is used to access memory and it contains a null segment selector.
#SS(0)	For an illegal address in the SS segment.
#PF (fault-code)	For a page fault.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

Real-Address Mode Exceptions

#GP(0)	If any part of the operand lies outside of the effective address space from 0 to 0FFFFH.
#SS(0)	For an illegal address in the SS segment.

Virtual 8086 Mode Exceptions

#GP(0)	If any part of the operand lies outside of the effective address space from 0 to 0FFFFH.
#SS(0)	For an illegal address in the SS segment.
#PF (fault-code)	For a page fault.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

Compatibility Mode Exceptions

Same exceptions as in Protected Mode.

64-Bit Mode Exceptions

#GP(0)	If the memory address is in a non-canonical form.
#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#PF (fault-code)	For a page fault.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

UCOMISD—Unordered Compare Scalar Double-Precision Floating-Point Values and Set EFLAGS

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 2E /r UCOMISD xmm1, xmm2/m64	A	V/V	SSE2	Compare low double-precision floating-point values in xmm1 and xmm2/mem64 and set the EFLAGS flags accordingly.
VEX.LIG.66.0F.WIG 2E /r VUCOMISD xmm1, xmm2/m64	A	V/V	AVX	Compare low double-precision floating-point values in xmm1 and xmm2/mem64 and set the EFLAGS flags accordingly.
EVEX.LIG.66.0F.W1 2E /r VUCOMISD xmm1, xmm2/m64{sae}	B	V/V	AVX512F	Compare low double-precision floating-point values in xmm1 and xmm2/m64 and set the EFLAGS flags accordingly.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	NA	ModRM:reg (r)	ModRM:r/m (r)	NA	NA
B	Tuple1 Scalar	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

Description

Performs an unordered compare of the double-precision floating-point values in the low quadwords of operand 1 (first operand) and operand 2 (second operand), and sets the ZF, PF, and CF flags in the EFLAGS register according to the result (unordered, greater than, less than, or equal). The OF, SF and AF flags in the EFLAGS register are set to 0. The unordered result is returned if either source operand is a NaN (QNaN or SNaN).

Operand 1 is an XMM register; operand 2 can be an XMM register or a 64 bit memory

location.

The UCOMISD instruction differs from the COMISD instruction in that it signals a SIMD floating-point invalid operation exception (#I) only when a source operand is an SNaN. The COMISD instruction signals an invalid numeric exception only if a source operand is either an SNaN or a QNaN.

The EFLAGS register is not updated if an unmasked SIMD floating-point exception is generated.

Note: VEX.vvvv and EVEX.vvvv are reserved and must be 1111b, otherwise instructions will #UD.

Software should ensure VCOMISD is encoded with VEX.L=0. Encoding VCOMISD with VEX.L=1 may encounter unpredictable behavior across different processor generations.

Operation

(V)UCOMISD (all versions)

```
RESULT ← UnorderedCompare(DEST[63:0] <> SRC[63:0]) {
(* Set EFLAGS *) CASE (RESULT) OF
    UNORDERED: ZF,PF,CF ← 111;
    GREATER_THAN: ZF,PF,CF ← 000;
    LESS_THAN: ZF,PF,CF ← 001;
    EQUAL: ZF,PF,CF ← 100;
ESAC;
OF, AF, SF ← 0; }
```

Intel C/C++ Compiler Intrinsic Equivalent

VUCOMISD int _mm_comi_round_sd(__m128d a, __m128d b, int imm, int sae);
UCOMISD int _mm_ucomieq_sd(__m128d a, __m128d b)
UCOMISD int _mm_ucomilt_sd(__m128d a, __m128d b)
UCOMISD int _mm_ucomile_sd(__m128d a, __m128d b)
UCOMISD int _mm_ucomigt_sd(__m128d a, __m128d b)
UCOMISD int _mm_ucomige_sd(__m128d a, __m128d b)
UCOMISD int _mm_ucomineq_sd(__m128d a, __m128d b)

SIMD Floating-Point Exceptions

Invalid (if SNaN operands), Denormal

Other Exceptions

VEX-encoded instructions, see Exceptions Type 3; additionally

#UD	If VEX.vvvv != 1111B.

EVEX-encoded instructions, see Exceptions Type E3NF.

UCOMISS—Unordered Compare Scalar Single-Precision Floating-Point Values and Set EFLAGS

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F 2E /r UCOMISS xmm1, xmm2/m32	A	V/V	SSE	Compare low single-precision floating-point values in xmm1 and xmm2/mem32 and set the EFLAGS flags accordingly.
VEX.LIG.0F.WIG 2E /r VUCOMISS xmm1, xmm2/m32	A	V/V	AVX	Compare low single-precision floating-point values in xmm1 and xmm2/mem32 and set the EFLAGS flags accordingly.
EVEX.LIG.0F.W0 2E /r VUCOMISS xmm1, xmm2/m32{sae}	B	V/V	AVX512F	Compare low single-precision floating-point values in xmm1 and xmm2/mem32 and set the EFLAGS flags accordingly.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	NA	ModRM:reg (r)	ModRM:r/m (r)	NA	NA
B	Tuple1 Scalar	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

Description

Compares the single-precision floating-point values in the low doublewords of operand 1 (first operand) and operand 2 (second operand), and sets the ZF, PF, and CF flags in the EFLAGS register according to the result (unordered, greater than, less than, or equal). The OF, SF and AF flags in the EFLAGS register are set to 0. The unordered result is returned if either source operand is a NaN (QNaN or SNaN).

Operand 1 is an XMM register; operand 2 can be an XMM register or a 32 bit memory location.

The UCOMISS instruction differs from the COMISS instruction in that it signals a SIMD floating-point invalid operation exception (#I) only if a source operand is an SNaN. The COMISS instruction signals an invalid numeric exception when a source operand is either a QNaN or SNaN.

The EFLAGS register is not updated if an unmasked SIMD floating-point exception is generated.

Note: VEX.vvvv and EVEX.vvvv are reserved and must be 1111b, otherwise instructions will #UD.

Software should ensure VCOMISS is encoded with VEX.L=0. Encoding VCOMISS with VEX.L=1 may encounter unpredictable behavior across different processor generations.

Operation

(V)UCOMISS (all versions)

```
RESULT ← UnorderedCompare(DEST[31:0] <> SRC[31:0]) {
(* Set EFLAGS *) CASE (RESULT) OF
    UNORDERED: ZF,PF,CF ← 111;
    GREATER_THAN: ZF,PF,CF ← 000;
    LESS_THAN: ZF,PF,CF ← 001;
    EQUAL: ZF,PF,CF ← 100;
ESAC;
OF, AF, SF ← 0; }
```

Intel C/C++ Compiler Intrinsic Equivalent

VUCOMISS int _mm_comi_round_ss(__m128 a, __m128 b, int imm, int sae);
UCOMISS int _mm_ucomieq_ss(__m128 a, __m128 b);
UCOMISS int _mm_ucomilt_ss(__m128 a, __m128 b);
UCOMISS int _mm_ucomile_ss(__m128 a, __m128 b);
UCOMISS int _mm_ucomigt_ss(__m128 a, __m128 b);
UCOMISS int _mm_ucomige_ss(__m128 a, __m128 b);
UCOMISS int _mm_ucomineq_ss(__m128 a, __m128 b);

SIMD Floating-Point Exceptions

Invalid (if SNaN Operands), Denormal

Other Exceptions

VEX-encoded instructions, see Exceptions Type 3; additionally

#UD	If VEX.vvvv != 1111B.

EVEX-encoded instructions, see Exceptions Type E3NF.

UD—Undefined Instruction

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
0F FF	UD0	ZO	Valid	Valid	Raise invalid opcode exception.
0F B9 /r	UD1 *r32, r/m32*	RM	Valid	Valid	Raise invalid opcode exception.
0F 0B	UD2	ZO	Valid	Valid	Raise invalid opcode exception.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
ZO	NA	NA	NA	NA
RM	ModRM:reg (r)	ModRM:r/m (r)	NA	NA

Description

Generates an invalid opcode exception. This instruction is provided for software testing to explicitly generate an invalid opcode exception. The opcodes for this instruction are reserved for this purpose.

Other than raising the invalid opcode exception, this instruction has no effect on processor state or memory.

Even though it is the execution of the UD instruction that causes the invalid opcode exception, the instruction pointer saved by delivery of the exception references the UD instruction (and not the following instruction).

This instruction's operation is the same in non-64-bit modes and 64-bit mode.

Operation

```
#UD (* Generates invalid opcode exception *);
```

Flags Affected

None.

Exceptions (All Operating Modes)

#UD	Raises an invalid opcode exception in all operating modes.

UNPCKHPD—Unpack and Interleave High Packed Double-Precision Floating-Point Values

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 15 /r UNPCKHPD xmm1, xmm2/m128	A	V/V	SSE2	Unpacks and Interleaves double-precision floating-point values from high quadwords of xmm1 and xmm2/m128.
VEX.NDS.128.66.0F.WIG 15 /r VUNPCKHPD xmm1,xmm2, xmm3/m128	B	V/V	AVX	Unpacks and Interleaves double-precision floating-point values from high quadwords of xmm2 and xmm3/m128.
VEX.NDS.256.66.0F.WIG 15 /r VUNPCKHPD ymm1,ymm2, ymm3/m256	B	V/V	AVX	Unpacks and Interleaves double-precision floating-point values from high quadwords of ymm2 and ymm3/m256.
EVEX.NDS.128.66.0F.W1 15 /r VUNPCKHPD xmm1 {k1}{z}, xmm2, xmm3/m128/m64bcst	C	V/V	AVX512VL AVX512F	Unpacks and Interleaves double precision floating-point values from high quadwords of xmm2 and xmm3/m128/m64bcst subject to writemask k1.
EVEX.NDS.256.66.0F.W1 15 /r VUNPCKHPD ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst	C	V/V	AVX512VL AVX512F	Unpacks and Interleaves double precision floating-point values from high quadwords of ymm2 and ymm3/m256/m64bcst subject to writemask k1.
EVEX.NDS.512.66.0F.W1 15 /r VUNPCKHPD zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst	C	V/V	AVX512F	Unpacks and Interleaves double-precision floating-point values from high quadwords of zmm2 and zmm3/m512/m64bcst subject to writemask k1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	NA	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
B	NA	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA
C	Full Vector	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA

Description

Performs an interleaved unpack of the high double-precision floating-point values from the first source operand and the second source operand. See Figure 4-15 in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2B.

128-bit Legacy SSE version: The second source can be an XMM register or an 128-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (MAXVL-1:128) of the corresponding ZMM register destination are unmodified. When unpacking from a memory operand, an implementation may fetch only the appropriate 64 bits; however, alignment to 16-byte boundary and normal segment checking will still be enforced.

VEX.128 encoded version: The first source operand is a XMM register. The second source operand can be a XMM register or a 128-bit memory location. The destination operand is a XMM register. The upper bits (MAXVL-1:128) of the corresponding ZMM register destination are zeroed.

VEX.256 encoded version: The first source operand is a YMM register. The second source operand can be a YMM register or a 256-bit memory location. The destination operand is a YMM register.

EVEX.512 encoded version: The first source operand is a ZMM register. The second source operand is a ZMM register, a 512-bit memory location, or a 512-bit vector broadcasted from a 64-bit memory location. The destination operand is a ZMM register, conditionally updated using writemask k1.

EVEX.256 encoded version: The first source operand is a YMM register. The second source operand is a YMM register, a 256-bit memory location, or a 256-bit vector broadcasted from a 64-bit memory location. The destination operand is a YMM register, conditionally updated using writemask k1.

EVEX.128 encoded version: The first source operand is a XMM register. The second source operand is a XMM register, a 128-bit memory location, or a 128-bit vector broadcasted from a 64-bit memory location. The destination operand is a XMM register, conditionally updated using writemask k1.

Operation

VUNPCKHPD (EVEX encoded versions when SRC2 is a register)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
IF VL >= 128
    TMP_DEST[63:0] ← SRC1[127:64]
    TMP_DEST[127:64] ← SRC2[127:64]
FI;
IF VL >= 256
    TMP_DEST[191:128] ← SRC1[255:192]
    TMP_DEST[255:192] ← SRC2[255:192]
FI;
IF VL >= 512
    TMP_DEST[319:256] ← SRC1[383:320]
    TMP_DEST[383:320] ← SRC2[383:320]
    TMP_DEST[447:384] ← SRC1[511:448]
    TMP_DEST[511:448] ← SRC2[511:448]
FI;
FOR j ← 0 TO KL-1
    i ← j * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+63:i] ← TMP_DEST[i+63:i]
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE *zeroing-masking*              ; zeroing-masking
                    DEST[i+63:i] ← 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0
```

VUNPCKHPD (EVEX encoded version when SRC2 is memory)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j ← 0 TO KL-1
    i ← j * 64
    IF (EVEX.b = 1)
        THEN TMP_SRC2[i+63:i] ← SRC2[63:0]
        ELSE TMP_SRC2[i+63:i] ← SRC2[i+63:i]
    FI;
ENDFOR;
IF VL >= 128
    TMP_DEST[63:0] ← SRC1[127:64]
    TMP_DEST[127:64] ← TMP_SRC2[127:64]
FI;
IF VL >= 256
    TMP_DEST[191:128] ← SRC1[255:192]
    TMP_DEST[255:192] ← TMP_SRC2[255:192]
FI;
IF VL >= 512
    TMP_DEST[319:256] ← SRC1[383:320]
    TMP_DEST[383:320] ← TMP_SRC2[383:320]
    TMP_DEST[447:384] ← SRC1[511:448]
    TMP_DEST[511:448] ← TMP_SRC2[511:448]
FI;

FOR j ← 0 TO KL-1
    i ← j * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+63:i] ← TMP_DEST[i+63:i]
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE *zeroing-masking*              ; zeroing-masking
                    DEST[i+63:i] ← 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0
```

VUNPCKHPD (VEX.256 encoded version)

```
DEST[63:0] ←SRC1[127:64]
DEST[127:64] ←SRC2[127:64]
DEST[191:128]←SRC1[255:192]
DEST[255:192]←SRC2[255:192]
DEST[MAXVL-1:256] ←0
```

VUNPCKHPD (VEX.128 encoded version)

```
DEST[63:0] ←SRC1[127:64]
DEST[127:64] ←SRC2[127:64]
DEST[MAXVL-1:128] ←0
```

UNPCKHPD (128-bit Legacy SSE version)

```
DEST[63:0] ←SRC1[127:64]
DEST[127:64] ←SRC2[127:64]
DEST[MAXVL-1:128] (Unmodified)
```

Intel C/C++ Compiler Intrinsic Equivalent

VUNPCKHPD __m512d _mm512_unpackhi_pd(__m512d a, __m512d b);
VUNPCKHPD __m512d _mm512_mask_unpackhi_pd(__m512d s, __mmask8 k, __m512d a, __m512d b);
VUNPCKHPD __m512d _mm512_maskz_unpackhi_pd(__mmask8 k, __m512d a, __m512d b);
VUNPCKHPD __m256d _mm256_unpackhi_pd(__m256d a, __m256d b)
VUNPCKHPD __m256d _mm256_mask_unpackhi_pd(__m256d s, __mmask8 k, __m256d a, __m256d b);
VUNPCKHPD __m256d _mm256_maskz_unpackhi_pd(__mmask8 k, __m256d a, __m256d b);
UNPCKHPD __m128d _mm_unpackhi_pd(__m128d a, __m128d b)
VUNPCKHPD __m128d _mm_mask_unpackhi_pd(__m128d s, __mmask8 k, __m128d a, __m128d b);
VUNPCKHPD __m128d _mm_maskz_unpackhi_pd(__mmask8 k, __m128d a, __m128d b);

SIMD Floating-Point Exceptions

None

Other Exceptions

Non-EVEX-encoded instructions, see Exceptions Type 4.
EVEX-encoded instructions, see Exceptions Type E4NF.

UNPCKHPS—Unpack and Interleave High Packed Single-Precision Floating-Point Values

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F 15 /r UNPCKHPS xmm1, xmm2/m128	A	V/V	SSE	Unpacks and Interleaves single-precision floating-point values from high quadwords of xmm1 and xmm2/m128.
VEX.NDS.128.0F.WIG 15 /r VUNPCKHPS xmm1, xmm2, xmm3/m128	B	V/V	AVX	Unpacks and Interleaves single-precision floating-point values from high quadwords of xmm2 and xmm3/m128.
VEX.NDS.256.0F.WIG 15 /r VUNPCKHPS ymm1, ymm2, ymm3/m256	B	V/V	AVX	Unpacks and Interleaves single-precision floating-point values from high quadwords of ymm2 and ymm3/m256.
EVEX.NDS.128.0F.W0 15 /r VUNPCKHPS xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst	C	V/V	AVX512VL AVX512F	Unpacks and Interleaves single-precision floating-point values from high quadwords of xmm2 and xmm3/m128/m32bcst and write result to xmm1 subject to writemask k1.
EVEX.NDS.256.0F.W0 15 /r VUNPCKHPS ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst	C	V/V	AVX512VL AVX512F	Unpacks and Interleaves single-precision floating-point values from high quadwords of ymm2 and ymm3/m256/m32bcst and write result to ymm1 subject to writemask k1.
EVEX.NDS.512.0F.W0 15 /r VUNPCKHPS zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst	C	V/V	AVX512F	Unpacks and Interleaves single-precision floating-point values from high quadwords of zmm2 and zmm3/m512/m32bcst and write result to zmm1 subject to writemask k1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	NA	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
B	NA	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA
C	Full Vector	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA

Description

Performs an interleaved unpack of the high single-precision floating-point values from the first source operand and the second source operand.

128-bit Legacy SSE version: The second source can be an XMM register or an 128-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (MAXVL-1:128) of the corresponding ZMM register destination are unmodified. When unpacking from a memory operand, an implementation may fetch only the appropriate 64 bits; however, alignment to 16-byte boundary and normal segment checking will still be enforced.

VEX.128 encoded version: The first source operand is a XMM register. The second source operand can be a XMM register or a 128-bit memory location. The destination operand is a XMM register. The upper bits (MAXVL-1:128) of the corresponding ZMM register destination are zeroed.

VEX.256 encoded version: The second source operand is an YMM register or an 256-bit memory location. The first source operand and destination operands are YMM registers.

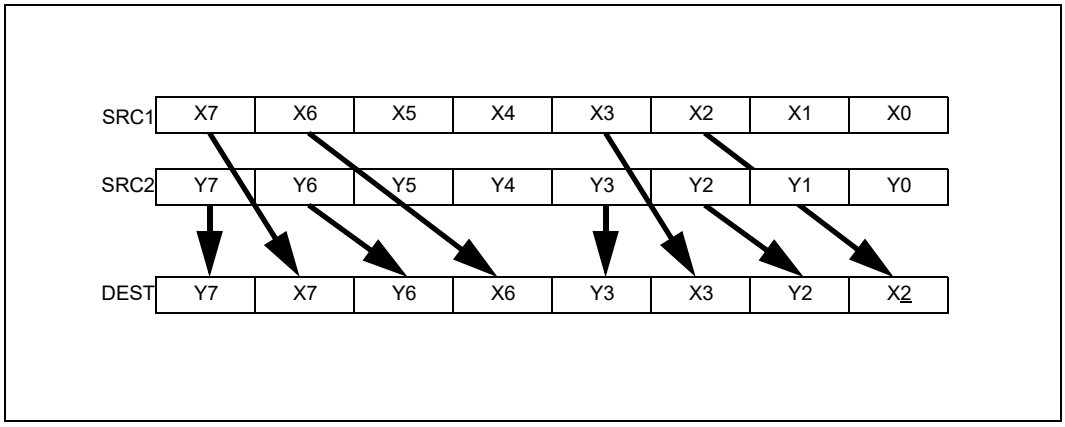


Figure 4-27. VUNPCKHPS Operation

EVEX.512 encoded version: The first source operand is a ZMM register. The second source operand is a ZMM register, a 512-bit memory location, or a 512-bit vector broadcasted from a 32-bit memory location. The destination operand is a ZMM register, conditionally updated using writemask k1.

EVEX.256 encoded version: The first source operand is a YMM register. The second source operand is a YMM register, a 256-bit memory location, or a 256-bit vector broadcasted from a 32-bit memory location. The destination operand is a YMM register, conditionally updated using writemask k1.

EVEX.128 encoded version: The first source operand is a XMM register. The second source operand is a XMM register, a 128-bit memory location, or a 128-bit vector broadcasted from a 32-bit memory location. The destination operand is a XMM register, conditionally updated using writemask k1.

Operation

VUNPCKHPS (EVEX encoded version when SRC2 is a register)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
IF VL >= 128
    TMP_DEST[31:0] ← SRC1[95:64]
    TMP_DEST[63:32] ← SRC2[95:64]
    TMP_DEST[95:64] ← SRC1[127:96]
    TMP_DEST[127:96] ← SRC2[127:96]
FI;
IF VL >= 256
    TMP_DEST[159:128] ← SRC1[223:192]
    TMP_DEST[191:160] ← SRC2[223:192]
    TMP_DEST[223:192] ← SRC1[255:224]
    TMP_DEST[255:224] ← SRC2[255:224]
FI;
IF VL >= 512
    TMP_DEST[287:256] ← SRC1[351:320]
    TMP_DEST[319:288] ← SRC2[351:320]
    TMP_DEST[351:320] ← SRC1[383:352]
    TMP_DEST[383:352] ← SRC2[383:352]
    TMP_DEST[415:384] ← SRC1[479:448]
    TMP_DEST[447:416] ← SRC2[479:448]
    TMP_DEST[479:448] ← SRC1[511:480]
    TMP_DEST[511:480] ← SRC2[511:480]
FI;
```

```
FOR j ← 0 TO KL-1
    i ← j * 32
    IF k1[j] OR *no writemask*
        THEN DEST[i+31:i] ← TMP_DEST[i+31:i]
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE *zeroing-masking*              ; zeroing-masking
                    DEST[i+31:i] ← 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0
```

VUNPCKHPS (EVEX encoded version when SRC2 is memory)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j ← 0 TO KL-1
    i ← j * 32
    IF (EVEX.b = 1)
        THEN TMP_SRC2[i+31:i] ← SRC2[31:0]
        ELSE TMP_SRC2[i+31:i] ← SRC2[i+31:i]
    FI;
ENDFOR;
IF VL >= 128
    TMP_DEST[31:0] ← SRC1[95:64]
    TMP_DEST[63:32] ← TMP_SRC2[95:64]
    TMP_DEST[95:64] ← SRC1[127:96]
    TMP_DEST[127:96] ← TMP_SRC2[127:96]
FI;
IF VL >= 256
    TMP_DEST[159:128] ← SRC1[223:192]
    TMP_DEST[191:160] ← TMP_SRC2[223:192]
    TMP_DEST[223:192] ← SRC1[255:224]
    TMP_DEST[255:224] ← TMP_SRC2[255:224]
FI;
IF VL >= 512
    TMP_DEST[287:256] ← SRC1[351:320]
    TMP_DEST[319:288] ← TMP_SRC2[351:320]
    TMP_DEST[351:320] ← SRC1[383:352]
    TMP_DEST[383:352] ← TMP_SRC2[383:352]
    TMP_DEST[415:384] ← SRC1[479:448]
    TMP_DEST[447:416] ← TMP_SRC2[479:448]
    TMP_DEST[479:448] ← SRC1[511:480]
    TMP_DEST[511:480] ← TMP_SRC2[511:480]
FI;
FOR j ← 0 TO KL-1
    i ← j * 32
    IF k1[j] OR *no writemask*
        THEN DEST[i+31:i] ← TMP_DEST[i+31:i]
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE *zeroing-masking*              ; zeroing-masking
                    DEST[i+31:i] ← 0
```

```
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0
```

VUNPCKHPS (VEX.256 encoded version)

```
DEST[31:0] ←SRC1[95:64]
DEST[63:32] ←SRC2[95:64]
DEST[95:64] ←SRC1[127:96]
DEST[127:96] ←SRC2[127:96]
DEST[159:128] ←SRC1[223:192]
DEST[191:160] ←SRC2[223:192]
DEST[223:192] ←SRC1[255:224]
DEST[255:224] ←SRC2[255:224]
DEST[MAXVL-1:256] ← 0
```

VUNPCKHPS (VEX.128 encoded version)

```
DEST[31:0] ←SRC1[95:64]
DEST[63:32] ←SRC2[95:64]
DEST[95:64] ←SRC1[127:96]
DEST[127:96] ←SRC2[127:96]
DEST[MAXVL-1:128] ←0
```

UNPCKHPS (128-bit Legacy SSE version)

```
DEST[31:0] ←SRC1[95:64]
DEST[63:32] ←SRC2[95:64]
DEST[95:64] ←SRC1[127:96]
DEST[127:96] ←SRC2[127:96]
DEST[MAXVL-1:128] (Unmodified)
```

Intel C/C++ Compiler Intrinsic Equivalent

VUNPCKHPS __m512 _mm512_unpackhi_ps(__m512 a, __m512 b);
VUNPCKHPS __m512 _mm512_mask_unpackhi_ps(__m512 s, __mmask16 k, __m512 a, __m512 b);
VUNPCKHPS __m512 _mm512_maskz_unpackhi_ps(__mmask16 k, __m512 a, __m512 b);
VUNPCKHPS __m256 _mm256_unpackhi_ps (__m256 a, __m256 b);
VUNPCKHPS __m256 _mm256_mask_unpackhi_ps(__m256 s, __mmask8 k, __m256 a, __m256 b);
VUNPCKHPS __m256 _mm256_maskz_unpackhi_ps(__mmask8 k, __m256 a, __m256 b);
UNPCKHPS __m128 _mm_unpackhi_ps (__m128 a, __m128 b);
VUNPCKHPS __m128 _mm_mask_unpackhi_ps(__m128 s, __mmask8 k, __m128 a, __m128 b);
VUNPCKHPS __m128 _mm_maskz_unpackhi_ps(__mmask8 k, __m128 a, __m128 b);

SIMD Floating-Point Exceptions

None

Other Exceptions

Non-EVEX-encoded instructions, see Exceptions Type 4.
EVEX-encoded instructions, see Exceptions Type E4NF.

UNPCKLPD—Unpack and Interleave Low Packed Double-Precision Floating-Point Values

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 14 /r UNPCKLPD xmm1, xmm2/m128	A	V/V	SSE2	Unpacks and Interleaves double-precision floating-point values from low quadwords of xmm1 and xmm2/m128.
VEX.NDS.128.66.0F.WIG 14 /r VUNPCKLPD xmm1,xmm2, xmm3/m128	B	V/V	AVX	Unpacks and Interleaves double-precision floating-point values from low quadwords of xmm2 and xmm3/m128.
VEX.NDS.256.66.0F.WIG 14 /r VUNPCKLPD ymm1,ymm2, ymm3/m256	B	V/V	AVX	Unpacks and Interleaves double-precision floating-point values from low quadwords of ymm2 and ymm3/m256.
EVEX.NDS.128.66.0F.W1 14 /r VUNPCKLPD xmm1 {k1}{z}, xmm2, xmm3/m128/m64bcst	C	V/V	AVX512VL AVX512F	Unpacks and Interleaves double precision floating-point values from low quadwords of xmm2 and xmm3/m128/m64bcst subject to write mask k1.
EVEX.NDS.256.66.0F.W1 14 /r VUNPCKLPD ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst	C	V/V	AVX512VL AVX512F	Unpacks and Interleaves double precision floating-point values from low quadwords of ymm2 and ymm3/m256/m64bcst subject to write mask k1.
EVEX.NDS.512.66.0F.W1 14 /r VUNPCKLPD zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst	C	V/V	AVX512F	Unpacks and Interleaves double-precision floating-point values from low quadwords of zmm2 and zmm3/m512/m64bcst subject to write mask k1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	NA	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
B	NA	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA
C	Full Vector	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA

Description

Performs an interleaved unpack of the low double-precision floating-point values from the first source operand and the second source operand.

128-bit Legacy SSE version: The second source can be an XMM register or an 128-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (MAXVL-1:128) of the corresponding ZMM register destination are unmodified. When unpacking from a memory operand, an implementation may fetch only the appropriate 64 bits; however, alignment to 16-byte boundary and normal segment checking will still be enforced.

VEX.128 encoded version: The first source operand is a XMM register. The second source operand can be a XMM register or a 128-bit memory location. The destination operand is a XMM register. The upper bits (MAXVL-1:128) of the corresponding ZMM register destination are zeroed.

VEX.256 encoded version: The first source operand is a YMM register. The second source operand can be a YMM register or a 256-bit memory location. The destination operand is a YMM register.

EVEX.512 encoded version: The first source operand is a ZMM register. The second source operand is a ZMM register, a 512-bit memory location, or a 512-bit vector broadcasted from a 64-bit memory location. The destination operand is a ZMM register, conditionally updated using writemask k1.

EVEX.256 encoded version: The first source operand is a YMM register. The second source operand is a YMM register, a 256-bit memory location, or a 256-bit vector broadcasted from a 64-bit memory location. The destination operand is a YMM register, conditionally updated using writemask k1.

EVEX.128 encoded version: The first source operand is an XMM register. The second source operand is a XMM register, a 128-bit memory location, or a 128-bit vector broadcasted from a 64-bit memory location. The destination operand is a XMM register, conditionally updated using writemask k1.

Operation

VUNPCKLPD (EVEX encoded versions when SRC2 is a register)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
IF VL >= 128
    TMP_DEST[63:0] ← SRC1[63:0]
    TMP_DEST[127:64] ← SRC2[63:0]
FI;
IF VL >= 256
    TMP_DEST[191:128] ← SRC1[191:128]
    TMP_DEST[255:192] ← SRC2[191:128]
FI;
IF VL >= 512
    TMP_DEST[319:256] ← SRC1[319:256]
    TMP_DEST[383:320] ← SRC2[319:256]
    TMP_DEST[447:384] ← SRC1[447:384]
    TMP_DEST[511:448] ← SRC2[447:384]
FI;

FOR j ← 0 TO KL-1
    i ← j * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+63:i] ← TMP_DEST[i+63:i]
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE *zeroing-masking*              ; zeroing-masking
                    DEST[i+63:i] ← 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0
```

VUNPCKLPD (EVEX encoded version when SRC2 is memory)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j ← 0 TO KL-1
    i ← j * 64
    IF (EVEX.b = 1)
        THEN TMP_SRC2[i+63:i] ← SRC2[63:0]
        ELSE TMP_SRC2[i+63:i] ← SRC2[i+63:i]
    FI;
ENDFOR;
IF VL >= 128
    TMP_DEST[63:0] ← SRC1[63:0]
    TMP_DEST[127:64] ← TMP_SRC2[63:0]
FI;
IF VL >= 256
    TMP_DEST[191:128] ← SRC1[191:128]
    TMP_DEST[255:192] ← TMP_SRC2[191:128]
FI;
IF VL >= 512
    TMP_DEST[319:256] ← SRC1[319:256]
    TMP_DEST[383:320] ← TMP_SRC2[319:256]
    TMP_DEST[447:384] ← SRC1[447:384]
    TMP_DEST[511:448] ← TMP_SRC2[447:384]
FI;
FOR j ← 0 TO KL-1
    i ← j * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+63:i] ← TMP_DEST[i+63:i]
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE *zeroing-masking*              ; zeroing-masking
                    DEST[i+63:i] ← 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0
```

VUNPCKLPD (VEX.256 encoded version)

```
DEST[63:0] ←SRC1[63:0]
DEST[127:64] ←SRC2[63:0]
DEST[191:128] ←SRC1[191:128]
DEST[255:192] ←SRC2[191:128]
DEST[MAXVL-1:256] ← 0
```

VUNPCKLPD (VEX.128 encoded version)

```
DEST[63:0] ←SRC1[63:0]
DEST[127:64] ←SRC2[63:0]
DEST[MAXVL-1:128] ←0
```

UNPCKLPD (128-bit Legacy SSE version)

```
DEST[63:0] ←SRC1[63:0]
DEST[127:64] ←SRC2[63:0]
DEST[MAXVL-1:128] (Unmodified)
```

Intel C/C++ Compiler Intrinsic Equivalent

VUNPCKLPD __m512d _mm512_unpacklo_pd(__m512d a, __m512d b);
VUNPCKLPD __m512d _mm512_mask_unpacklo_pd(__m512d s, __mmask8 k, __m512d a, __m512d b);
VUNPCKLPD __m512d _mm512_maskz_unpacklo_pd(__mmask8 k, __m512d a, __m512d b);
VUNPCKLPD __m256d _mm256_unpacklo_pd(__m256d a, __m256d b)
VUNPCKLPD __m256d _mm256_mask_unpacklo_pd(__m256d s, __mmask8 k, __m256d a, __m256d b);
VUNPCKLPD __m256d _mm256_maskz_unpacklo_pd(__mmask8 k, __m256d a, __m256d b);
UNPCKLPD __m128d _mm_unpacklo_pd(__m128d a, __m128d b)
VUNPCKLPD __m128d _mm_mask_unpacklo_pd(__m128d s, __mmask8 k, __m128d a, __m128d b);
VUNPCKLPD __m128d _mm_maskz_unpacklo_pd(__mmask8 k, __m128d a, __m128d b);

SIMD Floating-Point Exceptions

None

Other Exceptions

Non-EVEX-encoded instructions, see Exceptions Type 4.
EVEX-encoded instructions, see Exceptions Type E4NF.

UNPCKLPS—Unpack and Interleave Low Packed Single-Precision Floating-Point Values

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F 14 /r UNPCKLPS xmm1, xmm2/m128	A	V/V	SSE	Unpacks and Interleaves single-precision floating-point values from low quadwords of xmm1 and xmm2/m128.
VEX.NDS.128.0F.WIG 14 /r VUNPCKLPS xmm1,xmm2, xmm3/m128	B	V/V	AVX	Unpacks and Interleaves single-precision floating-point values from low quadwords of xmm2 and xmm3/m128.
VEX.NDS.256.0F.WIG 14 /r VUNPCKLPS ymm1,ymm2,ymm3/m256	B	V/V	AVX	Unpacks and Interleaves single-precision floating-point values from low quadwords of ymm2 and ymm3/m256.
EVEX.NDS.128.0F.W0 14 /r VUNPCKLPS xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst	C	V/V	AVX512VL AVX512F	Unpacks and Interleaves single-precision floating-point values from low quadwords of xmm2 and xmm3/mem and write result to xmm1 subject to write mask k1.
EVEX.NDS.256.0F.W0 14 /r VUNPCKLPS ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst	C	V/V	AVX512VL AVX512F	Unpacks and Interleaves single-precision floating-point values from low quadwords of ymm2 and ymm3/mem and write result to ymm1 subject to write mask k1.
EVEX.NDS.512.0F.W0 14 /r VUNPCKLPS zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst	C	V/V	AVX512F	Unpacks and Interleaves single-precision floating-point values from low quadwords of zmm2 and zmm3/m512/m32bcst and write result to zmm1 subject to write mask k1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	NA	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
B	NA	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA
C	Full Vector	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA

Description

Performs an interleaved unpack of the low single-precision floating-point values from the first source operand and the second source operand.

128-bit Legacy SSE version: The second source can be an XMM register or an 128-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (MAXVL-1:128) of the corresponding ZMM register destination are unmodified. When unpacking from a memory operand, an implementation may fetch only the appropriate 64 bits; however, alignment to 16-byte boundary and normal segment checking will still be enforced.

VEX.128 encoded version: The first source operand is a XMM register. The second source operand can be a XMM register or a 128-bit memory location. The destination operand is a XMM register. The upper bits (MAXVL-1:128) of the corresponding ZMM register destination are zeroed.

VEX.256 encoded version: The first source operand is a YMM register. The second source operand can be a YMM register or a 256-bit memory location. The destination operand is a YMM register.

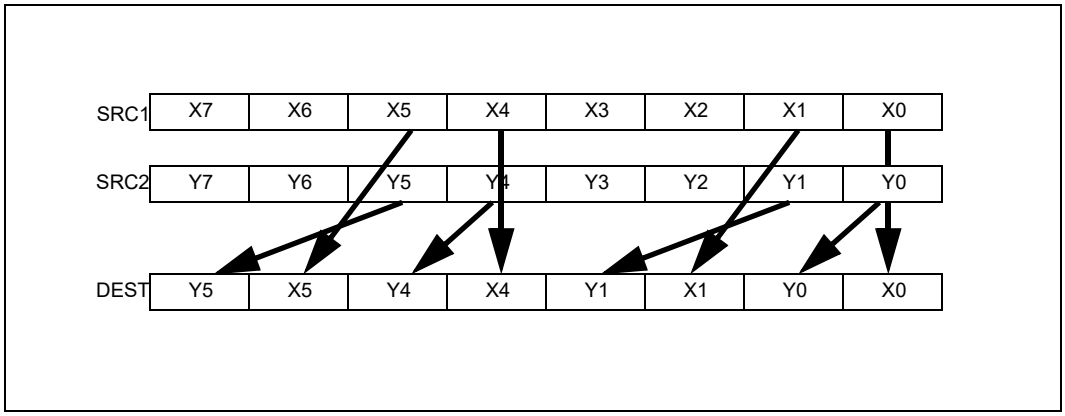


Figure 4-28. VUNPCKLPS Operation

EVEX.512 encoded version: The first source operand is a ZMM register. The second source operand is a ZMM register, a 512-bit memory location, or a 512-bit vector broadcasted from a 32-bit memory location. The destination operand is a ZMM register, conditionally updated using writemask k1.

EVEX.256 encoded version: The first source operand is a YMM register. The second source operand is a YMM register, a 256-bit memory location, or a 256-bit vector broadcasted from a 32-bit memory location. The destination operand is a YMM register, conditionally updated using writemask k1.

EVEX.128 encoded version: The first source operand is an XMM register. The second source operand is a XMM register, a 128-bit memory location, or a 128-bit vector broadcasted from a 32-bit memory location. The destination operand is a XMM register, conditionally updated using writemask k1.

Operation

VUNPCKLPS (EVEX encoded version when SRC2 is a ZMM register)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
IF VL >= 128
    TMP_DEST[31:0] ← SRC1[31:0]
    TMP_DEST[63:32] ← SRC2[31:0]
    TMP_DEST[95:64] ← SRC1[63:32]
    TMP_DEST[127:96] ← SRC2[63:32]
FI;
IF VL >= 256
    TMP_DEST[159:128] ← SRC1[159:128]
    TMP_DEST[191:160] ← SRC2[159:128]
    TMP_DEST[223:192] ← SRC1[191:160]
    TMP_DEST[255:224] ← SRC2[191:160]
FI;
IF VL >= 512
    TMP_DEST[287:256] ← SRC1[287:256]
    TMP_DEST[319:288] ← SRC2[287:256]
    TMP_DEST[351:320] ← SRC1[319:288]
    TMP_DEST[383:352] ← SRC2[319:288]
    TMP_DEST[415:384] ← SRC1[415:384]
    TMP_DEST[447:416] ← SRC2[415:384]
    TMP_DEST[479:448] ← SRC1[447:416]
    TMP_DEST[511:480] ← SRC2[447:416]
FI;
FOR j ← 0 TO KL-1
    i ← j * 32
```

```
    IF k1[j] OR *no writemask*
        THEN DEST[i+31:i] ← TMP_DEST[i+31:i]
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE *zeroing-masking*              ; zeroing-masking
                    DEST[i+31:i] ← 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0
```

VUNPCKLPS (EVEX encoded version when SRC2 is memory)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j ← 0 TO KL-1
    i ← j * 31
    IF (EVEX.b = 1)
        THEN TMP_SRC2[i+31:i] ← SRC2[31:0]
        ELSE TMP_SRC2[i+31:i] ← SRC2[i+31:i]
    FI;
ENDFOR;
IF VL >= 128
TMP_DEST[31:0] ← SRC1[31:0]
TMP_DEST[63:32] ← TMP_SRC2[31:0]
TMP_DEST[95:64] ← SRC1[63:32]
TMP_DEST[127:96] ← TMP_SRC2[63:32]
FI;
IF VL >= 256
    TMP_DEST[159:128] ← SRC1[159:128]
    TMP_DEST[191:160] ← TMP_SRC2[159:128]
    TMP_DEST[223:192] ← SRC1[191:160]
    TMP_DEST[255:224] ← TMP_SRC2[191:160]
FI;
IF VL >= 512
    TMP_DEST[287:256] ← SRC1[287:256]
    TMP_DEST[319:288] ← TMP_SRC2[287:256]
    TMP_DEST[351:320] ← SRC1[319:288]
    TMP_DEST[383:352] ← TMP_SRC2[319:288]
    TMP_DEST[415:384] ← SRC1[415:384]
    TMP_DEST[447:416] ← TMP_SRC2[415:384]
    TMP_DEST[479:448] ← SRC1[447:416]
    TMP_DEST[511:480] ← TMP_SRC2[447:416]
FI;
FOR j ← 0 TO KL-1
    i ← j * 32
    IF k1[j] OR *no writemask*
        THEN DEST[i+31:i] ← TMP_DEST[i+31:i]
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE *zeroing-masking*              ; zeroing-masking
                    DEST[i+31:i] ← 0
            FI
    FI;
```

ENDFOR
DEST[MAXVL-1:VL] ← 0

UNPCKLPS (VEX.256 encoded version)
DEST[31:0] ←SRC1[31:0]
DEST[63:32] ←SRC2[31:0]
DEST[95:64] ←SRC1[63:32]
DEST[127:96] ←SRC2[63:32]
DEST[159:128] ←SRC1[159:128]
DEST[191:160] ←SRC2[159:128]
DEST[223:192] ←SRC1[191:160]
DEST[255:224] ←SRC2[191:160]
DEST[MAXVL-1:256] ← 0

VUNPCKLPS (VEX.128 encoded version)
DEST[31:0] ←SRC1[31:0]
DEST[63:32] ←SRC2[31:0]
DEST[95:64] ←SRC1[63:32]
DEST[127:96] ←SRC2[63:32]
DEST[MAXVL-1:128] ←0

UNPCKLPS (128-bit Legacy SSE version)
DEST[31:0] ←SRC1[31:0]
DEST[63:32] ←SRC2[31:0]
DEST[95:64] ←SRC1[63:32]
DEST[127:96] ←SRC2[63:32]
DEST[MAXVL-1:128] (Unmodified)

Intel C/C++ Compiler Intrinsic Equivalent

VUNPCKLPS __m512 _mm512_unpacklo_ps(__m512 a, __m512 b);
VUNPCKLPS __m512 _mm512_mask_unpacklo_ps(__m512 s, __mmask16 k, __m512 a, __m512 b);
VUNPCKLPS __m512 _mm512_maskz_unpacklo_ps(__mmask16 k, __m512 a, __m512 b);
VUNPCKLPS __m256 _mm256_unpacklo_ps (__m256 a, __m256 b);
VUNPCKLPS __m256 _mm256_mask_unpacklo_ps(__m256 s, __mmask8 k, __m256 a, __m256 b);
VUNPCKLPS __m256 _mm256_maskz_unpacklo_ps(__mmask8 k, __m256 a, __m256 b);
UNPCKLPS __m128 _mm_unpacklo_ps (__m128 a, __m128 b);
VUNPCKLPS __m128 _mm_mask_unpacklo_ps(__m128 s, __mmask8 k, __m128 a, __m128 b);
VUNPCKLPS __m128 _mm_maskz_unpacklo_ps(__mmask8 k, __m128 a, __m128 b);

SIMD Floating-Point Exceptions

None

Other Exceptions

Non-EVEX-encoded instructions, see Exceptions Type 4.
EVEX-encoded instructions, see Exceptions Type E4NF.

9. Updates to Chapter 5, Volume 2C

Change bars show changes to Chapter 5 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2C:* Instruction Set Reference, V-Z.

Changes to this chapter: Rearranged ordering as necessary. Updates to the following instructions: VBROADCAST, VFIXUPIMMPS, VZEROALL, VZEROUPPER. Updated operand encoding table for instructions with tuple types, breaking out tuple types into a separate column. Corrected naming typos in operation section of various instructions.

CHAPTER 5
INSTRUCTION SET REFERENCE, V-Z

5.1 TERNARY BIT VECTOR LOGIC TABLE

VPTERNLOGD/VPTERNLOGQ instructions operate on dword/qword elements and take three bit vectors of the respective input data elements to form a set of 32/64 indices, where each 3-bit value provides an index into an 8-bit lookup table represented by the imm8 byte of the instruction. The 256 possible values of the imm8 byte is constructed as a 16x16 boolean logic table. The 16 rows of the table uses the lower 4 bits of imm8 as row index. The 16 columns are referenced by imm8[7:4]. The 16 columns of the table are present in two halves, with 8 columns shown in Table 5-1 for the column index value between 0:7, followed by Table 5-2 showing the 8 columns corresponding to column index 8:15. This section presents the two-halves of the 256-entry table using a short-hand notation representing simple or compound boolean logic expressions with three input bit source data.

The three input bit source data will be denoted with the capital letters: A, B, C; where A represents a bit from the first source operand (also the destination operand), B and C represent a bit from the 2nd and 3rd source operands.

Each map entry takes the form of a logic expression consisting of one of more component expressions. Each component expression consists of either a unary or binary boolean operator and associated operands. Each binary boolean operator is expressed in lowercase letters, and operands concatenated after the logic operator. The unary operator 'not' is expressed using '!'. Additionally, the conditional expression "A?B:C" expresses a result returning B if A is set, returning C otherwise.

A binary boolean operator is followed by two operands, e.g. andAB. For a compound binary expression that contain commutative components and comprising the same logic operator, the 2nd logic operator is omitted and three operands can be concatenated in sequence, e.g. andABC. When the 2nd operand of the first binary boolean expression comes from the result of another boolean expression, the 2nd boolean expression is concatenated after the uppercase operand of the first logic expression, e.g. norBnandAC. When the result is independent of an operand, that operand is omitted in the logic expression, e.g. zeros or norCB.

The 3-input expression "majorABC" returns 0 if two or more input bits are 0, returns 1 if two or more input bits are 1. The 3-input expression "minorABC" returns 1 if two or more input bits are 0, returns 0 if two or more input bits are 1.

The building-block bit logic functions used in Table 5-1 and Table 5-2 include;

- Constants: TRUE (1), FALSE (0);
- Unary function: Not (!);
- Binary functions: and, nand, or, nor, xor, xnor;
- Conditional function: Select (?:);
- Tertiary functions: major, minor.

Table 5-1. Low 8 columns of the 16x16 Map of VPTERNLOG Boolean Logic Operations

Imm	[7:4]							
[3:0]	*0H*	*1H*	*2H*	*3H*	*4H*	*5H*	*6H*	*7H*
00H	*FALSE*	*andAnorBC*	*norBnandAC*	*andA!B*	*norCnandBA*	*andA!C*	*andAxorBC*	*andAnandBC*
01H	*norABC*	*norCB*	*norBxorAC*	*A?!B:norBC*	*norCxorBA*	*A?!C:norBC*	*A?xorBC:norBC*	*A?nandBC:norBC*
02H	*andCnorBA*	*norBxnorAC*	*andC!B*	*norBnorAC*	*C?norBA:andBA*	*C?norBA:A*	*C?!B:andBA*	*C?!B:A*
03H	*norBA*	*norBandAC*	*C?!B:norBA*	*!B*	*C?norBA:xnorBA*	*A?!C:!B*	*A?xorBC:!B*	*A?nandBC:!B*
04H	*andBnorAC*	*norCxnorBA*	*B?norAC:andAC*	*B?norAC:A*	*andB!C*	*norCnorBA*	*B?!C:andAC*	*B?!C:A*
05H	*norCA*	*norCandBA*	*B?norAC:xnorAC*	*A?!B:!C*	*B?!C:norAC*	*!C*	*A?xorBC:!C*	*A?nandBC:!C*
06H	*norAxnorBC*	*A?norBC:xorBC*	*B?norAC:C*	*xorBorAC*	*C?norBA:B*	*xorCorBA*	*xorCB*	*B?!C:orAC*
07H	*norAandBC*	*minorABC*	*C?!B:!A*	*nandBorAC*	*B?!C:!A*	*nandCorBA*	*A?xorBC:nandBC*	*nandCB*
08H	*norAnandBC*	*A?norBC:andBC*	*andCxorBA*	*A?!B:andBC*	*andBxorAC*	*A?!C:andBC*	*A?xorBC:andBC*	*xorAandBC*
09H	*norAxorBC*	*A?norBC:xnorBC*	*C?xorBA:norBA*	*A?!B:xnorBC*	*B?xorAC:norAC*	*A?!C:xnorBC*	*xnorABC*	*A?nandBC:xnorBC*
0AH	*andC!A*	*A?norBC:C*	*andCnandBA*	*A?!B:C*	*C?!A:andBA*	*xorCA*	*xorCandBA*	*A?nandBC:C*
0BH	*C?!A:norBA*	*C?!A:!B*	*C?nandBA:norBA*	*C?nandBA:!B*	*B?xorAC:!A*	*B?xorAC:nandAC*	*C?nandBA:xnorBA*	*nandBxnorAC*
0CH	*andB!A*	*A?norBC:B*	*B?!A:andAC*	*xorBA*	*andBnandAC*	*A?!C:B*	*xorBandAC*	*A?nandBC:B*
0DH	*B?!A:norAC*	*B?!A:!C*	*B?!A:xnorAC*	*C?xorBA:nandBA*	*B?nandAC:norAC*	*B?nandAC:!C*	*B?nandAC:xnorAC*	*nandCxnorBA*
0EH	*norAnorBC*	*xorAorBC*	*B?!A:C*	*A?!B:orBC*	*C?!A:B*	*A?!C:orBC*	*B?nandAC:C*	*A?nandBC:orBC*
0FH	*!A*	*nandAorBC*	*C?nandBA:!A*	*nandBA*	*B?nandAC:!A*	*nandCA*	*nandAxnorBC*	*nandABC*

Table 5-2 shows the half of 256-entry map corresponding to column index values 8:15.

Table 5-2. Low 8 columns of the 16x16 Map of VPTERNLOG Boolean Logic Operations

Imm	[7:4]							
[3:0]	08H	09H	0AH	0BH	0CH	0DH	0EH	0FH
00H	andABC	andAxnorBC	andCA	B?andAC:A	andBA	C?andBA:A	andAorBC	A
01H	A?andBC:norBC	B?andAC:!C	A?C:norBC	C?A:!B	A?B:norBC	B?A:!C	xnorAorBC	orAnorBC
02H	andCxnorBA	B?andAC:xorAC	B?andAC:C	B?andAC:orAC	C?xnorBA:andBA	B?A:xorAC	B?A:C	B?A:orAC
03H	A?andBC:!B	xnorBandAC	A?C:!B	nandBnandAC	xnorBA	B?A:nandAC	A?orBC:!B	orA!B
04H	andBxnorAC	C?andBA:xorBA	B?xnorAC:andAC	B?xnorAC:A	C?andBA:B	C?andBA:orBA	C?A:B	C?A:orBA
05H	A?andBC:!C	xnorCandBA	xnorCA	C?A:nandBA	A?B:!C	nandCnandBA	A?orBC:!C	orA!C
06H	A?andBC:xorBC	xorABC	A?C:xorBC	B?xnorAC:orAC	A?B:xorBC	C?xnorBA:orBA	A?orBC:xorBC	orAxorBC
07H	xnorAandBC	A?xnorBC:nandBC	A?C:nandBC	nandBxorAC	A?B:nandBC	nandCxorBA	A?orBCnandBC	orAnandBC
08H	andCB	A?xnorBC:andBC	andCorAB	B?C:A	andBorAC	C?B:A	majorABC	orAandBC
09H	B?C:norAC	xnorCB	xnorCorBA	C?orBA:!B	xnorBorAC	B?orAC:!C	A?orBC:xnorBC	orAxnorBC
0AH	A?andBC:C	A?xnorBC:C	C	B?C:orAC	A?B:C	B?orAC:xorAC	orCandBA	orCA
0BH	B?C:!A	B?C:nandAC	orCnorBA	orC!B	B?orAC:!A	B?orAC:nandAC	orCxnorBA	nandBnorAC
0CH	A?andBC:B	A?xnorBC:B	A?C:B	C?orBA:xorBA	B	C?B:orBA	orBandAC	orBA
0DH	C?B!A	C?B:nandBA	C?orBA:!A	C?orBA:nandBA	orBnorAC	orB!C	orBxnorAC	nandCnorBA
0EH	A?andBC:orBC	A?xnorBC:orBC	A?C:orBC	orCxorBA	A?B:orBC	orBxorAC	orCB	orABC
0FH	nandAnandBC	nandAxorBC	orC!A	orCnandBA	orB!A	orBnandAC	nandAnorBC	TRUE

Table 5-1 and Table 5-2 translate each of the possible value of the imm8 byte to a Boolean expression. These tables can also be used by software to translate Boolean expressions to numerical constants to form the imm8 value needed to construct the VPTERNLOG syntax. There is a unique set of three byte constants (F0H, CCH, AAH) that can be used for this purpose as input operands in conjunction with the Boolean expressions defined in those tables. The reverse mapping can be expressed as:

Result_imm8 = Table_Lookup_Entry(0F0H, 0CCH, 0AAH)

Table_Lookup_Entry is the Boolean expression defined in Table 5-1 and Table 5-2.

5.2 INSTRUCTIONS (V-Z)

Chapter 5 continues an alphabetical discussion of Intel® 64 and IA-32 instructions (V-Z). See also: Chapter 3, "Instruction Set Reference, A-L," in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A*, and Chapter 4, "Instruction Set Reference, M-U," in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2B*.

VALIGND/VALIGNQ—Align Doubleword/Quadword Vectors

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.NDS.128.66.0F3A.W0 03 /r ib VALIGND xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst, imm8	A	V/V	AVX512VL AVX512F	Shift right and merge vectors xmm2 and xmm3/m128/m32bcst with double-word granularity using imm8 as number of elements to shift, and store the final result in xmm1, under writemask.
EVEX.NDS.128.66.0F3A.W1 03 /r ib VALIGNQ xmm1 {k1}{z}, xmm2, xmm3/m128/m64bcst, imm8	A	V/V	AVX512VL AVX512F	Shift right and merge vectors xmm2 and xmm3/m128/m64bcst with quad-word granularity using imm8 as number of elements to shift, and store the final result in xmm1, under writemask.
EVEX.NDS.256.66.0F3A.W0 03 /r ib VALIGND ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst, imm8	A	V/V	AVX512VL AVX512F	Shift right and merge vectors ymm2 and ymm3/m256/m32bcst with double-word granularity using imm8 as number of elements to shift, and store the final result in ymm1, under writemask.
EVEX.NDS.256.66.0F3A.W1 03 /r ib VALIGNQ ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst, imm8	A	V/V	AVX512VL AVX512F	Shift right and merge vectors ymm2 and ymm3/m256/m64bcst with quad-word granularity using imm8 as number of elements to shift, and store the final result in ymm1, under writemask.
EVEX.NDS.512.66.0F3A.W0 03 /r ib VALIGND zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst, imm8	A	V/V	AVX512F	Shift right and merge vectors zmm2 and zmm3/m512/m32bcst with double-word granularity using imm8 as number of elements to shift, and store the final result in zmm1, under writemask.
EVEX.NDS.512.66.0F3A.W1 03 /r ib VALIGNQ zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst, imm8	A	V/V	AVX512F	Shift right and merge vectors zmm2 and zmm3/m512/m64bcst with quad-word granularity using imm8 as number of elements to shift, and store the final result in zmm1, under writemask.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Full Vector	ModRM:reg (w)	EVEX.vvvv	ModRM:r/m (r)	NA

Description

Concatenates and shifts right doubleword/quadword elements of the first source operand (the second operand) and the second source operand (the third operand) into a 1024/512/256-bit intermediate vector. The low 512/256/128-bit of the intermediate vector is written to the destination operand (the first operand) using the writemask k1. The destination and first source operands are ZMM/YMM/XMM registers. The second source operand can be a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 32/64-bit memory location.

This instruction is writemasked, so only those elements with the corresponding bit set in vector mask register k1 are computed and stored into zmm1. Elements in zmm1 with the corresponding bit clear in k1 retain their previous values (merging-masking) or are set to 0 (zeroing-masking).

Operation

VALIGND (EVEX encoded versions)
(KL, VL) = (4, 128), (8, 256), (16, 512)

```
IF (SRC2 *is memory*) (AND EVEX.b = 1)
    THEN
        FOR j ← 0 TO KL-1
            i ←j * 32
            src[i+31:i] ← SRC2[31:0]
        ENDFOR;
    ELSE src ← SRC2
FI
; Concatenate sources
tmp[VL-1:0] ← src[VL-1:0]
tmp[2VL-1:VL] ← SRC1[VL-1:0]
; Shift right doubleword elements
IF VL = 128
    THEN SHIFT = imm8[1:0]
    ELSE
        IF VL = 256
            THEN SHIFT = imm8[2:0]
            ELSE SHIFT = imm8[3:0]
        FI
FI;
tmp[2VL-1:0] ← tmp[2VL-1:0] >> (32*SHIFT)
; Apply writemask
FOR j ← 0 TO KL-1
    i ←j * 32
    IF k1[j] OR *no writemask*
        THEN DEST[i+31:i] ← tmp[i+31:i]
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE                                ; zeroing-masking
                    DEST[i+31:i] ← 0
            FI
    FI;
ENDFOR;
DEST[MAXVL-1:VL] ← 0
```

VALIGNQ (EVEX encoded versions)

```
(KL, VL) = (2, 128), (4, 256),(8, 512)
IF (SRC2 *is memory*) (AND EVEX.b = 1)
    THEN
        FOR j ← 0 TO KL-1
            i ←j * 64
            src[i+63:i] ← SRC2[63:0]
        ENDFOR;
    ELSE src ← SRC2
FI
; Concatenate sources
tmp[VL-1:0] ← src[VL-1:0]
tmp[2VL-1:VL] ← SRC1[VL-1:0]
; Shift right quadword elements
IF VL = 128
    THEN SHIFT = imm8[0]
    ELSE
        IF VL = 256
            THEN SHIFT = imm8[1:0]
            ELSE SHIFT = imm8[2:0]
        FI
FI;
tmp[2VL-1:0] ← tmp[2VL-1:0] >> (64*SHIFT)
; Apply writemask
FOR j ← 0 TO KL-1
    i ←j * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+63:i] ← tmp[i+63:i]
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE                                ; zeroing-masking
                    DEST[i+63:i] ← 0
            FI
    FI;
ENDFOR;
DEST[MAXVL-1:VL] ← 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VALIGND __m512i _mm512_alignr_epi32(__m512i a, __m512i b, int cnt);
VALIGND __m512i _mm512_mask_alignr_epi32(__m512i s, __mmask16 k, __m512i a, __m512i b, int cnt);
VALIGND __m512i _mm512_maskz_alignr_epi32(__mmask16 k, __m512i a, __m512i b, int cnt);
VALIGND __m256i _mm256_mask_alignr_epi32(__m256i s, __mmask8 k, __m256i a, __m256i b, int cnt);
VALIGND __m256i _mm256_maskz_alignr_epi32(__mmask8 k, __m256i a, __m256i b, int cnt);
VALIGND __m128i _mm_mask_alignr_epi32(__m128i s, __mmask8 k, __m128i a, __m128i b, int cnt);
VALIGND __m128i _mm_maskz_alignr_epi32(__mmask8 k, __m128i a, __m128i b, int cnt);
VALIGNQ __m512i _mm512_alignr_epi64(__m512i a, __m512i b, int cnt);
VALIGNQ __m512i _mm512_mask_alignr_epi64(__m512i s, __mmask8 k, __m512i a, __m512i b, int cnt);
VALIGNQ __m512i _mm512_maskz_alignr_epi64(__mmask8 k, __m512i a, __m512i b, int cnt);
VALIGNQ __m256i _mm256_mask_alignr_epi64(__m256i s, __mmask8 k, __m256i a, __m256i b, int cnt);
VALIGNQ __m256i _mm256_maskz_alignr_epi64(__mmask8 k, __m256i a, __m256i b, int cnt);
VALIGNQ __m128i _mm_mask_alignr_epi64(__m128i s, __mmask8 k, __m128i a, __m128i b, int cnt);
VALIGNQ __m128i _mm_maskz_alignr_epi64(__mmask8 k, __m128i a, __m128i b, int cnt);

Exceptions

See Exceptions Type E4NF.

VBLENDMPD/VBLENDMPS—Blend Float64/Float32 Vectors Using an OpMask Control

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.NDS.128.66.0F38.W1 65 /r VBLENDMPD xmm1 {k1}{z}, xmm2, xmm3/m128/m64bcst	A	V/V	AVX512VL AVX512F	Blend double-precision vector xmm2 and double-precision vector xmm3/m128/m64bcst and store the result in xmm1, under control mask.
EVEX.NDS.256.66.0F38.W1 65 /r VBLENDMPD ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst	A	V/V	AVX512VL AVX512F	Blend double-precision vector ymm2 and double-precision vector ymm3/m256/m64bcst and store the result in ymm1, under control mask.
EVEX.NDS.512.66.0F38.W1 65 /r VBLENDMPD zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst	A	V/V	AVX512F	Blend double-precision vector zmm2 and double-precision vector zmm3/m512/m64bcst and store the result in zmm1, under control mask.
EVEX.NDS.128.66.0F38.W0 65 /r VBLENDMPS xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst	A	V/V	AVX512VL AVX512F	Blend single-precision vector xmm2 and single-precision vector xmm3/m128/m32bcst and store the result in xmm1, under control mask.
EVEX.NDS.256.66.0F38.W0 65 /r VBLENDMPS ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst	A	V/V	AVX512VL AVX512F	Blend single-precision vector ymm2 and single-precision vector ymm3/m256/m32bcst and store the result in ymm1, under control mask.
EVEX.NDS.512.66.0F38.W0 65 /r VBLENDMPS zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst	A	V/V	AVX512F	Blend single-precision vector zmm2 and single-precision vector zmm3/m512/m32bcst using k1 as select control and store the result in zmm1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Full Vector	ModRM:reg (w)	EVEX.vvvv	ModRM:r/m (r)	NA

Description

Performs an element-by-element blending between float64/float32 elements in the first source operand (the second operand) with the elements in the second source operand (the third operand) using an opmask register as select control. The blended result is written to the destination register.

The destination and first source operands are ZMM/YMM/XMM registers. The second source operand can be a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 64-bit memory location.

The opmask register is not used as a writemask for this instruction. Instead, the mask is used as an element selector: every element of the destination is conditionally selected between first source or second source using the value of the related mask bit (0 for first source operand, 1 for second source operand).

If EVEX.z is set, the elements with corresponding mask bit value of 0 in the destination operand are zeroed.

Operation

VBLENDMPD (EVEX encoded versions)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j ← 0 TO KL-1
    i ← j * 64
    IF k1[j] OR *no controlmask*
        THEN
            IF (EVEX.b = 1) AND (SRC2 *is memory*)
                THEN
                    DEST[i+63:i] ← SRC2[63:0]
                ELSE
                    DEST[i+63:i] ← SRC2[i+63:i]
            FI;
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN DEST[i+63:i] ← SRC1[i+63:i]
                ELSE                                ; zeroing-masking
                    DEST[i+63:i] ← 0
            FI;
    FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0
```

VBLENDMPS (EVEX encoded versions)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j ← 0 TO KL-1
    i ← j * 32
    IF k1[j] OR *no controlmask*
        THEN
            IF (EVEX.b = 1) AND (SRC2 *is memory*)
                THEN
                    DEST[i+31:i] ← SRC2[31:0]
                ELSE
                    DEST[i+31:i] ← SRC2[i+31:i]
            FI;
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN DEST[i+31:i] ← SRC1[i+31:i]
                ELSE                                ; zeroing-masking
                    DEST[i+31:i] ← 0
            FI;
    FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VBLENDMPD __m512d _mm512_mask_blend_pd(__mmask8 k, __m512d a, __m512d b);
VBLENDMPD __m256d _mm256_mask_blend_pd(__mmask8 k, __m256d a, __m256d b);
VBLENDMPD __m128d _mm_mask_blend_pd(__mmask8 k, __m128d a, __m128d b);
VBLENDMPS __m512 _mm512_mask_blend_ps(__mmask16 k, __m512 a, __m512 b);
VBLENDMPS __m256 _mm256_mask_blend_ps(__mmask8 k, __m256 a, __m256 b);
VBLENDMPS __m128 _mm_mask_blend_ps(__mmask8 k, __m128 a, __m128 b);

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type E4.

VBROADCAST—Load with Broadcast Floating-Point Data

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
VEX.128.66.0F38.W0 18 /r VBROADCASTSS xmm1, m32	A	V/V	AVX	Broadcast single-precision floating-point element in mem to four locations in xmm1.
VEX.256.66.0F38.W0 18 /r VBROADCASTSS ymm1, m32	A	V/V	AVX	Broadcast single-precision floating-point element in mem to eight locations in ymm1.
VEX.256.66.0F38.W0 19 /r VBROADCASTSD ymm1, m64	A	V/V	AVX	Broadcast double-precision floating-point element in mem to four locations in ymm1.
VEX.256.66.0F38.W0 1A /r VBROADCASTF128 ymm1, m128	A	V/V	AVX	Broadcast 128 bits of floating-point data in mem to low and high 128-bits in ymm1.
VEX.128.66.0F38.W0 18/r VBROADCASTSS *xmm1, xmm2*	A	V/V	AVX2	Broadcast the low single-precision floating-point element in the source operand to four locations in xmm1.
VEX.256.66.0F38.W0 18 /r VBROADCASTSS *ymm1, xmm2*	A	V/V	AVX2	Broadcast low single-precision floating-point element in the source operand to eight locations in ymm1.
VEX.256.66.0F38.W0 19 /r VBROADCASTSD *ymm1, xmm2*	A	V/V	AVX2	Broadcast low double-precision floating-point element in the source operand to four locations in ymm1.
EVEX.256.66.0F38.W1 19 /r VBROADCASTSD ymm1 {k1}{z}, xmm2/m64	B	V/V	AVX512VL AVX512F	Broadcast low double-precision floating-point element in xmm2/m64 to four locations in ymm1 using writemask k1.
EVEX.512.66.0F38.W1 19 /r VBROADCASTSD zmm1 {k1}{z}, xmm2/m64	B	V/V	AVX512F	Broadcast low double-precision floating-point element in xmm2/m64 to eight locations in zmm1 using writemask k1.
EVEX.256.66.0F38.W0 19 /r VBROADCASTF32X2 ymm1 {k1}{z}, xmm2/m64	C	V/V	AVX512VL AVX512DQ	Broadcast two single-precision floating-point elements in xmm2/m64 to locations in ymm1 using writemask k1.
EVEX.512.66.0F38.W0 19 /r VBROADCASTF32X2 zmm1 {k1}{z}, xmm2/m64	C	V/V	AVX512DQ	Broadcast two single-precision floating-point elements in xmm2/m64 to locations in zmm1 using writemask k1.
EVEX.128.66.0F38.W0 18 /r VBROADCASTSS xmm1 {k1}{z}, xmm2/m32	B	V/V	AVX512VL AVX512F	Broadcast low single-precision floating-point element in xmm2/m32 to all locations in xmm1 using writemask k1.
EVEX.256.66.0F38.W0 18 /r VBROADCASTSS ymm1 {k1}{z}, xmm2/m32	B	V/V	AVX512VL AVX512F	Broadcast low single-precision floating-point element in xmm2/m32 to all locations in ymm1 using writemask k1.
EVEX.512.66.0F38.W0 18 /r VBROADCASTSS zmm1 {k1}{z}, xmm2/m32	B	V/V	AVX512F	Broadcast low single-precision floating-point element in xmm2/m32 to all locations in zmm1 using writemask k1.
EVEX.256.66.0F38.W0 1A /r VBROADCASTF32X4 ymm1 {k1}{z}, m128	D	V/V	AVX512VL AVX512F	Broadcast 128 bits of 4 single-precision floating-point data in mem to locations in ymm1 using writemask k1.
EVEX.512.66.0F38.W0 1A /r VBROADCASTF32X4 zmm1 {k1}{z}, m128	D	V/V	AVX512F	Broadcast 128 bits of 4 single-precision floating-point data in mem to locations in zmm1 using writemask k1.
EVEX.256.66.0F38.W1 1A /r VBROADCASTF64X2 ymm1 {k1}{z}, m128	C	V/V	AVX512VL AVX512DQ	Broadcast 128 bits of 2 double-precision floating-point data in mem to locations in ymm1 using writemask k1.

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.512.66.0F38.W1 1A /r VBROADCASTF64X2 zmm1 {k1}{z}, m128	C	V/V	AVX512DQ	Broadcast 128 bits of 2 double-precision floating-point data in mem to locations in zmm1 using writemask k1.
EVEX.512.66.0F38.W0 1B /r VBROADCASTF32X8 zmm1 {k1}{z}, m256	E	V/V	AVX512DQ	Broadcast 256 bits of 8 single-precision floating-point data in mem to locations in zmm1 using writemask k1.
EVEX.512.66.0F38.W1 1B /r VBROADCASTF64X4 zmm1 {k1}{z}, m256	D	V/V	AVX512F	Broadcast 256 bits of 4 double-precision floating-point data in mem to locations in zmm1 using writemask k1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	NA	ModRM:reg (w)	ModRM:r/m (r)	NA	NA
B	Tuple1 Scalar	ModRM:reg (w)	ModRM:r/m (r)	NA	NA
C	Tuple2	ModRM:reg (w)	ModRM:r/m (r)	NA	NA
D	Tuple4	ModRM:reg (w)	ModRM:r/m (r)	NA	NA
E	Tuple8	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

Description

VBROADCASTSD/VBROADCASTSS/VBROADCASTF128 load floating-point values as one tuple from the source operand (second operand) in memory and broadcast to all elements of the destination operand (first operand).

VEX256-encoded versions: The destination operand is a YMM register. The source operand is either a 32-bit, 64-bit, or 128-bit memory location. Register source encodings are reserved and will #UD. Bits (MAXVL-1:256) of the destination register are zeroed.

EVEX-encoded versions: The destination operand is a ZMM/YMM/XMM register and updated according to the writemask k1. The source operand is either a 32-bit, 64-bit memory location or the low doubleword/quadword element of an XMM register.

VBROADCASTF32X2/VBROADCASTF32X4/VBROADCASTF64X2/VBROADCASTF32X8/VBROADCASTF64X4 load floating-point values as tuples from the source operand (the second operand) in memory or register and broadcast to all elements of the destination operand (the first operand). The destination operand is a YMM/ZMM register updated according to the writemask k1. The source operand is either a register or 64-bit/128-bit/256-bit memory location.

VBROADCASTSD and VBROADCASTF128,F32x4 and F64x2 are only supported as 256-bit and 512-bit wide versions and up. VBROADCASTSS is supported in 128-bit, 256-bit and 512-bit wide versions. F32x8 and F64x4 are only supported as 512-bit wide versions.

VBROADCASTF32X2/VBROADCASTF32X4/VBROADCASTF32X8 have 32-bit granularity. VBROADCASTF64X2 and VBROADCASTF64X4 have 64-bit granularity.

Note: VEX.vvvv and EVEX.vvvv are reserved and must be 1111b otherwise instructions will #UD.

If VBROADCASTSD or VBROADCASTF128 is encoded with VEX.L= 0, an attempt to execute the instruction encoded with VEX.L= 0 will cause an #UD exception.

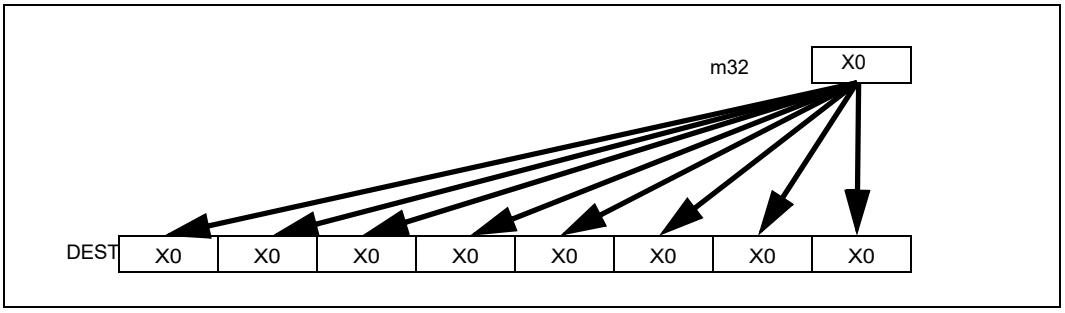


Figure 5-1. VBROADCASTSS Operation (VEX.256 encoded version)

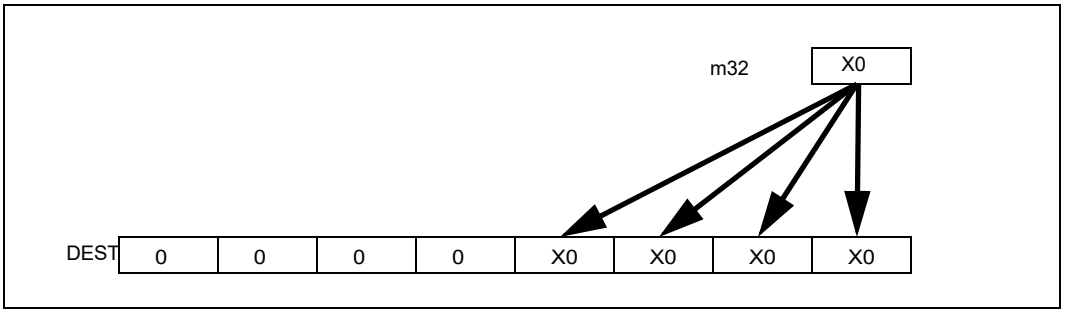


Figure 5-2. VBROADCASTSS Operation (VEX.128-bit version)

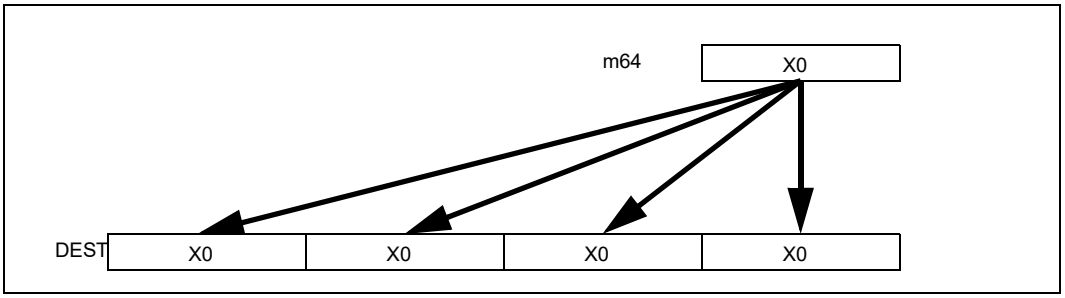


Figure 5-3. VBROADCASTSD Operation (VEX.256-bit version)

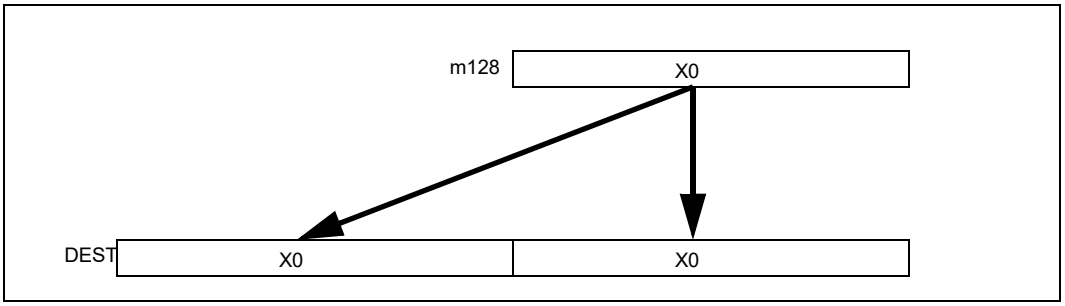


Figure 5-4. VBROADCASTF128 Operation (VEX.256-bit version)

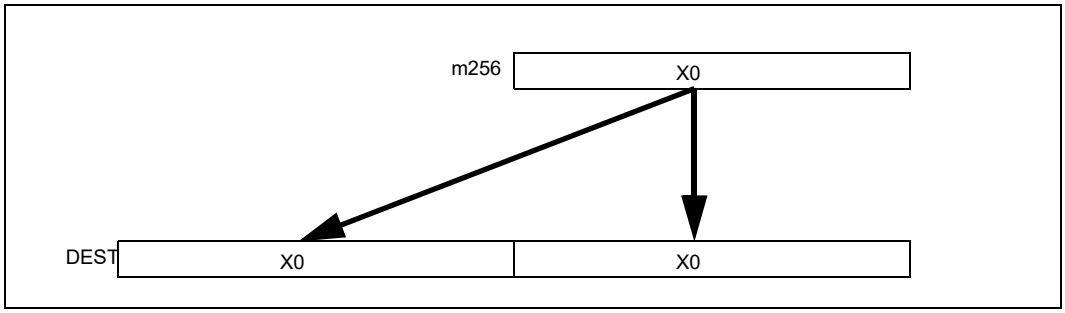


Figure 5-5. VBROADCASTF64X4 Operation (512-bit version with writemask all 1s)

Operation

VBROADCASTSS (128 bit version VEX and legacy)

```
temp ← SRC[31:0]
DEST[31:0] ← temp
DEST[63:32] ← temp
DEST[95:64] ← temp
DEST[127:96] ← temp
DEST[MAXVL-1:128] ← 0
```

VBROADCASTSS (VEX.256 encoded version)

```
temp ← SRC[31:0]
DEST[31:0] ← temp
DEST[63:32] ← temp
DEST[95:64] ← temp
DEST[127:96] ← temp
DEST[159:128] ← temp
DEST[191:160] ← temp
DEST[223:192] ← temp
DEST[255:224] ← temp
DEST[MAXVL-1:256] ← 0
```

VBROADCASTSS (EVEX encoded versions)

```
(KL, VL) (4, 128), (8, 256),= (16, 512)
FOR j ← 0 TO KL-1
    i ←j * 32
    IF k1[j] OR *no writemask*
        THEN DEST[i+31:i] ← SRC[31:0]
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE                            ; zeroing-masking
                    DEST[i+31:i] ← 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0
```

VBROADCASTSD (VEX.256 encoded version)

```
temp ← SRC[63:0]
DEST[63:0] ← temp
DEST[127:64] ← temp
DEST[191:128] ← temp
DEST[255:192] ← temp
DEST[MAXVL-1:256] ← 0
```

VBROADCASTSD (EVEX encoded versions)

```
(KL, VL) = (4, 256), (8, 512)
FOR j ← 0 TO KL-1
    i ←j * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+63:i] ← SRC[63:0]
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE                            ; zeroing-masking
                    DEST[i+63:i] ← 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0
```

VBROADCASTF32x2 (EVEX encoded versions)

```
(KL, VL) = (8, 256), (16, 512)
FOR j ← 0 TO KL-1
    i ←   j * 32
    n ←   (j mod 2) * 32
    IF k1[j] OR *no writemask*
        THEN DEST[i+31:i] ← SRC[n+31:n]
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE                            ; zeroing-masking
                    DEST[i+31:i] ← 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0
```

VBROADCASTF128 (VEX.256 encoded version)

```
temp ← SRC[127:0]
DEST[127:0] ← temp
DEST[255:128] ← temp
DEST[MAXVL-1:256] ← 0
```

VBROADCASTF32X4 (EVEX encoded versions)

```
(KL, VL) = (8, 256), (16, 512)
FOR j ← 0 TO KL-1
    i ←   j* 32
    n ←   (j modulo 4) * 32
    IF k1[j] OR *no writemask*
        THEN DEST[i+31:i] ← SRC[n+31:n]
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE                            ; zeroing-masking
                    DEST[i+31:i] ← 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0
```

VBROADCASTF64X2 (EVEX encoded versions)

```
(KL, VL) = (4, 256), (8, 512)
FOR j ← 0 TO KL-1
    i ← j * 64
    n ←(j modulo 2) * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+63:i] ← SRC[n+63:n]
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE                            ; zeroing-masking
                    DEST[i+63:i] = 0
            FI
    FI;
ENDFOR;
```

VBROADCASTF32X8 (EVEX.U1.512 encoded version)

```
FOR j ← 0 TO 15
    i ← j * 32
    n ←(j modulo 8) * 32
    IF k1[j] OR *no writemask*
        THEN DEST[i+31:i] ← SRC[n+31:n]
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE                            ; zeroing-masking
                    DEST[i+31:i] ← 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0
```

VBROADCASTF64X4 (EVEX.512 encoded version)

```
FOR j ← 0 TO 7
    i ← j * 64
    n ←    (j modulo 4) * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+63:i] ← SRC[n+63:n]
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE                                ; zeroing-masking
                    DEST[i+63:i] ← 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VBROADCASTF32x2 __m512 _mm512_broadcast_f32x2(__m128 a);
VBROADCASTF32x2 __m512 _mm512_mask_broadcast_f32x2(__m512 s, __mmask16 k, __m128 a);
VBROADCASTF32x2 __m512 _mm512_maskz_broadcast_f32x2(__mmask16 k, __m128 a);
VBROADCASTF32x2 __m256 _mm256_broadcast_f32x2(__m128 a);
VBROADCASTF32x2 __m256 _mm256_mask_broadcast_f32x2(__m256 s, __mmask8 k, __m128 a);
VBROADCASTF32x2 __m256 _mm256_maskz_broadcast_f32x2(__mmask8 k, __m128 a);
VBROADCASTF32x4 __m512 _mm512_broadcast_f32x4(__m128 a);
VBROADCASTF32x4 __m512 _mm512_mask_broadcast_f32x4(__m512 s, __mmask16 k, __m128 a);
VBROADCASTF32x4 __m512 _mm512_maskz_broadcast_f32x4(__mmask16 k, __m128 a);
VBROADCASTF32x4 __m256 _mm256_broadcast_f32x4(__m128 a);
VBROADCASTF32x4 __m256 _mm256_mask_broadcast_f32x4(__m256 s, __mmask8 k, __m128 a);
VBROADCASTF32x4 __m256 _mm256_maskz_broadcast_f32x4(__mmask8 k, __m128 a);
VBROADCASTF32x8 __m512 _mm512_broadcast_f32x8(__m256 a);
VBROADCASTF32x8 __m512 _mm512_mask_broadcast_f32x8(__m512 s, __mmask16 k, __m256 a);
VBROADCASTF32x8 __m512 _mm512_maskz_broadcast_f32x8(__mmask16 k, __m256 a);
VBROADCASTF64x2 __m512d _mm512_broadcast_f64x2(__m128d a);
VBROADCASTF64x2 __m512d _mm512_mask_broadcast_f64x2(__m512d s, __mmask8 k, __m128d a);
VBROADCASTF64x2 __m512d _mm512_maskz_broadcast_f64x2(__mmask8 k, __m128d a);
VBROADCASTF64x2 __m256d _mm256_broadcast_f64x2(__m128d a);
VBROADCASTF64x2 __m256d _mm256_mask_broadcast_f64x2(__m256d s, __mmask8 k, __m128d a);
VBROADCASTF64x2 __m256d _mm256_maskz_broadcast_f64x2(__mmask8 k, __m128d a);
VBROADCASTF64x4 __m512d _mm512_broadcast_f64x4(__m256d a);
VBROADCASTF64x4 __m512d _mm512_mask_broadcast_f64x4(__m512d s, __mmask8 k, __m256d a);
VBROADCASTF64x4 __m512d _mm512_maskz_broadcast_f64x4(__mmask8 k, __m256d a);
VBROADCASTSD __m512d _mm512_broadcastsd_pd(__m128d a);
VBROADCASTSD __m512d _mm512_mask_broadcastsd_pd(__m512d s, __mmask8 k, __m128d a);
VBROADCASTSD __m512d _mm512_maskz_broadcastsd_pd(__mmask8 k, __m128d a);
VBROADCASTSD __m256d _mm256_broadcastsd_pd(__m128d a);
VBROADCASTSD __m256d _mm256_mask_broadcastsd_pd(__m256d s, __mmask8 k, __m128d a);
VBROADCASTSD __m256d _mm256_maskz_broadcastsd_pd(__mmask8 k, __m128d a);
VBROADCASTSD __m256d _mm256_broadcast_sd(double *a);
VBROADCASTSS __m512 _mm512_broadcastss_ps(__m128 a);
VBROADCASTSS __m512 _mm512_mask_broadcastss_ps(__m512 s, __mmask16 k, __m128 a);
VBROADCASTSS __m512 _mm512_maskz_broadcastss_ps(__mmask16 k, __m128 a);
VBROADCASTSS __m256 _mm256_broadcastss_ps(__m128 a);
VBROADCASTSS __m256 _mm256_mask_broadcast_ss(__m256 s, __mmask8 k, __m128 a);
VBROADCASTSS __m256 _mm256_maskz_broadcast_ss(__mmask8 k, __m128 a);

VBROADCASTSS __m128 _mm_broadcastss_ps(__m128 a);
VBROADCASTSS __m128 _mm_mask_broadcast_ss(__m128 s, __mmask8 k, __m128 a);
VBROADCASTSS __m128 _mm_maskz_broadcast_ss(__mmask8 k, __m128 a);
VBROADCASTSS __m128 _mm_broadcast_ss(float *a);
VBROADCASTSS __m256 _mm256_broadcast_ss(float *a);
VBROADCASTF128 __m256 _mm256_broadcast_ps(__m128 * a);
VBROADCASTF128 __m256d _mm256_broadcast_pd(__m128d * a);

Exceptions

VEX-encoded instructions, see Exceptions Type 6;
EVEX-encoded instructions, see Exceptions Type E6.

#UD	If VEX.L = 0 for VBROADCASTSD or VBROADCASTF128.
	If EVEX.L'L = 0 for VBROADCASTSD/VBROADCASTF32X2/VBROADCASTF32X4/VBROADCASTF64X2.
	If EVEX.L'L < 10b for VBROADCASTF32X8/VBROADCASTF64X4.

VCOMPRESSPD—Store Sparse Packed Double-Precision Floating-Point Values into Dense Memory

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.128.66.0F38.W1 8A /r VCOMPRESSPD xmm1/m128 {k1}{z}, xmm2	A	V/V	AVX512VL AVX512F	Compress packed double-precision floating-point values from xmm2 to xmm1/m128 using writemask k1.
EVEX.256.66.0F38.W1 8A /r VCOMPRESSPD ymm1/m256 {k1}{z}, ymm2	A	V/V	AVX512VL AVX512F	Compress packed double-precision floating-point values from ymm2 to ymm1/m256 using writemask k1.
EVEX.512.66.0F38.W1 8A /r VCOMPRESSPD zmm1/m512 {k1}{z}, zmm2	A	V/V	AVX512F	Compress packed double-precision floating-point values from zmm2 using control mask k1 to zmm1/m512.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Tuple1 Scalar	ModRM:r/m (w)	ModRM:reg (r)	NA	NA

Description

Compress (store) up to 8 double-precision floating-point values from the source operand (the second operand) as a contiguous vector to the destination operand (the first operand) The source operand is a ZMM/YMM/XMM register, the destination operand can be a ZMM/YMM/XMM register or a 512/256/128-bit memory location.

The opmask register k1 selects the active elements (partial vector or possibly non-contiguous if less than 8 active elements) from the source operand to compress into a contiguous vector. The contiguous vector is written to the destination starting from the low element of the destination operand.

Memory destination version: Only the contiguous vector is written to the destination memory location. EVEX.z must be zero.

Register destination version: If the vector length of the contiguous vector is less than that of the input vector in the source operand, the upper bits of the destination register are unmodified if EVEX.z is not set, otherwise the upper bits are zeroed.

EVEX.vvvv is reserved and must be 1111b otherwise instructions will #UD.

Note that the compressed displacement assumes a pre-scaling (N) corresponding to the size of one single element instead of the size of the full vector.

Operation

VCOMPRESSPD (EVEX encoded versions) store form

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
SIZE ← 64
k ← 0
FOR j ← 0 TO KL-1
    i ← j * 64
    IF k1[j] OR *no writemask*
        THEN
            DEST[k+SIZE-1:k]← SRC[i+63:i]
            k ← k + SIZE
    FI;
ENDFOR
```

VCOMPRESSPD (EVEX encoded versions) reg-reg form

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
SIZE ← 64
k ← 0
FOR j ← 0 TO KL-1
    i ← j * 64
    IF k1[j] OR *no writemask*
        THEN
            DEST[k+SIZE-1:k]← SRC[i+63:i]
            k ← k + SIZE
    FI;
ENDFOR
IF *merging-masking*
        THEN *DEST[VL-1:k] remains unchanged*
        ELSE DEST[VL-1:k] ← 0
FI
DEST[MAXVL-1:VL] ← 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VCOMPRESSPD __m512d _mm512_mask_compress_pd(__m512d s, __mmask8 k, __m512d a);
VCOMPRESSPD __m512d _mm512_maskz_compress_pd(__mmask8 k, __m512d a);
VCOMPRESSPD void _mm512_mask_compressstoreu_pd(void * d, __mmask8 k, __m512d a);
VCOMPRESSPD __m256d _mm256_mask_compress_pd(__m256d s, __mmask8 k, __m256d a);
VCOMPRESSPD __m256d _mm256_maskz_compress_pd(__mmask8 k, __m256d a);
VCOMPRESSPD void _mm256_mask_compressstoreu_pd(void * d, __mmask8 k, __m256d a);
VCOMPRESSPD __m128d _mm_mask_compress_pd(__m128d s, __mmask8 k, __m128d a);
VCOMPRESSPD __m128d _mm_maskz_compress_pd(__mmask8 k, __m128d a);
VCOMPRESSPD void _mm_mask_compressstoreu_pd(void * d, __mmask8 k, __m128d a);

SIMD Floating-Point Exceptions

None

Other Exceptions

EVEX-encoded instructions, see Exceptions Type E4.nb.

#UD	If EVEX.vvvv != 1111B.

VCOMPRESSPS—Store Sparse Packed Single-Precision Floating-Point Values into Dense Memory

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.128.66.0F38.W0 8A /r VCOMPRESSPS xmm1/m128 {k1}{z}, xmm2	A	V/V	AVX512VL AVX512F	Compress packed single-precision floating-point values from xmm2 to xmm1/m128 using writemask k1.
EVEX.256.66.0F38.W0 8A /r VCOMPRESSPS ymm1/m256 {k1}{z}, ymm2	A	V/V	AVX512VL AVX512F	Compress packed single-precision floating-point values from ymm2 to ymm1/m256 using writemask k1.
EVEX.512.66.0F38.W0 8A /r VCOMPRESSPS zmm1/m512 {k1}{z}, zmm2	A	V/V	AVX512F	Compress packed single-precision floating-point values from zmm2 using control mask k1 to zmm1/m512.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Tuple1 Scalar	ModRM:r/m (w)	ModRM:reg (r)	NA	NA

Description

Compress (stores) up to 16 single-precision floating-point values from the source operand (the second operand) to the destination operand (the first operand). The source operand is a ZMM/YMM/XMM register, the destination operand can be a ZMM/YMM/XMM register or a 512/256/128-bit memory location.

The opmask register k1 selects the active elements (a partial vector or possibly non-contiguous if less than 16 active elements) from the source operand to compress into a contiguous vector. The contiguous vector is written to the destination starting from the low element of the destination operand.

Memory destination version: Only the contiguous vector is written to the destination memory location. EVEX.z must be zero.

Register destination version: If the vector length of the contiguous vector is less than that of the input vector in the source operand, the upper bits of the destination register are unmodified if EVEX.z is not set, otherwise the upper bits are zeroed.

EVEX.vvvv is reserved and must be 1111b otherwise instructions will #UD.

Note that the compressed displacement assumes a pre-scaling (N) corresponding to the size of one single element instead of the size of the full vector.

Operation

VCOMPRESSPS (EVEX encoded versions) store form

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
SIZE ← 32
k ← 0
FOR j ← 0 TO KL-1
    i ← j * 32
    IF k1[j] OR *no writemask*
        THEN
            DEST[k+SIZE-1:k]← SRC[i+31:i]
            k ← k + SIZE
    FI;
ENDFOR;
```

VCOMPRESSPS (EVEX encoded versions) reg-reg form

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
SIZE ← 32
k ← 0
FOR j ← 0 TO KL-1
    i ← j * 32
    IF k1[j] OR *no writemask*
        THEN
            DEST[k+SIZE-1:k]← SRC[i+31:i]
            k ← k + SIZE
    FI;
ENDFOR
IF *merging-masking*
    THEN *DEST[VL-1:k] remains unchanged*
    ELSE DEST[VL-1:k] ← 0
FI
DEST[MAXVL-1:VL] ← 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VCOMPRESSPS __m512 _mm512_mask_compress_ps(__m512 s, __mmask16 k, __m512 a);
VCOMPRESSPS __m512 _mm512_maskz_compress_ps(__mmask16 k, __m512 a);
VCOMPRESSPS void _mm512_mask_compressstoreu_ps(void * d, __mmask16 k, __m512 a);
VCOMPRESSPS __m256 _mm256_mask_compress_ps(__m256 s, __mmask8 k, __m256 a);
VCOMPRESSPS __m256 _mm256_maskz_compress_ps(__mmask8 k, __m256 a);
VCOMPRESSPS void _mm256_mask_compressstoreu_ps(void * d, __mmask8 k, __m256 a);
VCOMPRESSPS __m128 _mm_mask_compress_ps(__m128 s, __mmask8 k, __m128 a);
VCOMPRESSPS __m128 _mm_maskz_compress_ps(__mmask8 k, __m128 a);
VCOMPRESSPS void _mm_mask_compressstoreu_ps(void * d, __mmask8 k, __m128 a);

SIMD Floating-Point Exceptions

None

Other Exceptions

EVEX-encoded instructions, see Exceptions Type E4.nb.

#UD	If EVEX.vvvv != 1111B.

VCVTPD2QQ—Convert Packed Double-Precision Floating-Point Values to Packed Quadword Integers

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.128.66.0F.W1 7B /r VCVTPD2QQ xmm1 {k1}{z}, xmm2/m128/m64bcst	A	V/V	AVX512VL AVX512DQ	Convert two packed double-precision floating-point values from xmm2/m128/m64bcst to two packed quadword integers in xmm1 with writemask k1.
EVEX.256.66.0F.W1 7B /r VCVTPD2QQ ymm1 {k1}{z}, ymm2/m256/m64bcst	A	V/V	AVX512VL AVX512DQ	Convert four packed double-precision floating-point values from ymm2/m256/m64bcst to four packed quadword integers in ymm1 with writemask k1.
EVEX.512.66.0F.W1 7B /r VCVTPD2QQ zmm1 {k1}{z}, zmm2/m512/m64bcst{er}	A	V/V	AVX512DQ	Convert eight packed double-precision floating-point values from zmm2/m512/m64bcst to eight packed quadword integers in zmm1 with writemask k1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Full Vector	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

Description

Converts packed double-precision floating-point values in the source operand (second operand) to packed quad-word integers in the destination operand (first operand).

EVEX encoded versions: The source operand is a ZMM/YMM/XMM register or a 512/256/128-bit memory location. The destination operation is a ZMM/YMM/XMM register conditionally updated with writemask k1.

When a conversion is inexact, the value returned is rounded according to the rounding control bits in the MXCSR register or the embedded rounding control bits. If a converted result cannot be represented in the destination format, the floating-point invalid exception is raised, and if this exception is masked, the indefinite integer value (2^{w-1}, where w represents the number of bits in the destination format) is returned.

EVEX.vvvv is reserved and must be 1111b otherwise instructions will #UD.

Operation

VCVTPD2QQ (EVEX encoded version) when src operand is a register

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
IF (VL == 512) AND (EVEX.b == 1)
    THEN
        SET_RM(EVEX.RC);
    ELSE
        SET_RM(MXCSR.RM);
FI;

FOR j ← 0 TO KL-1
    i ← j * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+63:i] ←
            Convert_Double_Precision_Floating_Point_To_QuadInteger(SRC[i+63:i])
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE                            ; zeroing-masking
                    DEST[i+63:i] ← 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0
```

VCVTPD2QQ (EVEX encoded version) when src operand is a memory source

```
(KL, VL) = (2, 128), (4, 256), (8, 512)

FOR j ← 0 TO KL-1
    i ← j * 64
    IF k1[j] OR *no writemask*
        THEN
            IF (EVEX.b == 1)
                THEN
                    DEST[i+63:i] ←          Convert_Double_Precision_Floating_Point_To_QuadInteger(SRC[63:0])
                ELSE
                    DEST[i+63:i] ← Convert_Double_Precision_Floating_Point_To_QuadInteger(SRC[i+63:i])
            FI;
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE                            ; zeroing-masking
                    DEST[i+63:i] ← 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VCVTPD2QQ __m512i _mm512_cvtpd_epi64(__m512d a);
VCVTPD2QQ __m512i _mm512_mask_cvtpd_epi64(__m512i s, __mmask8 k, __m512d a);
VCVTPD2QQ __m512i _mm512_maskz_cvtpd_epi64(__mmask8 k, __m512d a);
VCVTPD2QQ __m512i _mm512_cvt_roundpd_epi64(__m512d a, int r);
VCVTPD2QQ __m512i _mm512_mask_cvt_roundpd_epi64(__m512i s, __mmask8 k, __m512d a, int r);
VCVTPD2QQ __m512i _mm512_maskz_cvt_roundpd_epi64(__mmask8 k, __m512d a, int r);
VCVTPD2QQ __m256i _mm256_mask_cvtpd_epi64(__m256i s, __mmask8 k, __m256d a);
VCVTPD2QQ __m256i _mm256_maskz_cvtpd_epi64(__mmask8 k, __m256d a);
VCVTPD2QQ __m128i _mm_mask_cvtpd_epi64(__m128i s, __mmask8 k, __m128d a);
VCVTPD2QQ __m128i _mm_maskz_cvtpd_epi64(__mmask8 k, __m128d a);
VCVTPD2QQ __m256i _mm256_cvtpd_epi64 (__m256d src)
VCVTPD2QQ __m128i _mm_cvtpd_epi64 (__m128d src)

SIMD Floating-Point Exceptions

Invalid, Precision

Other Exceptions

EVEX-encoded instructions, see Exceptions Type E2

#UD	If EVEX.vvvv != 1111B.

VCVTPD2UDQ—Convert Packed Double-Precision Floating-Point Values to Packed Unsigned Doubleword Integers

Opcode Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.128.0F.W1 79 /r VCVTPD2UDQ xmm1 {k1}{z}, xmm2/m128/m64bcst	A	V/V	AVX512VL AVX512F	Convert two packed double-precision floating-point values in xmm2/m128/m64bcst to two unsigned doubleword integers in xmm1 subject to writemask k1.
EVEX.256.0F.W1 79 /r VCVTPD2UDQ xmm1 {k1}{z}, ymm2/m256/m64bcst	A	V/V	AVX512VL AVX512F	Convert four packed double-precision floating-point values in ymm2/m256/m64bcst to four unsigned doubleword integers in xmm1 subject to writemask k1.
EVEX.512.0F.W1 79 /r VCVTPD2UDQ ymm1 {k1}{z}, zmm2/m512/m64bcst{er}	A	V/V	AVX512F	Convert eight packed double-precision floating-point values in zmm2/m512/m64bcst to eight unsigned doubleword integers in ymm1 subject to writemask k1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Full Vector	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

Description

Converts packed double-precision floating-point values in the source operand (the second operand) to packed unsigned doubleword integers in the destination operand (the first operand).

When a conversion is inexact, the value returned is rounded according to the rounding control bits in the MXCSR register or the embedded rounding control bits. If a converted result cannot be represented in the destination format, the floating-point invalid exception is raised, and if this exception is masked, the integer value $2^w - 1$ is returned, where w represents the number of bits in the destination format.

The source operand is a ZMM/YMM/XMM register, a 512/256/128-bit memory location, or a 512/256/128-bit vector broadcasted from a 64-bit memory location. The destination operand is a ZMM/YMM/XMM register conditionally updated with writemask k1. The upper bits (MAXVL-1:256) of the corresponding destination are zeroed.

EVEX.vvvv is reserved and must be 1111b otherwise instructions will #UD.

Operation

VCVTPD2UDQ (EVEX encoded versions) when src2 operand is a register

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
IF (VL = 512) AND (EVEX.b = 1)
    THEN
        SET_RM(EVEX.RC);
    ELSE
        SET_RM(MXCSR.RM);
FI;

FOR j ← 0 TO KL-1
    i ← j * 32
    k ← j * 64
    IF k1[j] OR *no writemask*
        THEN
            DEST[i+31:i] ←
            Convert_Double_Precision_Floating_Point_To_UInteger(SRC[k+63:k])
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE                            ; zeroing-masking
                    DEST[i+31:i] ← 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL/2] ← 0
```

VCVTPD2UDQ (EVEX encoded versions) when src operand is a memory source

```
(KL, VL) = (2, 128), (4, 256), (8, 512)

FOR j ← 0 TO KL-1
    i ← j * 32
    k ← j * 64
    IF k1[j] OR *no writemask*
        THEN
            IF (EVEX.b = 1)
                THEN
                    DEST[i+31:i] ←
            Convert_Double_Precision_Floating_Point_To_UInteger(SRC[63:0])
                ELSE
                    DEST[i+31:i] ←
            Convert_Double_Precision_Floating_Point_To_UInteger(SRC[k+63:k])
            FI;
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE                            ; zeroing-masking
                    DEST[i+31:i] ← 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL/2] ← 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VCVTPD2UDQ __m256i _mm512_cvtpd_epu32(__m512d a);
VCVTPD2UDQ __m256i _mm512_mask_cvtpd_epu32(__m256i s, __mmask8 k, __m512d a);
VCVTPD2UDQ __m256i _mm512_maskz_cvtpd_epu32(__mmask8 k, __m512d a);
VCVTPD2UDQ __m256i _mm512_cvt_roundpd_epu32(__m512d a, int r);
VCVTPD2UDQ __m256i _mm512_mask_cvt_roundpd_epu32(__m256i s, __mmask8 k, __m512d a, int r);
VCVTPD2UDQ __m256i _mm512_maskz_cvt_roundpd_epu32(__mmask8 k, __m512d a, int r);
VCVTPD2UDQ __m128i _mm256_mask_cvtpd_epu32(__m128i s, __mmask8 k, __m256d a);
VCVTPD2UDQ __m128i _mm256_maskz_cvtpd_epu32(__mmask8 k, __m256d a);
VCVTPD2UDQ __m128i _mm_mask_cvtpd_epu32(__m128i s, __mmask8 k, __m128d a);
VCVTPD2UDQ __m128i _mm_maskz_cvtpd_epu32(__mmask8 k, __m128d a);

SIMD Floating-Point Exceptions

Invalid, Precision

Other Exceptions

EVEX-encoded instructions, see Exceptions Type E2.

#UD	If EVEX.vvvv != 1111B.

VCVTPD2UQQ—Convert Packed Double-Precision Floating-Point Values to Packed Unsigned Quadword Integers

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.128.66.0F.W1 79 /r VCVTPD2UQQ xmm1 {k1}{z}, xmm2/m128/m64bcst	A	V/V	AVX512VL AVX512DQ	Convert two packed double-precision floating-point values from xmm2/mem to two packed unsigned quadword integers in xmm1 with writemask k1.
EVEX.256.66.0F.W1 79 /r VCVTPD2UQQ ymm1 {k1}{z}, ymm2/m256/m64bcst	A	V/V	AVX512VL AVX512DQ	Convert fourth packed double-precision floating-point values from ymm2/mem to four packed unsigned quadword integers in ymm1 with writemask k1.
EVEX.512.66.0F.W1 79 /r VCVTPD2UQQ zmm1 {k1}{z}, zmm2/m512/m64bcst{er}	A	V/V	AVX512DQ	Convert eight packed double-precision floating-point values from zmm2/mem to eight packed unsigned quadword integers in zmm1 with writemask k1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Full Vector	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

Description

Converts packed double-precision floating-point values in the source operand (second operand) to packed unsigned quadword integers in the destination operand (first operand).

When a conversion is inexact, the value returned is rounded according to the rounding control bits in the MXCSR register or the embedded rounding control bits. If a converted result cannot be represented in the destination format, the floating-point invalid exception is raised, and if this exception is masked, the integer value $2^w - 1$ is returned, where w represents the number of bits in the destination format.

The source operand is a ZMM/YMM/XMM register or a 512/256/128-bit memory location. The destination operation is a ZMM/YMM/XMM register conditionally updated with writemask k1.

EVEX.vvvv is reserved and must be 1111b otherwise instructions will #UD.

Operation

VCVTPD2UQQ (EVEX encoded versions) when src operand is a register

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
IF (VL == 512) AND (EVEX.b == 1)
    THEN
        SET_RM(EVEX.RC);
    ELSE
        SET_RM(MXCSR.RM);
FI;

FOR j ← 0 TO KL-1
    i ← j * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+63:i] ←
            Convert_Double_Precision_Floating_Point_To_UQuadInteger(SRC[i+63:i])
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE                            ; zeroing-masking
                    DEST[i+63:i] ← 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0
```

VCVTPD2UQQ (EVEX encoded versions) when src operand is a memory source

```
(KL, VL) = (2, 128), (4, 256), (8, 512)

FOR j ← 0 TO KL-1
    i ← j * 64
    IF k1[j] OR *no writemask*
        THEN
            IF (EVEX.b == 1)
                THEN
                    DEST[i+63:i] ←
            Convert_Double_Precision_Floating_Point_To_UQuadInteger(SRC[63:0])
                ELSE
                    DEST[i+63:i] ←
            Convert_Double_Precision_Floating_Point_To_UQuadInteger(SRC[i+63:i])
            FI;
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE                            ; zeroing-masking
                    DEST[i+63:i] ← 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VCVTPD2UQQ __m512i _mm512_cvtpd_epu64(__m512d a);
VCVTPD2UQQ __m512i _mm512_mask_cvtpd_epu64(__m512i s, __mmask8 k, __m512d a);
VCVTPD2UQQ __m512i _mm512_maskz_cvtpd_epu64(__mmask8 k, __m512d a);
VCVTPD2UQQ __m512i _mm512_cvt_roundpd_epu64(__m512d a, int r);
VCVTPD2UQQ __m512i _mm512_mask_cvt_roundpd_epu64(__m512i s, __mmask8 k, __m512d a, int r);
VCVTPD2UQQ __m512i _mm512_maskz_cvt_roundpd_epu64(__mmask8 k, __m512d a, int r);
VCVTPD2UQQ __m256i _mm256_mask_cvtpd_epu64(__m256i s, __mmask8 k, __m256d a);
VCVTPD2UQQ __m256i _mm256_maskz_cvtpd_epu64(__mmask8 k, __m256d a);
VCVTPD2UQQ __m128i _mm_mask_cvtpd_epu64(__m128i s, __mmask8 k, __m128d a);
VCVTPD2UQQ __m128i _mm_maskz_cvtpd_epu64(__mmask8 k, __m128d a);
VCVTPD2UQQ __m256i _mm256_cvtpd_epu64 (__m256d src)
VCVTPD2UQQ __m128i _mm_cvtpd_epu64 (__m128d src)

SIMD Floating-Point Exceptions

Invalid, Precision

Other Exceptions

EVEX-encoded instructions, see Exceptions Type E2

#UD	If EVEX.vvvv != 1111B.

VCVTPH2PS—Convert 16-bit FP values to Single-Precision FP values

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
VEX.128.66.0F38.W0 13 /r VCVTPH2PS xmm1, xmm2/m64	A	V/V	F16C	Convert four packed half precision (16-bit) floating-point values in xmm2/m64 to packed single-precision floating-point value in xmm1.
VEX.256.66.0F38.W0 13 /r VCVTPH2PS ymm1, xmm2/m128	A	V/V	F16C	Convert eight packed half precision (16-bit) floating-point values in xmm2/m128 to packed single-precision floating-point value in ymm1.
EVEX.128.66.0F38.W0 13 /r VCVTPH2PS xmm1 {k1}{z}, xmm2/m64	B	V/V	AVX512VL AVX512F	Convert four packed half precision (16-bit) floating-point values in xmm2/m64 to packed single-precision floating-point values in xmm1.
EVEX.256.66.0F38.W0 13 /r VCVTPH2PS ymm1 {k1}{z}, xmm2/m128	B	V/V	AVX512VL AVX512F	Convert eight packed half precision (16-bit) floating-point values in xmm2/m128 to packed single-precision floating-point values in ymm1.
EVEX.512.66.0F38.W0 13 /r VCVTPH2PS zmm1 {k1}{z}, ymm2/m256 {sae}	B	V/V	AVX512F	Convert sixteen packed half precision (16-bit) floating-point values in ymm2/m256 to packed single-precision floating-point values in zmm1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	NA	ModRM:reg (w)	ModRM:r/m (r)	NA	NA
B	Half Vector Mem	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

Description

Converts packed half precision (16-bits) floating-point values in the low-order bits of the source operand (the second operand) to packed single-precision floating-point values and writes the converted values into the destination operand (the first operand).

If case of a denormal operand, the correct normal result is returned. MXCSR.DAZ is ignored and is treated as if it 0. No denormal exception is reported on MXCSR.

VEX.128 version: The source operand is a XMM register or 64-bit memory location. The destination operand is a XMM register. The upper bits (MAXVL-1:128) of the corresponding destination register are zeroed.

VEX.256 version: The source operand is a XMM register or 128-bit memory location. The destination operand is a YMM register. Bits (MAXVL-1:256) of the corresponding destination register are zeroed.

EVEX encoded versions: The source operand is a YMM/XMM/XMM (low 64-bits) register or a 256/128/64-bit memory location. The destination operand is a ZMM/YMM/XMM register conditionally updated with writemask k1.

The diagram below illustrates how data is converted from four packed half precision (in 64 bits) to four single precision (in 128 bits) FP values.

Note: VEX.vvvv and EVEX.vvvv are reserved (must be 1111b).

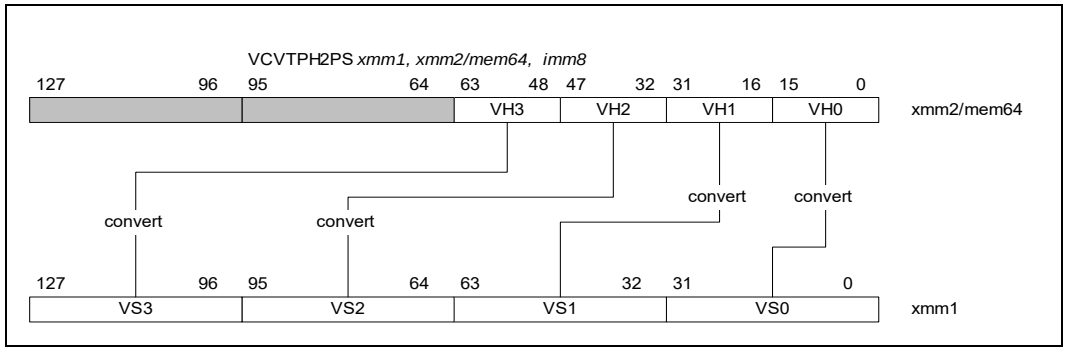


Figure 5-6. VCVTPH2PS (128-bit Version)

Operation

```
vCvt_h2s(SRC1[15:0])
{
RETURN Cvt_Half_Precision_To_Single_Precision(SRC1[15:0]);
}
```

VCVTPH2PS (EVEX encoded versions)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j ← 0 TO KL-1
    i ← j * 32
    k ← j * 16
    IF k1[j] OR *no writemask*
        THEN DEST[i+31:i] ←
            vCvt_h2s(SRC[k+15:k])
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE                            ; zeroing-masking
                    DEST[i+31:i] ← 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0
```

VCVTPH2PS (VEX.256 encoded version)

```
DEST[31:0] ←vCvt_h2s(SRC1[15:0]);
DEST[63:32] ←vCvt_h2s(SRC1[31:16]);
DEST[95:64] ←vCvt_h2s(SRC1[47:32]);
DEST[127:96] ←vCvt_h2s(SRC1[63:48]);
DEST[159:128] ←vCvt_h2s(SRC1[79:64]);
DEST[191:160] ←vCvt_h2s(SRC1[95:80]);
DEST[223:192] ←vCvt_h2s(SRC1[111:96]);
DEST[255:224] ←vCvt_h2s(SRC1[127:112]);
DEST[MAXVL-1:256] ← 0
```

VCVTPH2PS (VEX.128 encoded version)

```
DEST[31:0] ←vCvt_h2s(SRC1[15:0]);
DEST[63:32] ←vCvt_h2s(SRC1[31:16]);
DEST[95:64] ←vCvt_h2s(SRC1[47:32]);
DEST[127:96] ←vCvt_h2s(SRC1[63:48]);
DEST[MAXVL-1:128] ← 0
```

Flags Affected

None

Intel C/C++ Compiler Intrinsic Equivalent

VCVTPH2PS __m512 _mm512_cvtph_ps(__m256i a);
VCVTPH2PS __m512 _mm512_mask_cvtph_ps(__m512 s, __mmask16 k, __m256i a);
VCVTPH2PS __m512 _mm512_maskz_cvtph_ps(__mmask16 k, __m256i a);
VCVTPH2PS __m512 _mm512_cvt_roundph_ps(__m256i a, int sae);
VCVTPH2PS __m512 _mm512_mask_cvt_roundph_ps(__m512 s, __mmask16 k, __m256i a, int sae);
VCVTPH2PS __m512 _mm512_maskz_cvt_roundph_ps(__mmask16 k, __m256i a, int sae);
VCVTPH2PS __m256 _mm256_mask_cvtph_ps(__m256 s, __mmask8 k, __m128i a);
VCVTPH2PS __m256 _mm256_maskz_cvtph_ps(__mmask8 k, __m128i a);
VCVTPH2PS __m128 _mm_mask_cvtph_ps(__m128 s, __mmask8 k, __m128i a);
VCVTPH2PS __m128 _mm_maskz_cvtph_ps(__mmask8 k, __m128i a);
VCVTPH2PS __m128 _mm_cvtph_ps (__m128i m1);
VCVTPH2PS __m256 _mm256_cvtph_ps (__m128i m1)

SIMD Floating-Point Exceptions

Invalid

Other Exceptions

VEX-encoded instructions, see Exceptions Type 11 (do not report #AC);

EVEX-encoded instructions, see Exceptions Type E11.

#UD	If VEX.W=1.
#UD	If VEX.vvvv != 1111B or EVEX.vvvv != 1111B.

VCVTPS2PH—Convert Single-Precision FP value to 16-bit FP value

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
VEX.128.66.0F3A.W0 1D /r ib VCVTPS2PH xmm1/m64, xmm2, imm8	A	V/V	F16C	Convert four packed single-precision floating-point values in xmm2 to packed half-precision (16-bit) floating-point values in xmm1/m64. Imm8 provides rounding controls.
VEX.256.66.0F3A.W0 1D /r ib VCVTPS2PH xmm1/m128, ymm2, imm8	A	V/V	F16C	Convert eight packed single-precision floating-point values in ymm2 to packed half-precision (16-bit) floating-point values in xmm1/m128. Imm8 provides rounding controls.
EVEX.128.66.0F3A.W0 1D /r ib VCVTPS2PH xmm1/m64 {k1}{z}, xmm2, imm8	B	V/V	AVX512VL AVX512F	Convert four packed single-precision floating-point values in xmm2 to packed half-precision (16-bit) floating-point values in xmm1/m64. Imm8 provides rounding controls.
EVEX.256.66.0F3A.W0 1D /r ib VCVTPS2PH xmm1/m128 {k1}{z}, ymm2, imm8	B	V/V	AVX512VL AVX512F	Convert eight packed single-precision floating-point values in ymm2 to packed half-precision (16-bit) floating-point values in xmm1/m128. Imm8 provides rounding controls.
EVEX.512.66.0F3A.W0 1D /r ib VCVTPS2PH ymm1/m256 {k1}{z}, zmm2{sae}, imm8	B	V/V	AVX512F	Convert sixteen packed single-precision floating-point values in zmm2 to packed half-precision (16-bit) floating-point values in ymm1/m256. Imm8 provides rounding controls.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	NA	ModRM:r/m (w)	ModRM:reg (r)	Imm8	NA
B	Half Vector Mem	ModRM:r/m (w)	ModRM:reg (r)	Imm8	NA

Description

Convert packed single-precision floating values in the source operand to half-precision (16-bit) floating-point values and store to the destination operand. The rounding mode is specified using the immediate field (imm8).

Underflow results (i.e., tiny results) are converted to denormals. MXCSR.FTZ is ignored. If a source element is denormal relative to the input format with DM masked and at least one of PM or UM unmasked; a SIMD exception will be raised with DE, UE and PE set.

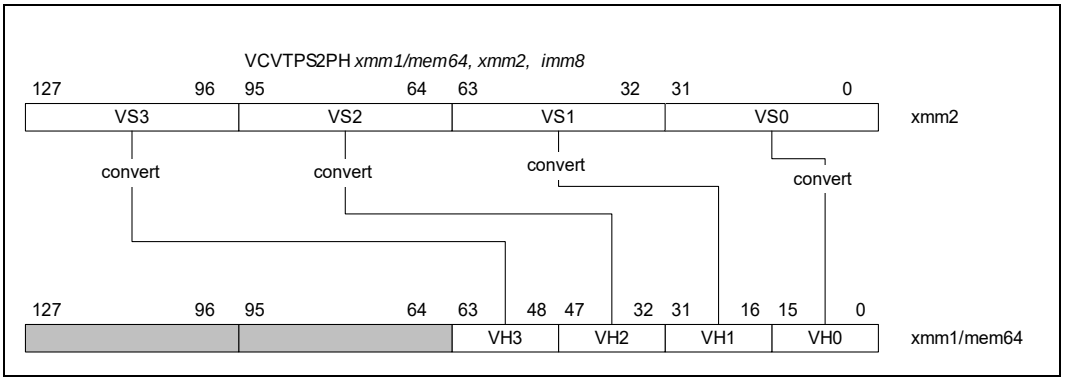


Figure 5-7. VCVTPS2PH (128-bit Version)

The immediate byte defines several bit fields that control rounding operation. The effect and encoding of the RC field are listed in Table 5-3.

Table 5-3. Immediate Byte Encoding for 16-bit Floating-Point Conversion Instructions

Bits	Field Name/value	Description	Comment
Imm[1:0]	RC=00B	Round to nearest even	If Imm[2] = 0
	RC=01B	Round down	
	RC=10B	Round up	
	RC=11B	Truncate	
Imm[2]	MS1=0	Use imm[1:0] for rounding	Ignore MXCSR.RC
	MS1=1	Use MXCSR.RC for rounding	
Imm[7:3]	Ignored	Ignored by processor	

VEX.128 version: The source operand is a XMM register. The destination operand is a XMM register or 64-bit memory location. If the destination operand is a register then the upper bits (MAXVL-1:64) of corresponding register are zeroed.

VEX.256 version: The source operand is a YMM register. The destination operand is a XMM register or 128-bit memory location. If the destination operand is a register, the upper bits (MAXVL-1:128) of the corresponding destination register are zeroed.

Note: VEX.vvvv and EVEX.vvvv are reserved (must be 1111b).

EVEX encoded versions: The source operand is a ZMM/YMM/XMM register. The destination operand is a YMM/XMM/XMM (low 64-bits) register or a 256/128/64-bit memory location, conditionally updated with writemask k1. Bits (MAXVL-1:256/128/64) of the corresponding destination register are zeroed.

Operation

```
vCvt_s2h(SRC1[31:0])
{
IF Imm[2] = 0
THEN    ; using Imm[1:0] for rounding control, see Table 5-3
    RETURN Cvt_Single_Precision_To_Half_Precision_FP_Imm(SRC1[31:0]);
ELSE    ; using MXCSR.RC for rounding control
    RETURN Cvt_Single_Precision_To_Half_Precision_FP_Mxcsr(SRC1[31:0]);
FI;
}
```

VCVTPS2PH (EVEX encoded versions) when dest is a register

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j ← 0 TO KL-1
    i ← j * 16
    k ← j * 32
    IF k1[j] OR *no writemask*
        THEN DEST[i+15:i] ←
            vCvt_s2h(SRC[k+31:k])
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+15:i] remains unchanged*
                ELSE                                ; zeroing-masking
                    DEST[i+15:i] ← 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL/2] ← 0
```

VCVTPS2PH (EVEX encoded versions) when dest is memory

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j ← 0 TO KL-1
    i ← j * 16
    k ← j * 32
    IF k1[j] OR *no writemask*
        THEN DEST[i+15:i] ←
            vCvt_s2h(SRC[k+31:k])
        ELSE
            *DEST[i+15:i] remains unchanged*     ; merging-masking
    FI;
ENDFOR
```

VCVTPS2PH (VEX.256 encoded version)

```
DEST[15:0] ←vCvt_s2h(SRC1[31:0]);
DEST[31:16] ←vCvt_s2h(SRC1[63:32]);
DEST[47:32] ←vCvt_s2h(SRC1[95:64]);
DEST[63:48] ←vCvt_s2h(SRC1[127:96]);
DEST[79:64] ←vCvt_s2h(SRC1[159:128]);
DEST[95:80] ←vCvt_s2h(SRC1[191:160]);
DEST[111:96] ←vCvt_s2h(SRC1[223:192]);
DEST[127:112] ←vCvt_s2h(SRC1[255:224]);
DEST[MAXVL-1:128] ← 0
```

VCVTPS2PH (VEX.128 encoded version)

```
DEST[15:0] ←vCvt_s2h(SRC1[31:0]);
DEST[31:16] ←vCvt_s2h(SRC1[63:32]);
DEST[47:32] ←vCvt_s2h(SRC1[95:64]);
DEST[63:48] ←vCvt_s2h(SRC1[127:96]);
DEST[MAXVL-1:64] ← 0
```

Flags Affected

None

Intel C/C++ Compiler Intrinsic Equivalent

VCVTPS2PH __m256i _mm512_cvtps_ph(__m512 a);
VCVTPS2PH __m256i _mm512_mask_cvtps_ph(__m256i s, __mmask16 k,__m512 a);
VCVTPS2PH __m256i _mm512_maskz_cvtps_ph(__mmask16 k,__m512 a);
VCVTPS2PH __m256i _mm512_cvt_roundps_ph(__m512 a, const int imm);
VCVTPS2PH __m256i _mm512_mask_cvt_roundps_ph(__m256i s, __mmask16 k,__m512 a, const int imm);
VCVTPS2PH __m256i _mm512_maskz_cvt_roundps_ph(__mmask16 k,__m512 a, const int imm);
VCVTPS2PH __m128i _mm256_mask_cvtps_ph(__m128i s, __mmask8 k,__m256 a);
VCVTPS2PH __m128i _mm256_maskz_cvtps_ph(__mmask8 k,__m256 a);
VCVTPS2PH __m128i _mm_mask_cvtps_ph(__m128i s, __mmask8 k,__m128 a);
VCVTPS2PH __m128i _mm_maskz_cvtps_ph(__mmask8 k,__m128 a);
VCVTPS2PH __m128i _mm_cvtps_ph (__m128 m1, const int imm);
VCVTPS2PH __m128i _mm256_cvtps_ph(__m256 m1, const int imm);

SIMD Floating-Point Exceptions

Invalid, Underflow, Overflow, Precision, Denormal (if MXCSR.DAZ=0);

Other Exceptions

VEX-encoded instructions, see Exceptions Type 11 (do not report #AC);

EVEX-encoded instructions, see Exceptions Type E11.

#UD	If VEX.W=1.
#UD	If VEX.vvvv != 1111B or EVEX.vvvv != 1111B.

VCVTPS2UDQ—Convert Packed Single-Precision Floating-Point Values to Packed Unsigned Doubleword Integer Values

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.128.0F.W0 79 /r VCVTPS2UDQ xmm1 {k1}{z}, xmm2/m128/m32bcst	A	V/V	AVX512VL AVX512F	Convert four packed single precision floating-point values from xmm2/m128/m32bcst to four packed unsigned doubleword values in xmm1 subject to writemask k1.
EVEX.256.0F.W0 79 /r VCVTPS2UDQ ymm1 {k1}{z}, ymm2/m256/m32bcst	A	V/V	AVX512VL AVX512F	Convert eight packed single precision floating-point values from ymm2/m256/m32bcst to eight packed unsigned doubleword values in ymm1 subject to writemask k1.
EVEX.512.0F.W0 79 /r VCVTPS2UDQ zmm1 {k1}{z}, zmm2/m512/m32bcst{er}	A	V/V	AVX512F	Convert sixteen packed single-precision floating-point values from zmm2/m512/m32bcst to sixteen packed unsigned doubleword values in zmm1 subject to writemask k1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Full Vector	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

Description

Converts sixteen packed single-precision floating-point values in the source operand to sixteen unsigned double-word integers in the destination operand.

When a conversion is inexact, the value returned is rounded according to the rounding control bits in the MXCSR register or the embedded rounding control bits. If a converted result cannot be represented in the destination format, the floating-point invalid exception is raised, and if this exception is masked, the integer value $2^w - 1$ is returned, where w represents the number of bits in the destination format.

The source operand is a ZMM/YMM/XMM register, a 512/256/128-bit memory location, or a 512/256/128-bit vector broadcasted from a 32-bit memory location. The destination operand is a ZMM/YMM/XMM register conditionally updated with writemask k1.

Note: EVEX.vvvv is reserved and must be 1111b otherwise instructions will #UD.

Operation

VCVTPS2UDQ (EVEX encoded versions) when src operand is a register

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
IF (VL = 512) AND (EVEX.b = 1)
    THEN
        SET_RM(EVEX.RC);
    ELSE
        SET_RM(MXCSR.RM);
FI;

FOR j ← 0 TO KL-1
    i ← j * 32
    IF k1[j] OR *no writemask*
        THEN DEST[i+31:i] ←
            Convert_Single_Precision_Floating_Point_To_UInteger(SRC[i+31:i])
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE                            ; zeroing-masking
                    DEST[i+31:i] ← 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0
```

VCVTPS2UDQ (EVEX encoded versions) when src operand is a memory source

```
(KL, VL) = (4, 128), (8, 256), (16, 512)

FOR j ← 0 TO KL-1
    i ← j * 32
    IF k1[j] OR *no writemask*
        THEN
            IF (EVEX.b = 1)
                THEN
                    DEST[i+31:i] ←
            Convert_Single_Precision_Floating_Point_To_UInteger(SRC[31:0])
                ELSE
                    DEST[i+31:i] ←
            Convert_Single_Precision_Floating_Point_To_UInteger(SRC[i+31:i])
            FI;
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE                            ; zeroing-masking
                    DEST[i+31:i] ← 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VCVTPS2UDQ __m512i _mm512_cvtps_epu32(__m512 a);
VCVTPS2UDQ __m512i _mm512_mask_cvtps_epu32(__m512i s, __mmask16 k, __m512 a);
VCVTPS2UDQ __m512i _mm512_maskz_cvtps_epu32(__mmask16 k, __m512 a);
VCVTPS2UDQ __m512i _mm512_cvt_roundps_epu32(__m512 a, int r);
VCVTPS2UDQ __m512i _mm512_mask_cvt_roundps_epu32(__m512i s, __mmask16 k, __m512 a, int r);
VCVTPS2UDQ __m512i _mm512_maskz_cvt_roundps_epu32(__mmask16 k, __m512 a, int r);
VCVTPS2UDQ __m256i _mm256_cvtps_epu32(__m256d a);
VCVTPS2UDQ __m256i _mm256_mask_cvtps_epu32(__m256i s, __mmask8 k, __m256 a);
VCVTPS2UDQ __m256i _mm256_maskz_cvtps_epu32(__mmask8 k, __m256 a);
VCVTPS2UDQ __m128i _mm_cvtps_epu32(__m128 a);
VCVTPS2UDQ __m128i _mm_mask_cvtps_epu32(__m128i s, __mmask8 k, __m128 a);
VCVTPS2UDQ __m128i _mm_maskz_cvtps_epu32(__mmask8 k, __m128 a);

SIMD Floating-Point Exceptions

Invalid, Precision

Other Exceptions

EVEX-encoded instructions, see Exceptions Type E2.

#UD	If EVEX.vvvv != 1111B.

VCVTPS2QQ—Convert Packed Single Precision Floating-Point Values to Packed Singed Quadword Integer Values

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.128.66.0F.W0 7B /r VCVTPS2QQ xmm1 {k1}{z}, xmm2/m64/m32bcst	A	V/V	AVX512VL AVX512DQ	Convert two packed single precision floating-point values from xmm2/m64/m32bcst to two packed signed quadword values in xmm1 subject to writemask k1.
EVEX.256.66.0F.W0 7B /r VCVTPS2QQ ymm1 {k1}{z}, xmm2/m128/m32bcst	A	V/V	AVX512VL AVX512DQ	Convert four packed single precision floating-point values from xmm2/m128/m32bcst to four packed signed quadword values in ymm1 subject to writemask k1.
EVEX.512.66.0F.W0 7B /r VCVTPS2QQ zmm1 {k1}{z}, ymm2/m256/m32bcst{er}	A	V/V	AVX512DQ	Convert eight packed single precision floating-point values from ymm2/m256/m32bcst to eight packed signed quadword values in zmm1 subject to writemask k1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Half Vector	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

Description

Converts eight packed single-precision floating-point values in the source operand to eight signed quadword integers in the destination operand.

When a conversion is inexact, the value returned is rounded according to the rounding control bits in the MXCSR register or the embedded rounding control bits. If a converted result cannot be represented in the destination format, the floating-point invalid exception is raised, and if this exception is masked, the indefinite integer value (2^{w-1}, where w represents the number of bits in the destination format) is returned.

The source operand is a YMM/XMM/XMM (low 64- bits) register or a 256/128/64-bit memory location. The destination operation is a ZMM/YMM/XMM register conditionally updated with writemask k1.

Note: EVEX.vvvv is reserved and must be 1111b otherwise instructions will #UD.

Operation

VCVTPS2QQ (EVEX encoded versions) when src operand is a register

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
IF (VL == 512) AND (EVEX.b == 1)
    THEN
        SET_RM(EVEX.RC);
    ELSE
        SET_RM(MXCSR.RM);
FI;
FOR j ← 0 TO KL-1
    i ← j * 64
    k ← j * 32
    IF k1[j] OR *no writemask*
        THEN DEST[i+63:i] ←
            Convert_Single_Precision_To_QuadInteger(SRC[k+31:k])
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE                                ; zeroing-masking
                    DEST[i+63:i] ← 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0
```

VCVTPS2QQ (EVEX encoded versions) when src operand is a memory source

```
(KL, VL) = (2, 128), (4, 256), (8, 512)

FOR j ← 0 TO KL-1
    i ← j * 64
    k ← j * 32
    IF k1[j] OR *no writemask*
        THEN
            IF (EVEX.b == 1)
                THEN
                    DEST[i+63:i] ←
            Convert_Single_Precision_To_QuadInteger(SRC[31:0])
                ELSE
                    DEST[i+63:i] ←
            Convert_Single_Precision_To_QuadInteger(SRC[k+31:k])
            FI;
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE                                ; zeroing-masking
                    DEST[i+63:i] ← 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VCVTPS2QQ __m512i _mm512_cvtps_epi64(__m512 a);
VCVTPS2QQ __m512i _mm512_mask_cvtps_epi64(__m512i s, __mmask16 k, __m512 a);
VCVTPS2QQ __m512i _mm512_maskz_cvtps_epi64(__mmask16 k, __m512 a);
VCVTPS2QQ __m512i _mm512_cvt_roundps_epi64(__m512 a, int r);
VCVTPS2QQ __m512i _mm512_mask_cvt_roundps_epi64(__m512i s, __mmask16 k, __m512 a, int r);
VCVTPS2QQ __m512i _mm512_maskz_cvt_roundps_epi64(__mmask16 k, __m512 a, int r);
VCVTPS2QQ __m256i _mm256_cvtps_epi64(__m256 a);
VCVTPS2QQ __m256i _mm256_mask_cvtps_epi64(__m256i s, __mmask8 k, __m256 a);
VCVTPS2QQ __m256i _mm256_maskz_cvtps_epi64(__mmask8 k, __m256 a);
VCVTPS2QQ __m128i _mm_cvtps_epi64(__m128 a);
VCVTPS2QQ __m128i _mm_mask_cvtps_epi64(__m128i s, __mmask8 k, __m128 a);
VCVTPS2QQ __m128i _mm_maskz_cvtps_epi64(__mmask8 k, __m128 a);

SIMD Floating-Point Exceptions

Invalid, Precision

Other Exceptions

EVEX-encoded instructions, see Exceptions Type E3

#UD	If EVEX.vvvv != 1111B.

VCVTPS2UQQ—Convert Packed Single Precision Floating-Point Values to Packed Unsigned Quadword Integer Values

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.128.66.0F.W0 79 /r VCVTPS2UQQ xmm1 {k1}{z}, xmm2/m64/m32bcst	A	V/V	AVX512VL AVX512DQ	Convert two packed single precision floating-point values from zmm2/m64/m32bcst to two packed unsigned quadword values in zmm1 subject to writemask k1.
EVEX.256.66.0F.W0 79 /r VCVTPS2UQQ ymm1 {k1}{z}, xmm2/m128/m32bcst	A	V/V	AVX512VL AVX512DQ	Convert four packed single precision floating-point values from xmm2/m128/m32bcst to four packed unsigned quadword values in ymm1 subject to writemask k1.
EVEX.512.66.0F.W0 79 /r VCVTPS2UQQ zmm1 {k1}{z}, ymm2/m256/m32bcst{er}	A	V/V	AVX512DQ	Convert eight packed single precision floating-point values from ymm2/m256/m32bcst to eight packed unsigned quadword values in zmm1 subject to writemask k1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Half Vector	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

Description

Converts up to eight packed single-precision floating-point values in the source operand to unsigned quadword integers in the destination operand.

When a conversion is inexact, the value returned is rounded according to the rounding control bits in the MXCSR register or the embedded rounding control bits. If a converted result cannot be represented in the destination format, the floating-point invalid exception is raised, and if this exception is masked, the integer value $2^w - 1$ is returned, where w represents the number of bits in the destination format.

The source operand is a YMM/XMM/XMM (low 64- bits) register or a 256/128/64-bit memory location. The destination operation is a ZMM/YMM/XMM register conditionally updated with writemask k1.

EVEX.vvvv is reserved and must be 1111b otherwise instructions will #UD.

Operation

VCVTPS2UQQ (EVEX encoded versions) when src operand is a register

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
IF (VL == 512) AND (EVEX.b == 1)
    THEN
        SET_RM(EVEX.RC);
    ELSE
        SET_RM(MXCSR.RM);
FI;
FOR j ← 0 TO KL-1
    i ← j * 64
    k ← j * 32
    IF k1[j] OR *no writemask*
        THEN DEST[i+63:i] ←
            Convert_Single_Precision_To_UQuadInteger(SRC[k+31:k])
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE                            ; zeroing-masking
                    DEST[i+63:i] ← 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0
```

VCVTPS2UQQ (EVEX encoded versions) when src operand is a memory source

```
(KL, VL) = (2, 128), (4, 256), (8, 512)

FOR j ← 0 TO KL-1
    i ← j * 64
    k ← j * 32
    IF k1[j] OR *no writemask*
        THEN
            IF (EVEX.b == 1)
                THEN
                    DEST[i+63:i] ←
            Convert_Single_Precision_To_UQuadInteger(SRC[31:0])
                ELSE
                    DEST[i+63:i] ←
            Convert_Single_Precision_To_UQuadInteger(SRC[k+31:k])
            FI;
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE                            ; zeroing-masking
                    DEST[i+63:i] ← 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VCVTPS2UQQ __m512i _mm512_cvtps_epu64(__m512 a);
VCVTPS2UQQ __m512i _mm512_mask_cvtps_epu64(__m512i s, __mmask16 k, __m512 a);
VCVTPS2UQQ __m512i _mm512_maskz_cvtps_epu64(__mmask16 k, __m512 a);
VCVTPS2UQQ __m512i _mm512_cvt_roundps_epu64(__m512 a, int r);
VCVTPS2UQQ __m512i _mm512_mask_cvt_roundps_epu64(__m512i s, __mmask16 k, __m512 a, int r);
VCVTPS2UQQ __m512i _mm512_maskz_cvt_roundps_epu64(__mmask16 k, __m512 a, int r);
VCVTPS2UQQ __m256i _mm256_cvtps_epu64(__m256 a);
VCVTPS2UQQ __m256i _mm256_mask_cvtps_epu64(__m256i s, __mmask8 k, __m256 a);
VCVTPS2UQQ __m256i _mm256_maskz_cvtps_epu64(__mmask8 k, __m256 a);
VCVTPS2UQQ __m128i _mm_cvtps_epu64(__m128 a);
VCVTPS2UQQ __m128i _mm_mask_cvtps_epu64(__m128i s, __mmask8 k, __m128 a);
VCVTPS2UQQ __m128i _mm_maskz_cvtps_epu64(__mmask8 k, __m128 a);

SIMD Floating-Point Exceptions

Invalid, Precision

Other Exceptions

EVEX-encoded instructions, see Exceptions Type E3

#UD	If EVEX.vvvv != 1111B.

VCVTQQ2PD—Convert Packed Quadword Integers to Packed Double-Precision Floating-Point Values

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.128.F3.0F.W1 E6 /r VCVTQQ2PD xmm1 {k1}{z}, xmm2/m128/m64bcst	A	V/V	AVX512VL AVX512DQ	Convert two packed quadword integers from xmm2/m128/m64bcst to packed double-precision floating-point values in xmm1 with writemask k1.
EVEX.256.F3.0F.W1 E6 /r VCVTQQ2PD ymm1 {k1}{z}, ymm2/m256/m64bcst	A	V/V	AVX512VL AVX512DQ	Convert four packed quadword integers from ymm2/m256/m64bcst to packed double-precision floating-point values in ymm1 with writemask k1.
EVEX.512.F3.0F.W1 E6 /r VCVTQQ2PD zmm1 {k1}{z}, zmm2/m512/m64bcst{er}	A	V/V	AVX512DQ	Convert eight packed quadword integers from zmm2/m512/m64bcst to eight packed double-precision floating-point values in zmm1 with writemask k1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Full Vector	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

Description

Converts packed quadword integers in the source operand (second operand) to packed double-precision floating-point values in the destination operand (first operand).

The source operand is a ZMM/YMM/XMM register or a 512/256/128-bit memory location. The destination operation is a ZMM/YMM/XMM register conditionally updated with writemask k1.

EVEX.vvvv is reserved and must be 1111b otherwise instructions will #UD.

Operation

VCVTQQ2PD (EVEX2 encoded versions) when src operand is a register

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
IF (VL == 512) AND (EVEX.b == 1)
    THEN
        SET_RM(EVEX.RC);
    ELSE
        SET_RM(MXCSR.RM);
FI;

FOR j ← 0 TO KL-1
    i ← j * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+63:i] ←
            Convert_QuadInteger_To_Double_Precision_Floating_Point(SRC[i+63:i])
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE                            ; zeroing-masking
                    DEST[i+63:i] ← 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0
```

VCVTQQ2PD (EVEX encoded versions) when src operand is a memory source

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j ← 0 TO KL-1
    i ← j * 64
    IF k1[j] OR *no writemask*
        THEN
            IF (EVEX.b == 1)
                THEN
                    DEST[i+63:i] ←
            Convert_QuadInteger_To_Double_Precision_Floating_Point(SRC[63:0])
                ELSE
                    DEST[i+63:i] ←
            Convert_QuadInteger_To_Double_Precision_Floating_Point(SRC[i+63:i])
            FI;
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE                                ; zeroing-masking
                    DEST[i+63:i] ← 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VCVTQQ2PD __m512d _mm512_cvtepi64_pd(__m512i a);
VCVTQQ2PD __m512d _mm512_mask_cvtepi64_pd(__m512d s, __mmask16 k, __m512i a);
VCVTQQ2PD __m512d _mm512_maskz_cvtepi64_pd(__mmask16 k, __m512i a);
VCVTQQ2PD __m512d _mm512_cvt_roundepi64_pd(__m512i a, int r);
VCVTQQ2PD __m512d _mm512_mask_cvt_roundepi_ps(__m512d s, __mmask8 k, __m512i a, int r);
VCVTQQ2PD __m512d _mm512_maskz_cvt_roundepi64_pd(__mmask8 k, __m512i a, int r);
VCVTQQ2PD __m256d _mm256_mask_cvtepi64_pd(__m256d s, __mmask8 k, __m256i a);
VCVTQQ2PD __m256d _mm256_maskz_cvtepi64_pd(__mmask8 k, __m256i a);
VCVTQQ2PD __m128d _mm_mask_cvtepi64_pd(__m128d s, __mmask8 k, __m128i a);
VCVTQQ2PD __m128d _mm_maskz_cvtepi64_pd(__mmask8 k, __m128i a);

SIMD Floating-Point Exceptions

Precision

Other Exceptions

EVEX-encoded instructions, see Exceptions Type E2

#UD	If EVEX.vvvv != 1111B.

VCVTQQ2PS—Convert Packed Quadword Integers to Packed Single-Precision Floating-Point Values

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.128.0F.W1 5B /r VCVTQQ2PS xmm1 {k1}{z}, xmm2/m128/m64bcst	A	V/V	AVX512VL AVX512DQ	Convert two packed quadword integers from xmm2/mem to packed single-precision floating-point values in xmm1 with writemask k1.
EVEX.256.0F.W1 5B /r VCVTQQ2PS xmm1 {k1}{z}, ymm2/m256/m64bcst	A	V/V	AVX512VL AVX512DQ	Convert four packed quadword integers from ymm2/mem to packed single-precision floating-point values in xmm1 with writemask k1.
EVEX.512.0F.W1 5B /r VCVTQQ2PS ymm1 {k1}{z}, zmm2/m512/m64bcst{er}	A	V/V	AVX512DQ	Convert eight packed quadword integers from zmm2/mem to eight packed single-precision floating-point values in ymm1 with writemask k1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Full Vector	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

Description

Converts packed quadword integers in the source operand (second operand) to packed single-precision floating-point values in the destination operand (first operand).

The source operand is a ZMM/YMM/XMM register or a 512/256/128-bit memory location. The destination operation is a YMM/XMM/XMM (lower 64 bits) register conditionally updated with writemask k1.

EVEX.vvvv is reserved and must be 1111b otherwise instructions will #UD.

Operation

VCVTQQ2PS (EVEX encoded versions) when src operand is a register
(KL, VL) = (2, 128), (4, 256), (8, 512)

```
FOR j ← 0 TO KL-1
    i ← j * 64
    k ← j * 32
    IF k1[j] OR *no writemask*
        THEN DEST[k+31:k] ←
            Convert_QuadInteger_To_Single_Precision_Floating_Point(SRC[i+63:i])
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[k+31:k] remains unchanged*
                ELSE                            ; zeroing-masking
                    DEST[k+31:k] ← 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL/2] ← 0
```

VCVTQQ2PS (EVEX encoded versions) when src operand is a memory source
(KL, VL) = (2, 128), (4, 256), (8, 512)

```
FOR j ← 0 TO KL-1
    i ← j * 64
    k ← j * 32
    IF k1[j] OR *no writemask*
        THEN
            IF (EVEX.b == 1)
                THEN
                    DEST[k+31:k] ←
            Convert_QuadInteger_To_Single_Precision_Floating_Point(SRC[63:0])
                ELSE
                    DEST[k+31:k] ←
            Convert_QuadInteger_To_Single_Precision_Floating_Point(SRC[i+63:i])
            FI;
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[k+31:k] remains unchanged*
                ELSE                                ; zeroing-masking
                    DEST[k+31:k] ← 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL/2] ← 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VCVTQQ2PS __m256 _mm512_cvtepi64_ps(__m512i a);
VCVTQQ2PS __m256 _mm512_mask_cvtepi64_ps(__m256 s, __mmask16 k, __m512i a);
VCVTQQ2PS __m256 _mm512_maskz_cvtepi64_ps(__mmask16 k, __m512i a);
VCVTQQ2PS __m256 _mm512_cvt_roundepi64_ps(__m512i a, int r);
VCVTQQ2PS __m256 _mm512_mask_cvt_roundepi_ps(__m256 s, __mmask8 k, __m512i a, int r);
VCVTQQ2PS __m256 _mm512_maskz_cvt_roundepi64_ps(__mmask8 k, __m512i a, int r);
VCVTQQ2PS __m128 _mm256_cvtepi64_ps(__m256i a);
VCVTQQ2PS __m128 _mm256_mask_cvtepi64_ps(__m128 s, __mmask8 k, __m256i a);
VCVTQQ2PS __m128 _mm256_maskz_cvtepi64_ps(__mmask8 k, __m256i a);
VCVTQQ2PS __m128 _mm_cvtepi64_ps(__m128i a);
VCVTQQ2PS __m128 _mm_mask_cvtepi64_ps(__m128 s, __mmask8 k, __m128i a);
VCVTQQ2PS __m128 _mm_maskz_cvtepi64_ps(__mmask8 k, __m128i a);

SIMD Floating-Point Exceptions

Precision

Other Exceptions

EVEX-encoded instructions, see Exceptions Type E2

#UD	If EVEX.vvvv != 1111B.

VCVTSD2USI—Convert Scalar Double-Precision Floating-Point Value to Unsigned Doubleword Integer

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.LIG.F2.0F.W0 79 /r VCVTSD2USI r32, xmm1/m64{er}	A	V/V	AVX512F	Convert one double-precision floating-point value from xmm1/m64 to one unsigned doubleword integer r32.
EVEX.LIG.F2.0F.W1 79 /r VCVTSD2USI r64, xmm1/m64{er}	A	V/N.E.[1]	AVX512F	Convert one double-precision floating-point value from xmm1/m64 to one unsigned quadword integer zero-extended into r64.

NOTES:

1. EVEX.W1 in non-64 bit is ignored; the instructions behaves as if the W0 version is used.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Tuple1 Fixed	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

Description

Converts a double-precision floating-point value in the source operand (the second operand) to an unsigned doubleword integer in the destination operand (the first operand). The source operand can be an XMM register or a 64-bit memory location. The destination operand is a general-purpose register. When the source operand is an XMM register, the double-precision floating-point value is contained in the low quadword of the register.

When a conversion is inexact, the value returned is rounded according to the rounding control bits in the MXCSR register or the embedded rounding control bits. If a converted result cannot be represented in the destination format, the floating-point invalid exception is raised, and if this exception is masked, the integer value $2^w - 1$ is returned, where w represents the number of bits in the destination format.

Operation

VCVTSD2USI (EVEX encoded version)

```
IF (SRC *is register*) AND (EVEX.b = 1)
    THEN
        SET_RM(EVEX.RC);
    ELSE
        SET_RM(MXCSR.RM);
FI;
IF 64-Bit Mode and OperandSize = 64
    THEN    DEST[63:0] ← Convert_Double_Precision_Floating_Point_To_UInteger(SRC[63:0]);
    ELSE    DEST[31:0] ← Convert_Double_Precision_Floating_Point_To_UInteger(SRC[63:0]);
FI
```

Intel C/C++ Compiler Intrinsic Equivalent

VCVTSD2USI unsigned int _mm_cvtsd_u32(__m128d);
VCVTSD2USI unsigned int _mm_cvt_roundsd_u32(__m128d, int r);
VCVTSD2USI unsigned __int64 _mm_cvtsd_u64(__m128d);
VCVTSD2USI unsigned __int64 _mm_cvt_roundsd_u64(__m128d, int r);

SIMD Floating-Point Exceptions

Invalid, Precision

Other Exceptions

EVEX-encoded instructions, see Exceptions Type E3NF.

VCVTSS2USI—Convert Scalar Single-Precision Floating-Point Value to Unsigned Doubleword Integer

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.LIG.F3.0F.W0 79 /r VCVTSS2USI r32, xmm1/m32{er}	A	V/V	AVX512F	Convert one single-precision floating-point value from xmm1/m32 to one unsigned doubleword integer in r32.
EVEX.LIG.F3.0F.W1 79 /r VCVTSS2USI r64, xmm1/m32{er}	A	V/N.E.[1]	AVX512F	Convert one single-precision floating-point value from xmm1/m32 to one unsigned quadword integer in r64.

NOTES:

1. EVEX.W1 in non-64 bit is ignored; the instructions behaves as if the W0 version is used.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Tuple1 Fixed	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

Description

Converts a single-precision floating-point value in the source operand (the second operand) to an unsigned double-word integer (or unsigned quadword integer if operand size is 64 bits) in the destination operand (the first operand). The source operand can be an XMM register or a memory location. The destination operand is a general-purpose register. When the source operand is an XMM register, the single-precision floating-point value is contained in the low doubleword of the register.

When a conversion is inexact, the value returned is rounded according to the rounding control bits in the MXCSR register or the embedded rounding control bits. If a converted result cannot be represented in the destination format, the floating-point invalid exception is raised, and if this exception is masked, the integer value $2^w - 1$ is returned, where w represents the number of bits in the destination format.

VEX.W1 and EVEX.W1 versions: promotes the instruction to produce 64-bit data in 64-bit mode.

Note: EVEX.vvvv is reserved and must be 1111b, otherwise instructions will #UD.

Operation

VCVTSS2USI (EVEX encoded version)

```
IF (SRC *is register*) AND (EVEX.b = 1)
    THEN
        SET_RM(EVEX.RC);
    ELSE
        SET_RM(MXCSR.RM);
FI;
IF 64-bit Mode and OperandSize = 64
THEN
    DEST[63:0] ← Convert_Single_Precision_Floating_Point_To_UInteger(SRC[31:0]);
ELSE
    DEST[31:0] ← Convert_Single_Precision_Floating_Point_To_UInteger(SRC[31:0]);
FI;
```

Intel C/C++ Compiler Intrinsic Equivalent

VCVTSS2USI unsigned _mm_cvtss_u32(__m128 a);
VCVTSS2USI unsigned _mm_cvt_roundss_u32(__m128 a, int r);
VCVTSS2USI unsigned __int64 _mm_cvtss_u64(__m128 a);
VCVTSS2USI unsigned __int64 _mm_cvt_roundss_u64(__m128 a, int r);

SIMD Floating-Point Exceptions

Invalid, Precision

Other Exceptions

EVEX-encoded instructions, see Exceptions Type E3NF.

VCVTTPD2QQ—Convert with Truncation Packed Double-Precision Floating-Point Values to Packed Quadword Integers

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.128.66.0F.W1 7A /r VCVTTPD2QQ xmm1 {k1}{z}, xmm2/m128/m64bcst	A	V/V	AVX512VL AVX512DQ	Convert two packed double-precision floating-point values from zmm2/m128/m64bcst to two packed quadword integers in zmm1 using truncation with writemask k1.
EVEX.256.66.0F.W1 7A /r VCVTTPD2QQ ymm1 {k1}{z}, ymm2/m256/m64bcst	A	V/V	AVX512VL AVX512DQ	Convert four packed double-precision floating-point values from ymm2/m256/m64bcst to four packed quadword integers in ymm1 using truncation with writemask k1.
EVEX.512.66.0F.W1 7A /r VCVTTPD2QQ zmm1 {k1}{z}, zmm2/m512/m64bcst{sae}	A	V/V	AVX512DQ	Convert eight packed double-precision floating-point values from zmm2/m512 to eight packed quadword integers in zmm1 using truncation with writemask k1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Full Vector	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

Description

Converts with truncation packed double-precision floating-point values in the source operand (second operand) to packed quadword integers in the destination operand (first operand).

EVEX encoded versions: The source operand is a ZMM/YMM/XMM register or a 512/256/128-bit memory location. The destination operand is a ZMM/YMM/XMM register conditionally updated with writemask k1.

When a conversion is inexact, the value returned is rounded according to the rounding control bits in the MXCSR register. If a converted result cannot be represented in the destination format, the floating-point invalid exception is raised, and if this exception is masked, the indefinite integer value (2^{w-1}, where w represents the number of bits in the destination format) is returned.

Note: EVEX.vvvv is reserved and must be 1111b, otherwise instructions will #UD.

Operation

VCVTTPD2QQ (EVEX encoded version) when src operand is a register

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j ← 0 TO KL-1
    i ← j * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+63:i] ←
            Convert_Double_Precision_Floating_Point_To_QuadInteger_Truncate(SRC[i+63:i])
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE                            ; zeroing-masking
                    DEST[i+63:i] ← 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0
```

VCVTTPD2QQ (EVEX encoded version) when src operand is a memory source
(KL, VL) = (2, 128), (4, 256), (8, 512)

```
FOR j ← 0 TO KL-1
    i ← j * 64
    IF k1[j] OR *no writemask*
        THEN
            IF (EVEX.b == 1)
                THEN
                    DEST[i+63:i] ←          Convert_Double_Precision_Floating_Point_To_QuadInteger_Truncate(SRC[63:0])
                ELSE
                    DEST[i+63:i] ← Convert_Double_Precision_Floating_Point_To_QuadInteger_Truncate(SRC[i+63:i])
            FI;
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE                                ; zeroing-masking
                    DEST[i+63:i] ← 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VCVTTPD2QQ __m512i _mm512_cvttpd_epi64(__m512d a);
VCVTTPD2QQ __m512i _mm512_mask_cvttpd_epi64(__m512i s, __mmask8 k, __m512d a);
VCVTTPD2QQ __m512i _mm512_maskz_cvttpd_epi64(__mmask8 k, __m512d a);
VCVTTPD2QQ __m512i _mm512_cvtt_roundpd_epi64(__m512d a, int sae);
VCVTTPD2QQ __m512i _mm512_mask_cvtt_roundpd_epi64(__m512i s, __mmask8 k, __m512d a, int sae);
VCVTTPD2QQ __m512i _mm512_maskz_cvtt_roundpd_epi64(__mmask8 k, __m512d a, int sae);
VCVTTPD2QQ __m256i _mm256_mask_cvttpd_epi64(__m256i s, __mmask8 k, __m256d a);
VCVTTPD2QQ __m256i _mm256_maskz_cvttpd_epi64(__mmask8 k, __m256d a);
VCVTTPD2QQ __m128i _mm_mask_cvttpd_epi64(__m128i s, __mmask8 k, __m128d a);
VCVTTPD2QQ __m128i _mm_maskz_cvttpd_epi64(__mmask8 k, __m128d a);

SIMD Floating-Point Exceptions

Invalid, Precision

Other Exceptions

EVEX-encoded instructions, see Exceptions Type E2.

#UD	If EVEX.vvvv != 1111B.

VCVTTPD2UDQ—Convert with Truncation Packed Double-Precision Floating-Point Values to Packed Unsigned Doubleword Integers

Opcode Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.128.0F.W1 78 /r VCVTTPD2UDQ xmm1 {k1}{z}, xmm2/m128/m64bcst	A	V/V	AVX512VL AVX512F	Convert two packed double-precision floating-point values in xmm2/m128/m64bcst to two unsigned doubleword integers in xmm1 using truncation subject to writemask k1.
EVEX.256.0F.W1 78 02 /r VCVTTPD2UDQ xmm1 {k1}{z}, ymm2/m256/m64bcst	A	V/V	AVX512VL AVX512F	Convert four packed double-precision floating-point values in ymm2/m256/m64bcst to four unsigned doubleword integers in xmm1 using truncation subject to writemask k1.
EVEX.512.0F.W1 78 /r VCVTTPD2UDQ ymm1 {k1}{z}, zmm2/m512/m64bcst{sae}	A	V/V	AVX512F	Convert eight packed double-precision floating-point values in zmm2/m512/m64bcst to eight unsigned doubleword integers in ymm1 using truncation subject to writemask k1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Full Vector	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

Description

Converts with truncation packed double-precision floating-point values in the source operand (the second operand) to packed unsigned doubleword integers in the destination operand (the first operand).

When a conversion is inexact, the value returned is rounded according to the rounding control bits in the MXCSR register. If a converted result cannot be represented in the destination format, the floating-point invalid exception is raised, and if this exception is masked, the integer value $2^w - 1$ is returned, where w represents the number of bits in the destination format.

The source operand is a ZMM/YMM/XMM register, a 512/256/128-bit memory location, or a 512/256/128-bit vector broadcasted from a 64-bit memory location. The destination operand is a YMM/XMM/XMM (low 64 bits) register conditionally updated with writemask k1. The upper bits (MAXVL-1:256) of the corresponding destination are zeroed.

Note: EVEX.vvvv is reserved and must be 1111b, otherwise instructions will #UD.

Operation

VCVTTPD2UDQ (EVEX encoded versions) when src2 operand is a register

```
(KL, VL) = (2, 128), (4, 256),(8, 512)
FOR j ← 0 TO KL-1
    i ← j * 32
    k ← j * 64
    IF k1[j] OR *no writemask*
        THEN
            DEST[i+31:i] ←
            Convert_Double_Precision_Floating_Point_To_UInteger_Truncate(SRC[k+63:k])
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE                            ; zeroing-masking
                    DEST[i+31:i] ← 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL/2] ← 0
```

VCVTTPD2UDQ (EVEX encoded versions) when src operand is a memory source

```
(KL, VL) = (2, 128), (4, 256),(8, 512)

FOR j ← 0 TO KL-1
    i ← j * 32
    k ← j * 64
    IF k1[j] OR *no writemask*
        THEN
            IF (EVEX.b = 1)
                THEN
                    DEST[i+31:i] ←
            Convert_Double_Precision_Floating_Point_To_UInteger_Truncate(SRC[63:0])
                ELSE
                    DEST[i+31:i] ←
            Convert_Double_Precision_Floating_Point_To_UInteger_Truncate(SRC[k+63:k])
            FI;
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE                            ; zeroing-masking
                    DEST[i+31:i] ← 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL/2] ← 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VCVTTPD2UDQ __m256i _mm512_cvttpd_epu32(__m512d a);
VCVTTPD2UDQ __m256i _mm512_mask_cvttpd_epu32(__m256i s, __mmask8 k, __m512d a);
VCVTTPD2UDQ __m256i _mm512_maskz_cvttpd_epu32(__mmask8 k, __m512d a);
VCVTTPD2UDQ __m256i _mm512_cvtt_roundpd_epu32(__m512d a, int sae);
VCVTTPD2UDQ __m256i _mm512_mask_cvtt_roundpd_epu32(__m256i s, __mmask8 k, __m512d a, int sae);
VCVTTPD2UDQ __m256i _mm512_maskz_cvtt_roundpd_epu32(__mmask8 k, __m512d a, int sae);
VCVTTPD2UDQ __m128i _mm256_mask_cvttpd_epu32(__m128i s, __mmask8 k, __m256d a);
VCVTTPD2UDQ __m128i _mm256_maskz_cvttpd_epu32(__mmask8 k, __m256d a);
VCVTTPD2UDQ __m128i _mm_mask_cvttpd_epu32(__m128i s, __mmask8 k, __m128d a);
VCVTTPD2UDQ __m128i _mm_maskz_cvttpd_epu32(__mmask8 k, __m128d a);

SIMD Floating-Point Exceptions

Invalid, Precision

Other Exceptions

EVEX-encoded instructions, see Exceptions Type E2.

#UD	If EVEX.vvvv != 1111B.

VCVTTPD2UQQ—Convert with Truncation Packed Double-Precision Floating-Point Values to Packed Unsigned Quadword Integers

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.128.66.0F.W1 78 /r VCVTTPD2UQQ xmm1 {k1}{z}, xmm2/m128/m64bcst	A	V/V	AVX512VL AVX512DQ	Convert two packed double-precision floating-point values from xmm2/m128/m64bcst to two packed unsigned quadword integers in xmm1 using truncation with writemask k1.
EVEX.256.66.0F.W1 78 /r VCVTTPD2UQQ ymm1 {k1}{z}, ymm2/m256/m64bcst	A	V/V	AVX512VL AVX512DQ	Convert four packed double-precision floating-point values from ymm2/m256/m64bcst to four packed unsigned quadword integers in ymm1 using truncation with writemask k1.
EVEX.512.66.0F.W1 78 /r VCVTTPD2UQQ zmm1 {k1}{z}, zmm2/m512/m64bcst{sae}	A	V/V	AVX512DQ	Convert eight packed double-precision floating-point values from zmm2/mem to eight packed unsigned quadword integers in zmm1 using truncation with writemask k1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Full Vector	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

Description

Converts with truncation packed double-precision floating-point values in the source operand (second operand) to packed unsigned quadword integers in the destination operand (first operand).

When a conversion is inexact, the value returned is rounded according to the rounding control bits in the MXCSR register. If a converted result cannot be represented in the destination format, the floating-point invalid exception is raised, and if this exception is masked, the integer value $2^w - 1$ is returned, where w represents the number of bits in the destination format.

EVEX encoded versions: The source operand is a ZMM/YMM/XMM register or a 512/256/128-bit memory location. The destination operation is a ZMM/YMM/XMM register conditionally updated with writemask k1.

Note: EVEX.vvvv is reserved and must be 1111b, otherwise instructions will #UD.

Operation

VCVTTPD2UQQ (EVEX encoded versions) when src operand is a register

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j ← 0 TO KL-1
    i ← j * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+63:i] ←
            Convert_Double_Precision_Floating_Point_To_UQuadInteger_Truncate(SRC[i+63:i])
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE                            ; zeroing-masking
                    DEST[i+63:i] ← 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0
```

VCVTTPD2UQQ (EVEX encoded versions) when src operand is a memory source
(KL, VL) = (2, 128), (4, 256), (8, 512)

```
FOR j ← 0 TO KL-1
    i ← j * 64
    IF k1[j] OR *no writemask*
        THEN
            IF (EVEX.b == 1)
                THEN
                    DEST[i+63:i] ←
            Convert_Double_Precision_Floating_Point_To_UQuadInteger_Truncate(SRC[63:0])
                ELSE
                    DEST[i+63:i] ←
            Convert_Double_Precision_Floating_Point_To_UQuadInteger_Truncate(SRC[i+63:i])
            FI;
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE                            ; zeroing-masking
                    DEST[i+63:i] ← 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VCVTTPD2UQQ _mm<size>[_mask[z]]_cvtt[_round]pd_epu64

VCVTTPD2UQQ __m512i _mm512_cvttpd_epu64(__m512d a);
VCVTTPD2UQQ __m512i _mm512_mask_cvttpd_epu64(__m512i s, __mmask8 k, __m512d a);
VCVTTPD2UQQ __m512i _mm512_maskz_cvttpd_epu64(__mmask8 k, __m512d a);
VCVTTPD2UQQ __m512i _mm512_cvtt_roundpd_epu64(__m512d a, int sae);
VCVTTPD2UQQ __m512i _mm512_mask_cvtt_roundpd_epu64(__m512i s, __mmask8 k, __m512d a, int sae);
VCVTTPD2UQQ __m512i _mm512_maskz_cvtt_roundpd_epu64(__mmask8 k, __m512d a, int sae);
VCVTTPD2UQQ __m256i _mm256_mask_cvttpd_epu64(__m256i s, __mmask8 k, __m256d a);
VCVTTPD2UQQ __m256i _mm256_maskz_cvttpd_epu64(__mmask8 k, __m256d a);
VCVTTPD2UQQ __m128i _mm_mask_cvttpd_epu64(__m128i s, __mmask8 k, __m128d a);
VCVTTPD2UQQ __m128i _mm_maskz_cvttpd_epu64(__mmask8 k, __m128d a);

SIMD Floating-Point Exceptions

Invalid, Precision

Other Exceptions

EVEX-encoded instructions, see Exceptions Type E2.

#UD	If EVEX.vvvv != 1111B.

VCVTTPS2UDQ—Convert with Truncation Packed Single-Precision Floating-Point Values to Packed Unsigned Doubleword Integer Values

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.128.0F.W0 78 /r VCVTTPS2UDQ xmm1 {k1}{z}, xmm2/m128/m32bcst	A	V/V	AVX512VL AVX512F	Convert four packed single precision floating-point values from xmm2/m128/m32bcst to four packed unsigned doubleword values in xmm1 using truncation subject to writemask k1.
EVEX.256.0F.W0 78 /r VCVTTPS2UDQ ymm1 {k1}{z}, ymm2/m256/m32bcst	A	V/V	AVX512VL AVX512F	Convert eight packed single precision floating-point values from ymm2/m256/m32bcst to eight packed unsigned doubleword values in ymm1 using truncation subject to writemask k1.
EVEX.512.0F.W0 78 /r VCVTTPS2UDQ zmm1 {k1}{z}, zmm2/m512/m32bcst{sae}	A	V/V	AVX512F	Convert sixteen packed single-precision floating-point values from zmm2/m512/m32bcst to sixteen packed unsigned doubleword values in zmm1 using truncation subject to writemask k1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Full Vector	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

Description

Converts with truncation packed single-precision floating-point values in the source operand to sixteen unsigned doubleword integers in the destination operand.

When a conversion is inexact, the value returned is rounded according to the rounding control bits in the MXCSR. If a converted result cannot be represented in the destination format, the floating-point invalid exception is raised, and if this exception is masked, the integer value $2^w - 1$ is returned, where w represents the number of bits in the destination format.

EVEX encoded versions: The source operand is a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 32-bit memory location. The destination operand is a ZMM/YMM/XMM register conditionally updated with writemask k1.

Note: EVEX.vvvv is reserved and must be 1111b otherwise instructions will #UD.

Operation

VCVTTPS2UDQ (EVEX encoded versions) when src operand is a register

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j ← 0 TO KL-1
    i ← j * 32
    IF k1[j] OR *no writemask*
        THEN DEST[i+31:i] ←
            Convert_Single_Precision_Floating_Point_To_UInteger_Truncate(SRC[i+31:i])
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE                                ; zeroing-masking
                    DEST[i+31:i] ← 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0
```

VCVTTPS2UDQ (EVEX encoded versions) when src operand is a memory source
(KL, VL) = (4, 128), (8, 256), (16, 512)

```
FOR j ← 0 TO KL-1
    i ← j * 32
    IF k1[j] OR *no writemask*
        THEN
            IF (EVEX.b = 1)
                THEN
                    DEST[i+31:i] ←
            Convert_Single_Precision_Floating_Point_To_UInteger_Truncate(SRC[31:0])
                ELSE
                    DEST[i+31:i] ←
            Convert_Single_Precision_Floating_Point_To_UInteger_Truncate(SRC[i+31:i])
            FI;
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE                                ; zeroing-masking
                    DEST[i+31:i] ← 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VCVTTPS2UDQ __m512i _mm512_cvttps_epu32(__m512 a);
VCVTTPS2UDQ __m512i _mm512_mask_cvttps_epu32(__m512i s, __mmask16 k, __m512 a);
VCVTTPS2UDQ __m512i _mm512_maskz_cvttps_epu32(__mmask16 k, __m512 a);
VCVTTPS2UDQ __m512i _mm512_cvtt_roundps_epu32(__m512 a, int sae);
VCVTTPS2UDQ __m512i _mm512_mask_cvtt_roundps_epu32(__m512i s, __mmask16 k, __m512 a, int sae);
VCVTTPS2UDQ __m512i _mm512_maskz_cvtt_roundps_epu32(__mmask16 k, __m512 a, int sae);
VCVTTPS2UDQ __m256i _mm256_mask_cvttps_epu32(__m256i s, __mmask8 k, __m256 a);
VCVTTPS2UDQ __m256i _mm256_maskz_cvttps_epu32(__mmask8 k, __m256 a);
VCVTTPS2UDQ __m128i _mm_mask_cvttps_epu32(__m128i s, __mmask8 k, __m128 a);
VCVTTPS2UDQ __m128i _mm_maskz_cvttps_epu32(__mmask8 k, __m128 a);

SIMD Floating-Point Exceptions

Invalid, Precision

Other Exceptions

EVEX-encoded instructions, see Exceptions Type E2.

#UD	If EVEX.vvvv != 1111B.

VCVTTPS2QQ—Convert with Truncation Packed Single Precision Floating-Point Values to Packed Singed Quadword Integer Values

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.128.66.0F.W0 7A /r VCVTTPS2QQ xmm1 {k1}{z}, xmm2/m64/m32bcst	A	V/V	AVX512VL AVX512DQ	Convert two packed single precision floating-point values from xmm2/m64/m32bcst to two packed signed quadword values in xmm1 using truncation subject to writemask k1.
EVEX.256.66.0F.W0 7A /r VCVTTPS2QQ ymm1 {k1}{z}, xmm2/m128/m32bcst	A	V/V	AVX512VL AVX512DQ	Convert four packed single precision floating-point values from xmm2/m128/m32bcst to four packed signed quadword values in ymm1 using truncation subject to writemask k1.
EVEX.512.66.0F.W0 7A /r VCVTTPS2QQ zmm1 {k1}{z}, ymm2/m256/m32bcst{sae}	A	V/V	AVX512DQ	Convert eight packed single precision floating-point values from ymm2/m256/m32bcst to eight packed signed quadword values in zmm1 using truncation subject to writemask k1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Half Vector	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

Description

Converts with truncation packed single-precision floating-point values in the source operand to eight signed quad-word integers in the destination operand.

When a conversion is inexact, the value returned is rounded according to the rounding control bits in the MXCSR register. If a converted result cannot be represented in the destination format, the floating-point invalid exception is raised, and if this exception is masked, the indefinite integer value (2^{w-1}, where w represents the number of bits in the destination format) is returned.

EVEX encoded versions: The source operand is a YMM/XMM/XMM (low 64 bits) register or a 256/128/64-bit memory location. The destination operation is a vector register conditionally updated with writemask k1.

Note: EVEX.vvvv is reserved and must be 1111b otherwise instructions will #UD.

Operation

VCVTTPS2QQ (EVEX encoded versions) when src operand is a register

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j ← 0 TO KL-1
    i ← j * 64
    k ← j * 32
    IF k1[j] OR *no writemask*
        THEN DEST[i+63:i] ←
            Convert_Single_Precision_To_QuadInteger_Truncate(SRC[k+31:k])
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE                                ; zeroing-masking
                    DEST[i+63:i] ← 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0
```

VCVTTPS2QQ (EVEX encoded versions) when src operand is a memory source
(KL, VL) = (2, 128), (4, 256), (8, 512)

```
FOR j ← 0 TO KL-1
    i ← j * 64
    k ← j * 32
    IF k1[j] OR *no writemask*
        THEN
            IF (EVEX.b == 1)
                THEN
                    DEST[i+63:i] ←
            Convert_Single_Precision_To_QuadInteger_Truncate(SRC[31:0])
                ELSE
                    DEST[i+63:i] ←
            Convert_Single_Precision_To_QuadInteger_Truncate(SRC[k+31:k])
            FI;
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE                                ; zeroing-masking
                    DEST[i+63:i] ← 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VCVTTPS2QQ __m512i _mm512_cvttps_epi64(__m256 a);
VCVTTPS2QQ __m512i _mm512_mask_cvttps_epi64(__m512i s, __mmask16 k, __m256 a);
VCVTTPS2QQ __m512i _mm512_maskz_cvttps_epi64(__mmask16 k, __m256 a);
VCVTTPS2QQ __m512i _mm512_cvtt_roundps_epi64(__m256 a, int sae);
VCVTTPS2QQ __m512i _mm512_mask_cvtt_roundps_epi64(__m512i s, __mmask16 k, __m256 a, int sae);
VCVTTPS2QQ __m512i _mm512_maskz_cvtt_roundps_epi64(__mmask16 k, __m256 a, int sae);
VCVTTPS2QQ __m256i _mm256_mask_cvttps_epi64(__m256i s, __mmask8 k, __m128 a);
VCVTTPS2QQ __m256i _mm256_maskz_cvttps_epi64(__mmask8 k, __m128 a);
VCVTTPS2QQ __m128i _mm_mask_cvttps_epi64(__m128i s, __mmask8 k, __m128 a);
VCVTTPS2QQ __m128i _mm_maskz_cvttps_epi64(__mmask8 k, __m128 a);

SIMD Floating-Point Exceptions

Invalid, Precision

Other Exceptions

EVEX-encoded instructions, see Exceptions Type E3.

#UD	If EVEX.vvvv != 1111B.

VCVTTPS2UQQ—Convert with Truncation Packed Single Precision Floating-Point Values to Packed Unsigned Quadword Integer Values

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.128.66.0F.W0 78 /r VCVTTPS2UQQ xmm1 {k1}{z}, xmm2/m64/m32bcst	A	V/V	AVX512VL AVX512DQ	Convert two packed single precision floating-point values from xmm2/m64/m32bcst to two packed unsigned quadword values in xmm1 using truncation subject to writemask k1.
EVEX.256.66.0F.W0 78 /r VCVTTPS2UQQ ymm1 {k1}{z}, xmm2/m128/m32bcst	A	V/V	AVX512VL AVX512DQ	Convert four packed single precision floating-point values from xmm2/m128/m32bcst to four packed unsigned quadword values in ymm1 using truncation subject to writemask k1.
EVEX.512.66.0F.W0 78 /r VCVTTPS2UQQ zmm1 {k1}{z}, ymm2/m256/m32bcst{sae}	A	V/V	AVX512DQ	Convert eight packed single precision floating-point values from ymm2/m256/m32bcst to eight packed unsigned quadword values in zmm1 using truncation subject to writemask k1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Half Vector	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

Description

Converts with truncation up to eight packed single-precision floating-point values in the source operand to unsigned quadword integers in the destination operand.

When a conversion is inexact, the value returned is rounded according to the rounding control bits in the MXCSR register. If a converted result cannot be represented in the destination format, the floating-point invalid exception is raised, and if this exception is masked, the integer value $2^w - 1$ is returned, where w represents the number of bits in the destination format.

EVEX encoded versions: The source operand is a YMM/XMM/XMM (low 64 bits) register or a 256/128/64-bit memory location. The destination operation is a vector register conditionally updated with writemask k1.

Note: EVEX.vvvv is reserved and must be 1111b otherwise instructions will #UD.

Operation

VCVTTPS2UQQ (EVEX encoded versions) when src operand is a register

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j ← 0 TO KL-1
    i ← j * 64
    k ← j * 32
    IF k1[j] OR *no writemask*
        THEN DEST[i+63:i] ←
            Convert_Single_Precision_To_UQuadInteger_Truncate(SRC[k+31:k])
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE                            ; zeroing-masking
                    DEST[i+63:i] ← 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0
```

VCVTTPS2UQQ (EVEX encoded versions) when src operand is a memory source
(KL, VL) = (2, 128), (4, 256), (8, 512)

```
FOR j ← 0 TO KL-1
    i ← j * 64
    k ← j * 32
    IF k1[j] OR *no writemask*
        THEN
            IF (EVEX.b == 1)
                THEN
                    DEST[i+63:i] ←
            Convert_Single_Precision_To_UQuadInteger_Truncate(SRC[31:0])
                ELSE
                    DEST[i+63:i] ←
            Convert_Single_Precision_To_UQuadInteger_Truncate(SRC[k+31:k])
            FI;
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE                                ; zeroing-masking
                    DEST[i+63:i] ← 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VCVTTPS2UQQ _mm<size>[_mask[z]]_cvtt[_round]ps_epu64
VCVTTPS2UQQ __m512i _mm512_cvttps_epu64(__m256 a);
VCVTTPS2UQQ __m512i _mm512_mask_cvttps_epu64(__m512i s, __mmask16 k, __m256 a);
VCVTTPS2UQQ __m512i _mm512_maskz_cvttps_epu64(__mmask16 k, __m256 a);
VCVTTPS2UQQ __m512i _mm512_cvtt_roundps_epu64(__m256 a, int sae);
VCVTTPS2UQQ __m512i _mm512_mask_cvtt_roundps_epu64(__m512i s, __mmask16 k, __m256 a, int sae);
VCVTTPS2UQQ __m512i _mm512_maskz_cvtt_roundps_epu64(__mmask16 k, __m256 a, int sae);
VCVTTPS2UQQ __m256i _mm256_mask_cvttps_epu64(__m256i s, __mmask8 k, __m128 a);
VCVTTPS2UQQ __m256i _mm256_maskz_cvttps_epu64(__mmask8 k, __m128 a);
VCVTTPS2UQQ __m128i _mm_mask_cvttps_epu64(__m128i s, __mmask8 k, __m128 a);
VCVTTPS2UQQ __m128i _mm_maskz_cvttps_epu64(__mmask8 k, __m128 a);

SIMD Floating-Point Exceptions

Invalid, Precision

Other Exceptions

EVEX-encoded instructions, see Exceptions Type E3.

#UD	If EVEX.vvvv != 1111B.

VCVTTSD2USI—Convert with Truncation Scalar Double-Precision Floating-Point Value to Unsigned Integer

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.LIG.F2.0F.W0 78 /r VCVTTSD2USI r32, xmm1/m64{sae}	A	V/V	AVX512F	Convert one double-precision floating-point value from xmm1/m64 to one unsigned doubleword integer r32 using truncation.
EVEX.LIG.F2.0F.W1 78 /r VCVTTSD2USI r64, xmm1/m64{sae}	A	V/N.E.[1]	AVX512F	Convert one double-precision floating-point value from xmm1/m64 to one unsigned quadword integer zero-extended into r64 using truncation.

NOTES:

1. For this specific instruction, EVEX.W in non-64 bit is ignored; the instructions behaves as if the W0 version is used.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Tuple1 Fixed	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

Description

Converts with truncation a double-precision floating-point value in the source operand (the second operand) to an unsigned doubleword integer (or unsigned quadword integer if operand size is 64 bits) in the destination operand (the first operand). The source operand can be an XMM register or a 64-bit memory location. The destination operand is a general-purpose register. When the source operand is an XMM register, the double-precision floating-point value is contained in the low quadword of the register.

When a conversion is inexact, the value returned is rounded according to the rounding control bits in the MXCSR register. If a converted result cannot be represented in the destination format, the floating-point invalid exception is raised, and if this exception is masked, the integer value $2^w - 1$ is returned, where w represents the number of bits in the destination format.

EVEX.W1 version: promotes the instruction to produce 64-bit data in 64-bit mode.

Operation

VCVTTSD2USI (EVEX encoded version)

```
IF 64-Bit Mode and OperandSize = 64
    THEN    DEST[63:0] ← Convert_Double_Precision_Floating_Point_To_UInteger_Truncate(SRC[63:0]);
    ELSE    DEST[31:0] ← Convert_Double_Precision_Floating_Point_To_UInteger_Truncate(SRC[63:0]);
FI
```

Intel C/C++ Compiler Intrinsic Equivalent

VCVTTSD2USI unsigned int _mm_cvttsd_u32(__m128d);
VCVTTSD2USI unsigned int _mm_cvtt_roundsd_u32(__m128d, int sae);
VCVTTSD2USI unsigned __int64 _mm_cvttsd_u64(__m128d);
VCVTTSD2USI unsigned __int64 _mm_cvtt_roundsd_u64(__m128d, int sae);

SIMD Floating-Point Exceptions

Invalid, Precision

Other Exceptions

EVEX-encoded instructions, see Exceptions Type E3NF.

VCVTTSS2USI—Convert with Truncation Scalar Single-Precision Floating-Point Value to Unsigned Integer

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.LIG.F3.0F.W0 78 /r VCVTTSS2USI r32, xmm1/m32{sae}	A	V/V	AVX512F	Convert one single-precision floating-point value from xmm1/m32 to one unsigned doubleword integer in r32 using truncation.
EVEX.LIG.F3.0F.W1 78 /r VCVTTSS2USI r64, xmm1/m32{sae}	A	V/N.E.[1]	AVX512F	Convert one single-precision floating-point value from xmm1/m32 to one unsigned quadword integer in r64 using truncation.

NOTES:

1. For this specific instruction, EVEX.W in non-64 bit is ignored; the instructions behaves as if the W0 version is used.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Tuple1 Fixed	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

Description

Converts with truncation a single-precision floating-point value in the source operand (the second operand) to an unsigned doubleword integer (or unsigned quadword integer if operand size is 64 bits) in the destination operand (the first operand). The source operand can be an XMM register or a memory location. The destination operand is a general-purpose register. When the source operand is an XMM register, the single-precision floating-point value is contained in the low doubleword of the register.

When a conversion is inexact, the value returned is rounded according to the rounding control bits in the MXCSR register. If a converted result cannot be represented in the destination format, the floating-point invalid exception is raised, and if this exception is masked, the integer value $2^w - 1$ is returned, where w represents the number of bits in the destination format.

EVEX.W1 version: promotes the instruction to produce 64-bit data in 64-bit mode.

Note: EVEX.vvvv is reserved and must be 1111b, otherwise instructions will #UD.

Operation

VCVTTSS2USI (EVEX encoded version)

```
IF 64-bit Mode and OperandSize = 64
THEN
    DEST[63:0] ← Convert_Single_Precision_Floating_Point_To_UInteger_Truncate(SRC[31:0]);
ELSE
    DEST[31:0] ← Convert_Single_Precision_Floating_Point_To_UInteger_Truncate(SRC[31:0]);
FI;
```

Intel C/C++ Compiler Intrinsic Equivalent

VCVTTSS2USI unsigned int _mm_cvttss_u32(__m128 a);
VCVTTSS2USI unsigned int _mm_cvtt_roundss_u32(__m128 a, int sae);
VCVTTSS2USI unsigned __int64 _mm_cvttss_u64(__m128 a);
VCVTTSS2USI unsigned __int64 _mm_cvtt_roundss_u64(__m128 a, int sae);

SIMD Floating-Point Exceptions

Invalid, Precision

Other Exceptions

EVEX-encoded instructions, see Exceptions Type E3NF.

VCVTUDQ2PD—Convert Packed Unsigned Doubleword Integers to Packed Double-Precision Floating-Point Values

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.128.F3.0F.W0 7A /r VCVTUDQ2PD xmm1 {k1}{z}, xmm2/m64/m32bcst	A	V/V	AVX512VL AVX512F	Convert two packed unsigned doubleword integers from ymm2/m64/m32bcst to packed double-precision floating-point values in zmm1 with writemask k1.
EVEX.256.F3.0F.W0 7A /r VCVTUDQ2PD ymm1 {k1}{z}, xmm2/m128/m32bcst	A	V/V	AVX512VL AVX512F	Convert four packed unsigned doubleword integers from xmm2/m128/m32bcst to packed double-precision floating-point values in zmm1 with writemask k1.
EVEX.512.F3.0F.W0 7A /r VCVTUDQ2PD zmm1 {k1}{z}, ymm2/m256/m32bcst	A	V/V	AVX512F	Convert eight packed unsigned doubleword integers from ymm2/m256/m32bcst to eight packed double-precision floating-point values in zmm1 with writemask k1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Half Vector	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

Description

Converts packed unsigned doubleword integers in the source operand (second operand) to packed double-precision floating-point values in the destination operand (first operand).

The source operand is a YMM/XMM/XMM (low 64 bits) register, a 256/128/64-bit memory location or a 256/128/64-bit vector broadcasted from a 32-bit memory location. The destination operand is a ZMM/YMM/XMM register conditionally updated with writemask k1.

Attempt to encode this instruction with EVEX embedded rounding is ignored.

Note: EVEX.vvvv is reserved and must be 1111b, otherwise instructions will #UD.

Operation

VCVTUDQ2PD (EVEX encoded versions) when src operand is a register

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j ← 0 TO KL-1
    i ← j * 64
    k ← j * 32
    IF k1[j] OR *no writemask*
        THEN DEST[i+63:i] ←
            Convert_UInteger_To_Double_Precision_Floating_Point(SRC[k+31:k])
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE                                ; zeroing-masking
                    DEST[i+63:i] ← 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0
```

VCVTUDQ2PD (EVEX encoded versions) when src operand is a memory source
(KL, VL) = (2, 128), (4, 256), (8, 512)

```
FOR j ← 0 TO KL-1
    i ← j * 64
    k ← j * 32
    IF k1[j] OR *no writemask*
        THEN
            IF (EVEX.b = 1)
                THEN
                    DEST[i+63:i] ←
            Convert_UInteger_To_Double_Precision_Floating_Point(SRC[31:0])
                ELSE
                    DEST[i+63:i] ←
            Convert_UInteger_To_Double_Precision_Floating_Point(SRC[k+31:k])
            FI;
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE                                ; zeroing-masking
                    DEST[i+63:i] ← 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VCVTUDQ2PD __m512d _mm512_cvtepu32_pd(__m256i a);
VCVTUDQ2PD __m512d _mm512_mask_cvtepu32_pd(__m512d s, __mmask8 k, __m256i a);
VCVTUDQ2PD __m512d _mm512_maskz_cvtepu32_pd(__mmask8 k, __m256i a);
VCVTUDQ2PD __m256d _mm256_cvtepu32_pd(__m128i a);
VCVTUDQ2PD __m256d _mm256_mask_cvtepu32_pd(__m256d s, __mmask8 k, __m128i a);
VCVTUDQ2PD __m256d _mm256_maskz_cvtepu32_pd(__mmask8 k, __m128i a);
VCVTUDQ2PD __m128d _mm_cvtepu32_pd(__m128i a);
VCVTUDQ2PD __m128d _mm_mask_cvtepu32_pd(__m128d s, __mmask8 k, __m128i a);
VCVTUDQ2PD __m128d _mm_maskz_cvtepu32_pd(__mmask8 k, __m128i a);

SIMD Floating-Point Exceptions

None

Other Exceptions

EVEX-encoded instructions, see Exceptions Type E5.

#UD	If EVEX.vvvv != 1111B.

VCVTUDQ2PS—Convert Packed Unsigned Doubleword Integers to Packed Single-Precision Floating-Point Values

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.128.F2.0F.W0 7A /r VCVTUDQ2PS xmm1 {k1}{z}, xmm2/m128/m32bcst	A	V/V	AVX512VL AVX512F	Convert four packed unsigned doubleword integers from xmm2/m128/m32bcst to packed single-precision floating-point values in xmm1 with writemask k1.
EVEX.256.F2.0F.W0 7A /r VCVTUDQ2PS ymm1 {k1}{z}, ymm2/m256/m32bcst	A	V/V	AVX512VL AVX512F	Convert eight packed unsigned doubleword integers from ymm2/m256/m32bcst to packed single-precision floating-point values in zmm1 with writemask k1.
EVEX.512.F2.0F.W0 7A /r VCVTUDQ2PS zmm1 {k1}{z}, zmm2/m512/m32bcst{er}	A	V/V	AVX512F	Convert sixteen packed unsigned doubleword integers from zmm2/m512/m32bcst to sixteen packed single-precision floating-point values in zmm1 with writemask k1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Full Vector	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

Description

Converts packed unsigned doubleword integers in the source operand (second operand) to single-precision floating-point values in the destination operand (first operand).

The source operand is a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 32-bit memory location. The destination operand is a ZMM/YMM/XMM register conditionally updated with writemask k1.

Note: EVEX.vvvv is reserved and must be 1111b, otherwise instructions will #UD.

Operation

VCVTUDQ2PS (EVEX encoded version) when src operand is a register

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
IF (VL = 512) AND (EVEX.b = 1)
    THEN
        SET_RM(EVEX.RC);
    ELSE
        SET_RM(MXCSR.RM);
FI;

FOR j ← 0 TO KL-1
    i ← j * 32
    IF k1[j] OR *no writemask*
        THEN DEST[i+31:i] ←
            Convert_UInteger_To_Single_Precision_Floating_Point(SRC[i+31:i])
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE                            ; zeroing-masking
                    DEST[i+31:i] ← 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0
```

VCVTUDQ2PS (EVEX encoded version) when src operand is a memory source
(KL, VL) = (4, 128), (8, 256), (16, 512)

```
FOR j ← 0 TO KL-1
    i ← j * 32
    IF k1[j] OR *no writemask*
        THEN
            IF (EVEX.b = 1)
                THEN
                    DEST[i+31:i] ←
            Convert_UInteger_To_Single_Precision_Floating_Point(SRC[31:0])
                ELSE
                    DEST[i+31:i] ←
            Convert_UInteger_To_Single_Precision_Floating_Point(SRC[i+31:i])
            FI;
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE                                ; zeroing-masking
                    DEST[i+31:i] ← 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VCVTUDQ2PS __m512 _mm512_cvtepu32_ps(__m512i a);
VCVTUDQ2PS __m512 _mm512_mask_cvtepu32_ps(__m512 s, __mmask16 k, __m512i a);
VCVTUDQ2PS __m512 _mm512_maskz_cvtepu32_ps(__mmask16 k, __m512i a);
VCVTUDQ2PS __m512 _mm512_cvt_roundepu32_ps(__m512i a, int r);
VCVTUDQ2PS __m512 _mm512_mask_cvt_roundepu32_ps(__m512 s, __mmask16 k, __m512i a, int r);
VCVTUDQ2PS __m512 _mm512_maskz_cvt_roundepu32_ps(__mmask16 k, __m512i a, int r);
VCVTUDQ2PS __m256 _mm256_cvtepu32_ps(__m256i a);
VCVTUDQ2PS __m256 _mm256_mask_cvtepu32_ps(__m256 s, __mmask8 k, __m256i a);
VCVTUDQ2PS __m256 _mm256_maskz_cvtepu32_ps(__mmask8 k, __m256i a);
VCVTUDQ2PS __m128 _mm_cvtepu32_ps(__m128i a);
VCVTUDQ2PS __m128 _mm_mask_cvtepu32_ps(__m128 s, __mmask8 k, __m128i a);
VCVTUDQ2PS __m128 _mm_maskz_cvtepu32_ps(__mmask8 k, __m128i a);

SIMD Floating-Point Exceptions

Precision

Other Exceptions

EVEX-encoded instructions, see Exceptions Type E2.

#UD	If EVEX.vvvv != 1111B.

VCVTUQQ2PD—Convert Packed Unsigned Quadword Integers to Packed Double-Precision Floating-Point Values

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.128.F3.0F.W1 7A /r VCVTUQQ2PD xmm1 {k1}{z}, xmm2/m128/m64bcst	A	V/V	AVX512VL AVX512DQ	Convert two packed unsigned quadword integers from xmm2/m128/m64bcst to two packed double-precision floating-point values in xmm1 with writemask k1.
EVEX.256.F3.0F.W1 7A /r VCVTUQQ2PD ymm1 {k1}{z}, ymm2/m256/m64bcst	A	V/V	AVX512VL AVX512DQ	Convert four packed unsigned quadword integers from ymm2/m256/m64bcst to packed double-precision floating-point values in ymm1 with writemask k1.
EVEX.512.F3.0F.W1 7A /r VCVTUQQ2PD zmm1 {k1}{z}, zmm2/m512/m64bcst{er}	A	V/V	AVX512DQ	Convert eight packed unsigned quadword integers from zmm2/m512/m64bcst to eight packed double-precision floating-point values in zmm1 with writemask k1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Full Vector	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

Description

Converts packed unsigned quadword integers in the source operand (second operand) to packed double-precision floating-point values in the destination operand (first operand).

The source operand is a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 64-bit memory location. The destination operand is a ZMM/YMM/XMM register conditionally updated with writemask k1.

Note: EVEX.vvvv is reserved and must be 1111b, otherwise instructions will #UD.

Operation

VCVTUQQ2PD (EVEX encoded version) when src operand is a register

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
IF (VL == 512) AND (EVEX.b == 1)
    THEN
        SET_RM(EVEX.RC);
    ELSE
        SET_RM(MXCSR.RM);
FI;

FOR j ← 0 TO KL-1
    i ← j * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+63:i] ←
            Convert_UQuadInteger_To_Double_Precision_Floating_Point(SRC[i+63:i])
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE                            ; zeroing-masking
                    DEST[i+63:i] ← 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0
```

VCVTUQQ2PD (EVEX encoded version) when src operand is a memory source
(KL, VL) = (2, 128), (4, 256), (8, 512)

```
FOR j ← 0 TO KL-1
    i ← j * 64
    IF k1[j] OR *no writemask*
        THEN
            IF (EVEX.b == 1)
                THEN
                    DEST[i+63:i] ←
            Convert_UQuadInteger_To_Double_Precision_Floating_Point(SRC[63:0])
                ELSE
                    DEST[i+63:i] ←
            Convert_UQuadInteger_To_Double_Precision_Floating_Point(SRC[i+63:i])
            FI;
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE                                ; zeroing-masking
                    DEST[i+63:i] ← 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VCVTUQQ2PD __m512d _mm512_cvtepu64_ps(__m512i a);
VCVTUQQ2PD __m512d _mm512_mask_cvtepu64_ps(__m512d s, __mmask8 k, __m512i a);
VCVTUQQ2PD __m512d _mm512_maskz_cvtepu64_ps(__mmask8 k, __m512i a);
VCVTUQQ2PD __m512d _mm512_cvt_roundepu64_ps(__m512i a, int r);
VCVTUQQ2PD __m512d _mm512_mask_cvt_roundepu64_ps(__m512d s, __mmask8 k, __m512i a, int r);
VCVTUQQ2PD __m512d _mm512_maskz_cvt_roundepu64_ps(__mmask8 k, __m512i a, int r);
VCVTUQQ2PD __m256d _mm256_cvtepu64_ps(__m256i a);
VCVTUQQ2PD __m256d _mm256_mask_cvtepu64_ps(__m256d s, __mmask8 k, __m256i a);
VCVTUQQ2PD __m256d _mm256_maskz_cvtepu64_ps(__mmask8 k, __m256i a);
VCVTUQQ2PD __m128d _mm_cvtepu64_ps(__m128i a);
VCVTUQQ2PD __m128d _mm_mask_cvtepu64_ps(__m128d s, __mmask8 k, __m128i a);
VCVTUQQ2PD __m128d _mm_maskz_cvtepu64_ps(__mmask8 k, __m128i a);

SIMD Floating-Point Exceptions

Precision

Other Exceptions

EVEX-encoded instructions, see Exceptions Type E2.

#UD	If EVEX.vvvv != 1111B.

VCVTUQQ2PS—Convert Packed Unsigned Quadword Integers to Packed Single-Precision Floating-Point Values

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.128.F2.0F.W1 7A /r VCVTUQQ2PS xmm1 {k1}{z}, xmm2/m128/m64bcst	A	V/V	AVX512VL AVX512DQ	Convert two packed unsigned quadword integers from xmm2/m128/m64bcst to packed single-precision floating-point values in zmm1 with writemask k1.
EVEX.256.F2.0F.W1 7A /r VCVTUQQ2PS xmm1 {k1}{z}, ymm2/m256/m64bcst	A	V/V	AVX512VL AVX512DQ	Convert four packed unsigned quadword integers from ymm2/m256/m64bcst to packed single-precision floating-point values in xmm1 with writemask k1.
EVEX.512.F2.0F.W1 7A /r VCVTUQQ2PS ymm1 {k1}{z}, zmm2/m512/m64bcst{er}	A	V/V	AVX512DQ	Convert eight packed unsigned quadword integers from zmm2/m512/m64bcst to eight packed single-precision floating-point values in zmm1 with writemask k1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Full Vector	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

Description

Converts packed unsigned quadword integers in the source operand (second operand) to single-precision floating-point values in the destination operand (first operand).

EVEX encoded versions: The source operand is a ZMM/YMM/XMM register or a 512/256/128-bit memory location. The destination operand is a YMM/XMM/XMM (low 64 bits) register conditionally updated with writemask k1.

Note: EVEX.vvvv is reserved and must be 1111b, otherwise instructions will #UD.

Operation

VCVTUQQ2PS (EVEX encoded version) when src operand is a register

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
IF (VL = 512) AND (EVEX.b = 1)
    THEN
        SET_RM(EVEX.RC);
    ELSE
        SET_RM(MXCSR.RM);
FI;

FOR j ← 0 TO KL-1
    i ← j * 32
    k ← j * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+31:i] ←
            Convert_UQuadInteger_To_Single_Precision_Floating_Point(SRC[k+63:k])
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE                            ; zeroing-masking
                    DEST[i+31:i] ← 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL/2] ← 0
```

VCVTUQQ2PS (EVEX encoded version) when src operand is a memory source
(KL, VL) = (2, 128), (4, 256), (8, 512)

```
FOR j ← 0 TO KL-1
    i ← j * 32
    k ← j * 64
    IF k1[j] OR *no writemask*
        THEN
            IF (EVEX.b = 1)
                THEN
                    DEST[i+31:i] ←
            Convert_UQuadInteger_To_Single_Precision_Floating_Point(SRC[63:0])
                ELSE
                    DEST[i+31:i] ←
            Convert_UQuadInteger_To_Single_Precision_Floating_Point(SRC[k+63:k])
            FI;
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE                                ; zeroing-masking
                    DEST[i+31:i] ← 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL/2] ← 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VCVTUQQ2PS __m256 _mm512_cvtepu64_ps(__m512i a);
VCVTUQQ2PS __m256 _mm512_mask_cvtepu64_ps(__m256 s, __mmask8 k, __m512i a);
VCVTUQQ2PS __m256 _mm512_maskz_cvtepu64_ps(__mmask8 k, __m512i a);
VCVTUQQ2PS __m256 _mm512_cvt_roundepu64_ps(__m512i a, int r);
VCVTUQQ2PS __m256 _mm512_mask_cvt_roundepu64_ps(__m256 s, __mmask8 k, __m512i a, int r);
VCVTUQQ2PS __m256 _mm512_maskz_cvt_roundepu64_ps(__mmask8 k, __m512i a, int r);
VCVTUQQ2PS __m128 _mm256_cvtepu64_ps(__m256i a);
VCVTUQQ2PS __m128 _mm256_mask_cvtepu64_ps(__m128 s, __mmask8 k, __m256i a);
VCVTUQQ2PS __m128 _mm256_maskz_cvtepu64_ps(__mmask8 k, __m256i a);
VCVTUQQ2PS __m128 _mm_cvtepu64_ps(__m128i a);
VCVTUQQ2PS __m128 _mm_mask_cvtepu64_ps(__m128 s, __mmask8 k, __m128i a);
VCVTUQQ2PS __m128 _mm_maskz_cvtepu64_ps(__mmask8 k, __m128i a);

SIMD Floating-Point Exceptions

Precision

Other Exceptions

EVEX-encoded instructions, see Exceptions Type E2.

#UD	If EVEX.vvvv != 1111B.

VCVTUSI2SD—Convert Unsigned Integer to Scalar Double-Precision Floating-Point Value

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.NDS.LIG.F2.0F.W0 7B /r VCVTUSI2SD xmm1, xmm2, r/m32	A	V/V	AVX512F	Convert one unsigned doubleword integer from r/m32 to one double-precision floating-point value in xmm1.
EVEX.NDS.LIG.F2.0F.W1 7B /r VCVTUSI2SD xmm1, xmm2, r/m64{er}	A	V/N.E.[1]	AVX512F	Convert one unsigned quadword integer from r/m64 to one double-precision floating-point value in xmm1.

NOTES:

1. For this specific instruction, EVEX.W in non-64 bit is ignored; the instructions behaves as if the W0 version is used.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Tuple1 Scalar	ModRM:reg (w)	EVEX.vvvv	ModRM:r/m (r)	NA

Description

Converts an unsigned doubleword integer (or unsigned quadword integer if operand size is 64 bits) in the second source operand to a double-precision floating-point value in the destination operand. The result is stored in the low quadword of the destination operand. When conversion is inexact, the value returned is rounded according to the rounding control bits in the MXCSR register.

The second source operand can be a general-purpose register or a 32/64-bit memory location. The first source and destination operands are XMM registers. Bits (127:64) of the XMM register destination are copied from corresponding bits in the first source operand. Bits (MAXVL-1:128) of the destination register are zeroed.

EVEX.W1 version: promotes the instruction to use 64-bit input value in 64-bit mode.

EVEX.W0 version: attempt to encode this instruction with EVEX embedded rounding is ignored.

Operation

VCVTUSI2SD (EVEX encoded version)

```
IF (SRC2 *is register*) AND (EVEX.b = 1)
    THEN
        SET_RM(EVEX.RC);
    ELSE
        SET_RM(MXCSR.RM);
FI;
IF 64-Bit Mode And OperandSize = 64
THEN
    DEST[63:0] ← Convert_UInteger_To_Double_Precision_Floating_Point(SRC2[63:0]);
ELSE
    DEST[63:0] ← Convert_UInteger_To_Double_Precision_Floating_Point(SRC2[31:0]);
FI;
DEST[127:64] ← SRC1[127:64]
DEST[MAXVL-1:128] ← 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VCVTUSI2SD __m128d _mm_cvtu32_sd(__m128d s, unsigned a);
VCVTUSI2SD __m128d _mm_cvtu64_sd(__m128d s, unsigned __int64 a);
VCVTUSI2SD __m128d _mm_cvt_roundu64_sd(__m128d s, unsigned __int64 a, int r);

SIMD Floating-Point Exceptions

Precision

Other Exceptions

See Exceptions Type E3NF if W1, else type E10NF.

VCVTUSI2SS—Convert Unsigned Integer to Scalar Single-Precision Floating-Point Value

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.NDS.LIG.F3.0F.W0 7B /r VCVTUSI2SS xmm1, xmm2, r/m32{er}	A	V/V	AVX512F	Convert one signed doubleword integer from r/m32 to one single-precision floating-point value in xmm1.
EVEX.NDS.LIG.F3.0F.W1 7B /r VCVTUSI2SS xmm1, xmm2, r/m64{er}	A	V/N.E.[1]	AVX512F	Convert one signed quadword integer from r/m64 to one single-precision floating-point value in xmm1.

NOTES:

1. For this specific instruction, EVEX.W in non-64 bit is ignored; the instructions behaves as if the W0 version is used.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Tuple1 Scalar	ModRM:reg (w)	VEX.vvvv	ModRM:r/m (r)	NA

Description

Converts a unsigned doubleword integer (or unsigned quadword integer if operand size is 64 bits) in the source operand (second operand) to a single-precision floating-point value in the destination operand (first operand). The source operand can be a general-purpose register or a memory location. The destination operand is an XMM register. The result is stored in the low doubleword of the destination operand. When a conversion is inexact, the value returned is rounded according to the rounding control bits in the MXCSR register or the embedded rounding control bits.

The second source operand can be a general-purpose register or a 32/64-bit memory location. The first source and destination operands are XMM registers. Bits (127:32) of the XMM register destination are copied from corresponding bits in the first source operand. Bits (MAXVL-1:128) of the destination register are zeroed.

EVEX.W1 version: promotes the instruction to use 64-bit input value in 64-bit mode.

Operation

VCVTUSI2SS (EVEX encoded version)

```
IF (SRC2 *is register*) AND (EVEX.b = 1)
    THEN
        SET_RM(EVEX.RC);
    ELSE
        SET_RM(MXCSR.RM);
FI;
IF 64-Bit Mode And OperandSize = 64
THEN
    DEST[31:0] ← Convert_UInteger_To_Single_Precision_Floating_Point(SRC[63:0]);
ELSE
    DEST[31:0] ← Convert_UInteger_To_Single_Precision_Floating_Point(SRC[31:0]);
FI;
DEST[127:32] ← SRC1[127:32]
DEST[MAXVL-1:128] ← 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VCVTUSI2SS __m128 _mm_cvtu32_ss(__m128 s, unsigned a);
VCVTUSI2SS __m128 _mm_cvt_roundu32_ss(__m128 s, unsigned a, int r);
VCVTUSI2SS __m128 _mm_cvtu64_ss(__m128 s, unsigned __int64 a);
VCVTUSI2SS __m128 _mm_cvt_roundu64_ss(__m128 s, unsigned __int64 a, int r);

SIMD Floating-Point Exceptions

Precision

Other Exceptions

See Exceptions Type E3NF.

VDBPSADBW—Double Block Packed Sum-Absolute-Differences (SAD) on Unsigned Bytes

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.NDS.128.66.0F3A.W0 42 /r ib VDBPSADBW xmm1 {k1}{z}, xmm2, xmm3/m128, imm8	FVM	V/V	AVX512VL AVX512BW	Compute packed SAD word results of unsigned bytes in dword block from xmm2 with unsigned bytes of dword blocks transformed from xmm3/m128 using the shuffle controls in imm8. Results are written to xmm1 under the writemask k1.
EVEX.NDS.256.66.0F3A.W0 42 /r ib VDBPSADBW ymm1 {k1}{z}, ymm2, ymm3/m256, imm8	FVM	V/V	AVX512VL AVX512BW	Compute packed SAD word results of unsigned bytes in dword block from ymm2 with unsigned bytes of dword blocks transformed from ymm3/m256 using the shuffle controls in imm8. Results are written to ymm1 under the writemask k1.
EVEX.NDS.512.66.0F3A.W0 42 /r ib VDBPSADBW zmm1 {k1}{z}, zmm2, zmm3/m512, imm8	FVM	V/V	AVX512BW	Compute packed SAD word results of unsigned bytes in dword block from zmm2 with unsigned bytes of dword blocks transformed from zmm3/m512 using the shuffle controls in imm8. Results are written to zmm1 under the writemask k1.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
FVM	ModRM:reg (w)	EVEX.vvvv	ModRM:r/m (r)	Imm8

Description

Compute packed SAD (sum of absolute differences) word results of unsigned bytes from two 32-bit dword elements. Packed SAD word results are calculated in multiples of qword superblocks, producing 4 SAD word results in each 64-bit superblock of the destination register.

Within each super block of packed word results, the SAD results from two 32-bit dword elements are calculated as follows:

- The lower two word results are calculated each from the SAD operation between a sliding dword element within a qword superblock from an intermediate vector with a stationary dword element in the corresponding qword superblock of the first source operand. The intermediate vector, see "Tmp1" in Figure 5-8, is constructed from the second source operand the imm8 byte as shuffle control to select dword elements within a 128-bit lane of the second source operand. The two sliding dword elements in a qword superblock of Tmp1 are located at byte offset 0 and 1 within the superblock, respectively. The stationary dword element in the qword superblock from the first source operand is located at byte offset 0.
- The next two word results are calculated each from the SAD operation between a sliding dword element within a qword superblock from the intermediate vector Tmp1 with a second stationary dword element in the corresponding qword superblock of the first source operand. The two sliding dword elements in a qword superblock of Tmp1 are located at byte offset 2and 3 within the superblock, respectively. The stationary dword element in the qword superblock from the first source operand is located at byte offset 4.
- The intermediate vector is constructed in 128-bits lanes. Within each 128-bit lane, each dword element of the intermediate vector is selected by a two-bit field within the imm8 byte on the corresponding 128-bits of the second source operand. The imm8 byte serves as dword shuffle control within each 128-bit lanes of the intermediate vector and the second source operand, similarly to PSHUFD.

The first source operand is a ZMM/YMM/XMM register. The second source operand is a ZMM/YMM/XMM register, or a 512/256/128-bit memory location. The destination operand is conditionally updated based on writemask k1 at 16-bit word granularity.

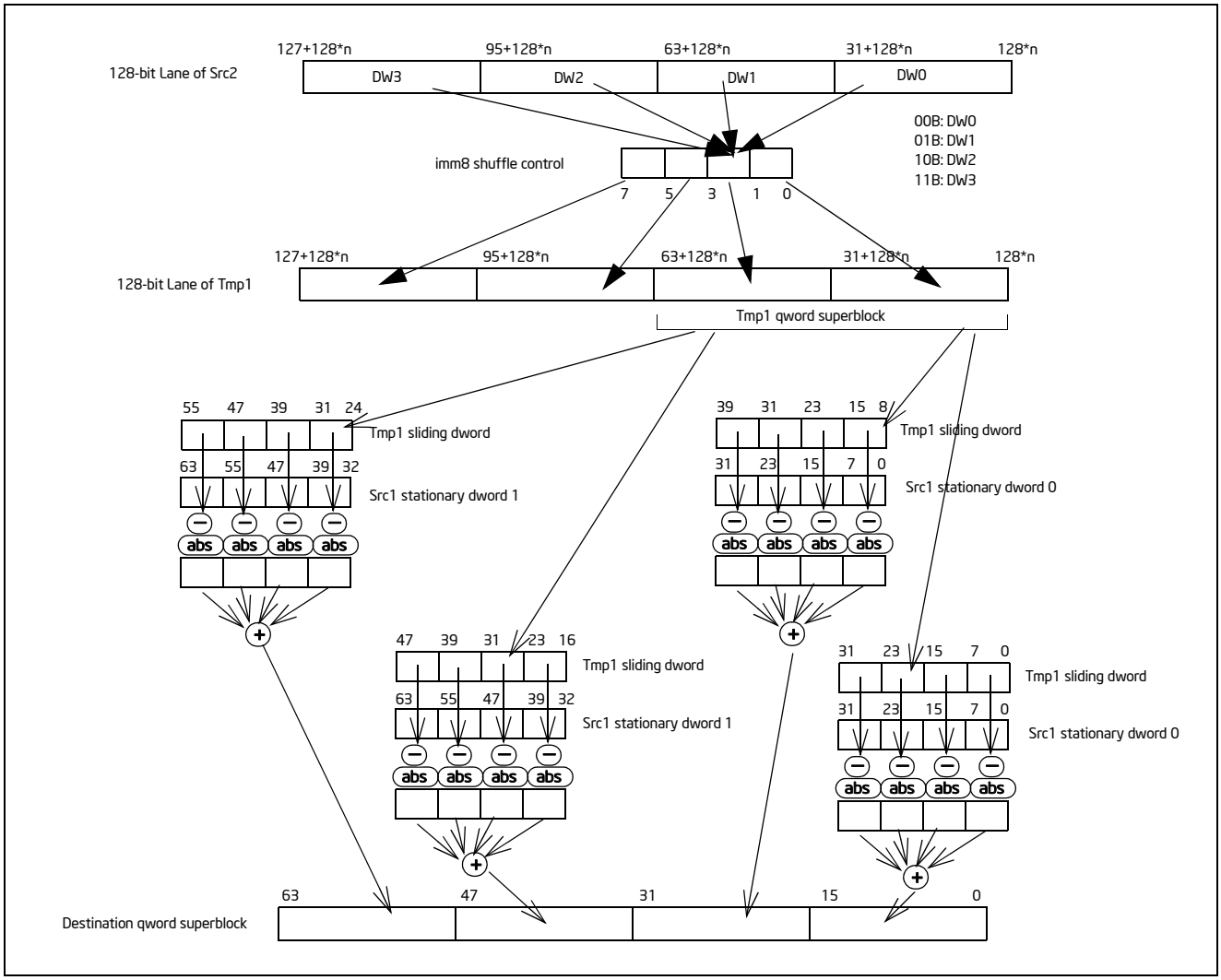


Figure 5-8. 64-bit Super Block of SAD Operation in VDBPSADBW

Operation

VDBPSADBW (EVEX encoded versions)

```
(KL, VL) = (8, 128), (16, 256), (32, 512)
Selection of quadruplets:
FOR l = 0 to VL step 128
    TMP1[l+31:l] ← select (SRC2[l+127: l], imm8[1:0])
    TMP1[l+63: l+32] ← select (SRC2[l+127: l], imm8[3:2])
    TMP1[l+95: l+64] ← select (SRC2[l+127: l], imm8[5:4])
    TMP1[l+127: l+96]← select (SRC2[l+127: l], imm8[7:6])
END FOR

SAD of quadruplets:

FOR l =0 to VL step 64
    TMP_DEST[l+15:l] ← ABS(SRC1[l+7: l] - TMP1[l+7: l]) +
        ABS(SRC1[l+15: l+8]- TMP1[l+15: l+8]) +
        ABS(SRC1[l+23: l+16]- TMP1[l+23: l+16]) +
        ABS(SRC1[l+31: l+24]- TMP1[l+31: l+24])

    TMP_DEST[l+31: l+16] ←ABS(SRC1[l+7: l] - TMP1[l+15: l+8]) +
        ABS(SRC1[l+15: l+8]- TMP1[l+23: l+16]) +
        ABS(SRC1[l+23: l+16]- TMP1[l+31: l+24]) +
        ABS(SRC1[l+31: l+24]- TMP1[l+39: l+32])
    TMP_DEST[l+47: l+32] ←ABS(SRC1[l+39: l+32] - TMP1[l+23: l+16]) +
        ABS(SRC1[l+47: l+40]- TMP1[l+31: l+24]) +
        ABS(SRC1[l+55: l+48]- TMP1[l+39: l+32]) +
        ABS(SRC1[l+63: l+56]- TMP1[l+47: l+40])

    TMP_DEST[l+63: l+48] ←ABS(SRC1[l+39: l+32] - TMP1[l+31: l+24]) +
        ABS(SRC1[l+47: l+40] - TMP1[l+39: l+32]) +
        ABS(SRC1[l+55: l+48] - TMP1[l+47: l+40]) +
        ABS(SRC1[l+63: l+56] - TMP1[l+55: l+48])
ENDFOR

FOR j ← 0 TO KL-1
    i ←  j * 16
    IF k1[j] OR *no writemask*
        THEN DEST[i+15:i] ← TMP_DEST[i+15:i]
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+15:i] remains unchanged*
                ELSE                                ; zeroing-masking
                    DEST[i+15:i] ← 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VDBPSADBW __m512i _mm512_dbsad_epu8(__m512i a, __m512i b);
VDBPSADBW __m512i _mm512_mask_dbsad_epu8(__m512i s, __mmask32 m, __m512i a, __m512i b);
VDBPSADBW __m512i _mm512_maskz_dbsad_epu8(__mmask32 m, __m512i a, __m512i b);
VDBPSADBW __m256i _mm256_dbsad_epu8(__m256i a, __m256i b);
VDBPSADBW __m256i _mm256_mask_dbsad_epu8(__m256i s, __mmask16 m, __m256i a, __m256i b);
VDBPSADBW __m256i _mm256_maskz_dbsad_epu8(__mmask16 m, __m256i a, __m256i b);
VDBPSADBW __m128i _mm_dbsad_epu8(__m128i a, __m128i b);
VDBPSADBW __m128i _mm_mask_dbsad_epu8(__m128i s, __mmask8 m, __m128i a, __m128i b);
VDBPSADBW __m128i _mm_maskz_dbsad_epu8(__mmask8 m, __m128i a, __m128i b);

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type E4NF.nb.

VEXPANDPD—Load Sparse Packed Double-Precision Floating-Point Values from Dense Memory

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.128.66.0F38.W1 88 /r VEXPANDPD xmm1 {k1}{z}, xmm2/m128	A	V/V	AVX512VL AVX512F	Expand packed double-precision floating-point values from xmm2/m128 to xmm1 using writemask k1.
EVEX.256.66.0F38.W1 88 /r VEXPANDPD ymm1 {k1}{z}, ymm2/m256	A	V/V	AVX512VL AVX512F	Expand packed double-precision floating-point values from ymm2/m256 to ymm1 using writemask k1.
EVEX.512.66.0F38.W1 88 /r VEXPANDPD zmm1 {k1}{z}, zmm2/m512	A	V/V	AVX512F	Expand packed double-precision floating-point values from zmm2/m512 to zmm1 using writemask k1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Tuple1 Scalar	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

Description

Expand (load) up to 8/4/2, contiguous, double-precision floating-point values of the input vector in the source operand (the second operand) to sparse elements in the destination operand (the first operand) selected by the writemask k1.

The destination operand is a ZMM/YMM/XMM register, the source operand can be a ZMM/YMM/XMM register or a 512/256/128-bit memory location.

The input vector starts from the lowest element in the source operand. The writemask register k1 selects the destination elements (a partial vector or sparse elements if less than 8 elements) to be replaced by the ascending elements in the input vector. Destination elements not selected by the writemask k1 are either unmodified or zeroed, depending on EVEX.z.

EVEX.vvvv is reserved and must be 1111b otherwise instructions will #UD.

Note that the compressed displacement assumes a pre-scaling (N) corresponding to the size of one single element instead of the size of the full vector.

Operation

VEXPANDPD (EVEX encoded versions)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
k ← 0
FOR j ← 0 TO KL-1
    i ← j * 64
    IF k1[j] OR *no writemask*
        THEN
            DEST[i+63:i] ← SRC[k+63:k];
            k ← k + 64
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE                                ; zeroing-masking
                    THEN DEST[i+63:i] ← 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] ←   0
```

Intel C/C++ Compiler Intrinsic Equivalent

VEXPANDPD __m512d _mm512_mask_expand_pd(__m512d s, __mmask8 k, __m512d a);
VEXPANDPD __m512d _mm512_maskz_expand_pd(__mmask8 k, __m512d a);
VEXPANDPD __m512d _mm512_mask_expandloadu_pd(__m512d s, __mmask8 k, void * a);
VEXPANDPD __m512d _mm512_maskz_expandloadu_pd(__mmask8 k, void * a);
VEXPANDPD __m256d _mm256_mask_expand_pd(__m256d s, __mmask8 k, __m256d a);
VEXPANDPD __m256d _mm256_maskz_expand_pd(__mmask8 k, __m256d a);
VEXPANDPD __m256d _mm256_mask_expandloadu_pd(__m256d s, __mmask8 k, void * a);
VEXPANDPD __m256d _mm256_maskz_expandloadu_pd(__mmask8 k, void * a);
VEXPANDPD __m128d _mm_mask_expand_pd(__m128d s, __mmask8 k, __m128d a);
VEXPANDPD __m128d _mm_maskz_expand_pd(__mmask8 k, __m128d a);
VEXPANDPD __m128d _mm_mask_expandloadu_pd(__m128d s, __mmask8 k, void * a);
VEXPANDPD __m128d _mm_maskz_expandloadu_pd(__mmask8 k, void * a);

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type E4.nb.

#UD	If EVEX.vvvv != 1111B.

VEXPANDPS—Load Sparse Packed Single-Precision Floating-Point Values from Dense Memory

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.128.66.0F38.W0 88 /r VEXPANDPS xmm1 {k1}{z}, xmm2/m128	A	V/V	AVX512VL AVX512F	Expand packed single-precision floating-point values from xmm2/m128 to xmm1 using writemask k1.
EVEX.256.66.0F38.W0 88 /r VEXPANDPS ymm1 {k1}{z}, ymm2/m256	A	V/V	AVX512VL AVX512F	Expand packed single-precision floating-point values from ymm2/m256 to ymm1 using writemask k1.
EVEX.512.66.0F38.W0 88 /r VEXPANDPS zmm1 {k1}{z}, zmm2/m512	A	V/V	AVX512F	Expand packed single-precision floating-point values from zmm2/m512 to zmm1 using writemask k1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Tuple1 Scalar	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

Description

Expand (load) up to 16/8/4, contiguous, single-precision floating-point values of the input vector in the source operand (the second operand) to sparse elements of the destination operand (the first operand) selected by the writemask k1.

The destination operand is a ZMM/YMM/XMM register, the source operand can be a ZMM/YMM/XMM register or a 512/256/128-bit memory location.

The input vector starts from the lowest element in the source operand. The writemask k1 selects the destination elements (a partial vector or sparse elements if less than 16 elements) to be replaced by the ascending elements in the input vector. Destination elements not selected by the writemask k1 are either unmodified or zeroed, depending on EVEX.z.

EVEX.vvvv is reserved and must be 1111b otherwise instructions will #UD.

Note that the compressed displacement assumes a pre-scaling (N) corresponding to the size of one single element instead of the size of the full vector.

Operation

VEXPANDPS (EVEX encoded versions)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
k ← 0
FOR j ← 0 TO KL-1
    i ← j * 32
    IF k1[j] OR *no writemask*
        THEN
            DEST[i+31:i] ← SRC[k+31:k];
            k ← k + 32
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE                                ; zeroing-masking
                    DEST[i+31:i] ← 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] ←  0
```

Intel C/C++ Compiler Intrinsic Equivalent

VEXPANDPS __m512 _mm512_mask_expand_ps(__m512 s, __mmask16 k, __m512 a);
VEXPANDPS __m512 _mm512_maskz_expand_ps(__mmask16 k, __m512 a);
VEXPANDPS __m512 _mm512_mask_expandloadu_ps(__m512 s, __mmask16 k, void * a);
VEXPANDPS __m512 _mm512_maskz_expandloadu_ps(__mmask16 k, void * a);
VEXPANDPD __m256 _mm256_mask_expand_ps(__m256 s, __mmask8 k, __m256 a);
VEXPANDPD __m256 _mm256_maskz_expand_ps(__mmask8 k, __m256 a);
VEXPANDPD __m256 _mm256_mask_expandloadu_ps(__m256 s, __mmask8 k, void * a);
VEXPANDPD __m256 _mm256_maskz_expandloadu_ps(__mmask8 k, void * a);
VEXPANDPD __m128 _mm_mask_expand_ps(__m128 s, __mmask8 k, __m128 a);
VEXPANDPD __m128 _mm_maskz_expand_ps(__mmask8 k, __m128 a);
VEXPANDPD __m128 _mm_mask_expandloadu_ps(__m128 s, __mmask8 k, void * a);
VEXPANDPD __m128 _mm_maskz_expandloadu_ps(__mmask8 k, void * a);

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type E4.nb.

#UD	If EVEX.vvvv != 1111B.

VERR/VERW—Verify a Segment for Reading or Writing

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
0F 00 /4	VERR *r/m16*	M	Valid	Valid	Set ZF=1 if segment specified with *r/m16* can be read.
0F 00 /5	VERW *r/m16*	M	Valid	Valid	Set ZF=1 if segment specified with *r/m16* can be written.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
M	ModRM:r/m (r)	NA	NA	NA

Description

Verifies whether the code or data segment specified with the source operand is readable (VERR) or writable (VERW) from the current privilege level (CPL). The source operand is a 16-bit register or a memory location that contains the segment selector for the segment to be verified. If the segment is accessible and readable (VERR) or writable (VERW), the ZF flag is set; otherwise, the ZF flag is cleared. Code segments are never verified as writable. This check cannot be performed on system segments.

To set the ZF flag, the following conditions must be met:

- The segment selector is not NULL.
- The selector must denote a descriptor within the bounds of the descriptor table (GDT or LDT).
- The selector must denote the descriptor of a code or data segment (not that of a system segment or gate).
- For the VERR instruction, the segment must be readable.
- For the VERW instruction, the segment must be a writable data segment.
- If the segment is not a conforming code segment, the segment's DPL must be greater than or equal to (have less or the same privilege as) both the CPL and the segment selector's RPL.

The validation performed is the same as is performed when a segment selector is loaded into the DS, ES, FS, or GS register, and the indicated access (read or write) is performed. The segment selector's value cannot result in a protection exception, enabling the software to anticipate possible segment access problems.

This instruction's operation is the same in non-64-bit modes and 64-bit mode. The operand size is fixed at 16 bits.

Operation

```
IF SRC(Offset) > (GDTR(Limit) or (LDTR(Limit))
    THEN ZF ← 0; FI;

Read segment descriptor;

IF SegmentDescriptor(DescriptorType) = 0 (* System segment *)
or (SegmentDescriptor(Type) ≠ conforming code segment)
and (CPL > DPL) or (RPL > DPL)
    THEN
        ZF ← 0;
    ELSE
        IF ((Instruction = VERR) and (Segment readable))
        or ((Instruction = VERW) and (Segment writable))
            THEN
                ZF ← 1;
        FI;
FI;
```

Flags Affected

The ZF flag is set to 1 if the segment is accessible and readable (VERR) or writable (VERW); otherwise, it is set to 0.

Protected Mode Exceptions

The only exceptions generated for these instructions are those related to illegal addressing of the source operand.

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
	If the DS, ES, FS, or GS register is used to access memory and it contains a NULL segment selector.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If the LOCK prefix is used.

Real-Address Mode Exceptions

#UD	The VERR and VERW instructions are not recognized in real-address mode.
	If the LOCK prefix is used.

Virtual-8086 Mode Exceptions

#UD	The VERR and VERW instructions are not recognized in virtual-8086 mode.
	If the LOCK prefix is used.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If the LOCK prefix is used.

VEXP2PD—Approximation to the Exponential 2^x of Packed Double-Precision Floating-Point Values with Less Than 2^-23 Relative Error

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.512.66.0F38.W1 C8 /r VEXP2PD zmm1 {k1}{z}, zmm2/m512/m64bcst {sae}	A	V/V	AVX512ER	Computes approximations to the exponential 2^x (with less than 2^-23 of maximum relative error) of the packed double-precision floating-point values from zmm2/m512/m64bcst and stores the floating-point result in zmm1with writemask k1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Full Vector	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA

Description

Computes the approximate base-2 exponential evaluation of the double-precision floating-point values in the source operand (the second operand) and stores the results to the destination operand (the first operand) using the writemask k1. The approximate base-2 exponential is evaluated with less than 2^-23 of relative error.

Denormal input values are treated as zeros and do not signal #DE, irrespective of MXCSR.DAZ. Denormal results are flushed to zeros and do not signal #UE, irrespective of MXCSR.FZ.

The source operand is a ZMM register, a 512-bit memory location or a 512-bit vector broadcasted from a 64-bit memory location. The destination operand is a ZMM register, conditionally updated using writemask k1.

EVEX.vvvv is reserved and must be 1111b otherwise instructions will #UD.

A numerically exact implementation of VEXP2xx can be found at https://software.intel.com/en-us/articles/reference-implementations-for-IA-approximation-instructions-vrcp14-vrsqrt14-vrcp28-vrsqrt28-vexp2.

Operation

VEXP2PD

```
(KL, VL) = (8, 512)
FOR j ← 0 TO KL-1
    i ← j * 64
    IF k1[j] OR *no writemask* THEN
            IF (EVEX.b = 1) AND (SRC *is memory*)
                THEN DEST[i+63:i] ← EXP2_23_DP(SRC[63:0])
                ELSE DEST[i+63:i] ← EXP2_23_DP(SRC[i+63:i])
            FI;
    ELSE
        IF *merging-masking*                    ; merging-masking
            THEN *DEST[i+63:i] remains unchanged*
            ELSE                                ; zeroing-masking
                DEST[i+63:i] ← 0
        FI;
    FI;
ENDFOR;
```

Table 5-4. Special Values Behavior

Source Input	Result	Comments
NaN	QNaN(src)	If (SRC = SNaN) then #I
+∞	+∞	
+/-0	1.0f	*Exact result*
-∞	+0.0f	
Integral value N	2^ (N)	*Exact result*

Intel C/C++ Compiler Intrinsic Equivalent

VEXP2PD __m512d _mm512_exp2a23_round_pd (__m512d a, int sae);
VEXP2PD __m512d _mm512_mask_exp2a23_round_pd (__m512d a, __mmask8 m, __m512d b, int sae);
VEXP2PD __m512d _mm512_maskz_exp2a23_round_pd (__mmask8 m, __m512d b, int sae);

SIMD Floating-Point Exceptions

Invalid (if SNaN input), Overflow

Other Exceptions

See Exceptions Type E2.

VEXP2PS—Approximation to the Exponential 2^x of Packed Single-Precision Floating-Point Values with Less Than 2^-23 Relative Error

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.512.66.0F38.W0 C8 /r VEXP2PS zmm1 {k1}{z}, zmm2/m512/m32bcst {sae}	A	V/V	AVX512ER	Computes approximations to the exponential 2^x (with less than 2^-23 of maximum relative error) of the packed single-precision floating-point values from zmm2/m512/m32bcst and stores the floating-point result in zmm1with writemask k1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Full Vector	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA

Description

Computes the approximate base-2 exponential evaluation of the single-precision floating-point values in the source operand (the second operand) and store the results in the destination operand (the first operand) using the writemask k1. The approximate base-2 exponential is evaluated with less than 2^-23 of relative error.

Denormal input values are treated as zeros and do not signal #DE, irrespective of MXCSR.DAZ. Denormal results are flushed to zeros and do not signal #UE, irrespective of MXCSR.FZ.

The source operand is a ZMM register, a 512-bit memory location, or a 512-bit vector broadcasted from a 32-bit memory location. The destination operand is a ZMM register, conditionally updated using writemask k1.

EVEX.vvvv is reserved and must be 1111b otherwise instructions will #UD.

A numerically exact implementation of VEXP2xx can be found at https://software.intel.com/en-us/articles/reference-implementations-for-IA-approximation-instructions-vrcp14-vrsqrt14-vrcp28-vrsqrt28-vexp2.

Operation

VEXP2PS

```
(KL, VL) = (16, 512)
FOR j ← 0 TO KL-1
    i ← j * 32
    IF k1[j] OR *no writemask* THEN
            IF (EVEX.b = 1) AND (SRC *is memory*)
                THEN DEST[i+31:i] ← EXP2_23_SP(SRC[31:0])
                ELSE DEST[i+31:i] ← EXP2_23_SP(SRC[i+31:i])
            FI;
    ELSE
        IF *merging-masking*                    ; merging-masking
            THEN *DEST[i+31:i] remains unchanged*
            ELSE                                ; zeroing-masking
                DEST[i+31:i] ← 0
        FI;
    FI;
ENDFOR;
```

Table 5-5. Special Values Behavior

Source Input	Result	Comments
NaN	QNaN(src)	If (SRC = SNaN) then #I
+∞	+∞	
+/-0	1.0f	*Exact result*
-∞	+0.0f	
Integral value N	2^ (N)	*Exact result*

Intel C/C++ Compiler Intrinsic Equivalent

VEXP2PS __m512 _mm512_exp2a23_round_ps (__m512 a, int sae);
VEXP2PS __m512 _mm512_mask_exp2a23_round_ps (__m512 a, __mmask16 m, __m512 b, int sae);
VEXP2PS __m512 _mm512_maskz_exp2a23_round_ps (__mmask16 m, __m512 b, int sae);

SIMD Floating-Point Exceptions

Invalid (if SNaN input), Overflow

Other Exceptions

See Exceptions Type E2.

VEXTRACTF128/VEXTRACTF32x4/VEXTRACTF64x2/VEXTRACTF32x8/VEXTRACTF64x4—Extract Packed Floating-Point Values

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
VEX.256.66.0F3A.W0 19 /r ib VEXTRACTF128 xmm1/m128, ymm2, imm8	A	V/V	AVX	Extract 128 bits of packed floating-point values from ymm2 and store results in xmm1/m128.
EVEX.256.66.0F3A.W0 19 /r ib VEXTRACTF32X4 xmm1/m128 {k1}{z}, ymm2, imm8	C	V/V	AVX512VL AVX512F	Extract 128 bits of packed single-precision floating-point values from ymm2 and store results in xmm1/m128 subject to writemask k1.
EVEX.512.66.0F3A.W0 19 /r ib VEXTRACTF32x4 xmm1/m128 {k1}{z}, zmm2, imm8	C	V/V	AVX512F	Extract 128 bits of packed single-precision floating-point values from zmm2 and store results in xmm1/m128 subject to writemask k1.
EVEX.256.66.0F3A.W1 19 /r ib VEXTRACTF64X2 xmm1/m128 {k1}{z}, ymm2, imm8	B	V/V	AVX512VL AVX512DQ	Extract 128 bits of packed double-precision floating-point values from ymm2 and store results in xmm1/m128 subject to writemask k1.
EVEX.512.66.0F3A.W1 19 /r ib VEXTRACTF64X2 xmm1/m128 {k1}{z}, zmm2, imm8	B	V/V	AVX512DQ	Extract 128 bits of packed double-precision floating-point values from zmm2 and store results in xmm1/m128 subject to writemask k1.
EVEX.512.66.0F3A.W0 1B /r ib VEXTRACTF32X8 ymm1/m256 {k1}{z}, zmm2, imm8	D	V/V	AVX512DQ	Extract 256 bits of packed single-precision floating-point values from zmm2 and store results in ymm1/m256 subject to writemask k1.
EVEX.512.66.0F3A.W1 1B /r ib VEXTRACTF64x4 ymm1/m256 {k1}{z}, zmm2, imm8	C	V/V	AVX512F	Extract 256 bits of packed double-precision floating-point values from zmm2 and store results in ymm1/m256 subject to writemask k1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	NA	ModRM:r/m (w)	ModRM:reg (r)	Imm8	NA
B	Tuple2	ModRM:r/m (w)	ModRM:reg (r)	Imm8	NA
C	Tuple4	ModRM:r/m (w)	ModRM:reg (r)	Imm8	NA
D	Tuple8	ModRM:r/m (w)	ModRM:reg (r)	Imm8	NA

Description

VEXTRACTF128/VEXTRACTF32x4 and VEXTRACTF64x2 extract 128-bits of single-precision floating-point values from the source operand (the second operand) and store to the low 128-bit of the destination operand (the first operand). The 128-bit data extraction occurs at an 128-bit granular offset specified by imm8[0] (256-bit) or imm8[1:0] as the multiply factor. The destination may be either a vector register or an 128-bit memory location.

VEXTRACTF32x4: The low 128-bit of the destination operand is updated at 32-bit granularity according to the writemask.

VEXTRACTF32x8 and VEXTRACTF64x4 extract 256-bits of double-precision floating-point values from the source operand (second operand) and store to the low 256-bit of the destination operand (the first operand). The 256-bit data extraction occurs at an 256-bit granular offset specified by imm8[0] (256-bit) or imm8[0] as the multiply factor The destination may be either a vector register or a 256-bit memory location.

VEXTRACTF64x4: The low 256-bit of the destination operand is updated at 64-bit granularity according to the writemask.

VEX.vvvv and EVEX.vvvv are reserved and must be 1111b otherwise instructions will #UD.

The high 6 bits of the immediate are ignored.

If VEXTRACTF128 is encoded with VEX.L= 0, an attempt to execute the instruction encoded with VEX.L= 0 will cause an #UD exception.

Operation

VEXTRACTF32x4 (EVEX encoded versions) when destination is a register

```
VL = 256, 512
IF VL = 256
    CASE (imm8[0]) OF
        0: TMP_DEST[127:0] ← SRC1[127:0]
        1: TMP_DEST[127:0] ← SRC1[255:128]
    ESAC.
FI;
IF VL = 512
    CASE (imm8[1:0]) OF
        00: TMP_DEST[127:0] ← SRC1[127:0]
        01: TMP_DEST[127:0] ← SRC1[255:128]
        10: TMP_DEST[127:0] ← SRC1[383:256]
        11: TMP_DEST[127:0] ← SRC1[511:384]
    ESAC.
FI;
FOR j ← 0 TO 3
    i ← j * 32
    IF k1[j] OR *no writemask*
        THEN DEST[i+31:i] ← TMP_DEST[i+31:i]
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE *zeroing-masking*              ; zeroing-masking
                    DEST[i+31:i] ← 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:128] ← 0
```

VEXTRACTF32x4 (EVEX encoded versions) when destination is memory

```
VL = 256, 512
IF VL = 256
    CASE (imm8[0]) OF
        0: TMP_DEST[127:0] ← SRC1[127:0]
        1: TMP_DEST[127:0] ← SRC1[255:128]
    ESAC.
FI;
IF VL = 512
    CASE (imm8[1:0]) OF
        00: TMP_DEST[127:0] ← SRC1[127:0]
        01: TMP_DEST[127:0] ← SRC1[255:128]
        10: TMP_DEST[127:0] ← SRC1[383:256]
        11: TMP_DEST[127:0] ← SRC1[511:384]
    ESAC.
FI;

FOR j ← 0 TO 3
    i ← j * 32
    IF k1[j] OR *no writemask*
        THEN DEST[i+31:i] ← TMP_DEST[i+31:i]
        ELSE *DEST[i+31:i] remains unchanged*       ; merging-masking
    FI;
```

```
ENDFOR
```

VEXTRACTF64x2 (EVEX encoded versions) when destination is a register

```
VL = 256, 512
IF VL = 256
    CASE (imm8[0]) OF
        0: TMP_DEST[127:0] ← SRC1[127:0]
        1: TMP_DEST[127:0] ← SRC1[255:128]
    ESAC.
FI;
IF VL = 512
    CASE (imm8[1:0]) OF
        00: TMP_DEST[127:0] ← SRC1[127:0]
        01: TMP_DEST[127:0] ← SRC1[255:128]
        10: TMP_DEST[127:0] ← SRC1[383:256]
        11: TMP_DEST[127:0] ← SRC1[511:384]
    ESAC.
FI;

FOR j ← 0 TO 1
    i ← j * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+63:i] ← TMP_DEST[i+63:i]
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE *zeroing-masking*              ; zeroing-masking
                    DEST[i+63:i] ← 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:128] ← 0
```

VEXTRACTF64x2 (EVEX encoded versions) when destination is memory

```
VL = 256, 512
IF VL = 256
    CASE (imm8[0]) OF
        0: TMP_DEST[127:0] ← SRC1[127:0]
        1: TMP_DEST[127:0] ← SRC1[255:128]
    ESAC.
FI;
IF VL = 512
    CASE (imm8[1:0]) OF
        00: TMP_DEST[127:0] ← SRC1[127:0]
        01: TMP_DEST[127:0] ← SRC1[255:128]
        10: TMP_DEST[127:0] ← SRC1[383:256]
        11: TMP_DEST[127:0] ← SRC1[511:384]
    ESAC.
FI;

FOR j ← 0 TO 1
    i ← j * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+63:i] ← TMP_DEST[i+63:i]
```

```
        ELSE *DEST[i+63:i] remains unchanged*           ; merging-masking
    FI;
ENDFOR
```

VEXTRACTF32x8 (EVEX.U1.512 encoded version) when destination is a register

```
VL = 512
CASE (imm8[0]) OF
    0: TMP_DEST[255:0] ← SRC1[255:0]
    1: TMP_DEST[255:0] ← SRC1[511:256]
ESAC.

FOR j ← 0 TO 7
    i ← j * 32
    IF k1[j] OR *no writemask*
        THEN DEST[i+31:i] ← TMP_DEST[i+31:i]
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE *zeroing-masking*              ; zeroing-masking
                    DEST[i+31:i] ← 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:256] ← 0
```

VEXTRACTF32x8 (EVEX.U1.512 encoded version) when destination is memory

```
CASE (imm8[0]) OF
    0: TMP_DEST[255:0] ← SRC1[255:0]
    1: TMP_DEST[255:0] ← SRC1[511:256]
ESAC.

FOR j ← 0 TO 7
    i ← j * 32
    IF k1[j] OR *no writemask*
        THEN DEST[i+31:i] ← TMP_DEST[i+31:i]
        ELSE *DEST[i+31:i] remains unchanged*           ; merging-masking
    FI;
ENDFOR
```

VEXTRACTF64x4 (EVEX.512 encoded version) when destination is a register

```
VL = 512
CASE (imm8[0]) OF
    0: TMP_DEST[255:0] ← SRC1[255:0]
    1: TMP_DEST[255:0] ← SRC1[511:256]
ESAC.

FOR j ← 0 TO 3
    i ← j * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+63:i] ← TMP_DEST[i+63:i]
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE *zeroing-masking*              ; zeroing-masking
```

```
                        DEST[i+63:i] ← 0
                FI
    FI;
ENDFOR
DEST[MAXVL-1:256] ← 0
```

VEXTRACTF64x4 (EVEX.512 encoded version) when destination is memory

```
CASE (imm8[0]) OF
    0: TMP_DEST[255:0] ← SRC1[255:0]
    1: TMP_DEST[255:0] ← SRC1[511:256]
ESAC.

FOR j ← 0 TO 3
    i ← j * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+63:i] ← TMP_DEST[i+63:i]
        ELSE      ; merging-masking
            *DEST[i+63:i] remains unchanged*
    FI;
ENDFOR
```

VEXTRACTF128 (memory destination form)

```
CASE (imm8[0]) OF
    0: DEST[127:0] ←SRC1[127:0]
    1: DEST[127:0] ←SRC1[255:128]
ESAC.
```

VEXTRACTF128 (register destination form)

```
CASE (imm8[0]) OF
    0: DEST[127:0] ←SRC1[127:0]
    1: DEST[127:0] ←SRC1[255:128]
ESAC.
DEST[MAXVL-1:128] ←0
```

Intel C/C++ Compiler Intrinsic Equivalent

VEXTRACTF32x4 __m128 _mm512_extractf32x4_ps(__m512 a, const int nidx);
VEXTRACTF32x4 __m128 _mm512_mask_extractf32x4_ps(__m128 s, __mmask8 k, __m512 a, const int nidx);
VEXTRACTF32x4 __m128 _mm512_maskz_extractf32x4_ps(__mmask8 k, __m512 a, const int nidx);
VEXTRACTF32x4 __m128 _mm256_extractf32x4_ps(__m256 a, const int nidx);
VEXTRACTF32x4 __m128 _mm256_mask_extractf32x4_ps(__m128 s, __mmask8 k, __m256 a, const int nidx);
VEXTRACTF32x4 __m128 _mm256_maskz_extractf32x4_ps(__mmask8 k, __m256 a, const int nidx);
VEXTRACTF32x8 __m256 _mm512_extractf32x8_ps(__m512 a, const int nidx);
VEXTRACTF32x8 __m256 _mm512_mask_extractf32x8_ps(__m256 s, __mmask8 k, __m512 a, const int nidx);
VEXTRACTF32x8 __m256 _mm512_maskz_extractf32x8_ps(__mmask8 k, __m512 a, const int nidx);
VEXTRACTF64x2 __m128d _mm512_extractf64x2_pd(__m512d a, const int nidx);
VEXTRACTF64x2 __m128d _mm512_mask_extractf64x2_pd(__m128d s, __mmask8 k, __m512d a, const int nidx);
VEXTRACTF64x2 __m128d _mm512_maskz_extractf64x2_pd(__mmask8 k, __m512d a, const int nidx);
VEXTRACTF64x2 __m128d _mm256_extractf64x2_pd(__m256d a, const int nidx);
VEXTRACTF64x2 __m128d _mm256_mask_extractf64x2_pd(__m128d s, __mmask8 k, __m256d a, const int nidx);
VEXTRACTF64x2 __m128d _mm256_maskz_extractf64x2_pd(__mmask8 k, __m256d a, const int nidx);
VEXTRACTF64x4 __m256d _mm512_extractf64x4_pd(__m512d a, const int nidx);
VEXTRACTF64x4 __m256d _mm512_mask_extractf64x4_pd(__m256d s, __mmask8 k, __m512d a, const int nidx);
VEXTRACTF64x4 __m256d _mm512_maskz_extractf64x4_pd(__mmask8 k, __m512d a, const int nidx);
VEXTRACTF128 __m128 _mm256_extractf128_ps (__m256 a, int offset);

VEXTRACTF128 __m128d _mm256_extractf128_pd (__m256d a, int offset);
VEXTRACTF128 __m128i_mm256_extractf128_si256(__m256i a, int offset);

SIMD Floating-Point Exceptions

None

Other Exceptions

VEX-encoded instructions, see Exceptions Type 6;

EVEX-encoded instructions, see Exceptions Type E6NF.

#UD	IF VEX.L = 0.
#UD	If VEX.vvvv != 1111B or EVEX.vvvv != 1111B.

VEXTRACTI128/VEXTRACTI32x4/VEXTRACTI64x2/VEXTRACTI32x8/VEXTRACTI64x4—Extract packed Integer Values

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
VEX.256.66.0F3A.W0 39 /r ib VEXTRACTI128 xmm1/m128, ymm2, imm8	A	V/V	AVX2	Extract 128 bits of integer data from ymm2 and store results in xmm1/m128.
EVEX.256.66.0F3A.W0 39 /r ib VEXTRACTI32X4 xmm1/m128 {k1}{z}, ymm2, imm8	C	V/V	AVX512VL AVX512F	Extract 128 bits of double-word integer values from ymm2 and store results in xmm1/m128 subject to writemask k1.
EVEX.512.66.0F3A.W0 39 /r ib VEXTRACTI32x4 xmm1/m128 {k1}{z}, zmm2, imm8	C	V/V	AVX512F	Extract 128 bits of double-word integer values from zmm2 and store results in xmm1/m128 subject to writemask k1.
EVEX.256.66.0F3A.W1 39 /r ib VEXTRACTI64X2 xmm1/m128 {k1}{z}, ymm2, imm8	B	V/V	AVX512VL AVX512DQ	Extract 128 bits of quad-word integer values from ymm2 and store results in xmm1/m128 subject to writemask k1.
EVEX.512.66.0F3A.W1 39 /r ib VEXTRACTI64X2 xmm1/m128 {k1}{z}, zmm2, imm8	B	V/V	AVX512DQ	Extract 128 bits of quad-word integer values from zmm2 and store results in xmm1/m128 subject to writemask k1.
EVEX.512.66.0F3A.W0 3B /r ib VEXTRACTI32X8 ymm1/m256 {k1}{z}, zmm2, imm8	D	V/V	AVX512DQ	Extract 256 bits of double-word integer values from zmm2 and store results in ymm1/m256 subject to writemask k1.
EVEX.512.66.0F3A.W1 3B /r ib VEXTRACTI64x4 ymm1/m256 {k1}{z}, zmm2, imm8	C	V/V	AVX512F	Extract 256 bits of quad-word integer values from zmm2 and store results in ymm1/m256 subject to writemask k1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	NA	ModRM:r/m (w)	ModRM:reg (r)	Imm8	NA
B	Tuple2	ModRM:r/m (w)	ModRM:reg (r)	Imm8	NA
C	Tuple4	ModRM:r/m (w)	ModRM:reg (r)	Imm8	NA
D	Tuple8	ModRM:r/m (w)	ModRM:reg (r)	Imm8	NA

Description

VEXTRACTI128/VEXTRACTI32x4 and VEXTRACTI64x2 extract 128-bits of doubleword integer values from the source operand (the second operand) and store to the low 128-bit of the destination operand (the first operand). The 128-bit data extraction occurs at an 128-bit granular offset specified by imm8[0] (256-bit) or imm8[1:0] as the multiply factor. The destination may be either a vector register or an 128-bit memory location.

VEXTRACTI32x4: The low 128-bit of the destination operand is updated at 32-bit granularity according to the writemask.

VEXTRACTI64x2: The low 128-bit of the destination operand is updated at 64-bit granularity according to the writemask.

VEXTRACTI32x8 and VEXTRACTI64x4 extract 256-bits of quadword integer values from the source operand (the second operand) and store to the low 256-bit of the destination operand (the first operand). The 256-bit data extraction occurs at an 256-bit granular offset specified by imm8[0] (256-bit) or imm8[0] as the multiply factor The destination may be either a vector register or a 256-bit memory location.

VEXTRACTI32x8: The low 256-bit of the destination operand is updated at 32-bit granularity according to the writemask.

VEXTRACTI64x4: The low 256-bit of the destination operand is updated at 64-bit granularity according to the writemask.

VEX.vvvv and EVEX.vvvv are reserved and must be 1111b otherwise instructions will #UD.

The high 7 bits (6 bits in EVEX.512) of the immediate are ignored.

If VEXTRACTI128 is encoded with VEX.L= 0, an attempt to execute the instruction encoded with VEX.L= 0 will cause an #UD exception.

Operation

VEXTRACTI32x4 (EVEX encoded versions) when destination is a register

```
VL = 256, 512
IF VL = 256
    CASE (imm8[0]) OF
        0: TMP_DEST[127:0] ← SRC1[127:0]
        1: TMP_DEST[127:0] ← SRC1[255:128]
    ESAC.
FI;
IF VL = 512
    CASE (imm8[1:0]) OF
        00: TMP_DEST[127:0] ← SRC1[127:0]
        01: TMP_DEST[127:0] ← SRC1[255:128]
        10: TMP_DEST[127:0] ← SRC1[383:256]
        11: TMP_DEST[127:0] ← SRC1[511:384]
    ESAC.
FI;
FOR j ← 0 TO 3
    i ← j * 32
    IF k1[j] OR *no writemask*
        THEN DEST[i+31:i] ← TMP_DEST[i+31:i]
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE *zeroing-masking*              ; zeroing-masking
                    DEST[i+31:i] ← 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:128] ← 0
```

VEXTRACTI32x4 (EVEX encoded versions) when destination is memory

```
VL = 256, 512
IF VL = 256
    CASE (imm8[0]) OF
        0: TMP_DEST[127:0] ← SRC1[127:0]
        1: TMP_DEST[127:0] ← SRC1[255:128]
    ESAC.
FI;
IF VL = 512
    CASE (imm8[1:0]) OF
        00: TMP_DEST[127:0] ← SRC1[127:0]
        01: TMP_DEST[127:0] ← SRC1[255:128]
        10: TMP_DEST[127:0] ← SRC1[383:256]
        11: TMP_DEST[127:0] ← SRC1[511:384]
    ESAC.
```

```
FI;

FOR j ← 0 TO 3
    i ← j * 32
    IF k1[j] OR *no writemask*
        THEN DEST[i+31:i] ← TMP_DEST[i+31:i]
        ELSE *DEST[i+31:i] remains unchanged*          ; merging-masking
    FI;
ENDFOR
```

VEXTRACTI64x2 (EVEX encoded versions) when destination is a register

```
VL = 256, 512
IF VL = 256
    CASE (imm8[0]) OF
        0: TMP_DEST[127:0] ← SRC1[127:0]
        1: TMP_DEST[127:0] ← SRC1[255:128]
    ESAC.
FI;
IF VL = 512
    CASE (imm8[1:0]) OF
        00: TMP_DEST[127:0] ← SRC1[127:0]
        01: TMP_DEST[127:0] ← SRC1[255:128]
        10: TMP_DEST[127:0] ← SRC1[383:256]
        11: TMP_DEST[127:0] ← SRC1[511:384]
    ESAC.
FI;

FOR j ← 0 TO 1
    i ← j * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+63:i] ← TMP_DEST[i+63:i]
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE *zeroing-masking*              ; zeroing-masking
                    DEST[i+63:i] ← 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:128] ← 0
```

VEXTRACTI64x2 (EVEX encoded versions) when destination is memory

```
VL = 256, 512
IF VL = 256
    CASE (imm8[0]) OF
        0: TMP_DEST[127:0] ← SRC1[127:0]
        1: TMP_DEST[127:0] ← SRC1[255:128]
    ESAC.
FI;
IF VL = 512
    CASE (imm8[1:0]) OF
        00: TMP_DEST[127:0] ← SRC1[127:0]
        01: TMP_DEST[127:0] ← SRC1[255:128]
        10: TMP_DEST[127:0] ← SRC1[383:256]
        11: TMP_DEST[127:0] ← SRC1[511:384]
    ESAC.
FI;

FOR j ← 0 TO 1
    i ← j * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+63:i] ← TMP_DEST[i+63:i]
        ELSE *DEST[i+63:i] remains unchanged*          ; merging-masking
    FI;
ENDFOR
```

VEXTRACTI32x8 (EVEX.U1.512 encoded version) when destination is a register

```
VL = 512
CASE (imm8[0]) OF
    0: TMP_DEST[255:0] ← SRC1[255:0]
    1: TMP_DEST[255:0] ← SRC1[511:256]
ESAC.

FOR j ← 0 TO 7
    i ← j * 32
    IF k1[j] OR *no writemask*
        THEN DEST[i+31:i] ← TMP_DEST[i+31:i]
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE *zeroing-masking*              ; zeroing-masking
                    DEST[i+31:i] ← 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:256] ← 0
```

VEXTRACTI32x8 (EVEX.U1.512 encoded version) when destination is memory

```
CASE (imm8[0]) OF
    0: TMP_DEST[255:0] ← SRC1[255:0]
    1: TMP_DEST[255:0] ← SRC1[511:256]
ESAC.

FOR j ← 0 TO 7
    i ← j * 32
    IF k1[j] OR *no writemask*
        THEN DEST[i+31:i] ← TMP_DEST[i+31:i]
        ELSE *DEST[i+31:i] remains unchanged*          ; merging-masking
    FI;
ENDFOR
```

VEXTRACTI64x4 (EVEX.512 encoded version) when destination is a register

```
VL = 512
CASE (imm8[0]) OF
    0: TMP_DEST[255:0] ← SRC1[255:0]
    1: TMP_DEST[255:0] ← SRC1[511:256]
ESAC.

FOR j ← 0 TO 3
    i ← j * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+63:i] ← TMP_DEST[i+63:i]
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE *zeroing-masking*              ; zeroing-masking
                    DEST[i+63:i] ← 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:256] ← 0
```

VEXTRACTI64x4 (EVEX.512 encoded version) when destination is memory

```
CASE (imm8[0]) OF
    0: TMP_DEST[255:0] ← SRC1[255:0]
    1: TMP_DEST[255:0] ← SRC1[511:256]
ESAC.
FOR j ← 0 TO 3
    i ← j * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+63:i] ← TMP_DEST[i+63:i]
        ELSE *DEST[i+63:i] remains unchanged*          ; merging-masking
    FI;
ENDFOR
```

VEXTRACTI128 (memory destination form)

```
CASE (imm8[0]) OF
    0: DEST[127:0] ←SRC1[127:0]
    1: DEST[127:0] ←SRC1[255:128]
ESAC.
```

VEXTRACTI128 (register destination form)

```
CASE (imm8[0]) OF
    0: DEST[127:0] ←SRC1[127:0]
    1: DEST[127:0] ←SRC1[255:128]
ESAC.
DEST[MAXVL-1:128] ←0
```

Intel C/C++ Compiler Intrinsic Equivalent

VEXTRACTI32x4 __m128i _mm512_extracti32x4_epi32(__m512i a, const int nidx);
VEXTRACTI32x4 __m128i _mm512_mask_extracti32x4_epi32(__m128i s, __mmask8 k, __m512i a, const int nidx);
VEXTRACTI32x4 __m128i _mm512_maskz_extracti32x4_epi32(__mmask8 k, __m512i a, const int nidx);
VEXTRACTI32x4 __m128i _mm256_extracti32x4_epi32(__m256i a, const int nidx);
VEXTRACTI32x4 __m128i _mm256_mask_extracti32x4_epi32(__m128i s, __mmask8 k, __m256i a, const int nidx);
VEXTRACTI32x4 __m128i _mm256_maskz_extracti32x4_epi32(__mmask8 k, __m256i a, const int nidx);
VEXTRACTI32x8 __m256i _mm512_extracti32x8_epi32(__m512i a, const int nidx);
VEXTRACTI32x8 __m256i _mm512_mask_extracti32x8_epi32(__m256i s, __mmask8 k, __m512i a, const int nidx);
VEXTRACTI32x8 __m256i _mm512_maskz_extracti32x8_epi32(__mmask8 k, __m512i a, const int nidx);
VEXTRACTI64x2 __m128i _mm512_extracti64x2_epi64(__m512i a, const int nidx);
VEXTRACTI64x2 __m128i _mm512_mask_extracti64x2_epi64(__m128i s, __mmask8 k, __m512i a, const int nidx);
VEXTRACTI64x2 __m128i _mm512_maskz_extracti64x2_epi64(__mmask8 k, __m512i a, const int nidx);
VEXTRACTI64x2 __m128i _mm256_extracti64x2_epi64(__m256i a, const int nidx);
VEXTRACTI64x2 __m128i _mm256_mask_extracti64x2_epi64(__m128i s, __mmask8 k, __m256i a, const int nidx);
VEXTRACTI64x2 __m128i _mm256_maskz_extracti64x2_epi64(__mmask8 k, __m256i a, const int nidx);
VEXTRACTI64x4 __m256i _mm512_extracti64x4_epi64(__m512i a, const int nidx);
VEXTRACTI64x4 __m256i _mm512_mask_extracti64x4_epi64(__m256i s, __mmask8 k, __m512i a, const int nidx);
VEXTRACTI64x4 __m256i _mm512_maskz_extracti64x4_epi64(__mmask8 k, __m512i a, const int nidx);
VEXTRACTI128 __m128i _mm256_extracti128_si256(__m256i a, int offset);

SIMD Floating-Point Exceptions

None

Other Exceptions

VEX-encoded instructions, see Exceptions Type 6;
EVEX-encoded instructions, see Exceptions Type E6NF.

#UD	IF VEX.L = 0.
#UD	If VEX.vvvv != 1111B or EVEX.vvvv != 1111B.

VFIXUPIMMPD—Fix Up Special Packed Float64 Values

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.NDS.128.66.0F3A.W1 54 /r ib VFIXUPIMMPD xmm1 {k1}{z}, xmm2, xmm3/m128/m64bcst, imm8	A	V/V	AVX512VL AVX512F	Fix up special numbers in float64 vector xmm1, float64 vector xmm2 and int64 vector xmm3/m128/m64bcst and store the result in xmm1, under writemask.
EVEX.NDS.256.66.0F3A.W1 54 /r ib VFIXUPIMMPD ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst, imm8	A	V/V	AVX512VL AVX512F	Fix up special numbers in float64 vector ymm1, float64 vector ymm2 and int64 vector ymm3/m256/m64bcst and store the result in ymm1, under writemask.
EVEX.NDS.512.66.0F3A.W1 54 /r ib VFIXUPIMMPD zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst{sae}, imm8	A	V/V	AVX512F	Fix up elements of float64 vector in zmm2 using int64 vector table in zmm3/m512/m64bcst, combine with preserved elements from zmm1, and store the result in zmm1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Full Vector	ModRM:reg (r, w)	EVEX.vvvv	ModRM:r/m (r)	Imm8

Description

Perform fix-up of quad-word elements encoded in double-precision floating-point format in the first source operand (the second operand) using a 32-bit, two-level look-up table specified in the corresponding quadword element of the second source operand (the third operand) with exception reporting specifier imm8. The elements that are fixed-up are selected by mask bits of 1 specified in the opmask k1. Mask bits of 0 in the opmask k1 or table response action of 0000b preserves the corresponding element of the first operand. The fixed-up elements from the first source operand and the preserved element in the first operand are combined as the final results in the destination operand (the first operand).

The destination and the first source operands are ZMM/YMM/XMM registers. The second source operand can be a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 64-bit memory location.

The two-level look-up table perform a fix-up of each DP FP input data in the first source operand by decoding the input data encoding into 8 token types. A response table is defined for each token type that converts the input encoding in the first source operand with one of 16 response actions.

This instruction is specifically intended for use in fixing up the results of arithmetic calculations involving one source so that they match the spec, although it is generally useful for fixing up the results of multiple-instruction sequences to reflect special-number inputs. For example, consider rcp(0). Input 0 to rcp, and you should get INF according to the DX10 spec. However, evaluating rcp via Newton-Raphson, where x=approx(1/0), yields an incorrect result. To deal with this, VFIXUPIMMPD can be used after the N-R reciprocal sequence to set the result to the correct value (i.e. INF when the input is 0).

If MXCSR.DAZ is not set, denormal input elements in the first source operand are considered as normal inputs and do not trigger any fixup nor fault reporting.

Imm8 is used to set the required flags reporting. It supports #ZE and #IE fault reporting (see details below).

MXCSR mask bits are ignored and are treated as if all mask bits are set to masked response). If any of the imm8 bits is set and the condition met for fault reporting, MXCSR.IE or MXCSR.ZE might be updated.

This instruction is writemasked, so only those elements with the corresponding bit set in vector mask register k1 are computed and stored into zmm1. Elements in the destination with the corresponding bit clear in k1 retain their previous values or are set to 0.

Operation

```
enum TOKEN_TYPE
{
    QNAN_TOKEN ← 0,
    SNAN_TOKEN ← 1,
    ZERO_VALUE_TOKEN ← 2,
    POS_ONE_VALUE_TOKEN ← 3,
    NEG_INF_TOKEN ← 4,
    POS_INF_TOKEN ← 5,
    NEG_VALUE_TOKEN ← 6,
    POS_VALUE_TOKEN ← 7
}

FIXUPIMM_DP (dest[63:0], src1[63:0],tbl3[63:0], imm8 [7:0]){
    tsrc[63:0] ← ((src1[62:52] = 0) AND (MXCSR.DAZ =1)) ? 0.0 : src1[63:0]
    CASE(tsrc[63:0] of TOKEN_TYPE) {
        QNAN_TOKEN: j ← 0;
        SNAN_TOKEN: j ← 1;
        ZERO_VALUE_TOKEN: j ← 2;
        POS_ONE_VALUE_TOKEN: j ← 3;
        NEG_INF_TOKEN: j ← 4;
        POS_INF_TOKEN: j ← 5;
        NEG_VALUE_TOKEN: j ← 6;
        POS_VALUE_TOKEN: j ← 7;
    }    ; end source special CASE(tsrc...)

    ; The required response from src3 table is extracted
    token_response[3:0] = tbl3[3+4*j:4*j];

    CASE(token_response[3:0]) {
        0000: dest[63:0] ← dest[63:0];          ; preserve content of DEST
        0001: dest[63:0] ← tsrc[63:0];          ; pass through src1 normal input value, denormal as zero
        0010: dest[63:0] ← QNaN(tsrc[63:0]);
        0011: dest[63:0] ← QNAN_Indefinite;
        0100: dest[63:0] ← -INF;
        0101: dest[63:0] ← +INF;
        0110: dest[63:0] ← tsrc.sign? –INF : +INF;
        0111: dest[63:0] ← -0;
        1000: dest[63:0] ← +0;
        1001: dest[63:0] ← -1;
        1010: dest[63:0] ← +1;
        1011: dest[63:0] ← ½;
        1100: dest[63:0] ← 90.0;
        1101: dest[63:0] ← PI/2;
        1110: dest[63:0] ← MAX_FLOAT;
        1111: dest[63:0] ← -MAX_FLOAT;
    }        ; end of token_response CASE
```

```
    ; The required fault reporting from imm8 is extracted
    ; TOKENs are mutually exclusive and TOKENs priority defines the order.
    ; Multiple faults related to a single token can occur simultaneously.
    IF (tsrc[63:0] of TOKEN_TYPE: ZERO_VALUE_TOKEN) AND imm8[0] then set #ZE;
    IF (tsrc[63:0] of TOKEN_TYPE: ZERO_VALUE_TOKEN) AND imm8[1] then set #IE;
    IF (tsrc[63:0] of TOKEN_TYPE: ONE_VALUE_TOKEN) AND imm8[2] then set #ZE;
    IF (tsrc[63:0] of TOKEN_TYPE: ONE_VALUE_TOKEN) AND imm8[3] then set #IE;
    IF (tsrc[63:0] of TOKEN_TYPE: SNAN_TOKEN) AND imm8[4] then set #IE;
    IF (tsrc[63:0] of TOKEN_TYPE: NEG_INF_TOKEN) AND imm8[5] then set #IE;
    IF (tsrc[63:0] of TOKEN_TYPE: NEG_VALUE_TOKEN) AND imm8[6] then set #IE;
    IF (tsrc[63:0] of TOKEN_TYPE: POS_INF_TOKEN) AND imm8[7] then set #IE;
        ; end fault reporting
    return dest[63:0];
}       ; end of FIXUPIMM_DP()
```

VFIXUPIMMPD

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j ← 0 TO KL-1
    i ← j * 64
    IF k1[j] OR *no writemask*
        THEN
            IF (EVEX.b = 1) AND (SRC2 *is memory*)
                THEN
                    DEST[i+63:i] ← FIXUPIMM_DP(DEST[i+63:i], SRC1[i+63:i], SRC2[63:0], imm8 [7:0])
                ELSE
                    DEST[i+63:i] ← FIXUPIMM_DP(DEST[i+63:i], SRC1[i+63:i], SRC2[i+63:i], imm8 [7:0])
            FI;
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE  DEST[i+63:i] ← 0          ; zeroing-masking
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0
```

Immediate Control Description:

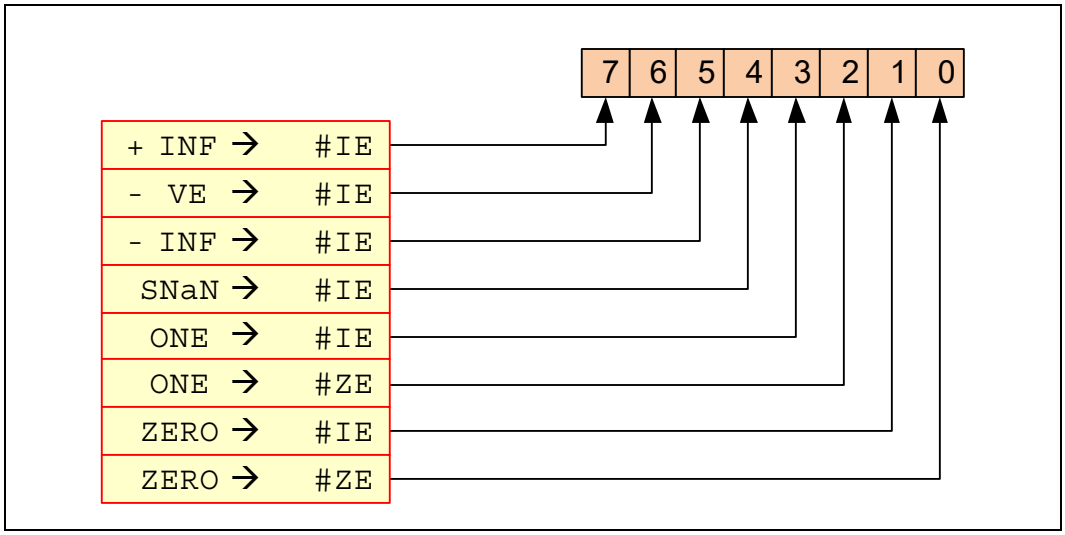


Figure 5-9. VFIXUPIMMPD Immediate Control Description

Intel C/C++ Compiler Intrinsic Equivalent

VFIXUPIMMPD __m512d _mm512_fixupimm_pd(__m512d a, __m512i tbl, int imm);
VFIXUPIMMPD __m512d _mm512_mask_fixupimm_pd(__m512d s, __mmask8 k, __m512d a, __m512i tbl, int imm);
VFIXUPIMMPD __m512d _mm512_maskz_fixupimm_pd(__mmask8 k, __m512d a, __m512i tbl, int imm);
VFIXUPIMMPD __m512d _mm512_fixupimm_round_pd(__m512d a, __m512i tbl, int imm, int sae);
VFIXUPIMMPD __m512d _mm512_mask_fixupimm_round_pd(__m512d s, __mmask8 k, __m512d a, __m512i tbl, int imm, int sae);
VFIXUPIMMPD __m512d _mm512_maskz_fixupimm_round_pd(__mmask8 k, __m512d a, __m512i tbl, int imm, int sae);
VFIXUPIMMPD __m256d _mm256_fixupimm_pd(__m256d a, __m256i tbl, int imm);
VFIXUPIMMPD __m256d _mm256_mask_fixupimm_pd(__m256d s, __mmask8 k, __m256d a, __m256i tbl, int imm);
VFIXUPIMMPD __m256d _mm256_maskz_fixupimm_pd(__mmask8 k, __m256d a, __m256i tbl, int imm);
VFIXUPIMMPD __m128d _mm_fixupimm_pd(__m128d a, __m128i tbl, int imm);
VFIXUPIMMPD __m128d _mm_mask_fixupimm_pd(__m128d s, __mmask8 k, __m128d a, __m128i tbl, int imm);
VFIXUPIMMPD __m128d _mm_maskz_fixupimm_pd(__mmask8 k, __m128d a, __m128i tbl, int imm);

SIMD Floating-Point Exceptions

Zero, Invalid

Other Exceptions

See Exceptions Type E2.

VFIXUPIMMPS—Fix Up Special Packed Float32 Values

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.NDS.128.66.0F3A.W0 54 /r VFIXUPIMMPS xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst, imm8	A	V/V	AVX512VL AVX512F	Fix up special numbers in float32 vector xmm1, float32 vector xmm2 and int32 vector xmm3/m128/m32bcst and store the result in xmm1, under writemask.
EVEX.NDS.256.66.0F3A.W0 54 /r VFIXUPIMMPS ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst, imm8	A	V/V	AVX512VL AVX512F	Fix up special numbers in float32 vector ymm1, float32 vector ymm2 and int32 vector ymm3/m256/m32bcst and store the result in ymm1, under writemask.
EVEX.NDS.512.66.0F3A.W0 54 /r ib VFIXUPIMMPS zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst{sae}, imm8	A	V/V	AVX512F	Fix up elements of float32 vector in zmm2 using int32 vector table in zmm3/m512/m32bcst, combine with preserved elements from zmm1, and store the result in zmm1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Full Vector	ModRM:reg (r, w)	EVEX.vvvv	ModRM:r/m (r)	Imm8

Description

Perform fix-up of doubleword elements encoded in single-precision floating-point format in the first source operand (the second operand) using a 32-bit, two-level look-up table specified in the corresponding doubleword element of the second source operand (the third operand) with exception reporting specifier imm8. The elements that are fixed-up are selected by mask bits of 1 specified in the opmask k1. Mask bits of 0 in the opmask k1 or table response action of 0000b preserves the corresponding element of the first operand. The fixed-up elements from the first source operand and the preserved element in the first operand are combined as the final results in the destination operand (the first operand).

The destination and the first source operands are ZMM/YMM/XMM registers. The second source operand can be a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 64-bit memory location.

The two-level look-up table perform a fix-up of each SP FP input data in the first source operand by decoding the input data encoding into 8 token types. A response table is defined for each token type that converts the input encoding in the first source operand with one of 16 response actions.

This instruction is specifically intended for use in fixing up the results of arithmetic calculations involving one source so that they match the spec, although it is generally useful for fixing up the results of multiple-instruction sequences to reflect special-number inputs. For example, consider rcp(0). Input 0 to rcp, and you should get INF according to the DX10 spec. However, evaluating rcp via Newton-Raphson, where x=approx(1/0), yields an incorrect result. To deal with this, VFIXUPIMMPS can be used after the N-R reciprocal sequence to set the result to the correct value (i.e. INF when the input is 0).

If MXCSR.DAZ is not set, denormal input elements in the first source operand are considered as normal inputs and do not trigger any fixup nor fault reporting.

Imm8 is used to set the required flags reporting. It supports #ZE and #IE fault reporting (see details below).

MXCSR.DAZ is used and refer to zmm2 only (i.e. zmm1 is not considered as zero in case MXCSR.DAZ is set).

MXCSR mask bits are ignored and are treated as if all mask bits are set to masked response). If any of the imm8 bits is set and the condition met for fault reporting, MXCSR.IE or MXCSR.ZE might be updated.

Operation

```
enum TOKEN_TYPE
{
    QNAN_TOKEN ← 0,
    SNAN_TOKEN ← 1,
    ZERO_VALUE_TOKEN ← 2,
    POS_ONE_VALUE_TOKEN ← 3,
    NEG_INF_TOKEN ← 4,
    POS_INF_TOKEN ← 5,
    NEG_VALUE_TOKEN ← 6,
    POS_VALUE_TOKEN ← 7
}

FIXUPIMM_SP ( dest[31:0], src1[31:0],tbl3[31:0], imm8 [7:0]){
    tsrc[31:0] ← ((src1[30:23] = 0) AND (MXCSR.DAZ =1)) ? 0.0 : src1[31:0]
    CASE(tsrc[31:0] of TOKEN_TYPE) {
        QNAN_TOKEN: j ← 0;
        SNAN_TOKEN: j ← 1;
        ZERO_VALUE_TOKEN: j ← 2;
        POS_ONE_VALUE_TOKEN: j ← 3;
        NEG_INF_TOKEN: j ← 4;
        POS_INF_TOKEN: j ← 5;
        NEG_VALUE_TOKEN: j ← 6;
        POS_VALUE_TOKEN: j ← 7;
    }           ; end source special CASE(tsrc...)

    ; The required response from src3 table is extracted
    token_response[3:0] = tbl3[3+4*j:4*j];

    CASE(token_response[3:0]) {
        0000: dest[31:0] ← dest[31:0];        ; preserve content of DEST
        0001: dest[31:0] ← tsrc[31:0];        ; pass through src1 normal input value, denormal as zero
        0010: dest[31:0] ← QNaN(tsrc[31:0]);
        0011: dest[31:0] ← QNAN_Indefinite;
        0100: dest[31:0] ← -INF;
        0101: dest[31:0] ← +INF;
        0110: dest[31:0] ← tsrc.sign? –INF : +INF;
        0111: dest[31:0] ← -0;
        1000: dest[31:0] ← +0;
        1001: dest[31:0] ← -1;
        1010: dest[31:0] ← +1;
        1011:  dest[31:0] ← ½;
        1100: dest[31:0] ← 90.0;
        1101: dest[31:0] ← PI/2;
        1110: dest[31:0] ← MAX_FLOAT;
        1111: dest[31:0] ← -MAX_FLOAT;
    }           ; end of token_response CASE
```

```
    ; The required fault reporting from imm8 is extracted
    ; TOKENs are mutually exclusive and TOKENs priority defines the order.
    ; Multiple faults related to a single token can occur simultaneously.
    IF (tsrc[31:0] of TOKEN_TYPE: ZERO_VALUE_TOKEN) AND imm8[0] then set #ZE;
    IF (tsrc[31:0] of TOKEN_TYPE: ZERO_VALUE_TOKEN) AND imm8[1] then set #IE;
    IF (tsrc[31:0] of TOKEN_TYPE: ONE_VALUE_TOKEN) AND imm8[2] then set #ZE;
    IF (tsrc[31:0] of TOKEN_TYPE: ONE_VALUE_TOKEN) AND imm8[3] then set #IE;
    IF (tsrc[31:0] of TOKEN_TYPE: SNAN_TOKEN) AND imm8[4] then set #IE;
    IF (tsrc[31:0] of TOKEN_TYPE: NEG_INF_TOKEN) AND imm8[5] then set #IE;
    IF (tsrc[31:0] of TOKEN_TYPE: NEG_VALUE_TOKEN) AND imm8[6] then set #IE;
    IF (tsrc[31:0] of TOKEN_TYPE: POS_INF_TOKEN) AND imm8[7] then set #IE;
        ; end fault reporting
    return dest[31:0];
}       ; end of FIXUPIMM_SP()
```

VFIXUPIMMPS (EVEX)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)

FOR j ← 0 TO KL-1
    i ← j * 32
    IF k1[j] OR *no writemask*
        THEN
            IF (EVEX.b = 1) AND (SRC2 *is memory*)
                THEN
                    DEST[i+31:i] ← FIXUPIMM_SP(DEST[i+31:i], SRC1[i+31:i], SRC2[31:0], imm8 [7:0])
                ELSE
                    DEST[i+31:i] ← FIXUPIMM_SP(DEST[i+31:i], SRC1[i+31:i], SRC2[i+31:i], imm8 [7:0])
            FI;
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE  DEST[i+31:i] ← 0              ; zeroing-masking
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0
```

Immediate Control Description:

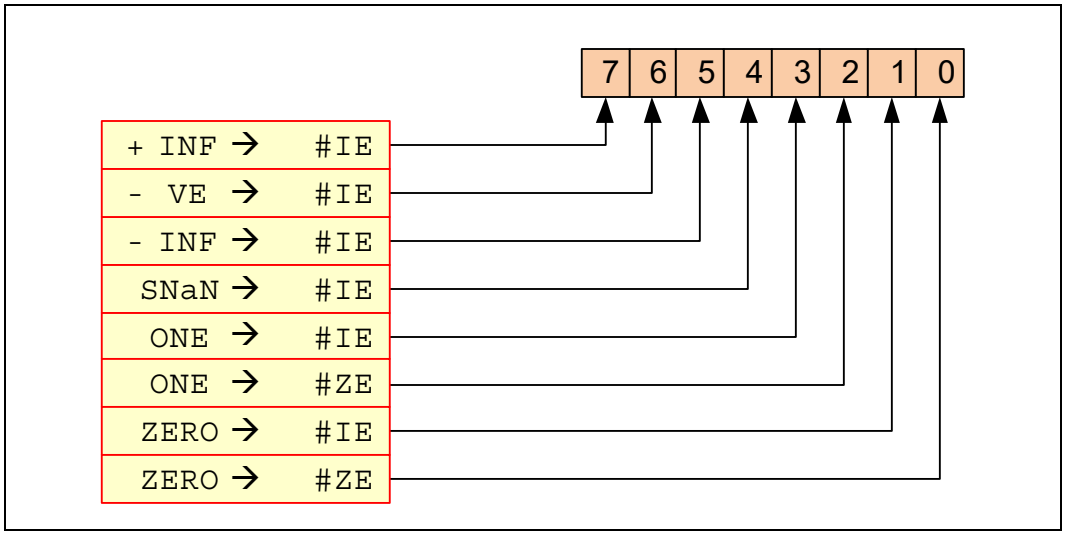


Figure 5-10. VFIXUPIMMPS Immediate Control Description

Intel C/C++ Compiler Intrinsic Equivalent

VFIXUPIMMPS __m512 _mm512_fixupimm_ps(__m512 a, __m512i tbl, int imm);
VFIXUPIMMPS __m512 _mm512_mask_fixupimm_ps(__m512 s, __mmask16 k, __m512 a, __m512i tbl, int imm);
VFIXUPIMMPS __m512 _mm512_maskz_fixupimm_ps(__mmask16 k, __m512 a, __m512i tbl, int imm);
VFIXUPIMMPS __m512 _mm512_fixupimm_round_ps(__m512 a, __m512i tbl, int imm, int sae);
VFIXUPIMMPS __m512 _mm512_mask_fixupimm_round_ps(__m512 s, __mmask16 k, __m512 a, __m512i tbl, int imm, int sae);
VFIXUPIMMPS __m512 _mm512_maskz_fixupimm_round_ps(__mmask16 k, __m512 a, __m512i tbl, int imm, int sae);
VFIXUPIMMPS __m256 _mm256_fixupimm_ps(__m256 a, __m256i tbl, int imm);
VFIXUPIMMPS __m256 _mm256_mask_fixupimm_ps(__m256 s, __mmask8 k, __m256 a, __m256i tbl, int imm);
VFIXUPIMMPS __m256 _mm256_maskz_fixupimm_ps(__mmask8 k, __m256 a, __m256i tbl, int imm);
VFIXUPIMMPS __m128 _mm_fixupimm_ps(__m128 a, __m128i tbl, int imm);
VFIXUPIMMPS __m128 _mm_mask_fixupimm_ps(__m128 s, __mmask8 k, __m128 a, __m128i tbl, int imm);
VFIXUPIMMPS __m128 _mm_maskz_fixupimm_ps(__mmask8 k, __m128 a, __m128i tbl, int imm);

SIMD Floating-Point Exceptions

Zero, Invalid

Other Exceptions

See Exceptions Type E2.

VFIXUPIMMSD—Fix Up Special Scalar Float64 Value

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.NDS.LIG.66.0F3A.W1 55 /r ib VFIXUPIMMSD xmm1 {k1}{z}, xmm2, xmm3/m64{sae}, imm8	A	V/V	AVX512F	Fix up a float64 number in the low quadword element of xmm2 using scalar int32 table in xmm3/m64 and store the result in xmm1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Tuple1 Scalar	ModRM:reg (r, w)	EVEX.vvvv	ModRM:r/m (r)	Imm8

Description

Perform a fix-up of the low quadword element encoded in double-precision floating-point format in the first source operand (the second operand) using a 32-bit, two-level look-up table specified in the low quadword element of the second source operand (the third operand) with exception reporting specifier imm8. The element that is fixed-up is selected by mask bit of 1 specified in the opmask k1. Mask bit of 0 in the opmask k1 or table response action of 0000b preserves the corresponding element of the first operand. The fixed-up element from the first source operand or the preserved element in the first operand becomes the low quadword element of the destination operand (the first operand). Bits 127:64 of the destination operand is copied from the corresponding bits of the first source operand. The destination and first source operands are XMM registers. The second source operand can be a XMM register or a 64- bit memory location.

The two-level look-up table perform a fix-up of each DP FP input data in the first source operand by decoding the input data encoding into 8 token types. A response table is defined for each token type that converts the input encoding in the first source operand with one of 16 response actions.

This instruction is specifically intended for use in fixing up the results of arithmetic calculations involving one source so that they match the spec, although it is generally useful for fixing up the results of multiple-instruction sequences to reflect special-number inputs. For example, consider rcp(0). Input 0 to rcp, and you should get INF according to the DX10 spec. However, evaluating rcp via Newton-Raphson, where x=approx(1/0), yields an incorrect result. To deal with this, VFIXUPIMMPD can be used after the N-R reciprocal sequence to set the result to the correct value (i.e. INF when the input is 0).

If MXCSR.DAZ is not set, denormal input elements in the first source operand are considered as normal inputs and do not trigger any fixup nor fault reporting.

Imm8 is used to set the required flags reporting. It supports #ZE and #IE fault reporting (see details below).

MXCSR.DAZ is used and refer to zmm2 only (i.e. zmm1 is not considered as zero in case MXCSR.DAZ is set).

MXCSR mask bits are ignored and are treated as if all mask bits are set to masked response). If any of the imm8 bits is set and the condition met for fault reporting, MXCSR.IE or MXCSR.ZE might be updated.

Operation

```
enum TOKEN_TYPE
{
    QNAN_TOKEN ← 0,
    SNAN_TOKEN ← 1,
    ZERO_VALUE_TOKEN ← 2,
    POS_ONE_VALUE_TOKEN ← 3,
    NEG_INF_TOKEN ← 4,
    POS_INF_TOKEN ← 5,
    NEG_VALUE_TOKEN ← 6,
    POS_VALUE_TOKEN ← 7
}
```

```
FIXUPIMM_DP (dest[63:0], src1[63:0],tbl3[63:0], imm8 [7:0]){
    tsrc[63:0] ← ((src1[62:52] = 0) AND (MXCSR.DAZ =1)) ? 0.0 : src1[63:0]
    CASE(tsrc[63:0] of TOKEN_TYPE) {
        QNAN_TOKEN: j ← 0;
        SNAN_TOKEN: j ← 1;
        ZERO_VALUE_TOKEN: j ← 2;
        POS_ONE_VALUE_TOKEN: j ← 3;
        NEG_INF_TOKEN: j ← 4;
        POS_INF_TOKEN: j ← 5;
        NEG_VALUE_TOKEN: j ← 6;
        POS_VALUE_TOKEN: j ← 7;
    }           ; end source special CASE(tsrc...)

    ; The required response from src3 table is extracted
    token_response[3:0] = tbl3[3+4*j:4*j];

    CASE(token_response[3:0]) {
        0000: dest[63:0] ← dest[63:0]          ; preserve content of DEST
        0001: dest[63:0] ← tsrc[63:0];         ; pass through src1 normal input value, denormal as zero
        0010: dest[63:0] ← QNaN(tsrc[63:0]);
        0011: dest[63:0] ← QNAN_Indefinite;
        0100:dest[63:0] ← -INF;
        0101: dest[63:0] ← +INF;
        0110: dest[63:0] ← tsrc.sign? –INF : +INF;
        0111: dest[63:0] ← -0;
        1000: dest[63:0] ← +0;
        1001: dest[63:0] ← -1;
        1010: dest[63:0] ← +1;
        1011: dest[63:0] ← ½;
        1100: dest[63:0] ← 90.0;
        1101: dest[63:0] ← PI/2;
        1110: dest[63:0] ← MAX_FLOAT;
        1111: dest[63:0] ← -MAX_FLOAT;
    }           ; end of token_response CASE

    ; The required fault reporting from imm8 is extracted
    ; TOKENs are mutually exclusive and TOKENs priority defines the order.
    ; Multiple faults related to a single token can occur simultaneously.
    IF (tsrc[63:0] of TOKEN_TYPE: ZERO_VALUE_TOKEN) AND imm8[0] then set #ZE;
    IF (tsrc[63:0] of TOKEN_TYPE: ZERO_VALUE_TOKEN) AND imm8[1] then set #IE;
    IF (tsrc[63:0] of TOKEN_TYPE: ONE_VALUE_TOKEN) AND imm8[2] then set #ZE;
    IF (tsrc[63:0] of TOKEN_TYPE: ONE_VALUE_TOKEN) AND imm8[3] then set #IE;
    IF (tsrc[63:0] of TOKEN_TYPE: SNAN_TOKEN) AND imm8[4] then set #IE;
    IF (tsrc[63:0] of TOKEN_TYPE: NEG_INF_TOKEN) AND imm8[5] then set #IE;
    IF (tsrc[63:0] of TOKEN_TYPE: NEG_VALUE_TOKEN) AND imm8[6] then set #IE;
    IF (tsrc[63:0] of TOKEN_TYPE: POS_INF_TOKEN) AND imm8[7] then set #IE;
        ; end fault reporting
    return dest[63:0];
}       ; end of FIXUPIMM_DP()
```

VFIXUPIMMSD (EVEX encoded version)

```
IF k1[0] OR *no writemask*
    THEN DEST[63:0] ← FIXUPIMM_DP(DEST[63:0], SRC1[63:0], SRC2[63:0], imm8 [7:0])
    ELSE
        IF *merging-masking*                    ; merging-masking
            THEN *DEST[63:0] remains unchanged*
            ELSE  DEST[63:0] ← 0                ; zeroing-masking
        FI
FI;
DEST[127:64] ← SRC1[127:64]
DEST[MAXVL-1:128] ← 0
```

Immediate Control Description:

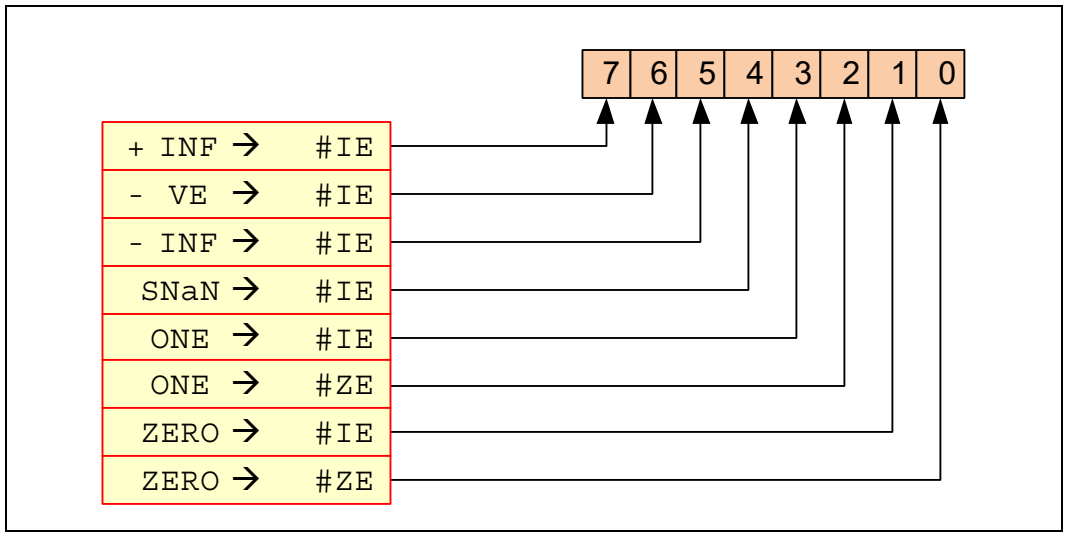


Figure 5-11. VFIXUPIMMSD Immediate Control Description

Intel C/C++ Compiler Intrinsic Equivalent

VFIXUPIMMSD __m128d _mm_fixupimm_sd(__m128d a, __m128i tbl, int imm);
VFIXUPIMMSD __m128d _mm_mask_fixupimm_sd(__m128d s, __mmask8 k, __m128d a, __m128i tbl, int imm);
VFIXUPIMMSD __m128d _mm_maskz_fixupimm_sd(__mmask8 k, __m128d a, __m128i tbl, int imm);
VFIXUPIMMSD __m128d _mm_fixupimm_round_sd(__m128d a, __m128i tbl, int imm, int sae);
VFIXUPIMMSD __m128d _mm_mask_fixupimm_round_sd(__m128d s, __mmask8 k, __m128d a, __m128i tbl, int imm, int sae);
VFIXUPIMMSD __m128d _mm_maskz_fixupimm_round_sd(__mmask8 k, __m128d a, __m128i tbl, int imm, int sae);

SIMD Floating-Point Exceptions

Zero, Invalid

Other Exceptions

See Exceptions Type E3.

VFIXUPIMMSS—Fix Up Special Scalar Float32 Value

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.NDS.LIG.66.0F3A.W0 55 /r ib VFIXUPIMMSS xmm1 {k1}{z}, xmm2, xmm3/m32{sae}, imm8	A	V/V	AVX512F	Fix up a float32 number in the low doubleword element in xmm2 using scalar int32 table in xmm3/m32 and store the result in xmm1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Tuple1 Scalar	ModRM:reg (r, w)	EVEX.vvvv	ModRM:r/m (r)	Imm8

Description

Perform a fix-up of the low doubleword element encoded in single-precision floating-point format in the first source operand (the second operand) using a 32-bit, two-level look-up table specified in the low doubleword element of the second source operand (the third operand) with exception reporting specifier imm8. The element that is fixed-up is selected by mask bit of 1 specified in the opmask k1. Mask bit of 0 in the opmask k1 or table response action of 0000b preserves the corresponding element of the first operand. The fixed-up element from the first source operand or the preserved element in the first operand becomes the low doubleword element of the destination operand (the first operand) Bits 127:32 of the destination operand is copied from the corresponding bits of the first source operand. The destination and first source operands are XMM registers. The second source operand can be a XMM register or a 32-bit memory location.

The two-level look-up table perform a fix-up of each SP FP input data in the first source operand by decoding the input data encoding into 8 token types. A response table is defined for each token type that converts the input encoding in the first source operand with one of 16 response actions.

This instruction is specifically intended for use in fixing up the results of arithmetic calculations involving one source so that they match the spec, although it is generally useful for fixing up the results of multiple-instruction sequences to reflect special-number inputs. For example, consider rcp(0). Input 0 to rcp, and you should get INF according to the DX10 spec. However, evaluating rcp via Newton-Raphson, where x=approx(1/0), yields an incorrect result. To deal with this, VFIXUPIMMPD can be used after the N-R reciprocal sequence to set the result to the correct value (i.e. INF when the input is 0).

If MXCSR.DAZ is not set, denormal input elements in the first source operand are considered as normal inputs and do not trigger any fixup nor fault reporting.

Imm8 is used to set the required flags reporting. It supports #ZE and #IE fault reporting (see details below).

MXCSR.DAZ is used and refer to zmm2 only (i.e. zmm1 is not considered as zero in case MXCSR.DAZ is set).

MXCSR mask bits are ignored and are treated as if all mask bits are set to masked response). If any of the imm8 bits is set and the condition met for fault reporting, MXCSR.IE or MXCSR.ZE might be updated.

Operation

```
enum TOKEN_TYPE
{
    QNAN_TOKEN ← 0,
    SNAN_TOKEN ← 1,
    ZERO_VALUE_TOKEN ← 2,
    POS_ONE_VALUE_TOKEN ← 3,
    NEG_INF_TOKEN ← 4,
    POS_INF_TOKEN ← 5,
    NEG_VALUE_TOKEN ← 6,
    POS_VALUE_TOKEN ← 7
}
```

```
FIXUPIMM_SP (dest[31:0], src1[31:0],tbl3[31:0], imm8 [7:0]){
    tsrc[31:0] ← ((src1[30:23] = 0) AND (MXCSR.DAZ =1)) ? 0.0 : src1[31:0]
    CASE(tsrc[63:0] of TOKEN_TYPE) {
        QNAN_TOKEN: j ← 0;
        SNAN_TOKEN: j ← 1;
        ZERO_VALUE_TOKEN: j ← 2;
        POS_ONE_VALUE_TOKEN: j ← 3;
        NEG_INF_TOKEN: j ← 4;
        POS_INF_TOKEN: j ← 5;
        NEG_VALUE_TOKEN: j ← 6;
        POS_VALUE_TOKEN: j = 7;
    }           ; end source special CASE(tsrc...)

    ; The required response from src3 table is extracted
    token_response[3:0] = tbl3[3+4*j:4*j];

    CASE(token_response[3:0]) {
        0000: dest[31:0] ← dest[31:0];        ; preserve content of DEST
        0001: dest[31:0] ← tsrc[31:0];        ; pass through src1 normal input value, denormal as zero
        0010: dest[31:0] ← QNaN(tsrc[31:0]);
        0011: dest[31:0] ← QNAN_Indefinite;
        0100: dest[31:0] ← -INF;
        0101: dest[31:0] ← +INF;
        0110: dest[31:0] ← tsrc.sign? –INF : +INF;
        0111: dest[31:0] ← -0;
        1000: dest[31:0] ← +0;
        1001: dest[31:0] ← -1;
        1010: dest[31:0] ← +1;
        1011: dest[31:0] ← ½;
        1100: dest[31:0] ← 90.0;
        1101: dest[31:0] ← PI/2;
        1110: dest[31:0] ← MAX_FLOAT;
        1111: dest[31:0] ← -MAX_FLOAT;
    }           ; end of token_response CASE

    ; The required fault reporting from imm8 is extracted
    ; TOKENs are mutually exclusive and TOKENs priority defines the order.
    ; Multiple faults related to a single token can occur simultaneously.
    IF (tsrc[31:0] of TOKEN_TYPE: ZERO_VALUE_TOKEN) AND imm8[0] then set #ZE;
    IF (tsrc[31:0] of TOKEN_TYPE: ZERO_VALUE_TOKEN) AND imm8[1] then set #IE;
    IF (tsrc[31:0] of TOKEN_TYPE: ONE_VALUE_TOKEN) AND imm8[2] then set #ZE;
    IF (tsrc[31:0] of TOKEN_TYPE: ONE_VALUE_TOKEN) AND imm8[3] then set #IE;
    IF (tsrc[31:0] of TOKEN_TYPE: SNAN_TOKEN) AND imm8[4] then set #IE;
    IF (tsrc[31:0] of TOKEN_TYPE: NEG_INF_TOKEN) AND imm8[5] then set #IE;
    IF (tsrc[31:0] of TOKEN_TYPE: NEG_VALUE_TOKEN) AND imm8[6] then set #IE;
    IF (tsrc[31:0] of TOKEN_TYPE: POS_INF_TOKEN) AND imm8[7] then set #IE;
        ; end fault reporting
    return dest[31:0];
}       ; end of FIXUPIMM_SP()
```

VFIXUPIMMSS (EVEX encoded version)

```
IF k1[0] OR *no writemask*
    THEN DEST[31:0] ← FIXUPIMM_SP(DEST[31:0], SRC1[31:0], SRC2[31:0], imm8 [7:0])
    ELSE
        IF *merging-masking*                    ; merging-masking
            THEN *DEST[31:0] remains unchanged*
            ELSE  DEST[31:0] ← 0                ; zeroing-masking
        FI
FI;
DEST[127:32] ← SRC1[127:32]
DEST[MAXVL-1:128] ← 0
```

Immediate Control Description:

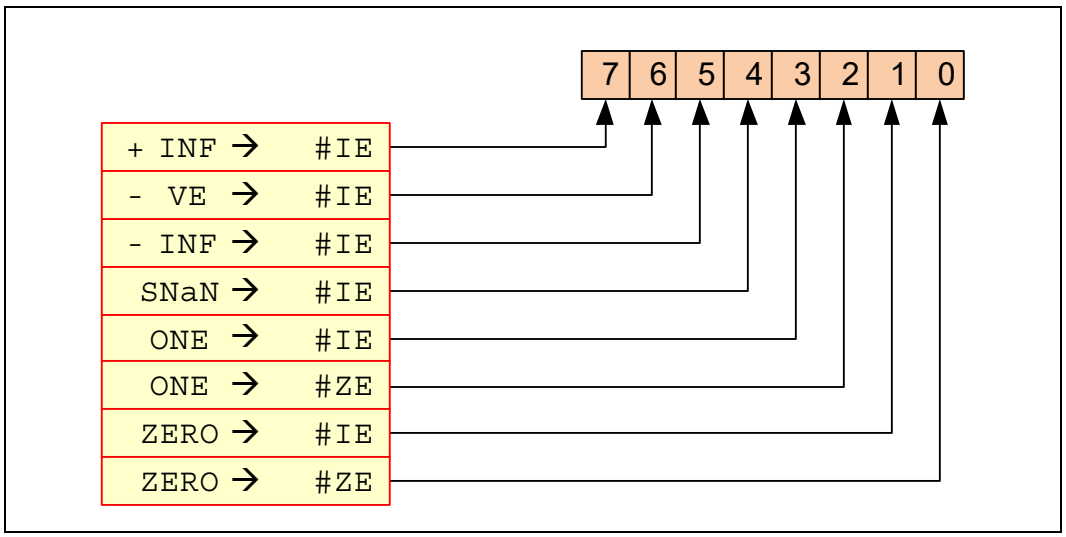


Figure 5-12. VFIXUPIMMSS Immediate Control Description

Intel C/C++ Compiler Intrinsic Equivalent

VFIXUPIMMSS __m128 _mm_fixupimm_ss(__m128 a, __m128i tbl, int imm);
VFIXUPIMMSS __m128 _mm_mask_fixupimm_ss(__m128 s, __mmask8 k, __m128 a, __m128i tbl, int imm);
VFIXUPIMMSS __m128 _mm_maskz_fixupimm_ss(__mmask8 k, __m128 a, __m128i tbl, int imm);
VFIXUPIMMSS __m128 _mm_fixupimm_round_ss(__m128 a, __m128i tbl, int imm, int sae);
VFIXUPIMMSS __m128 _mm_mask_fixupimm_round_ss(__m128 s, __mmask8 k, __m128 a, __m128i tbl, int imm, int sae);
VFIXUPIMMSS __m128 _mm_maskz_fixupimm_round_ss(__mmask8 k, __m128 a, __m128i tbl, int imm, int sae);

SIMD Floating-Point Exceptions

Zero, Invalid

Other Exceptions

See Exceptions Type E3.

VFMADD132PD/VFMADD213PD/VFMADD231PD—Fused Multiply-Add of Packed Double-Precision Floating-Point Values

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
VEX.NDS.128.66.0F38.W1 98 /r VFMADD132PD xmm1, xmm2, xmm3/m128	A	V/V	FMA	Multiply packed double-precision floating-point values from xmm1 and xmm3/mem, add to xmm2 and put result in xmm1.
VEX.NDS.128.66.0F38.W1 A8 /r VFMADD213PD xmm1, xmm2, xmm3/m128	A	V/V	FMA	Multiply packed double-precision floating-point values from xmm1 and xmm2, add to xmm3/mem and put result in xmm1.
VEX.NDS.128.66.0F38.W1 B8 /r VFMADD231PD xmm1, xmm2, xmm3/m128	A	V/V	FMA	Multiply packed double-precision floating-point values from xmm2 and xmm3/mem, add to xmm1 and put result in xmm1.
VEX.NDS.256.66.0F38.W1 98 /r VFMADD132PD ymm1, ymm2, ymm3/m256	A	V/V	FMA	Multiply packed double-precision floating-point values from ymm1 and ymm3/mem, add to ymm2 and put result in ymm1.
VEX.NDS.256.66.0F38.W1 A8 /r VFMADD213PD ymm1, ymm2, ymm3/m256	A	V/V	FMA	Multiply packed double-precision floating-point values from ymm1 and ymm2, add to ymm3/mem and put result in ymm1.
VEX.NDS.256.66.0F38.W1 B8 /r VFMADD231PD ymm1, ymm2, ymm3/m256	A	V/V	FMA	Multiply packed double-precision floating-point values from ymm2 and ymm3/mem, add to ymm1 and put result in ymm1.
EVEX.NDS.128.66.0F38.W1 98 /r VFMADD132PD xmm1 {k1}{z}, xmm2, xmm3/m128/m64bcst	A	V/V	AVX512VL AVX512F	Multiply packed double-precision floating-point values from xmm1 and xmm3/m128/m64bcst, add to xmm2 and put result in xmm1.
EVEX.NDS.128.66.0F38.W1 A8 /r VFMADD213PD xmm1 {k1}{z}, xmm2, xmm3/m128/m64bcst	B	V/V	AVX512VL AVX512F	Multiply packed double-precision floating-point values from xmm1 and xmm2, add to xmm3/m128/m64bcst and put result in xmm1.
EVEX.NDS.128.66.0F38.W1 B8 /r VFMADD231PD xmm1 {k1}{z}, xmm2, xmm3/m128/m64bcst	B	V/V	AVX512VL AVX512F	Multiply packed double-precision floating-point values from xmm2 and xmm3/m128/m64bcst, add to xmm1 and put result in xmm1.
EVEX.NDS.256.66.0F38.W1 98 /r VFMADD132PD ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst	B	V/V	AVX512VL AVX512F	Multiply packed double-precision floating-point values from ymm1 and ymm3/m256/m64bcst, add to ymm2 and put result in ymm1.
EVEX.NDS.256.66.0F38.W1 A8 /r VFMADD213PD ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst	B	V/V	AVX512VL AVX512F	Multiply packed double-precision floating-point values from ymm1 and ymm2, add to ymm3/m256/m64bcst and put result in ymm1.
EVEX.NDS.256.66.0F38.W1 B8 /r VFMADD231PD ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst	B	V/V	AVX512VL AVX512F	Multiply packed double-precision floating-point values from ymm2 and ymm3/m256/m64bcst, add to ymm1 and put result in ymm1.
EVEX.NDS.512.66.0F38.W1 98 /r VFMADD132PD zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst{er}	B	V/V	AVX512F	Multiply packed double-precision floating-point values from zmm1 and zmm3/m512/m64bcst, add to zmm2 and put result in zmm1.
EVEX.NDS.512.66.0F38.W1 A8 /r VFMADD213PD zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst{er}	B	V/V	AVX512F	Multiply packed double-precision floating-point values from zmm1 and zmm2, add to zmm3/m512/m64bcst and put result in zmm1.
EVEX.NDS.512.66.0F38.W1 B8 /r VFMADD231PD zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst{er}	B	V/V	AVX512F	Multiply packed double-precision floating-point values from zmm2 and zmm3/m512/m64bcst, add to zmm1 and put result in zmm1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	NA	ModRM:reg (r, w)	VEX.vvvv (r)	ModRM:r/m (r)	NA
B	Full Vector	ModRM:reg (r, w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA

Description

Performs a set of SIMD multiply-add computation on packed double-precision floating-point values using three source operands and writes the multiply-add results in the destination operand. The destination operand is also the first source operand. The second operand must be a SIMD register. The third source operand can be a SIMD register or a memory location.

VFMADD132PD: Multiplies the two, four or eight packed double-precision floating-point values from the first source operand to the two, four or eight packed double-precision floating-point values in the third source operand, adds the infinite precision intermediate result to the two, four or eight packed double-precision floating-point values in the second source operand, performs rounding and stores the resulting two, four or eight packed double-precision floating-point values to the destination operand (first source operand).

VFMADD213PD: Multiplies the two, four or eight packed double-precision floating-point values from the second source operand to the two, four or eight packed double-precision floating-point values in the first source operand, adds the infinite precision intermediate result to the two, four or eight packed double-precision floating-point values in the third source operand, performs rounding and stores the resulting two, four or eight packed double-precision floating-point values to the destination operand (first source operand).

VFMADD231PD: Multiplies the two, four or eight packed double-precision floating-point values from the second source to the two, four or eight packed double-precision floating-point values in the third source operand, adds the infinite precision intermediate result to the two, four or eight packed double-precision floating-point values in the first source operand, performs rounding and stores the resulting two, four or eight packed double-precision floating-point values to the destination operand (first source operand).

EVEX encoded versions: The destination operand (also first source operand) is a ZMM register and encoded in reg_field. The second source operand is a ZMM register and encoded in EVEX.vvvv. The third source operand is a ZMM register, a 512-bit memory location, or a 512-bit vector broadcasted from a 64-bit memory location. The destination operand is conditionally updated with write mask k1.

VEX.256 encoded version: The destination operand (also first source operand) is a YMM register and encoded in reg_field. The second source operand is a YMM register and encoded in VEX.vvvv. The third source operand is a YMM register or a 256-bit memory location and encoded in rm_field.

VEX.128 encoded version: The destination operand (also first source operand) is a XMM register and encoded in reg_field. The second source operand is a XMM register and encoded in VEX.vvvv. The third source operand is a XMM register or a 128-bit memory location and encoded in rm_field. The upper 128 bits of the YMM destination register are zeroed.

Operation

In the operations below, "*" and "+" symbols represent multiplication and addition with infinite precision inputs and outputs (no rounding).

VFMADD132PD DEST, SRC2, SRC3 (VEX encoded version)

```
IF (VEX.128) THEN
    MAXNUM ←2
ELSEIF (VEX.256)
    MAXNUM ← 4
FI
For i = 0 to MAXNUM-1 {
    n ← 64*i;
    DEST[n+63:n] ← RoundFPControl_MXCSR(DEST[n+63:n]*SRC3[n+63:n] + SRC2[n+63:n])
}
IF (VEX.128) THEN
    DEST[MAXVL-1:128] ← 0
ELSEIF (VEX.256)
    DEST[MAXVL-1:256] ← 0
FI
```

VFMADD213PD DEST, SRC2, SRC3 (VEX encoded version)

```
IF (VEX.128) THEN
    MAXNUM ←2
ELSEIF (VEX.256)
    MAXNUM ← 4
FI
For i = 0 to MAXNUM-1 {
    n ← 64*i;
    DEST[n+63:n] ← RoundFPControl_MXCSR(SRC2[n+63:n]*DEST[n+63:n] + SRC3[n+63:n])
}
IF (VEX.128) THEN
    DEST[MAXVL-1:128] ← 0
ELSEIF (VEX.256)
    DEST[MAXVL-1:256] ← 0
FI
```

VFMADD231PD DEST, SRC2, SRC3 (VEX encoded version)

```
IF (VEX.128) THEN
    MAXNUM ←2
ELSEIF (VEX.256)
    MAXNUM ← 4
FI
For i = 0 to MAXNUM-1 {
    n ← 64*i;
    DEST[n+63:n] ← RoundFPControl_MXCSR(SRC2[n+63:n]*SRC3[n+63:n] + DEST[n+63:n])
}
IF (VEX.128) THEN
    DEST[MAXVL-1:128] ← 0
ELSEIF (VEX.256)
    DEST[MAXVL-1:256] ← 0
FI
```

VFMADD132PD DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a register)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
IF (VL = 512) AND (EVEX.b = 1)
    THEN
        SET_RM(EVEX.RC);
    ELSE
        SET_RM(MXCSR.RM);
FI;
FOR j ← 0 TO KL-1
    i ← j * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+63:i] ←
            RoundFPControl(DEST[i+63:i]*SRC3[i+63:i] + SRC2[i+63:i])
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE                            ; zeroing-masking
                    DEST[i+63:i] ← 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0
```

VFMADD132PD DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a memory source)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)

FOR j ← 0 TO KL-1
    i ← j * 64
    IF k1[j] OR *no writemask*
        THEN
            IF (EVEX.b = 1)
                THEN
                    DEST[i+63:i] ←
            RoundFPControl_MXCSR(DEST[i+63:i]*SRC3[63:0] + SRC2[i+63:i])
                ELSE
                    DEST[i+63:i] ←
            RoundFPControl_MXCSR(DEST[i+63:i]*SRC3[i+63:i] + SRC2[i+63:i])
            FI;
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE                            ; zeroing-masking
                    DEST[i+63:i] ← 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0
```

VFMADD213PD DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a is a register)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
IF (VL = 512) AND (EVEX.b = 1)
    THEN
        SET_RM(EVEX.RC);
    ELSE
        SET_RM(MXCSR.RM);
FI;
FOR j ← 0 TO KL-1
    i ← j * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+63:i] ←
            RoundFPControl(SRC2[i+63:i]*DEST[i+63:i] + SRC3[i+63:i])
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE                            ; zeroing-masking
                    DEST[i+63:i] ← 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0
```

VFMADD213PD DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a memory source)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)

FOR j ← 0 TO KL-1
    i ← j * 64
    IF k1[j] OR *no writemask*
        THEN
            IF (EVEX.b = 1)
                THEN
                    DEST[i+63:i] ←
            RoundFPControl_MXCSR(SRC2[i+63:i]*DEST[i+63:i] + SRC3[63:0])
                ELSE
                    DEST[i+63:i] ←
            RoundFPControl_MXCSR(SRC2[i+63:i]*DEST[i+63:i] + SRC3[i+63:i])
            FI;
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE                            ; zeroing-masking
                    DEST[i+63:i] ← 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0
```

VFMADD231PD DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a register)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
IF (VL = 512) AND (EVEX.b = 1)
    THEN
        SET_RM(EVEX.RC);
    ELSE
        SET_RM(MXCSR.RM);
FI;
FOR j ← 0 TO KL-1
    i ← j * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+63:i] ←
            RoundFPControl(SRC2[i+63:i]*SRC3[i+63:i] + DEST[i+63:i])
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE                            ; zeroing-masking
                    DEST[i+63:i] ← 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0
```

VFMADD231PD DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a memory source)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)

FOR j ← 0 TO KL-1
    i ← j * 64
    IF k1[j] OR *no writemask*
        THEN
            IF (EVEX.b = 1)
                THEN
                    DEST[i+63:i] ←
            RoundFPControl_MXCSR(SRC2[i+63:i]*SRC3[63:0] + DEST[i+63:i])
                ELSE
                    DEST[i+63:i] ←
            RoundFPControl_MXCSR(SRC2[i+63:i]*SRC3[i+63:i] + DEST[i+63:i])
            FI;
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE                            ; zeroing-masking
                    DEST[i+63:i] ← 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VFMADDxxxPD __m512d _mm512_fmadd_pd(__m512d a, __m512d b, __m512d c);
VFMADDxxxPD __m512d _mm512_fmadd_round_pd(__m512d a, __m512d b, __m512d c, int r);
VFMADDxxxPD __m512d _mm512_mask_fmadd_pd(__m512d a, __mmask8 k, __m512d b, __m512d c);
VFMADDxxxPD __m512d _mm512_maskz_fmadd_pd(__mmask8 k, __m512d a, __m512d b, __m512d c);
VFMADDxxxPD __m512d _mm512_mask3_fmadd_pd(__m512d a, __m512d b, __m512d c, __mmask8 k);
VFMADDxxxPD __m512d _mm512_mask_fmadd_round_pd(__m512d a, __mmask8 k, __m512d b, __m512d c, int r);
VFMADDxxxPD __m512d _mm512_maskz_fmadd_round_pd(__mmask8 k, __m512d a, __m512d b, __m512d c, int r);
VFMADDxxxPD __m512d _mm512_mask3_fmadd_round_pd(__m512d a, __m512d b, __m512d c, __mmask8 k, int r);
VFMADDxxxPD __m256d _mm256_mask_fmadd_pd(__m256d a, __mmask8 k, __m256d b, __m256d c);
VFMADDxxxPD __m256d _mm256_maskz_fmadd_pd(__mmask8 k, __m256d a, __m256d b, __m256d c);
VFMADDxxxPD __m256d _mm256_mask3_fmadd_pd(__m256d a, __m256d b, __m256d c, __mmask8 k);
VFMADDxxxPD __m128d _mm_mask_fmadd_pd(__m128d a, __mmask8 k, __m128d b, __m128d c);
VFMADDxxxPD __m128d _mm_maskz_fmadd_pd(__mmask8 k, __m128d a, __m128d b, __m128d c);
VFMADDxxxPD __m128d _mm_mask3_fmadd_pd(__m128d a, __m128d b, __m128d c, __mmask8 k);
VFMADDxxxPD __m128d _mm_fmadd_pd (__m128d a, __m128d b, __m128d c);
VFMADDxxxPD __m256d _mm256_fmadd_pd (__m256d a, __m256d b, __m256d c);

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal

Other Exceptions

VEX-encoded instructions, see Exceptions Type 2.
EVEX-encoded instructions, see Exceptions Type E2.

VFMADD132PS/VFMADD213PS/VFMADD231PS—Fused Multiply-Add of Packed Single-Precision Floating-Point Values

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
VEX.NDS.128.66.0F38.W0 98 /r VFMADD132PS xmm1, xmm2, xmm3/m128	A	V/V	FMA	Multiply packed single-precision floating-point values from xmm1 and xmm3/mem, add to xmm2 and put result in xmm1.
VEX.NDS.128.66.0F38.W0 A8 /r VFMADD213PS xmm1, xmm2, xmm3/m128	A	V/V	FMA	Multiply packed single-precision floating-point values from xmm1 and xmm2, add to xmm3/mem and put result in xmm1.
VEX.NDS.128.66.0F38.W0 B8 /r VFMADD231PS xmm1, xmm2, xmm3/m128	A	V/V	FMA	Multiply packed single-precision floating-point values from xmm2 and xmm3/mem, add to xmm1 and put result in xmm1.
VEX.NDS.256.66.0F38.W0 98 /r VFMADD132PS ymm1, ymm2, ymm3/m256	A	V/V	FMA	Multiply packed single-precision floating-point values from ymm1 and ymm3/mem, add to ymm2 and put result in ymm1.
VEX.NDS.256.66.0F38.W0 A8 /r VFMADD213PS ymm1, ymm2, ymm3/m256	A	V/V	FMA	Multiply packed single-precision floating-point values from ymm1 and ymm2, add to ymm3/mem and put result in ymm1.
VEX.NDS.256.66.0F38.0 B8 /r VFMADD231PS ymm1, ymm2, ymm3/m256	A	V/V	FMA	Multiply packed single-precision floating-point values from ymm2 and ymm3/mem, add to ymm1 and put result in ymm1.
EVEX.NDS.128.66.0F38.W0 98 /r VFMADD132PS xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst	B	V/V	AVX512VL AVX512F	Multiply packed single-precision floating-point values from xmm1 and xmm3/m128/m32bcst, add to xmm2 and put result in xmm1.
EVEX.NDS.128.66.0F38.W0 A8 /r VFMADD213PS xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst	B	V/V	AVX512VL AVX512F	Multiply packed single-precision floating-point values from xmm1 and xmm2, add to xmm3/m128/m32bcst and put result in xmm1.
EVEX.NDS.128.66.0F38.W0 B8 /r VFMADD231PS xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst	B	V/V	AVX512VL AVX512F	Multiply packed single-precision floating-point values from xmm2 and xmm3/m128/m32bcst, add to xmm1 and put result in xmm1.
EVEX.NDS.256.66.0F38.W0 98 /r VFMADD132PS ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst	B	V/V	AVX512VL AVX512F	Multiply packed single-precision floating-point values from ymm1 and ymm3/m256/m32bcst, add to ymm2 and put result in ymm1.
EVEX.NDS.256.66.0F38.W0 A8 /r VFMADD213PS ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst	B	V/V	AVX512VL AVX512F	Multiply packed single-precision floating-point values from ymm1 and ymm2, add to ymm3/m256/m32bcst and put result in ymm1.
EVEX.NDS.256.66.0F38.W0 B8 /r VFMADD231PS ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst	B	V/V	AVX512VL AVX512F	Multiply packed single-precision floating-point values from ymm2 and ymm3/m256/m32bcst, add to ymm1 and put result in ymm1.
EVEX.NDS.512.66.0F38.W0 98 /r VFMADD132PS zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst{er}	B	V/V	AVX512F	Multiply packed single-precision floating-point values from zmm1 and zmm3/m512/m32bcst, add to zmm2 and put result in zmm1.
EVEX.NDS.512.66.0F38.W0 A8 /r VFMADD213PS zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst{er}	B	V/V	AVX512F	Multiply packed single-precision floating-point values from zmm1 and zmm2, add to zmm3/m512/m32bcst and put result in zmm1.
EVEX.NDS.512.66.0F38.W0 B8 /r VFMADD231PS zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst{er}	B	V/V	AVX512F	Multiply packed single-precision floating-point values from zmm2 and zmm3/m512/m32bcst, add to zmm1 and put result in zmm1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	NA	ModRM:reg (r, w)	VEX.vvvv (r)	ModRM:r/m (r)	NA
B	Full Vector	ModRM:reg (r, w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA

Description

Performs a set of SIMD multiply-add computation on packed single-precision floating-point values using three source operands and writes the multiply-add results in the destination operand. The destination operand is also the first source operand. The second operand must be a SIMD register. The third source operand can be a SIMD register or a memory location.

VFMADD132PS: Multiplies the four, eight or sixteen packed single-precision floating-point values from the first source operand to the four, eight or sixteen packed single-precision floating-point values in the third source operand, adds the infinite precision intermediate result to the four, eight or sixteen packed single-precision floating-point values in the second source operand, performs rounding and stores the resulting four, eight or sixteen packed single-precision floating-point values to the destination operand (first source operand).

VFMADD213PS: Multiplies the four, eight or sixteen packed single-precision floating-point values from the second source operand to the four, eight or sixteen packed single-precision floating-point values in the first source operand, adds the infinite precision intermediate result to the four, eight or sixteen packed single-precision floating-point values in the third source operand, performs rounding and stores the resulting the four, eight or sixteen packed single-precision floating-point values to the destination operand (first source operand).

VFMADD231PS: Multiplies the four, eight or sixteen packed single-precision floating-point values from the second source operand to the four, eight or sixteen packed single-precision floating-point values in the third source operand, adds the infinite precision intermediate result to the four, eight or sixteen packed single-precision floating-point values in the first source operand, performs rounding and stores the resulting four, eight or sixteen packed single-precision floating-point values to the destination operand (first source operand).

EVEX encoded versions: The destination operand (also first source operand) is a ZMM register and encoded in reg_field. The second source operand is a ZMM register and encoded in EVEX.vvvv. The third source operand is a ZMM register, a 512-bit memory location, or a 512-bit vector broadcasted from a 32-bit memory location. The destination operand is conditionally updated with write mask k1.

VEX.256 encoded version: The destination operand (also first source operand) is a YMM register and encoded in reg_field. The second source operand is a YMM register and encoded in VEX.vvvv. The third source operand is a YMM register or a 256-bit memory location and encoded in rm_field.

VEX.128 encoded version: The destination operand (also first source operand) is a XMM register and encoded in reg_field. The second source operand is a XMM register and encoded in VEX.vvvv. The third source operand is a XMM register or a 128-bit memory location and encoded in rm_field. The upper 128 bits of the YMM destination register are zeroed.

Operation

In the operations below, "*" and "+" symbols represent multiplication and addition with infinite precision inputs and outputs (no rounding).

VFMADD132PS DEST, SRC2, SRC3

```
IF (VEX.128) THEN
    MAXNUM ←4
ELSEIF (VEX.256)
    MAXNUM ← 8
FI
For i = 0 to MAXNUM-1 {
    n ← 32*i;
    DEST[n+31:n] ← RoundFPControl_MXCSR(DEST[n+31:n]*SRC3[n+31:n] + SRC2[n+31:n])
}
IF (VEX.128) THEN
    DEST[MAXVL-1:128] ← 0
ELSEIF (VEX.256)
    DEST[MAXVL-1:256] ← 0
FI
```

VFMADD213PS DEST, SRC2, SRC3

```
IF (VEX.128) THEN
    MAXNUM ←4
ELSEIF (VEX.256)
    MAXNUM ← 8
FI
For i = 0 to MAXNUM-1 {
    n ← 32*i;
    DEST[n+31:n] ← RoundFPControl_MXCSR(SRC2[n+31:n]*DEST[n+31:n] + SRC3[n+31:n])
}
IF (VEX.128) THEN
    DEST[MAXVL-1:128] ← 0
ELSEIF (VEX.256)
    DEST[MAXVL-1:256] ← 0
FI
```

VFMADD231PS DEST, SRC2, SRC3

```
IF (VEX.128) THEN
    MAXNUM ←4
ELSEIF (VEX.256)
    MAXNUM ← 8
FI
For i = 0 to MAXNUM-1 {
    n ← 32*i;
    DEST[n+31:n] ← RoundFPControl_MXCSR(SRC2[n+31:n]*SRC3[n+31:n] + DEST[n+31:n])
}
IF (VEX.128) THEN
    DEST[MAXVL-1:128] ← 0
ELSEIF (VEX.256)
    DEST[MAXVL-1:256] ← 0
FI
```

VFMADD132PS DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a register)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
IF (VL = 512) AND (EVEX.b = 1)
    THEN
        SET_RM(EVEX.RC);
    ELSE
        SET_RM(MXCSR.RM);
FI;
FOR j ← 0 TO KL-1
    i ← j * 32
    IF k1[j] OR *no writemask*
        THEN DEST[i+31:i] ←
            RoundFPControl(DEST[i+31:i]*SRC3[i+31:i] + SRC2[i+31:i])
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE                            ; zeroing-masking
                    DEST[i+31:i] ← 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0
```

VFMADD132PS DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a memory source)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)

FOR j ← 0 TO KL-1
    i ← j * 32
    IF k1[j] OR *no writemask*
        THEN
            IF (EVEX.b = 1)
                THEN
                    DEST[i+31:i] ←
            RoundFPControl_MXCSR(DEST[i+31:i]*SRC3[31:0] + SRC2[i+31:i])
                ELSE
                    DEST[i+31:i] ←
            RoundFPControl_MXCSR(DEST[i+31:i]*SRC3[i+31:i] + SRC2[i+31:i])
            FI;
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE                            ; zeroing-masking
                    DEST[i+31:i] ← 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0
```

VFMADD213PS DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a register)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
IF (VL = 512) AND (EVEX.b = 1)
    THEN
        SET_RM(EVEX.RC);
    ELSE
        SET_RM(MXCSR.RM);
FI;
FOR j ← 0 TO KL-1
    i ← j * 32
    IF k1[j] OR *no writemask*
        THEN DEST[i+31:i] ←
            RoundFPControl(SRC2[i+31:i]*DEST[i+31:i] + SRC3[i+31:i])
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE                            ; zeroing-masking
                    DEST[i+31:i] ← 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0
```

VFMADD213PS DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a memory source)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)

FOR j ← 0 TO KL-1
    i ← j * 32
    IF k1[j] OR *no writemask*
        THEN
            IF (EVEX.b = 1)
                THEN
                    DEST[i+31:i] ←
            RoundFPControl_MXCSR(SRC2[i+31:i]*DEST[i+31:i] + SRC3[31:0])
                ELSE
                    DEST[i+31:i] ←
            RoundFPControl_MXCSR(SRC2[i+31:i]*DEST[i+31:i] + SRC3[i+31:i])
            FI;
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE                            ; zeroing-masking
                    DEST[i+31:i] ← 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0
```

VFMADD231PS DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a register)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
IF (VL = 512) AND (EVEX.b = 1)
    THEN
        SET_RM(EVEX.RC);
    ELSE
        SET_RM(MXCSR.RM);
FI;
FOR j ← 0 TO KL-1
    i ← j * 32
    IF k1[j] OR *no writemask*
        THEN DEST[i+31:i] ←
            RoundFPControl(SRC2[i+31:i]*SRC3[i+31:i] + DEST[i+31:i])
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE                            ; zeroing-masking
                    DEST[i+31:i] ← 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0
```

VFMADD231PS DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a memory source)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)

FOR j ← 0 TO KL-1
    i ← j * 32
    IF k1[j] OR *no writemask*
        THEN
            IF (EVEX.b = 1)
                THEN
                    DEST[i+31:i] ←
            RoundFPControl_MXCSR(SRC2[i+31:i]*SRC3[31:0] + DEST[i+31:i])
                ELSE
                    DEST[i+31:i] ←
            RoundFPControl_MXCSR(SRC2[i+31:i]*SRC3[i+31:i] + DEST[i+31:i])
            FI;
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE                            ; zeroing-masking
                    DEST[i+31:i] ← 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VFMADDxxxPS __m512 _mm512_fmadd_ps(__m512 a, __m512 b, __m512 c);
VFMADDxxxPS __m512 _mm512_fmadd_round_ps(__m512 a, __m512 b, __m512 c, int r);
VFMADDxxxPS __m512 _mm512_mask_fmadd_ps(__m512 a, __mmask16 k, __m512 b, __m512 c);
VFMADDxxxPS __m512 _mm512_maskz_fmadd_ps(__mmask16 k, __m512 a, __m512 b, __m512 c);
VFMADDxxxPS __m512 _mm512_mask3_fmadd_ps(__m512 a, __m512 b, __m512 c, __mmask16 k);
VFMADDxxxPS __m512 _mm512_mask_fmadd_round_ps(__m512 a, __mmask16 k, __m512 b, __m512 c, int r);
VFMADDxxxPS __m512 _mm512_maskz_fmadd_round_ps(__mmask16 k, __m512 a, __m512 b, __m512 c, int r);
VFMADDxxxPS __m512 _mm512_mask3_fmadd_round_ps(__m512 a, __m512 b, __m512 c, __mmask16 k, int r);
VFMADDxxxPS __m256 _mm256_mask_fmadd_ps(__m256 a, __mmask8 k, __m256 b, __m256 c);
VFMADDxxxPS __m256 _mm256_maskz_fmadd_ps(__mmask8 k, __m256 a, __m256 b, __m256 c);
VFMADDxxxPS __m256 _mm256_mask3_fmadd_ps(__m256 a, __m256 b, __m256 c, __mmask8 k);
VFMADDxxxPS __m128 _mm_mask_fmadd_ps(__m128 a, __mmask8 k, __m128 b, __m128 c);
VFMADDxxxPS __m128 _mm_maskz_fmadd_ps(__mmask8 k, __m128 a, __m128 b, __m128 c);
VFMADDxxxPS __m128 _mm_mask3_fmadd_ps(__m128 a, __m128 b, __m128 c, __mmask8 k);
VFMADDxxxPS __m128 _mm_fmadd_ps (__m128 a, __m128 b, __m128 c);
VFMADDxxxPS __m256 _mm256_fmadd_ps (__m256 a, __m256 b, __m256 c);

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal

Other Exceptions

VEX-encoded instructions, see Exceptions Type 2.
EVEX-encoded instructions, see Exceptions Type E2.

VFMADD132SD/VFMADD213SD/VFMADD231SD—Fused Multiply-Add of Scalar Double-Precision Floating-Point Values

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
VEX.DDS.LIG.66.0F38.W1 99 /r VFMADD132SD xmm1, xmm2, xmm3/m64	A	V/V	FMA	Multiply scalar double-precision floating-point value from xmm1 and xmm3/m64, add to xmm2 and put result in xmm1.
VEX.DDS.LIG.66.0F38.W1 A9 /r VFMADD213SD xmm1, xmm2, xmm3/m64	A	V/V	FMA	Multiply scalar double-precision floating-point value from xmm1 and xmm2, add to xmm3/m64 and put result in xmm1.
VEX.DDS.LIG.66.0F38.W1 B9 /r VFMADD231SD xmm1, xmm2, xmm3/m64	A	V/V	FMA	Multiply scalar double-precision floating-point value from xmm2 and xmm3/m64, add to xmm1 and put result in xmm1.
EVEX.DDS.LIG.66.0F38.W1 99 /r VFMADD132SD xmm1 {k1}{z}, xmm2, xmm3/m64{er}	B	V/V	AVX512F	Multiply scalar double-precision floating-point value from xmm1 and xmm3/m64, add to xmm2 and put result in xmm1.
EVEX.DDS.LIG.66.0F38.W1 A9 /r VFMADD213SD xmm1 {k1}{z}, xmm2, xmm3/m64{er}	B	V/V	AVX512F	Multiply scalar double-precision floating-point value from xmm1 and xmm2, add to xmm3/m64 and put result in xmm1.
EVEX.DDS.LIG.66.0F38.W1 B9 /r VFMADD231SD xmm1 {k1}{z}, xmm2, xmm3/m64{er}	B	V/V	AVX512F	Multiply scalar double-precision floating-point value from xmm2 and xmm3/m64, add to xmm1 and put result in xmm1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	NA	ModRM:reg (r, w)	VEX.vvvv (r)	ModRM:r/m (r)	NA
B	Tuple1 Scalar	ModRM:reg (r, w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA

Description

Performs a SIMD multiply-add computation on the low double-precision floating-point values using three source operands and writes the multiply-add result in the destination operand. The destination operand is also the first source operand. The first and second operand are XMM registers. The third source operand can be an XMM register or a 64-bit memory location.

VFMADD132SD: Multiplies the low double-precision floating-point value from the first source operand to the low double-precision floating-point value in the third source operand, adds the infinite precision intermediate result to the low double-precision floating-point values in the second source operand, performs rounding and stores the resulting double-precision floating-point value to the destination operand (first source operand).

VFMADD213SD: Multiplies the low double-precision floating-point value from the second source operand to the low double-precision floating-point value in the first source operand, adds the infinite precision intermediate result to the low double-precision floating-point value in the third source operand, performs rounding and stores the resulting double-precision floating-point value to the destination operand (first source operand).

VFMADD231SD: Multiplies the low double-precision floating-point value from the second source to the low double-precision floating-point value in the third source operand, adds the infinite precision intermediate result to the low double-precision floating-point value in the first source operand, performs rounding and stores the resulting double-precision floating-point value to the destination operand (first source operand).

VEX.128 and EVEX encoded version: The destination operand (also first source operand) is encoded in reg_field. The second source operand is encoded in VEX.vvvv/EVEX.vvvv. The third source operand is encoded in rm_field. Bits 127:64 of the destination are unchanged. Bits MAXVL-1:128 of the destination register are zeroed.

EVEX encoded version: The low quadword element of the destination is updated according to the writemask.

Operation

In the operations below, "*" and "+" symbols represent multiplication and addition with infinite precision inputs and outputs (no rounding).

VFMADD132SD DEST, SRC2, SRC3 (EVEX encoded version)

```
IF (EVEX.b = 1) and SRC3 *is a register*
    THEN
        SET_RM(EVEX.RC);
    ELSE
        SET_RM(MXCSR.RM);
FI;
IF k1[0] or *no writemask*
    THEN    DEST[63:0] ← RoundFPControl(DEST[63:0]*SRC3[63:0] + SRC2[63:0])
    ELSE
        IF *merging-masking*                ; merging-masking
            THEN *DEST[63:0] remains unchanged*
            ELSE                            ; zeroing-masking
                THEN DEST[63:0] ← 0
        FI;
FI;
DEST[127:64] ← DEST[127:64]
DEST[MAXVL-1:128] ← 0
```

VFMADD213SD DEST, SRC2, SRC3 (EVEX encoded version)

```
IF (EVEX.b = 1) and SRC3 *is a register*
    THEN
        SET_RM(EVEX.RC);
    ELSE
        SET_RM(MXCSR.RM);
FI;
IF k1[0] or *no writemask*
    THEN    DEST[63:0] ← RoundFPControl(SRC2[63:0]*DEST[63:0] + SRC3[63:0])
    ELSE
        IF *merging-masking*                ; merging-masking
            THEN *DEST[63:0] remains unchanged*
            ELSE                            ; zeroing-masking
                THEN DEST[63:0] ← 0
        FI;
FI;
DEST[127:64] ← DEST[127:64]
DEST[MAXVL-1:128] ← 0
```

VFMADD231SD DEST, SRC2, SRC3 (EVEX encoded version)

```
IF (EVEX.b = 1) and SRC3 *is a register*
    THEN
        SET_RM(EVEX.RC);
    ELSE
        SET_RM(MXCSR.RM);
FI;
IF k1[0] or *no writemask*
    THEN    DEST[63:0] ← RoundFPControl(SRC2[63:0]*SRC3[63:0] + DEST[63:0])
    ELSE
        IF *merging-masking*                    ; merging-masking
            THEN *DEST[63:0] remains unchanged*
            ELSE                                ; zeroing-masking
                THEN DEST[63:0] ← 0
        FI;
FI;
DEST[127:64] ← DEST[127:64]
DEST[MAXVL-1:128] ← 0
```

VFMADD132SD DEST, SRC2, SRC3 (VEX encoded version)

```
DEST[63:0] ← MAXVL-1:128RoundFPControl_MXCSR(DEST[63:0]*SRC3[63:0] + SRC2[63:0])
DEST[127:63] ← DEST[127:63]
DEST[MAXVL-1:128] ← 0
```

VFMADD213SD DEST, SRC2, SRC3 (VEX encoded version)

```
DEST[63:0] ← RoundFPControl_MXCSR(SRC2[63:0]*DEST[63:0] + SRC3[63:0])
DEST[127:63] ← DEST[127:63]
DEST[MAXVL-1:128] ← 0
```

VFMADD231SD DEST, SRC2, SRC3 (VEX encoded version)

```
DEST[63:0] ← RoundFPControl_MXCSR(SRC2[63:0]*SRC3[63:0] + DEST[63:0])
DEST[127:63] ← DEST[127:63]
DEST[MAXVL-1:128] ← 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VFMADDxxxSD __m128d _mm_fmadd_round_sd(__m128d a, __m128d b, __m128d c, int r);
VFMADDxxxSD __m128d _mm_mask_fmadd_sd(__m128d a, __mmask8 k, __m128d b, __m128d c);
VFMADDxxxSD __m128d _mm_maskz_fmadd_sd(__mmask8 k, __m128d a, __m128d b, __m128d c);
VFMADDxxxSD __m128d _mm_mask3_fmadd_sd(__m128d a, __m128d b, __m128d c, __mmask8 k);
VFMADDxxxSD __m128d _mm_mask_fmadd_round_sd(__m128d a, __mmask8 k, __m128d b, __m128d c, int r);
VFMADDxxxSD __m128d _mm_maskz_fmadd_round_sd(__mmask8 k, __m128d a, __m128d b, __m128d c, int r);
VFMADDxxxSD __m128d _mm_mask3_fmadd_round_sd(__m128d a, __m128d b, __m128d c, __mmask8 k, int r);
VFMADDxxxSD __m128d _mm_fmadd_sd (__m128d a, __m128d b, __m128d c);

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal

Other Exceptions

VEX-encoded instructions, see Exceptions Type 3.
EVEX-encoded instructions, see Exceptions Type E3.

VFMADD132SS/VFMADD213SS/VFMADD231SS—Fused Multiply-Add of Scalar Single-Precision Floating-Point Values

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
VEX.DDS.LIG.66.0F38.W0 99 /r VFMADD132SS xmm1, xmm2, xmm3/m32	A	V/V	FMA	Multiply scalar single-precision floating-point value from xmm1 and xmm3/m32, add to xmm2 and put result in xmm1.
VEX.DDS.LIG.66.0F38.W0 A9 /r VFMADD213SS xmm1, xmm2, xmm3/m32	A	V/V	FMA	Multiply scalar single-precision floating-point value from xmm1 and xmm2, add to xmm3/m32 and put result in xmm1.
VEX.DDS.LIG.66.0F38.W0 B9 /r VFMADD231SS xmm1, xmm2, xmm3/m32	A	V/V	FMA	Multiply scalar single-precision floating-point value from xmm2 and xmm3/m32, add to xmm1 and put result in xmm1.
EVEX.DDS.LIG.66.0F38.W0 99 /r VFMADD132SS xmm1 {k1}{z}, xmm2, xmm3/m32{er}	B	V/V	AVX512F	Multiply scalar single-precision floating-point value from xmm1 and xmm3/m32, add to xmm2 and put result in xmm1.
EVEX.DDS.LIG.66.0F38.W0 A9 /r VFMADD213SS xmm1 {k1}{z}, xmm2, xmm3/m32{er}	B	V/V	AVX512F	Multiply scalar single-precision floating-point value from xmm1 and xmm2, add to xmm3/m32 and put result in xmm1.
EVEX.DDS.LIG.66.0F38.W0 B9 /r VFMADD231SS xmm1 {k1}{z}, xmm2, xmm3/m32{er}	B	V/V	AVX512F	Multiply scalar single-precision floating-point value from xmm2 and xmm3/m32, add to xmm1 and put result in xmm1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	NA	ModRM:reg (r, w)	VEX.vvvv (r)	ModRM:r/m (r)	NA
B	Tuple1 Scalar	ModRM:reg (r, w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA

Description

Performs a SIMD multiply-add computation on single-precision floating-point values using three source operands and writes the multiply-add results in the destination operand. The destination operand is also the first source operand. The first and second operands are XMM registers. The third source operand can be a XMM register or a 32-bit memory location.

VFMADD132SS: Multiplies the low single-precision floating-point value from the first source operand to the low single-precision floating-point value in the third source operand, adds the infinite precision intermediate result to the low single-precision floating-point value in the second source operand, performs rounding and stores the resulting single-precision floating-point value to the destination operand (first source operand).

VFMADD213SS: Multiplies the low single-precision floating-point value from the second source operand to the low single-precision floating-point value in the first source operand, adds the infinite precision intermediate result to the low single-precision floating-point value in the third source operand, performs rounding and stores the resulting single-precision floating-point value to the destination operand (first source operand).

VFMADD231SS: Multiplies the low single-precision floating-point value from the second source operand to the low single-precision floating-point value in the third source operand, adds the infinite precision intermediate result to the low single-precision floating-point value in the first source operand, performs rounding and stores the resulting single-precision floating-point value to the destination operand (first source operand).

VEX.128 and EVEX encoded version: The destination operand (also first source operand) is encoded in reg_field. The second source operand is encoded in VEX.vvvv/EVEX.vvvv. The third source operand is encoded in rm_field. Bits 127:32 of the destination are unchanged. Bits MAXVL-1:128 of the destination register are zeroed.

EVEX encoded version: The low doubleword element of the destination is updated according to the writemask.

Compiler tools may optionally support a complementary mnemonic for each instruction mnemonic listed in the opcode/instruction column of the summary table. The behavior of the complementary mnemonic in situations involving NANs are governed by the definition of the instruction mnemonic defined in the opcode/instruction column.

Operation

In the operations below, "*" and "+" symbols represent multiplication and addition with infinite precision inputs and outputs (no rounding).

VFMADD132SS DEST, SRC2, SRC3 (EVEX encoded version)

```
IF (EVEX.b = 1) and SRC3 *is a register*
    THEN
        SET_RM(EVEX.RC);
    ELSE
        SET_RM(MXCSR.RM);
FI;
IF k1[0] or *no writemask*
    THEN    DEST[31:0] ← RoundFPControl(DEST[31:0]*SRC3[31:0] + SRC2[31:0])
    ELSE
        IF *merging-masking*                ; merging-masking
            THEN *DEST[31:0] remains unchanged*
            ELSE                            ; zeroing-masking
                THEN DEST[31:0] ← 0
        FI;
FI;
DEST[127:32] ← DEST[127:32]
DEST[MAXVL-1:128] ← 0
```

VFMADD213SS DEST, SRC2, SRC3 (EVEX encoded version)

```
IF (EVEX.b = 1) and SRC3 *is a register*
    THEN
        SET_RM(EVEX.RC);
    ELSE
        SET_RM(MXCSR.RM);
FI;
IF k1[0] or *no writemask*
    THEN    DEST[31:0] ← RoundFPControl(SRC2[31:0]*DEST[31:0] + SRC3[31:0])
    ELSE
        IF *merging-masking*                ; merging-masking
            THEN *DEST[31:0] remains unchanged*
            ELSE                            ; zeroing-masking
                THEN DEST[31:0] ← 0
        FI;
FI;
DEST[127:32] ← DEST[127:32]
DEST[MAXVL-1:128] ← 0
```

VFMADD231SS DEST, SRC2, SRC3 (EVEX encoded version)

```
IF (EVEX.b = 1) and SRC3 *is a register*
    THEN
        SET_RM(EVEX.RC);
    ELSE
        SET_RM(MXCSR.RM);
FI;
IF k1[0] or *no writemask*
    THEN    DEST[31:0] ← RoundFPControl(SRC2[31:0]*SRC3[31:0] + DEST[31:0])
    ELSE
        IF *merging-masking*                ; merging-masking
            THEN *DEST[31:0]] remains unchanged*
            ELSE                            ; zeroing-masking
                THEN DEST[31:0] ← 0
        FI;
FI;
DEST[127:32] ← DEST[127:32]
DEST[MAXVL-1:128] ← 0
```

VFMADD132SS DEST, SRC2, SRC3 (VEX encoded version)

```
DEST[31:0] ←RoundFPControl_MXCSR(DEST[31:0]*SRC3[31:0] + SRC2[31:0])
DEST[127:32] ←DEST[127:32]
DEST[MAXVL-1:128] ←0
```

VFMADD213SS DEST, SRC2, SRC3 (VEX encoded version)

```
DEST[31:0] ←RoundFPControl_MXCSR(SRC2[31:0]*DEST[31:0] + SRC3[31:0])
DEST[127:32] ←DEST[127:32]
DEST[MAXVL-1:128] ←0
```

VFMADD231SS DEST, SRC2, SRC3 (VEX encoded version)

```
DEST[31:0] ←RoundFPControl_MXCSR(SRC2[31:0]*SRC3[31:0] + DEST[31:0])
DEST[127:32] ←DEST[127:32]
DEST[MAXVL-1:128] ←0
```

Intel C/C++ Compiler Intrinsic Equivalent

VFMADDxxxSS __m128 _mm_fmadd_round_ss(__m128 a, __m128 b, __m128 c, int r);
VFMADDxxxSS __m128 _mm_mask_fmadd_ss(__m128 a, __mmask8 k, __m128 b, __m128 c);
VFMADDxxxSS __m128 _mm_maskz_fmadd_ss(__mmask8 k, __m128 a, __m128 b, __m128 c);
VFMADDxxxSS __m128 _mm_mask3_fmadd_ss(__m128 a, __m128 b, __m128 c, __mmask8 k);
VFMADDxxxSS __m128 _mm_mask_fmadd_round_ss(__m128 a, __mmask8 k, __m128 b, __m128 c, int r);
VFMADDxxxSS __m128 _mm_maskz_fmadd_round_ss(__mmask8 k, __m128 a, __m128 b, __m128 c, int r);
VFMADDxxxSS __m128 _mm_mask3_fmadd_round_ss(__m128 a, __m128 b, __m128 c, __mmask8 k, int r);
VFMADDxxxSS __m128 _mm_fmadd_ss (__m128 a, __m128 b, __m128 c);

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal

Other Exceptions

VEX-encoded instructions, see Exceptions Type 3.

EVEX-encoded instructions, see Exceptions Type E3.

VFMADDSUB132PD/VFMADDSUB213PD/VFMADDSUB231PD—Fused Multiply-Alternating Add/Subtract of Packed Double-Precision Floating-Point Values

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
VEX.DDS.128.66.0F38.W1 96 /r VFMADDSUB132PD xmm1, xmm2, xmm3/m128	A	V/V	FMA	Multiply packed double-precision floating-point values from xmm1 and xmm3/mem, add/subtract elements in xmm2 and put result in xmm1.
VEX.DDS.128.66.0F38.W1 A6 /r VFMADDSUB213PD xmm1, xmm2, xmm3/m128	A	V/V	FMA	Multiply packed double-precision floating-point values from xmm1 and xmm2, add/subtract elements in xmm3/mem and put result in xmm1.
VEX.DDS.128.66.0F38.W1 B6 /r VFMADDSUB231PD xmm1, xmm2, xmm3/m128	A	V/V	FMA	Multiply packed double-precision floating-point values from xmm2 and xmm3/mem, add/subtract elements in xmm1 and put result in xmm1.
VEX.DDS.256.66.0F38.W1 96 /r VFMADDSUB132PD ymm1, ymm2, ymm3/m256	A	V/V	FMA	Multiply packed double-precision floating-point values from ymm1 and ymm3/mem, add/subtract elements in ymm2 and put result in ymm1.
VEX.DDS.256.66.0F38.W1 A6 /r VFMADDSUB213PD ymm1, ymm2, ymm3/m256	A	V/V	FMA	Multiply packed double-precision floating-point values from ymm1 and ymm2, add/subtract elements in ymm3/mem and put result in ymm1.
VEX.DDS.256.66.0F38.W1 B6 /r VFMADDSUB231PD ymm1, ymm2, ymm3/m256	A	V/V	FMA	Multiply packed double-precision floating-point values from ymm2 and ymm3/mem, add/subtract elements in ymm1 and put result in ymm1.
EVEX.DDS.128.66.0F38.W1 A6 /r VFMADDSUB213PD xmm1 {k1}{z}, xmm2, xmm3/m128/m64bcst	B	V/V	AVX512VL AVX512F	Multiply packed double-precision floating-point values from xmm1 and xmm2, add/subtract elements in xmm3/m128/m64bcst and put result in xmm1 subject to writemask k1.
EVEX.DDS.128.66.0F38.W1 B6 /r VFMADDSUB231PD xmm1 {k1}{z}, xmm2, xmm3/m128/m64bcst	B	V/V	AVX512VL AVX512F	Multiply packed double-precision floating-point values from xmm2 and xmm3/m128/m64bcst, add/subtract elements in xmm1 and put result in xmm1 subject to writemask k1.
EVEX.DDS.128.66.0F38.W1 96 /r VFMADDSUB132PD xmm1 {k1}{z}, xmm2, xmm3/m128/m64bcst	B	V/V	AVX512VL AVX512F	Multiply packed double-precision floating-point values from xmm1 and xmm3/m128/m64bcst, add/subtract elements in xmm2 and put result in xmm1 subject to writemask k1.
EVEX.DDS.256.66.0F38.W1 A6 /r VFMADDSUB213PD ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst	B	V/V	AVX512VL AVX512F	Multiply packed double-precision floating-point values from ymm1 and ymm2, add/subtract elements in ymm3/m256/m64bcst and put result in ymm1 subject to writemask k1.
EVEX.DDS.256.66.0F38.W1 B6 /r VFMADDSUB231PD ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst	B	V/V	AVX512VL AVX512F	Multiply packed double-precision floating-point values from ymm2 and ymm3/m256/m64bcst, add/subtract elements in ymm1 and put result in ymm1 subject to writemask k1.
EVEX.DDS.256.66.0F38.W1 96 /r VFMADDSUB132PD ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst	B	V/V	AVX512VL AVX512F	Multiply packed double-precision floating-point values from ymm1 and ymm3/m256/m64bcst, add/subtract elements in ymm2 and put result in ymm1 subject to writemask k1.

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.DDS.512.66.0F38.W1 A6 /r VFMADDSUB213PD zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst{er}	B	V/V	AVX512F	Multiply packed double-precision floating-point values from zmm1and zmm2, add/subtract elements in zmm3/m512/m64bcst and put result in zmm1 subject to writemask k1.
EVEX.DDS.512.66.0F38.W1 B6 /r VFMADDSUB231PD zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst{er}	B	V/V	AVX512F	Multiply packed double-precision floating-point values from zmm2 and zmm3/m512/m64bcst, add/subtract elements in zmm1 and put result in zmm1 subject to writemask k1.
EVEX.DDS.512.66.0F38.W1 96 /r VFMADDSUB132PD zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst{er}	B	V/V	AVX512F	Multiply packed double-precision floating-point values from zmm1 and zmm3/m512/m64bcst, add/subtract elements in zmm2 and put result in zmm1 subject to writemask k1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	NA	ModRM:reg (r, w)	VEX.vvvv (r)	ModRM:r/m (r)	NA
B	Full Vector	ModRM:reg (r, w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA

Description

VFMADDSUB132PD: Multiplies the two, four, or eight packed double-precision floating-point values from the first source operand to the two or four packed double-precision floating-point values in the third source operand. From the infinite precision intermediate result, adds the odd double-precision floating-point elements and subtracts the even double-precision floating-point values in the second source operand, performs rounding and stores the resulting two or four packed double-precision floating-point values to the destination operand (first source operand).

VFMADDSUB213PD: Multiplies the two, four, or eight packed double-precision floating-point values from the second source operand to the two or four packed double-precision floating-point values in the first source operand. From the infinite precision intermediate result, adds the odd double-precision floating-point elements and subtracts the even double-precision floating-point values in the third source operand, performs rounding and stores the resulting two or four packed double-precision floating-point values to the destination operand (first source operand).

VFMADDSUB231PD: Multiplies the two, four, or eight packed double-precision floating-point values from the second source operand to the two or four packed double-precision floating-point values in the third source operand. From the infinite precision intermediate result, adds the odd double-precision floating-point elements and subtracts the even double-precision floating-point values in the first source operand, performs rounding and stores the resulting two or four packed double-precision floating-point values to the destination operand (first source operand).

EVEX encoded versions: The destination operand (also first source operand) and the second source operand are ZMM/YMM/XMM register. The third source operand is a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 64-bit memory location. The destination operand is conditionally updated with write mask k1.

VEX.256 encoded version: The destination operand (also first source operand) is a YMM register and encoded in reg_field. The second source operand is a YMM register and encoded in VEX.vvvv. The third source operand is a YMM register or a 256-bit memory location and encoded in rm_field.

VEX.128 encoded version: The destination operand (also first source operand) is a XMM register and encoded in reg_field. The second source operand is a XMM register and encoded in VEX.vvvv. The third source operand is a XMM register or a 128-bit memory location and encoded in rm_field. The upper 128 bits of the YMM destination register are zeroed.

Compiler tools may optionally support a complementary mnemonic for each instruction mnemonic listed in the opcode/instruction column of the summary table. The behavior of the complementary mnemonic in situations involving NANs are governed by the definition of the instruction mnemonic defined in the opcode/instruction column.

Operation

In the operations below, "*" and "-" symbols represent multiplication and subtraction with infinite precision inputs and outputs (no rounding).

VFMADDSUB132PD DEST, SRC2, SRC3

```
IF (VEX.128) THEN
    DEST[63:0] ←RoundFPControl_MXCSR(DEST[63:0]*SRC3[63:0] - SRC2[63:0])
    DEST[127:64] ←RoundFPControl_MXCSR(DEST[127:64]*SRC3[127:64] + SRC2[127:64])
    DEST[MAXVL-1:128] ←0
ELSEIF (VEX.256)
    DEST[63:0] ←RoundFPControl_MXCSR(DEST[63:0]*SRC3[63:0] - SRC2[63:0])
    DEST[127:64] ←RoundFPControl_MXCSR(DEST[127:64]*SRC3[127:64] + SRC2[127:64])
    DEST[191:128] ←RoundFPControl_MXCSR(DEST[191:128]*SRC3[191:128] - SRC2[191:128])
    DEST[255:192] ←RoundFPControl_MXCSR(DEST[255:192]*SRC3[255:192] + SRC2[255:192]
FI
```

VFMADDSUB213PD DEST, SRC2, SRC3

```
IF (VEX.128) THEN
    DEST[63:0] ←RoundFPControl_MXCSR(SRC2[63:0]*DEST[63:0] - SRC3[63:0])
    DEST[127:64] ←RoundFPControl_MXCSR(SRC2[127:64]*DEST[127:64] + SRC3[127:64])
    DEST[MAXVL-1:128] ←0
ELSEIF (VEX.256)
    DEST[63:0] ←RoundFPControl_MXCSR(SRC2[63:0]*DEST[63:0] - SRC3[63:0])
    DEST[127:64] ←RoundFPControl_MXCSR(SRC2[127:64]*DEST[127:64] + SRC3[127:64])
    DEST[191:128] ←RoundFPControl_MXCSR(SRC2[191:128]*DEST[191:128] - SRC3[191:128])
    DEST[255:192] ←RoundFPControl_MXCSR(SRC2[255:192]*DEST[255:192] + SRC3[255:192]
FI
```

VFMADDSUB231PD DEST, SRC2, SRC3

```
IF (VEX.128) THEN
    DEST[63:0] ←RoundFPControl_MXCSR(SRC2[63:0]*SRC3[63:0] - DEST[63:0])
    DEST[127:64] ←RoundFPControl_MXCSR(SRC2[127:64]*SRC3[127:64] + DEST[127:64])
    DEST[MAXVL-1:128] ←0
ELSEIF (VEX.256)
    DEST[63:0] ←RoundFPControl_MXCSR(SRC2[63:0]*SRC3[63:0] - DEST[63:0])
    DEST[127:64] ←RoundFPControl_MXCSR(SRC2[127:64]*SRC3[127:64] + DEST[127:64])
    DEST[191:128] ←RoundFPControl_MXCSR(SRC2[191:128]*SRC3[191:128] - DEST[191:128])
    DEST[255:192] ←RoundFPControl_MXCSR(SRC2[255:192]*SRC3[255:192] + DEST[255:192]
FI
```

VFMADDSUB132PD DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a register)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
IF (VL = 512) AND (EVEX.b = 1)
    THEN
        SET_RM(EVEX.RC);
    ELSE
        SET_RM(MXCSR.RM);
FI;
FOR j ← 0 TO KL-1
    i ← j * 64
    IF k1[j] OR *no writemask*
        THEN
            IF j *is even*
                THEN DEST[i+63:i] ←
                    RoundFPControl(DEST[i+63:i]*SRC3[i+63:i] - SRC2[i+63:i])
                ELSE DEST[i+63:i] ←
                    RoundFPControl(DEST[i+63:i]*SRC3[i+63:i] + SRC2[i+63:i])
            FI
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE                                ; zeroing-masking
                    DEST[i+63:i] ← 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0
```

VFMADDSUB132PD DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a memory source)
(KL, VL) = (2, 128), (4, 256), (8, 512)

```
FOR j ← 0 TO KL-1
    i ← j * 64
    IF k1[j] OR *no writemask*
        THEN
            IF j *is even*
                THEN
                    IF (EVEX.b = 1)
                        THEN
                            DEST[i+63:i] ←
                    RoundFPControl_MXCSR(DEST[i+63:i]*SRC3[63:0] - SRC2[i+63:i])
                        ELSE
                            DEST[i+63:i] ←
                    RoundFPControl_MXCSR(DEST[i+63:i]*SRC3[i+63:i] - SRC2[i+63:i])
                FI;
                ELSE
                    IF (EVEX.b = 1)
                        THEN
                            DEST[i+63:i] ←
                    RoundFPControl_MXCSR(DEST[i+63:i]*SRC3[63:0] + SRC2[i+63:i])
                        ELSE
                            DEST[i+63:i] ←
                    RoundFPControl_MXCSR(DEST[i+63:i]*SRC3[i+63:i] + SRC2[i+63:i])
                FI;
            FI

        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE                                ; zeroing-masking
                    DEST[i+63:i] ← 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0
```

VFMADDSUB213PD DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a register)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
IF (VL = 512) AND (EVEX.b = 1)
    THEN
        SET_RM(EVEX.RC);
    ELSE
        SET_RM(MXCSR.RM);
FI;
FOR j ← 0 TO KL-1
    i ← j * 64
    IF k1[j] OR *no writemask*
        THEN
            IF j *is even*
                THEN DEST[i+63:i] ←
                    RoundFPControl(SRC2[i+63:i]*DEST[i+63:i] - SRC3[i+63:i])
                ELSE DEST[i+63:i] ←
                    RoundFPControl(SRC2[i+63:i]*DEST[i+63:i] + SRC3[i+63:i])
            FI
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE                                ; zeroing-masking
                    DEST[i+63:i] ← 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0
```

VFMADDSUB213PD DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a memory source)
(KL, VL) = (2, 128), (4, 256), (8, 512)

```
FOR j ← 0 TO KL-1
    i ← j * 64
    IF k1[j] OR *no writemask*
        THEN
            IF j *is even*
                THEN
                    IF (EVEX.b = 1)
                        THEN
                            DEST[i+63:i] ←
                    RoundFPControl_MXCSR(SRC2[i+63:i]*DEST[i+63:i] - SRC3[63:0])
                        ELSE
                            DEST[i+63:i] ←
                    RoundFPControl_MXCSR(SRC2[i+63:i]*DEST[i+63:i] - SRC3[i+63:i])
                    FI;
                ELSE
                    IF (EVEX.b = 1)
                        THEN
                            DEST[i+63:i] ←
                    RoundFPControl_MXCSR(SRC2[i+63:i]*DEST[i+63:i] + SRC3[63:0])
                        ELSE
                            DEST[i+63:i] ←
                    RoundFPControl_MXCSR(SRC2[i+63:i]*DEST[i+63:i] + SRC3[i+63:i])
                    FI;
            FI
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE                                ; zeroing-masking
                    DEST[i+63:i] ← 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0
```

VFMADDSUB231PD DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a register)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
IF (VL = 512) AND (EVEX.b = 1)
    THEN
        SET_RM(EVEX.RC);
    ELSE
        SET_RM(MXCSR.RM);
FI;
FOR j ← 0 TO KL-1
    i ← j * 64
    IF k1[j] OR *no writemask*
        THEN
            IF j *is even*
                THEN DEST[i+63:i] ←
                    RoundFPControl(SRC2[i+63:i]*SRC3[i+63:i] - DEST[i+63:i])
                ELSE DEST[i+63:i] ←
                    RoundFPControl(SRC2[i+63:i]*SRC3[i+63:i] + DEST[i+63:i])
            FI
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE                            ; zeroing-masking
                    DEST[i+63:i] ← 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0
```

VFMADDSUB231PD DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a memory source)
(KL, VL) = (2, 128), (4, 256), (8, 512)

```
FOR j ← 0 TO KL-1
    i ← j * 64
    IF k1[j] OR *no writemask*
        THEN
            IF j *is even*
                THEN
                    IF (EVEX.b = 1)
                        THEN
                            DEST[i+63:i] ←
                        RoundFPControl_MXCSR(SRC2[i+63:i]*SRC3[63:0] - DEST[i+63:i])
                        ELSE
                            DEST[i+63:i] ←
                        RoundFPControl_MXCSR(SRC2[i+63:i]*SRC3[i+63:i] - DEST[i+63:i])
                    FI;
                ELSE
                    IF (EVEX.b = 1)
                        THEN
                            DEST[i+63:i] ←
                        RoundFPControl_MXCSR(SRC2[i+63:i]*SRC3[63:0] + DEST[i+63:i])
                        ELSE
                            DEST[i+63:i] ←
                        RoundFPControl_MXCSR(SRC2[i+63:i]*SRC3[i+63:i] + DEST[i+63:i])
                    FI;
            FI
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE                                ; zeroing-masking
                    DEST[i+63:i] ← 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VFMADDSUBxxxPD __m512d _mm512_fmaddsub_pd(__m512d a, __m512d b, __m512d c);
VFMADDSUBxxxPD __m512d _mm512_fmaddsub_round_pd(__m512d a, __m512d b, __m512d c, int r);
VFMADDSUBxxxPD __m512d _mm512_mask_fmaddsub_pd(__m512d a, __mmask8 k, __m512d b, __m512d c);
VFMADDSUBxxxPD __m512d _mm512_maskz_fmaddsub_pd(__mmask8 k, __m512d a, __m512d b, __m512d c);
VFMADDSUBxxxPD __m512d _mm512_mask3_fmaddsub_pd(__m512d a, __m512d b, __m512d c, __mmask8 k);
VFMADDSUBxxxPD __m512d _mm512_mask_fmaddsub_round_pd(__m512d a, __mmask8 k, __m512d b, __m512d c, int r);
VFMADDSUBxxxPD __m512d _mm512_maskz_fmaddsub_round_pd(__mmask8 k, __m512d a, __m512d b, __m512d c, int r);
VFMADDSUBxxxPD __m512d _mm512_mask3_fmaddsub_round_pd(__m512d a, __m512d b, __m512d c, __mmask8 k, int r);
VFMADDSUBxxxPD __m256d _mm256_mask_fmaddsub_pd(__m256d a, __mmask8 k, __m256d b, __m256d c);
VFMADDSUBxxxPD __m256d _mm256_maskz_fmaddsub_pd(__mmask8 k, __m256d a, __m256d b, __m256d c);
VFMADDSUBxxxPD __m256d _mm256_mask3_fmaddsub_pd(__m256d a, __m256d b, __m256d c, __mmask8 k);
VFMADDSUBxxxPD __m128d _mm_mask_fmaddsub_pd(__m128d a, __mmask8 k, __m128d b, __m128d c);
VFMADDSUBxxxPD __m128d _mm_maskz_fmaddsub_pd(__mmask8 k, __m128d a, __m128d b, __m128d c);
VFMADDSUBxxxPD __m128d _mm_mask3_fmaddsub_pd(__m128d a, __m128d b, __m128d c, __mmask8 k);
VFMADDSUBxxxPD __m128d _mm_fmaddsub_pd (__m128d a, __m128d b, __m128d c);
VFMADDSUBxxxPD __m256d _mm256_fmaddsub_pd (__m256d a, __m256d b, __m256d c);

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal

Other Exceptions

VEX-encoded instructions, see Exceptions Type 2.
EVEX-encoded instructions, see Exceptions Type E2.

VFMADDSUB132PS/VFMADDSUB213PS/VFMADDSUB231PS—Fused Multiply-Alternating Add/Subtract of Packed Single-Precision Floating-Point Values

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
VEX.DDS.128.66.0F38.W0 96 /r VFMADDSUB132PS xmm1, xmm2, xmm3/m128	A	V/V	FMA	Multiply packed single-precision floating-point values from xmm1 and xmm3/mem, add/subtract elements in xmm2 and put result in xmm1.
VEX.DDS.128.66.0F38.W0 A6 /r VFMADDSUB213PS xmm1, xmm2, xmm3/m128	A	V/V	FMA	Multiply packed single-precision floating-point values from xmm1 and xmm2, add/subtract elements in xmm3/mem and put result in xmm1.
VEX.DDS.128.66.0F38.W0 B6 /r VFMADDSUB231PS xmm1, xmm2, xmm3/m128	A	V/V	FMA	Multiply packed single-precision floating-point values from xmm2 and xmm3/mem, add/subtract elements in xmm1 and put result in xmm1.
VEX.DDS.256.66.0F38.W0 96 /r VFMADDSUB132PS ymm1, ymm2, ymm3/m256	A	V/V	FMA	Multiply packed single-precision floating-point values from ymm1 and ymm3/mem, add/subtract elements in ymm2 and put result in ymm1.
VEX.DDS.256.66.0F38.W0 A6 /r VFMADDSUB213PS ymm1, ymm2, ymm3/m256	A	V/V	FMA	Multiply packed single-precision floating-point values from ymm1 and ymm2, add/subtract elements in ymm3/mem and put result in ymm1.
VEX.DDS.256.66.0F38.W0 B6 /r VFMADDSUB231PS ymm1, ymm2, ymm3/m256	A	V/V	FMA	Multiply packed single-precision floating-point values from ymm2 and ymm3/mem, add/subtract elements in ymm1 and put result in ymm1.
EVEX.DDS.128.66.0F38.W0 A6 /r VFMADDSUB213PS xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst	B	V/V	AVX512VL AVX512F	Multiply packed single-precision floating-point values from xmm1 and xmm2, add/subtract elements in xmm3/m128/m32bcst and put result in xmm1 subject to writemask k1.
EVEX.DDS.128.66.0F38.W0 B6 /r VFMADDSUB231PS xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst	B	V/V	AVX512VL AVX512F	Multiply packed single-precision floating-point values from xmm2 and xmm3/m128/m32bcst, add/subtract elements in xmm1 and put result in xmm1 subject to writemask k1.
EVEX.DDS.128.66.0F38.W0 96 /r VFMADDSUB132PS xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst	B	V/V	AVX512VL AVX512F	Multiply packed single-precision floating-point values from xmm1 and xmm3/m128/m32bcst, add/subtract elements in zmm2 and put result in xmm1 subject to writemask k1.
EVEX.DDS.256.66.0F38.W0 A6 /r VFMADDSUB213PS ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst	B	V/V	AVX512VL AVX512F	Multiply packed single-precision floating-point values from ymm1 and ymm2, add/subtract elements in ymm3/m256/m32bcst and put result in ymm1 subject to writemask k1.
EVEX.DDS.256.66.0F38.W0 B6 /r VFMADDSUB231PS ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst	B	V/V	AVX512VL AVX512F	Multiply packed single-precision floating-point values from ymm2 and ymm3/m256/m32bcst, add/subtract elements in ymm1 and put result in ymm1 subject to writemask k1.
EVEX.DDS.256.66.0F38.W0 96 /r VFMADDSUB132PS ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst	B	V/V	AVX512VL AVX512F	Multiply packed single-precision floating-point values from ymm1 and ymm3/m256/m32bcst, add/subtract elements in ymm2 and put result in ymm1 subject to writemask k1.
EVEX.DDS.512.66.0F38.W0 A6 /r VFMADDSUB213PS zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst{er}	B	V/V	AVX512F	Multiply packed single-precision floating-point values from zmm1 and zmm2, add/subtract elements in zmm3/m512/m32bcst and put result in zmm1 subject to writemask k1.
EVEX.DDS.512.66.0F38.W0 B6 /r VFMADDSUB231PS zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst{er}	B	V/V	AVX512F	Multiply packed single-precision floating-point values from zmm2 and zmm3/m512/m32bcst, add/subtract elements in zmm1 and put result in zmm1 subject to writemask k1.
EVEX.DDS.512.66.0F38.W0 96 /r VFMADDSUB132PS zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst{er}	B	V/V	AVX512F	Multiply packed single-precision floating-point values from zmm1 and zmm3/m512/m32bcst, add/subtract elements in zmm2 and put result in zmm1 subject to writemask k1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	NA	ModRM:reg (r, w)	VEX.vvvv (r)	ModRM:r/m (r)	NA
B	Full Vector	ModRM:reg (r, w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA

Description

VFMADDSUB132PS: Multiplies the four, eight or sixteen packed single-precision floating-point values from the first source operand to the corresponding packed single-precision floating-point values in the third source operand. From the infinite precision intermediate result, adds the odd single-precision floating-point elements and subtracts the even single-precision floating-point values in the second source operand, performs rounding and stores the resulting packed single-precision floating-point values to the destination operand (first source operand).

VFMADDSUB213PS: Multiplies the four, eight or sixteen packed single-precision floating-point values from the second source operand to the corresponding packed single-precision floating-point values in the first source operand. From the infinite precision intermediate result, adds the odd single-precision floating-point elements and subtracts the even single-precision floating-point values in the third source operand, performs rounding and stores the resulting packed single-precision floating-point values to the destination operand (first source operand).

VFMADDSUB231PS: Multiplies the four, eight or sixteen packed single-precision floating-point values from the second source operand to the corresponding packed single-precision floating-point values in the third source operand. From the infinite precision intermediate result, adds the odd single-precision floating-point elements and subtracts the even single-precision floating-point values in the first source operand, performs rounding and stores the resulting packed single-precision floating-point values to the destination operand (first source operand).

EVEX encoded versions: The destination operand (also first source operand) and the second source operand are ZMM/YMM/XMM register. The third source operand is a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 32-bit memory location. The destination operand is conditionally updated with write mask k1.

VEX.256 encoded version: The destination operand (also first source operand) is a YMM register and encoded in reg_field. The second source operand is a YMM register and encoded in VEX.vvvv. The third source operand is a YMM register or a 256-bit memory location and encoded in rm_field.

VEX.128 encoded version: The destination operand (also first source operand) is a XMM register and encoded in reg_field. The second source operand is a XMM register and encoded in VEX.vvvv. The third source operand is a XMM register or a 128-bit memory location and encoded in rm_field. The upper 128 bits of the YMM destination register are zeroed.

Compiler tools may optionally support a complementary mnemonic for each instruction mnemonic listed in the opcode/instruction column of the summary table. The behavior of the complementary mnemonic in situations involving NANs are governed by the definition of the instruction mnemonic defined in the opcode/instruction column.

Operation

In the operations below, "*" and "+" symbols represent multiplication and addition with infinite precision inputs and outputs (no rounding).

VFMADDSUB132PS DEST, SRC2, SRC3

```
IF (VEX.128) THEN
    MAXNUM ←2
ELSEIF (VEX.256)
    MAXNUM ← 4
FI
For i = 0 to MAXNUM -1{
    n ← 64*i;
    DEST[n+31:n] ←RoundFPControl_MXCSR(DEST[n+31:n]*SRC3[n+31:n] - SRC2[n+31:n])
    DEST[n+63:n+32] ←RoundFPControl_MXCSR(DEST[n+63:n+32]*SRC3[n+63:n+32] + SRC2[n+63:n+32])
}
IF (VEX.128) THEN
    DEST[MAXVL-1:128] ←0
ELSEIF (VEX.256)
    DEST[MAXVL-1:256] ← 0
FI
```

VFMADDSUB213PS DEST, SRC2, SRC3

```
IF (VEX.128) THEN
    MAXNUM ←2
ELSEIF (VEX.256)
    MAXNUM ← 4
FI
For i = 0 to MAXNUM -1{
    n ← 64*i;
    DEST[n+31:n] ←RoundFPControl_MXCSR(SRC2[n+31:n]*DEST[n+31:n] - SRC3[n+31:n])
    DEST[n+63:n+32] ←RoundFPControl_MXCSR(SRC2[n+63:n+32]*DEST[n+63:n+32] + SRC3[n+63:n+32])
}
IF (VEX.128) THEN
    DEST[MAXVL-1:128] ←0
ELSEIF (VEX.256)
    DEST[MAXVL-1:256] ← 0
FI
```

VFMADDSUB231PS DEST, SRC2, SRC3

```
IF (VEX.128) THEN
    MAXNUM ←2
ELSEIF (VEX.256)
    MAXNUM ← 4
FI
For i = 0 to MAXNUM -1{
    n ← 64*i;
    DEST[n+31:n] ←RoundFPControl_MXCSR(SRC2[n+31:n]*SRC3[n+31:n] - DEST[n+31:n])
    DEST[n+63:n+32] ←RoundFPControl_MXCSR(SRC2[n+63:n+32]*SRC3[n+63:n+32] + DEST[n+63:n+32])
}
IF (VEX.128) THEN
    DEST[MAXVL-1:128] ←0
ELSEIF (VEX.256)
    DEST[MAXVL-1:256] ← 0
FI
```

VFMADDSUB132PS DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a register)

```
(KL, VL) (4, 128), (8, 256),= (16, 512)
IF (VL = 512) AND (EVEX.b = 1)
    THEN
        SET_RM(EVEX.RC);
    ELSE
        SET_RM(MXCSR.RM);
FI;
FOR j ← 0 TO KL-1
    i ← j * 32
    IF k1[j] OR *no writemask*
        THEN
            IF j *is even*
                THEN DEST[i+31:i] ←
                    RoundFPControl(DEST[i+31:i]*SRC3[i+31:i] - SRC2[i+31:i])
                ELSE DEST[i+31:i] ←
                    RoundFPControl(DEST[i+31:i]*SRC3[i+31:i] + SRC2[i+31:i])
            FI
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE                            ; zeroing-masking
                    DEST[i+31:i] ← 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0
```

VFMADDSUB132PS DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a memory source)
(KL, VL) = (4, 128), (8, 256), (16, 512)

```
FOR j ← 0 TO KL-1
    i ← j * 32
    IF k1[j] OR *no writemask*
        THEN
            IF j *is even*
                THEN
                    IF (EVEX.b = 1)
                        THEN
                            DEST[i+31:i] ←
                    RoundFPControl_MXCSR(DEST[i+31:i]*SRC3[31:0] - SRC2[i+31:i])
                        ELSE
                            DEST[i+31:i] ←
                    RoundFPControl_MXCSR(DEST[i+31:i]*SRC3[i+31:i] - SRC2[i+31:i])
                    FI;
                ELSE
                    IF (EVEX.b = 1)
                        THEN
                            DEST[i+31:i] ←
                    RoundFPControl_MXCSR(DEST[i+31:i]*SRC3[31:0] + SRC2[i+31:i])
                        ELSE
                            DEST[i+31:i] ←
                    RoundFPControl_MXCSR(DEST[i+31:i]*SRC3[i+31:i] + SRC2[i+31:i])
                    FI;
            FI

        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE                                ; zeroing-masking
                    DEST[i+31:i] ← 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0
```

VFMADDSUB213PS DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a register)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
IF (VL = 512) AND (EVEX.b = 1)
    THEN
        SET_RM(EVEX.RC);
    ELSE
        SET_RM(MXCSR.RM);
FI;
FOR j ← 0 TO KL-1
    i ← j * 32
    IF k1[j] OR *no writemask*
        THEN
            IF j *is even*
                THEN DEST[i+31:i] ←
                    RoundFPControl(SRC2[i+31:i]*DEST[i+31:i] - SRC3[i+31:i])
                ELSE DEST[i+31:i] ←
                    RoundFPControl(SRC2[i+31:i]*DEST[i+31:i] + SRC3[i+31:i])
            FI
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE                                ; zeroing-masking
                    DEST[i+31:i] ← 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0
```

VFMADDSUB213PS DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a memory source)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)

FOR j ← 0 TO KL-1
    i ← j * 32
    IF k1[j] OR *no writemask*
        THEN
            IF j *is even*
                THEN
                    IF (EVEX.b = 1)
                        THEN
                            DEST[i+31:i] ←
                    RoundFPControl_MXCSR(SRC2[i+31:i]*DEST[i+31:i] - SRC3[31:0])
                        ELSE
                            DEST[i+31:i] ←
                    RoundFPControl_MXCSR(SRC2[i+31:i]*DEST[i+31:i] - SRC3[i+31:i])
                    FI;
                ELSE
                    IF (EVEX.b = 1)
                        THEN
                            DEST[i+31:i] ←
                    RoundFPControl_MXCSR(SRC2[i+31:i]*DEST[i+31:i] + SRC3[31:0])
                        ELSE
                            DEST[i+31:i] ←
                    RoundFPControl_MXCSR(SRC2[i+31:i]*DEST[i+31:i] + SRC3[i+31:i])
                    FI;
```

```
                FI
            ELSE
                IF *merging-masking*                    ; merging-masking
                    THEN *DEST[i+31:i] remains unchanged*
                    ELSE                                ; zeroing-masking
                        DEST[i+31:i] ← 0
                FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0
```

VFMADDSUB231PS DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a register)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
IF (VL = 512) AND (EVEX.b = 1)
    THEN
        SET_RM(EVEX.RC);
    ELSE
        SET_RM(MXCSR.RM);
FI;
FOR j ← 0 TO KL-1
    i ← j * 32
    IF k1[j] OR *no writemask*
        THEN
            IF j *is even*
                THEN DEST[i+31:i] ←
                    RoundFPControl(SRC2[i+31:i]*SRC3[i+31:i] - DEST[i+31:i])
                ELSE DEST[i+31:i] ←
                    RoundFPControl(SRC2[i+31:i]*SRC3[i+31:i] + DEST[i+31:i])
            FI
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE                                ; zeroing-masking
                    DEST[i+31:i] ← 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0
```

VFMADDSUB231PS DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a memory source)
(KL, VL) = (4, 128), (8, 256), (16, 512)

```
FOR j ← 0 TO KL-1
    i ← j * 32
    IF k1[j] OR *no writemask*
        THEN
            IF j *is even*
                THEN
                    IF (EVEX.b = 1)
                        THEN
                            DEST[i+31:i] ←
                    RoundFPControl_MXCSR(SRC2[i+31:i]*SRC3[31:0] - DEST[i+31:i])
                        ELSE
                            DEST[i+31:i] ←
                    RoundFPControl_MXCSR(SRC2[i+31:i]*SRC3[i+31:i] - DEST[i+31:i])
                    FI;
                ELSE
                    IF (EVEX.b = 1)
                        THEN
                            DEST[i+31:i] ←
                    RoundFPControl_MXCSR(SRC2[i+31:i]*SRC3[31:0] + DEST[i+31:i])
                        ELSE
                            DEST[i+31:i] ←
                    RoundFPControl_MXCSR(SRC2[i+31:i]*SRC3[i+31:i] + DEST[i+31:i])
                    FI;
            FI
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE                                ; zeroing-masking
                    DEST[i+31:i] ← 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VFMADDSUBxxxPS __m512 _mm512_fmaddsub_ps(__m512 a, __m512 b, __m512 c);
VFMADDSUBxxxPS __m512 _mm512_fmaddsub_round_ps(__m512 a, __m512 b, __m512 c, int r);
VFMADDSUBxxxPS __m512 _mm512_mask_fmaddsub_ps(__m512 a, __mmask16 k, __m512 b, __m512 c);
VFMADDSUBxxxPS __m512 _mm512_maskz_fmaddsub_ps(__mmask16 k, __m512 a, __m512 b, __m512 c);
VFMADDSUBxxxPS __m512 _mm512_mask3_fmaddsub_ps(__m512 a, __m512 b, __m512 c, __mmask16 k);
VFMADDSUBxxxPS __m512 _mm512_mask_fmaddsub_round_ps(__m512 a, __mmask16 k, __m512 b, __m512 c, int r);
VFMADDSUBxxxPS __m512 _mm512_maskz_fmaddsub_round_ps(__mmask16 k, __m512 a, __m512 b, __m512 c, int r);
VFMADDSUBxxxPS __m512 _mm512_mask3_fmaddsub_round_ps(__m512 a, __m512 b, __m512 c, __mmask16 k, int r);
VFMADDSUBxxxPS __m256 _mm256_mask_fmaddsub_ps(__m256 a, __mmask8 k, __m256 b, __m256 c);
VFMADDSUBxxxPS __m256 _mm256_maskz_fmaddsub_ps(__mmask8 k, __m256 a, __m256 b, __m256 c);
VFMADDSUBxxxPS __m256 _mm256_mask3_fmaddsub_ps(__m256 a, __m256 b, __m256 c, __mmask8 k);
VFMADDSUBxxxPS __m128 _mm_mask_fmaddsub_ps(__m128 a, __mmask8 k, __m128 b, __m128 c);
VFMADDSUBxxxPS __m128 _mm_maskz_fmaddsub_ps(__mmask8 k, __m128 a, __m128 b, __m128 c);
VFMADDSUBxxxPS __m128 _mm_mask3_fmaddsub_ps(__m128 a, __m128 b, __m128 c, __mmask8 k);
VFMADDSUBxxxPS __m128 _mm_fmaddsub_ps (__m128 a, __m128 b, __m128 c);
VFMADDSUBxxxPS __m256 _mm256_fmaddsub_ps (__m256 a, __m256 b, __m256 c);

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal

Other Exceptions

VEX-encoded instructions, see Exceptions Type 2.
EVEX-encoded instructions, see Exceptions Type E2.

VFMSUBADD132PD/VFMSUBADD213PD/VFMSUBADD231PD—Fused Multiply-Alternating Subtract/Add of Packed Double-Precision Floating-Point Values

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
VEX.DDS.128.66.0F38.W1 97 /r VFMSUBADD132PD xmm1, xmm2, xmm3/m128	A	V/V	FMA	Multiply packed double-precision floating-point values from xmm1 and xmm3/mem, subtract/add elements in xmm2 and put result in xmm1.
VEX.DDS.128.66.0F38.W1 A7 /r VFMSUBADD213PD xmm1, xmm2, xmm3/m128	A	V/V	FMA	Multiply packed double-precision floating-point values from xmm1 and xmm2, subtract/add elements in xmm3/mem and put result in xmm1.
VEX.DDS.128.66.0F38.W1 B7 /r VFMSUBADD231PD xmm1, xmm2, xmm3/m128	A	V/V	FMA	Multiply packed double-precision floating-point values from xmm2 and xmm3/mem, subtract/add elements in xmm1 and put result in xmm1.
VEX.DDS.256.66.0F38.W1 97 /r VFMSUBADD132PD ymm1, ymm2, ymm3/m256	A	V/V	FMA	Multiply packed double-precision floating-point values from ymm1 and ymm3/mem, subtract/add elements in ymm2 and put result in ymm1.
VEX.DDS.256.66.0F38.W1 A7 /r VFMSUBADD213PD ymm1, ymm2, ymm3/m256	A	V/V	FMA	Multiply packed double-precision floating-point values from ymm1 and ymm2, subtract/add elements in ymm3/mem and put result in ymm1.
VEX.DDS.256.66.0F38.W1 B7 /r VFMSUBADD231PD ymm1, ymm2, ymm3/m256	A	V/V	FMA	Multiply packed double-precision floating-point values from ymm2 and ymm3/mem, subtract/add elements in ymm1 and put result in ymm1.
EVEX.DDS.128.66.0F38.W1 97 /r VFMSUBADD132PD xmm1 {k1}{z}, xmm2, xmm3/m128/m64bcst	B	V/V	AVX512VL AVX512F	Multiply packed double-precision floating-point values from xmm1 and xmm3/m128/m64bcst, subtract/add elements in xmm2 and put result in xmm1 subject to writemask k1.
EVEX.DDS.128.66.0F38.W1 A7 /r VFMSUBADD213PD xmm1 {k1}{z}, xmm2, xmm3/m128/m64bcst	B	V/V	AVX512VL AVX512F	Multiply packed double-precision floating-point values from xmm1 and xmm2, subtract/add elements in xmm3/m128/m64bcst and put result in xmm1 subject to writemask k1.
EVEX.DDS.128.66.0F38.W1 B7 /r VFMSUBADD231PD xmm1 {k1}{z}, xmm2, xmm3/m128/m64bcst	B	V/V	AVX512VL AVX512F	Multiply packed double-precision floating-point values from xmm2 and xmm3/m128/m64bcst, subtract/add elements in xmm1 and put result in xmm1 subject to writemask k1.
EVEX.DDS.256.66.0F38.W1 97 /r VFMSUBADD132PD ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst	B	V/V	AVX512VL AVX512F	Multiply packed double-precision floating-point values from ymm1 and ymm3/m256/m64bcst, subtract/add elements in ymm2 and put result in ymm1 subject to writemask k1.
EVEX.DDS.256.66.0F38.W1 A7 /r VFMSUBADD213PD ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst	B	V/V	AVX512VL AVX512F	Multiply packed double-precision floating-point values from ymm1 and ymm2, subtract/add elements in ymm3/m256/m64bcst and put result in ymm1 subject to writemask k1.
EVEX.DDS.256.66.0F38.W1 B7 /r VFMSUBADD231PD ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst	B	V/V	AVX512VL AVX512F	Multiply packed double-precision floating-point values from ymm2 and ymm3/m256/m64bcst, subtract/add elements in ymm1 and put result in ymm1 subject to writemask k1.

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.DDS.512.66.0F38.W1 97 /r VFMSUBADD132PD zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst{er}	B	V/V	AVX512F	Multiply packed double-precision floating-point values from zmm1 and zmm3/m512/m64bcst, subtract/add elements in zmm2 and put result in zmm1 subject to writemask k1.
EVEX.DDS.512.66.0F38.W1 A7 /r VFMSUBADD213PD zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst{er}	B	V/V	AVX512F	Multiply packed double-precision floating-point values from zmm1 and zmm2, subtract/add elements in zmm3/m512/m64bcst and put result in zmm1 subject to writemask k1.
EVEX.DDS.512.66.0F38.W1 B7 /r VFMSUBADD231PD zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst{er}	B	V/V	AVX512F	Multiply packed double-precision floating-point values from zmm2 and zmm3/m512/m64bcst, subtract/add elements in zmm1 and put result in zmm1 subject to writemask k1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	NA	ModRM:reg (r, w)	VEX.vvvv (r)	ModRM:r/m (r)	NA
B	Full Vector	ModRM:reg (r, w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA

Description

VFMSUBADD132PD: Multiplies the two, four, or eight packed double-precision floating-point values from the first source operand to the two or four packed double-precision floating-point values in the third source operand. From the infinite precision intermediate result, subtracts the odd double-precision floating-point elements and adds the even double-precision floating-point values in the second source operand, performs rounding and stores the resulting two or four packed double-precision floating-point values to the destination operand (first source operand).

VFMSUBADD213PD: Multiplies the two, four, or eight packed double-precision floating-point values from the second source operand to the two or four packed double-precision floating-point values in the first source operand. From the infinite precision intermediate result, subtracts the odd double-precision floating-point elements and adds the even double-precision floating-point values in the third source operand, performs rounding and stores the resulting two or four packed double-precision floating-point values to the destination operand (first source operand).

VFMSUBADD231PD: Multiplies the two, four, or eight packed double-precision floating-point values from the second source operand to the two or four packed double-precision floating-point values in the third source operand. From the infinite precision intermediate result, subtracts the odd double-precision floating-point elements and adds the even double-precision floating-point values in the first source operand, performs rounding and stores the resulting two or four packed double-precision floating-point values to the destination operand (first source operand).

EVEX encoded versions: The destination operand (also first source operand) and the second source operand are ZMM/YMM/XMM register. The third source operand is a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 64-bit memory location. The destination operand is conditionally updated with write mask k1.

VEX.256 encoded version: The destination operand (also first source operand) is a YMM register and encoded in reg_field. The second source operand is a YMM register and encoded in VEX.vvvv. The third source operand is a YMM register or a 256-bit memory location and encoded in rm_field.

VEX.128 encoded version: The destination operand (also first source operand) is a XMM register and encoded in reg_field. The second source operand is a XMM register and encoded in VEX.vvvv. The third source operand is a XMM register or a 128-bit memory location and encoded in rm_field. The upper 128 bits of the YMM destination register are zeroed.

Compiler tools may optionally support a complementary mnemonic for each instruction mnemonic listed in the opcode/instruction column of the summary table. The behavior of the complementary mnemonic in situations involving NANs are governed by the definition of the instruction mnemonic defined in the opcode/instruction column.

Operation

In the operations below, "*" and "+" symbols represent multiplication and addition with infinite precision inputs and outputs (no rounding).

VFMSUBADD132PD DEST, SRC2, SRC3

```
IF (VEX.128) THEN
    DEST[63:0] ←RoundFPControl_MXCSR(DEST[63:0]*SRC3[63:0] + SRC2[63:0])
    DEST[127:64] ←RoundFPControl_MXCSR(DEST[127:64]*SRC3[127:64] - SRC2[127:64])
    DEST[MAXVL-1:128] ←0
ELSEIF (VEX.256)
    DEST[63:0] ←RoundFPControl_MXCSR(DEST[63:0]*SRC3[63:0] + SRC2[63:0])
    DEST[127:64] ←RoundFPControl_MXCSR(DEST[127:64]*SRC3[127:64] - SRC2[127:64])
    DEST[191:128] ←RoundFPControl_MXCSR(DEST[191:128]*SRC3[191:128] + SRC2[191:128])
    DEST[255:192] ←RoundFPControl_MXCSR(DEST[255:192]*SRC3[255:192] - SRC2[255:192]
FI
```

```
VFMSUBADD213PD DEST, SRC2, SRC3
IF (VEX.128) THEN
    DEST[63:0] ←RoundFPControl_MXCSR(SRC2[63:0]*DEST[63:0] + SRC3[63:0])
    DEST[127:64] ←RoundFPControl_MXCSR(SRC2[127:64]*DEST[127:64] - SRC3[127:64])
    DEST[MAXVL-1:128] ←0
ELSEIF (VEX.256)
    DEST[63:0] ←RoundFPControl_MXCSR(SRC2[63:0]*DEST[63:0] + SRC3[63:0])
    DEST[127:64] ←RoundFPControl_MXCSR(SRC2[127:64]*DEST[127:64] - SRC3[127:64])
    DEST[191:128] ←RoundFPControl_MXCSR(SRC2[191:128]*DEST[191:128] + SRC3[191:128])
    DEST[255:192] ←RoundFPControl_MXCSR(SRC2[255:192]*DEST[255:192] - SRC3[255:192]
FI
```

VFMSUBADD231PD DEST, SRC2, SRC3

```
IF (VEX.128) THEN
    DEST[63:0] ←RoundFPControl_MXCSR(SRC2[63:0]*SRC3[63:0] + DEST[63:0])
    DEST[127:64] ←RoundFPControl_MXCSR(SRC2[127:64]*SRC3[127:64] - DEST[127:64])
    DEST[MAXVL-1:128] ←0
ELSEIF (VEX.256)
    DEST[63:0] ←RoundFPControl_MXCSR(SRC2[63:0]*SRC3[63:0] + DEST[63:0])
    DEST[127:64] ←RoundFPControl_MXCSR(SRC2[127:64]*SRC3[127:64] - DEST[127:64])
    DEST[191:128] ←RoundFPControl_MXCSR(SRC2[191:128]*SRC3[191:128] + DEST[191:128])
    DEST[255:192] ←RoundFPControl_MXCSR(SRC2[255:192]*SRC3[255:192] - DEST[255:192]
FI
```

VFMSUBADD132PD DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a register)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
IF (VL = 512) AND (EVEX.b = 1)
    THEN
        SET_RM(EVEX.RC);
    ELSE
        SET_RM(MXCSR.RM);
FI;
FOR j ← 0 TO KL-1
    i ← j * 64
    IF k1[j] OR *no writemask*
        THEN
            IF j *is even*
                THEN DEST[i+63:i] ←
                    RoundFPControl(DEST[i+63:i]*SRC3[i+63:i] + SRC2[i+63:i])
                ELSE DEST[i+63:i] ←
                    RoundFPControl(DEST[i+63:i]*SRC3[i+63:i] - SRC2[i+63:i])
            FI
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE                                ; zeroing-masking
                    DEST[i+63:i] ← 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0
```

VFMSUBADD132PD DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a memory source)
(KL, VL) = (2, 128), (4, 256), (8, 512)

```
FOR j ← 0 TO KL-1
    i ← j * 64
    IF k1[j] OR *no writemask*
        THEN
            IF j *is even*
                THEN
                    IF (EVEX.b = 1)
                        THEN
                            DEST[i+63:i] ←
                    RoundFPControl_MXCSR(DEST[i+63:i]*SRC3[63:0] + SRC2[i+63:i])
                        ELSE
                            DEST[i+63:i] ←
                    RoundFPControl_MXCSR(DEST[i+63:i]*SRC3[i+63:i] + SRC2[i+63:i])
                    FI;
                ELSE
                    IF (EVEX.b = 1)
                        THEN
                            DEST[i+63:i] ←
                    RoundFPControl_MXCSR(DEST[i+63:i]*SRC3[63:0] - SRC2[i+63:i])
                        ELSE
                            DEST[i+63:i] ←
                    RoundFPControl_MXCSR(DEST[i+63:i]*SRC3[i+63:i] - SRC2[i+63:i])
                    FI;
            FI

        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE                                ; zeroing-masking
                    DEST[i+63:i] ← 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0
```

VFMSUBADD213PD DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a register)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
IF (VL = 512) AND (EVEX.b = 1)
    THEN
        SET_RM(EVEX.RC);
    ELSE
        SET_RM(MXCSR.RM);
FI;
FOR j ← 0 TO KL-1
    i ← j * 64
    IF k1[j] OR *no writemask*
        THEN
            IF j *is even*
                THEN DEST[i+63:i] ←
                    RoundFPControl(SRC2[i+63:i]*DEST[i+63:i] + SRC3[i+63:i])
                ELSE DEST[i+63:i] ←
                    RoundFPControl(SRC2[i+63:i]*DEST[i+63:i] - SRC3[i+63:i])
            FI
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE                            ; zeroing-masking
                    DEST[i+63:i] ← 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0
```

VFMSUBADD213PD DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a memory source)
(KL, VL) = (2, 128), (4, 256), (8, 512)

```
FOR j ← 0 TO KL-1
    i ← j * 64
    IF k1[j] OR *no writemask*
        THEN
            IF j *is even*
                THEN
                    IF (EVEX.b = 1)
                        THEN
                            DEST[i+63:i] ←
                    RoundFPControl_MXCSR(SRC2[i+63:i]*DEST[i+63:i] + SRC3[63:0])
                        ELSE
                            DEST[i+63:i] ←
                    RoundFPControl_MXCSR(SRC2[i+63:i]*DEST[i+63:i] + SRC3[i+63:i])
                    FI;
                ELSE
                    IF (EVEX.b = 1)
                        THEN
                            DEST[i+63:i] ←
                    RoundFPControl_MXCSR(SRC2[i+63:i]*DEST[i+63:i] - SRC3[63:0])
                        ELSE
                            DEST[i+63:i] ←
                    RoundFPControl_MXCSR(SRC2[i+63:i]*DEST[i+63:i] - SRC3[i+63:i])
                    FI;
            FI
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE                                ; zeroing-masking
                    DEST[i+63:i] ← 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0
```

VFMSUBADD231PD DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a register)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
IF (VL = 512) AND (EVEX.b = 1)
    THEN
        SET_RM(EVEX.RC);
    ELSE
        SET_RM(MXCSR.RM);
FI;
FOR j ← 0 TO KL-1
    i ← j * 64
    IF k1[j] OR *no writemask*
        THEN
            IF j *is even*
                THEN DEST[i+63:i] ←
                    RoundFPControl(SRC2[i+63:i]*SRC3[i+63:i] + DEST[i+63:i])
                ELSE DEST[i+63:i] ←
                    RoundFPControl(SRC2[i+63:i]*SRC3[i+63:i] - DEST[i+63:i])
            FI
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE                            ; zeroing-masking
                    DEST[i+63:i] ← 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0
```

VFMSUBADD231PD DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a memory source)
(KL, VL) = (2, 128), (4, 256), (8, 512)

```
FOR j ← 0 TO KL-1
    i ← j * 64
    IF k1[j] OR *no writemask*
        THEN
            IF j *is even*
                THEN
                    IF (EVEX.b = 1)
                        THEN
                            DEST[i+63:i] ←
                    RoundFPControl_MXCSR(SRC2[i+63:i]*SRC3[63:0] + DEST[i+63:i])
                        ELSE
                            DEST[i+63:i] ←
                    RoundFPControl_MXCSR(SRC2[i+63:i]*SRC3[i+63:i] + DEST[i+63:i])
                    FI;
                ELSE
                    IF (EVEX.b = 1)
                        THEN
                            DEST[i+63:i] ←
                    RoundFPControl_MXCSR(SRC2[i+63:i]*SRC3[63:0] - DEST[i+63:i])

                        ELSE
                            DEST[i+63:i] ←
                    RoundFPControl_MXCSR(SRC2[i+63:i]*SRC3[i+63:i] - DEST[i+63:i])
                    FI;
            FI
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE                                ; zeroing-masking
                    DEST[i+63:i] ← 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VFMSUBADDxxxPD __m512d _mm512_fmsubadd_pd(__m512d a, __m512d b, __m512d c);
VFMSUBADDxxxPD __m512d _mm512_fmsubadd_round_pd(__m512d a, __m512d b, __m512d c, int r);
VFMSUBADDxxxPD __m512d _mm512_mask_fmsubadd_pd(__m512d a, __mmask8 k, __m512d b, __m512d c);
VFMSUBADDxxxPD __m512d _mm512_maskz_fmsubadd_pd(__mmask8 k, __m512d a, __m512d b, __m512d c);
VFMSUBADDxxxPD __m512d _mm512_mask3_fmsubadd_pd(__m512d a, __m512d b, __m512d c, __mmask8 k);
VFMSUBADDxxxPD __m512d _mm512_mask_fmsubadd_round_pd(__m512d a, __mmask8 k, __m512d b, __m512d c, int r);
VFMSUBADDxxxPD __m512d _mm512_maskz_fmsubadd_round_pd(__mmask8 k, __m512d a, __m512d b, __m512d c, int r);
VFMSUBADDxxxPD __m512d _mm512_mask3_fmsubadd_round_pd(__m512d a, __m512d b, __m512d c, __mmask8 k, int r);
VFMSUBADDxxxPD __m256d _mm256_mask_fmsubadd_pd(__m256d a, __mmask8 k, __m256d b, __m256d c);
VFMSUBADDxxxPD __m256d _mm256_maskz_fmsubadd_pd(__mmask8 k, __m256d a, __m256d b, __m256d c);
VFMSUBADDxxxPD __m256d _mm256_mask3_fmsubadd_pd(__m256d a, __m256d b, __m256d c, __mmask8 k);
VFMSUBADDxxxPD __m128d _mm_mask_fmsubadd_pd(__m128d a, __mmask8 k, __m128d b, __m128d c);
VFMSUBADDxxxPD __m128d _mm_maskz_fmsubadd_pd(__mmask8 k, __m128d a, __m128d b, __m128d c);
VFMSUBADDxxxPD __m128d _mm_mask3_fmsubadd_pd(__m128d a, __m128d b, __m128d c, __mmask8 k);
VFMSUBADDxxxPD __m128d _mm_fmsubadd_pd (__m128d a, __m128d b, __m128d c);
VFMSUBADDxxxPD __m256d _mm256_fmsubadd_pd (__m256d a, __m256d b, __m256d c);

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal

Other Exceptions

VEX-encoded instructions, see Exceptions Type 2.
EVEX-encoded instructions, see Exceptions Type E2.

VFMSUBADD132PS/VFMSUBADD213PS/VFMSUBADD231PS—Fused Multiply-Alternating Subtract/Add of Packed Single-Precision Floating-Point Values

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
VEX.DDS.128.66.0F38.W0 97 /r VFMSUBADD132PS xmm1, xmm2, xmm3/m128	A	V/V	FMA	Multiply packed single-precision floating-point values from xmm1 and xmm3/mem, subtract/add elements in xmm2 and put result in xmm1.
VEX.DDS.128.66.0F38.W0 A7 /r VFMSUBADD213PS xmm1, xmm2, xmm3/m128	A	V/V	FMA	Multiply packed single-precision floating-point values from xmm1 and xmm2, subtract/add elements in xmm3/mem and put result in xmm1.
VEX.DDS.128.66.0F38.W0 B7 /r VFMSUBADD231PS xmm1, xmm2, xmm3/m128	A	V/V	FMA	Multiply packed single-precision floating-point values from xmm2 and xmm3/mem, subtract/add elements in xmm1 and put result in xmm1.
VEX.DDS.256.66.0F38.W0 97 /r VFMSUBADD132PS ymm1, ymm2, ymm3/m256	A	V/V	FMA	Multiply packed single-precision floating-point values from ymm1 and ymm3/mem, subtract/add elements in ymm2 and put result in ymm1.
VEX.DDS.256.66.0F38.W0 A7 /r VFMSUBADD213PS ymm1, ymm2, ymm3/m256	A	V/V	FMA	Multiply packed single-precision floating-point values from ymm1 and ymm2, subtract/add elements in ymm3/mem and put result in ymm1.
VEX.DDS.256.66.0F38.W0 B7 /r VFMSUBADD231PS ymm1, ymm2, ymm3/m256	A	V/V	FMA	Multiply packed single-precision floating-point values from ymm2 and ymm3/mem, subtract/add elements in ymm1 and put result in ymm1.
EVEX.DDS.128.66.0F38.W0 97 /r VFMSUBADD132PS xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst	B	V/V	AVX512VL AVX512F	Multiply packed single-precision floating-point values from xmm1 and xmm3/m128/m32bcst, subtract/add elements in xmm2 and put result in xmm1 subject to writemask k1.
EVEX.DDS.128.66.0F38.W0 A7 /r VFMSUBADD213PS xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst	B	V/V	AVX512VL AVX512F	Multiply packed single-precision floating-point values from xmm1 and xmm2, subtract/add elements in xmm3/m128/m32bcst and put result in xmm1 subject to writemask k1.
EVEX.DDS.128.66.0F38.W0 B7 /r VFMSUBADD231PS xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst	B	V/V	AVX512VL AVX512F	Multiply packed single-precision floating-point values from xmm2 and xmm3/m128/m32bcst, subtract/add elements in xmm1 and put result in xmm1 subject to writemask k1.
EVEX.DDS.256.66.0F38.W0 97 /r VFMSUBADD132PS ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst	B	V/V	AVX512VL AVX512F	Multiply packed single-precision floating-point values from ymm1 and ymm3/m256/m32bcst, subtract/add elements in ymm2 and put result in ymm1 subject to writemask k1.
EVEX.DDS.256.66.0F38.W0 A7 /r VFMSUBADD213PS ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst	B	V/V	AVX512VL AVX512F	Multiply packed single-precision floating-point values from ymm1 and ymm2, subtract/add elements in ymm3/m256/m32bcst and put result in ymm1 subject to writemask k1.
EVEX.DDS.256.66.0F38.W0 B7 /r VFMSUBADD231PS ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst	B	V/V	AVX512VL AVX512F	Multiply packed single-precision floating-point values from ymm2 and ymm3/m256/m32bcst, subtract/add elements in ymm1 and put result in ymm1 subject to writemask k1.
EVEX.DDS.512.66.0F38.W0 97 /r VFMSUBADD132PS zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst{er}	B	V/V	AVX512F	Multiply packed single-precision floating-point values from zmm1 and zmm3/m512/m32bcst, subtract/add elements in zmm2 and put result in zmm1 subject to writemask k1.
EVEX.DDS.512.66.0F38.W0 A7 /r VFMSUBADD213PS zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst{er}	B	V/V	AVX512F	Multiply packed single-precision floating-point values from zmm1 and zmm2, subtract/add elements in zmm3/m512/m32bcst and put result in zmm1 subject to writemask k1.
EVEX.DDS.512.66.0F38.W0 B7 /r VFMSUBADD231PS zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst{er}	B	V/V	AVX512F	Multiply packed single-precision floating-point values from zmm2 and zmm3/m512/m32bcst, subtract/add elements in zmm1 and put result in zmm1 subject to writemask k1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	NA	ModRM:reg (r, w)	VEX.vvvv (r)	ModRM:r/m (r)	NA
B	Full Vector	ModRM:reg (r, w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA

Description

VFMSUBADD132PS: Multiplies the four, eight or sixteen packed single-precision floating-point values from the first source operand to the corresponding packed single-precision floating-point values in the third source operand. From the infinite precision intermediate result, subtracts the odd single-precision floating-point elements and adds the even single-precision floating-point values in the second source operand, performs rounding and stores the resulting packed single-precision floating-point values to the destination operand (first source operand).

VFMSUBADD213PS: Multiplies the four, eight or sixteen packed single-precision floating-point values from the second source operand to the corresponding packed single-precision floating-point values in the first source operand. From the infinite precision intermediate result, subtracts the odd single-precision floating-point elements and adds the even single-precision floating-point values in the third source operand, performs rounding and stores the resulting packed single-precision floating-point values to the destination operand (first source operand).

VFMSUBADD231PS: Multiplies the four, eight or sixteen packed single-precision floating-point values from the second source operand to the corresponding packed single-precision floating-point values in the third source operand. From the infinite precision intermediate result, subtracts the odd single-precision floating-point elements and adds the even single-precision floating-point values in the first source operand, performs rounding and stores the resulting packed single-precision floating-point values to the destination operand (first source operand).

EVEX encoded versions: The destination operand (also first source operand) and the second source operand are ZMM/YMM/XMM register. The third source operand is a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 32-bit memory location. The destination operand is conditionally updated with write mask k1.

VEX.256 encoded version: The destination operand (also first source operand) is a YMM register and encoded in reg_field. The second source operand is a YMM register and encoded in VEX.vvvv. The third source operand is a YMM register or a 256-bit memory location and encoded in rm_field.

VEX.128 encoded version: The destination operand (also first source operand) is a XMM register and encoded in reg_field. The second source operand is a XMM register and encoded in VEX.vvvv. The third source operand is a XMM register or a 128-bit memory location and encoded in rm_field. The upper 128 bits of the YMM destination register are zeroed.

Compiler tools may optionally support a complementary mnemonic for each instruction mnemonic listed in the opcode/instruction column of the summary table. The behavior of the complementary mnemonic in situations involving NANs are governed by the definition of the instruction mnemonic defined in the opcode/instruction column.

Operation

In the operations below, "*" and "+" symbols represent multiplication and addition with infinite precision inputs and outputs (no rounding).

VFMSUBADD132PS DEST, SRC2, SRC3

```
IF (VEX.128) THEN
    MAXNUM ←2
ELSEIF (VEX.256)
    MAXNUM ← 4
FI
For i = 0 to MAXNUM -1{
    n ← 64*i;
    DEST[n+31:n] ←RoundFPControl_MXCSR(DEST[n+31:n]*SRC3[n+31:n] + SRC2[n+31:n])
    DEST[n+63:n+32] ←RoundFPControl_MXCSR(DEST[n+63:n+32]*SRC3[n+63:n+32] -SRC2[n+63:n+32])
}
IF (VEX.128) THEN
    DEST[MAXVL-1:128] ←0
ELSEIF (VEX.256)
    DEST[MAXVL-1:256] ← 0
FI
```

VFMSUBADD213PS DEST, SRC2, SRC3

```
IF (VEX.128) THEN
    MAXNUM ←2
ELSEIF (VEX.256)
    MAXNUM ← 4
FI
For i = 0 to MAXNUM -1{
    n ← 64*i;
    DEST[n+31:n] ←RoundFPControl_MXCSR(SRC2[n+31:n]*DEST[n+31:n] +SRC3[n+31:n])
    DEST[n+63:n+32] ←RoundFPControl_MXCSR(SRC2[n+63:n+32]*DEST[n+63:n+32] -SRC3[n+63:n+32])
}
IF (VEX.128) THEN
    DEST[MAXVL-1:128] ←0
ELSEIF (VEX.256)
    DEST[MAXVL-1:256] ← 0
FI
```

VFMSUBADD231PS DEST, SRC2, SRC3

```
IF (VEX.128) THEN
    MAXNUM ←2
ELSEIF (VEX.256)
    MAXNUM ← 4
FI
For i = 0 to MAXNUM -1{
    n ← 64*i;
    DEST[n+31:n] ←RoundFPControl_MXCSR(SRC2[n+31:n]*SRC3[n+31:n] + DEST[n+31:n])
    DEST[n+63:n+32] ←RoundFPControl_MXCSR(SRC2[n+63:n+32]*SRC3[n+63:n+32] -DEST[n+63:n+32])
}
IF (VEX.128) THEN
    DEST[MAXVL-1:128] ←0
ELSEIF (VEX.256)
    DEST[MAXVL-1:256] ← 0
FI
```

VFMSUBADD132PS DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a register)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
IF (VL = 512) AND (EVEX.b = 1)
    THEN
        SET_RM(EVEX.RC);
    ELSE
        SET_RM(MXCSR.RM);
FI;
FOR j ← 0 TO KL-1
    i ← j * 32
    IF k1[j] OR *no writemask*
        THEN
            IF j *is even*
                THEN DEST[i+31:i] ←
                    RoundFPControl(DEST[i+31:i]*SRC3[i+31:i] + SRC2[i+31:i])
                ELSE DEST[i+31:i] ←
                    RoundFPControl(DEST[i+31:i]*SRC3[i+31:i] - SRC2[i+31:i])
            FI
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE                            ; zeroing-masking
                    DEST[i+31:i] ← 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0
```

VFMSUBADD132PS DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a memory source)
(KL, VL) = (4, 128), (8, 256), (16, 512)

```
FOR j ← 0 TO KL-1
    i ← j * 32
    IF k1[j] OR *no writemask*
        THEN
            IF j *is even*
                THEN
                    IF (EVEX.b = 1)
                        THEN
                            DEST[i+31:i] ←
                        RoundFPControl_MXCSR(DEST[i+31:i]*SRC3[31:0] + SRC2[i+31:i])
                        ELSE
                            DEST[i+31:i] ←
                        RoundFPControl_MXCSR(DEST[i+31:i]*SRC3[i+31:i] + SRC2[i+31:i])
                    FI;
                ELSE
                    IF (EVEX.b = 1)
                        THEN
                            DEST[i+31:i] ←
                    RoundFPControl_MXCSR(DEST[i+31:i]*SRC3[31:0] - SRC2[i+31:i])
                        ELSE
                            DEST[i+31:i] ←
                    RoundFPControl_MXCSR(DEST[i+31:i]*SRC3[i+31:i] - SRC2[i+31:i])
                    FI;
            FI

        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE                                ; zeroing-masking
                    DEST[i+31:i] ← 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0
```

VFMSUBADD213PS DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a register)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
IF (VL = 512) AND (EVEX.b = 1)
    THEN
        SET_RM(EVEX.RC);
    ELSE
        SET_RM(MXCSR.RM);
FI;
FOR j ← 0 TO KL-1
    i ← j * 32
    IF k1[j] OR *no writemask*
        THEN
            IF j *is even*
                THEN DEST[i+31:i] ←
                    RoundFPControl(SRC2[i+31:i]*DEST[i+31:i] + SRC3[i+31:i])
                ELSE DEST[i+31:i] ←
                    RoundFPControl(SRC2[i+31:i]*DEST[i+31:i] - SRC3[i+31:i])
            FI
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE                            ; zeroing-masking
                    DEST[i+31:i] ← 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0
```

VFMSUBADD213PS DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a memory source)
(KL, VL) = (4, 128), (8, 256), (16, 512)

```
FOR j ← 0 TO KL-1
    i ← j * 32
    IF k1[j] OR *no writemask*
        THEN
            IF j *is even*
                THEN
                    IF (EVEX.b = 1)
                        THEN
                            DEST[i+31:i] ←
                    RoundFPControl_MXCSR(SRC2[i+31:i]*DEST[i+31:i] + SRC3[31:0])
                    ELSE
                        DEST[i+31:i] ←
                    RoundFPControl_MXCSR(SRC2[i+31:i]*DEST[i+31:i] + SRC3[i+31:i])
                FI;
                ELSE
                    IF (EVEX.b = 1)
                        THEN
                            DEST[i+31:i] ←
                    RoundFPControl_MXCSR(SRC2[i+31:i]*DEST[i+31:i] - SRC3[i+31:i])
                        ELSE
                            DEST[i+31:i] ←
                    RoundFPControl_MXCSR(SRC2[i+31:i]*DEST[i+31:i] - SRC3[31:0])
                    FI;
            FI
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE                                ; zeroing-masking
                    DEST[i+31:i] ← 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0
```

VFMSUBADD231PS DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a register)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
IF (VL = 512) AND (EVEX.b = 1)
    THEN
        SET_RM(EVEX.RC);
    ELSE
        SET_RM(MXCSR.RM);
FI;
FOR j ← 0 TO KL-1
    i ← j * 32
    IF k1[j] OR *no writemask*
        THEN
            IF j *is even*
                THEN DEST[i+31:i] ←
                    RoundFPControl(SRC2[i+31:i]*SRC3[i+31:i] + DEST[i+31:i])
                ELSE DEST[i+31:i] ←
                    RoundFPControl(SRC2[i+31:i]*SRC3[i+31:i] - DEST[i+31:i])
            FI
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE                                ; zeroing-masking
                    DEST[i+31:i] ← 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0
```

VFMSUBADD231PS DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a memory source)
(KL, VL) = (4, 128), (8, 256), (16, 512)

```
FOR j ← 0 TO KL-1
    i ← j * 32
    IF k1[j] OR *no writemask*
        THEN
            IF j *is even*
                THEN
                    IF (EVEX.b = 1)
                        THEN
                            DEST[i+31:i] ←
                        RoundFPControl_MXCSR(SRC2[i+31:i]*SRC3[31:0] + DEST[i+31:i])
                        ELSE
                            DEST[i+31:i] ←
                        RoundFPControl_MXCSR(SRC2[i+31:i]*SRC3[i+31:i] + DEST[i+31:i])
                    FI;
                ELSE
                    IF (EVEX.b = 1)
                        THEN
                            DEST[i+31:i] ←
                    RoundFPControl_MXCSR(SRC2[i+31:i]*SRC3[31:0] - DEST[i+31:i])
                        ELSE
                            DEST[i+31:i] ←
                    RoundFPControl_MXCSR(SRC2[i+31:i]*SRC3[i+31:i] - DEST[i+31:i])
                    FI;
            FI
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE                                ; zeroing-masking
                    DEST[i+31:i] ← 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VFMSUBADDxxxPS __m512 _mm512_fmsubadd_ps(__m512 a, __m512 b, __m512 c);
VFMSUBADDxxxPS __m512 _mm512_fmsubadd_round_ps(__m512 a, __m512 b, __m512 c, int r);
VFMSUBADDxxxPS __m512 _mm512_mask_fmsubadd_ps(__m512 a, __mmask16 k, __m512 b, __m512 c);
VFMSUBADDxxxPS __m512 _mm512_maskz_fmsubadd_ps(__mmask16 k, __m512 a, __m512 b, __m512 c);
VFMSUBADDxxxPS __m512 _mm512_mask3_fmsubadd_ps(__m512 a, __m512 b, __m512 c, __mmask16 k);
VFMSUBADDxxxPS __m512 _mm512_mask_fmsubadd_round_ps(__m512 a, __mmask16 k, __m512 b, __m512 c, int r);
VFMSUBADDxxxPS __m512 _mm512_maskz_fmsubadd_round_ps(__mmask16 k, __m512 a, __m512 b, __m512 c, int r);
VFMSUBADDxxxPS __m512 _mm512_mask3_fmsubadd_round_ps(__m512 a, __m512 b, __m512 c, __mmask16 k, int r);
VFMSUBADDxxxPS __m256 _mm256_mask_fmsubadd_ps(__m256 a, __mmask8 k, __m256 b, __m256 c);
VFMSUBADDxxxPS __m256 _mm256_maskz_fmsubadd_ps(__mmask8 k, __m256 a, __m256 b, __m256 c);
VFMSUBADDxxxPS __m256 _mm256_mask3_fmsubadd_ps(__m256 a, __m256 b, __m256 c, __mmask8 k);
VFMSUBADDxxxPS __m128 _mm_mask_fmsubadd_ps(__m128 a, __mmask8 k, __m128 b, __m128 c);
VFMSUBADDxxxPS __m128 _mm_maskz_fmsubadd_ps(__mmask8 k, __m128 a, __m128 b, __m128 c);
VFMSUBADDxxxPS __m128 _mm_mask3_fmsubadd_ps(__m128 a, __m128 b, __m128 c, __mmask8 k);
VFMSUBADDxxxPS __m128 _mm_fmsubadd_ps (__m128 a, __m128 b, __m128 c);
VFMSUBADDxxxPS __m256 _mm256_fmsubadd_ps (__m256 a, __m256 b, __m256 c);

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal

Other Exceptions

VEX-encoded instructions, see Exceptions Type 2.
EVEX-encoded instructions, see Exceptions Type E2.

VFMSUB132PD/VFMSUB213PD/VFMSUB231PD—Fused Multiply-Subtract of Packed Double-Precision Floating-Point Values

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
VEX.NDS.128.66.0F38.W1 9A /r VFMSUB132PD xmm1, xmm2, xmm3/m128	A	V/V	FMA	Multiply packed double-precision floating-point values from xmm1 and xmm3/mem, subtract xmm2 and put result in xmm1.
VEX.NDS.128.66.0F38.W1 AA /r VFMSUB213PD xmm1, xmm2, xmm3/m128	A	V/V	FMA	Multiply packed double-precision floating-point values from xmm1 and xmm2, subtract xmm3/mem and put result in xmm1.
VEX.NDS.128.66.0F38.W1 BA /r VFMSUB231PD xmm1, xmm2, xmm3/m128	A	V/V	FMA	Multiply packed double-precision floating-point values from xmm2 and xmm3/mem, subtract xmm1 and put result in xmm1.
VEX.NDS.256.66.0F38.W1 9A /r VFMSUB132PD ymm1, ymm2, ymm3/m256	A	V/V	FMA	Multiply packed double-precision floating-point values from ymm1 and ymm3/mem, subtract ymm2 and put result in ymm1.
VEX.NDS.256.66.0F38.W1 AA /r VFMSUB213PD ymm1, ymm2, ymm3/m256	A	V/V	FMA	Multiply packed double-precision floating-point values from ymm1 and ymm2, subtract ymm3/mem and put result in ymm1.
VEX.NDS.256.66.0F38.W1 BA /r VFMSUB231PD ymm1, ymm2, ymm3/m256	A	V/V	FMA	Multiply packed double-precision floating-point values from ymm2 and ymm3/mem, subtract ymm1 and put result in ymm1.S
EVEX.NDS.128.66.0F38.W1 9A /r VFMSUB132PD xmm1 {k1}{z}, xmm2, xmm3/m128/m64bcst	B	V/V	AVX512VL AVX512F	Multiply packed double-precision floating-point values from xmm1 and xmm3/m128/m64bcst, subtract xmm2 and put result in xmm1 subject to writemask k1.
EVEX.NDS.128.66.0F38.W1 AA /r VFMSUB213PD xmm1 {k1}{z}, xmm2, xmm3/m128/m64bcst	B	V/V	AVX512VL AVX512F	Multiply packed double-precision floating-point values from xmm1 and xmm2, subtract xmm3/m128/m64bcst and put result in xmm1 subject to writemask k1.
EVEX.NDS.128.66.0F38.W1 BA /r VFMSUB231PD xmm1 {k1}{z}, xmm2, xmm3/m128/m64bcst	B	V/V	AVX512VL AVX512F	Multiply packed double-precision floating-point values from xmm2 and xmm3/m128/m64bcst, subtract xmm1 and put result in xmm1 subject to writemask k1.
EVEX.NDS.256.66.0F38.W1 9A /r VFMSUB132PD ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst	B	V/V	AVX512VL AVX512F	Multiply packed double-precision floating-point values from ymm1 and ymm3/m256/m64bcst, subtract ymm2 and put result in ymm1 subject to writemask k1.
EVEX.NDS.256.66.0F38.W1 AA /r VFMSUB213PD ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst	B	V/V	AVX512VL AVX512F	Multiply packed double-precision floating-point values from ymm1 and ymm2, subtract ymm3/m256/m64bcst and put result in ymm1 subject to writemask k1.
EVEX.NDS.256.66.0F38.W1 BA /r VFMSUB231PD ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst	B	V/V	AVX512VL AVX512F	Multiply packed double-precision floating-point values from ymm2 and ymm3/m256/m64bcst, subtract ymm1 and put result in ymm1 subject to writemask k1.
EVEX.NDS.512.66.0F38.W1 9A /r VFMSUB132PD zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst{er}	B	V/V	AVX512F	Multiply packed double-precision floating-point values from zmm1 and zmm3/m512/m64bcst, subtract zmm2 and put result in zmm1 subject to writemask k1.
EVEX.NDS.512.66.0F38.W1 AA /r VFMSUB213PD zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst{er}	B	V/V	AVX512F	Multiply packed double-precision floating-point values from zmm1 and zmm2, subtract zmm3/m512/m64bcst and put result in zmm1 subject to writemask k1.
EVEX.NDS.512.66.0F38.W1 BA /r VFMSUB231PD zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst{er}	B	V/V	AVX512F	Multiply packed double-precision floating-point values from zmm2 and zmm3/m512/m64bcst, subtract zmm1 and put result in zmm1 subject to writemask k1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	NA	ModRM:reg (r, w)	VEX.vvvv (r)	ModRM:r/m (r)	NA
B	Full Vector	ModRM:reg (r, w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA

Description

Performs a set of SIMD multiply-subtract computation on packed double-precision floating-point values using three source operands and writes the multiply-subtract results in the destination operand. The destination operand is also the first source operand. The second operand must be a SIMD register. The third source operand can be a SIMD register or a memory location.

VFMSUB132PD: Multiplies the two, four or eight packed double-precision floating-point values from the first source operand to the two, four or eight packed double-precision floating-point values in the third source operand. From the infinite precision intermediate result, subtracts the two, four or eight packed double-precision floating-point values in the second source operand, performs rounding and stores the resulting two, four or eight packed double-precision floating-point values to the destination operand (first source operand).

VFMSUB213PD: Multiplies the two, four or eight packed double-precision floating-point values from the second source operand to the two, four or eight packed double-precision floating-point values in the first source operand. From the infinite precision intermediate result, subtracts the two, four or eight packed double-precision floating-point values in the third source operand, performs rounding and stores the resulting two, four or eight packed double-precision floating-point values to the destination operand (first source operand).

VFMSUB231PD: Multiplies the two, four or eight packed double-precision floating-point values from the second source to the two, four or eight packed double-precision floating-point values in the third source operand. From the infinite precision intermediate result, subtracts the two, four or eight packed double-precision floating-point values in the first source operand, performs rounding and stores the resulting two, four or eight packed double-precision floating-point values to the destination operand (first source operand).

EVEX encoded versions: The destination operand (also first source operand) and the second source operand are ZMM/YMM/XMM register. The third source operand is a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 64-bit memory location. The destination operand is conditionally updated with write mask k1.

VEX.256 encoded version: The destination operand (also first source operand) is a YMM register and encoded in reg_field. The second source operand is a YMM register and encoded in VEX.vvvv. The third source operand is a YMM register or a 256-bit memory location and encoded in rm_field.

VEX.128 encoded version: The destination operand (also first source operand) is a XMM register and encoded in reg_field. The second source operand is a XMM register and encoded in VEX.vvvv. The third source operand is a XMM register or a 128-bit memory location and encoded in rm_field. The upper 128 bits of the YMM destination register are zeroed.

Operation

In the operations below, "*" and "-" symbols represent multiplication and subtraction with infinite precision inputs and outputs (no rounding).

VFMSUB132PD DEST, SRC2, SRC3 (VEX encoded versions)

```
IF (VEX.128) THEN
    MAXNUM ←2
ELSEIF (VEX.256)
    MAXNUM ← 4
FI
For i = 0 to MAXNUM-1 {
    n ← 64*i;
    DEST[n+63:n] ← RoundFPControl_MXCSR(DEST[n+63:n]*SRC3[n+63:n] - SRC2[n+63:n])
}
IF (VEX.128) THEN
    DEST[MAXVL-1:128] ← 0
ELSEIF (VEX.256)
    DEST[MAXVL-1:256] ← 0
FI
```

VFMSUB213PD DEST, SRC2, SRC3 (VEX encoded versions)

```
IF (VEX.128) THEN
    MAXNUM ←2
ELSEIF (VEX.256)
    MAXNUM ← 4
FI
For i = 0 to MAXNUM-1 {
    n ← 64*i;
    DEST[n+63:n] ← RoundFPControl_MXCSR(SRC2[n+63:n]*DEST[n+63:n] - SRC3[n+63:n])
}
IF (VEX.128) THEN
    DEST[MAXVL-1:128] ← 0
ELSEIF (VEX.256)
    DEST[MAXVL-1:256] ← 0
FI
```

VFMSUB231PD DEST, SRC2, SRC3 (VEX encoded versions)

```
IF (VEX.128) THEN
    MAXNUM ←2
ELSEIF (VEX.256)
    MAXNUM ← 4
FI
For i = 0 to MAXNUM-1 {
    n ← 64*i;
    DEST[n+63:n] ← RoundFPControl_MXCSR(SRC2[n+63:n]*SRC3[n+63:n] - DEST[n+63:n])
}
IF (VEX.128) THEN
    DEST[MAXVL-1:128] ← 0
ELSEIF (VEX.256)
    DEST[MAXVL-1:256] ← 0
FI
```

VFMSUB132PD DEST, SRC2, SRC3 (EVEX encoded versions, when src3 operand is a register)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
IF (VL = 512) AND (EVEX.b = 1)
    THEN
        SET_RM(EVEX.RC);
    ELSE
        SET_RM(MXCSR.RM);
FI;
FOR j ← 0 TO KL-1
    i ← j * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+63:i] ←
            RoundFPControl(DEST[i+63:i]*SRC3[i+63:i] - SRC2[i+63:i])
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE                                ; zeroing-masking
                    DEST[i+63:i] ← 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0
```

VFMSUB132PD DEST, SRC2, SRC3 (EVEX encoded versions, when src3 operand is a memory source)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)

FOR j ← 0 TO KL-1
    i ← j * 64
    IF k1[j] OR *no writemask*
        THEN
            IF (EVEX.b = 1)
                THEN
                    DEST[i+63:i] ←
            RoundFPControl_MXCSR(DEST[i+63:i]*SRC3[63:0] - SRC2[i+63:i])
                ELSE
                    DEST[i+63:i] ←
            RoundFPControl_MXCSR(DEST[i+63:i]*SRC3[i+63:i] - SRC2[i+63:i])
            FI;
            ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE                                ; zeroing-masking
                    DEST[i+63:i] ← 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0
```

VFMSUB213PD DEST, SRC2, SRC3 (EVEX encoded versions, when src3 operand is a register)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
IF (VL = 512) AND (EVEX.b = 1)
    THEN
        SET_RM(EVEX.RC);
    ELSE
        SET_RM(MXCSR.RM);
FI;
FOR j ← 0 TO KL-1
    i ← j * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+63:i] ←
            RoundFPControl(SRC2[i+63:i]*DEST[i+63:i] - SRC3[i+63:i])
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE                                ; zeroing-masking
                    DEST[i+63:i] ← 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0
```

VFMSUB213PD DEST, SRC2, SRC3 (EVEX encoded versions, when src3 operand is a memory source)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)

FOR j ← 0 TO KL-1
    i ← j * 64
    IF k1[j] OR *no writemask*
        THEN
            IF (EVEX.b = 1)
                THEN
                    DEST[i+63:i] ←
            RoundFPControl_MXCSR(SRC2[i+63:i]*DEST[i+63:i] - SRC3[63:0])
+31:i])
                ELSE
                    DEST[i+63:i] ←
            RoundFPControl_MXCSR(SRC2[i+63:i]*DEST[i+63:i] - SRC3[i+63:i])
            FI;
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE                                ; zeroing-masking
                    DEST[i+63:i] ← 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0
```

VFMSUB231PD DEST, SRC2, SRC3 (EVEX encoded versions, when src3 operand is a register)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
IF (VL = 512) AND (EVEX.b = 1)
    THEN
        SET_RM(EVEX.RC);
    ELSE
        SET_RM(MXCSR.RM);
FI;
FOR j ← 0 TO KL-1
    i ← j * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+63:i] ←
            RoundFPControl(SRC2[i+63:i]*SRC3[i+63:i] - DEST[i+63:i])
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE                                ; zeroing-masking
                    DEST[i+63:i] ← 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0
```

VFMSUB231PD DEST, SRC2, SRC3 (EVEX encoded versions, when src3 operand is a memory source)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)

FOR j ← 0 TO KL-1
    i ← j * 64
    IF k1[j] OR *no writemask*
        THEN
            IF (EVEX.b = 1)
                THEN
                    DEST[i+63:i] ←
            RoundFPControl_MXCSR(SRC2[i+63:i]*SRC3[63:0] - DEST[i+63:i])
                ELSE
                    DEST[i+63:i] ←
            RoundFPControl_MXCSR(SRC2[i+63:i]*SRC3[i+63:i] - DEST[i+63:i])
            FI;
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE                                ; zeroing-masking
                    DEST[i+63:i] ← 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VFMSUBxxxPD __m512d _mm512_fmsub_pd(__m512d a, __m512d b, __m512d c);
VFMSUBxxxPD __m512d _mm512_fmsub_round_pd(__m512d a, __m512d b, __m512d c, int r);
VFMSUBxxxPD __m512d _mm512_mask_fmsub_pd(__m512d a, __mmask8 k, __m512d b, __m512d c);
VFMSUBxxxPD __m512d _mm512_maskz_fmsub_pd(__mmask8 k, __m512d a, __m512d b, __m512d c);
VFMSUBxxxPD __m512d _mm512_mask3_fmsub_pd(__m512d a, __m512d b, __m512d c, __mmask8 k);
VFMSUBxxxPD __m512d _mm512_mask_fmsub_round_pd(__m512d a, __mmask8 k, __m512d b, __m512d c, int r);
VFMSUBxxxPD __m512d _mm512_maskz_fmsub_round_pd(__mmask8 k, __m512d a, __m512d b, __m512d c, int r);
VFMSUBxxxPD __m512d _mm512_mask3_fmsub_round_pd(__m512d a, __m512d b, __m512d c, __mmask8 k, int r);
VFMSUBxxxPD __m256d _mm256_mask_fmsub_pd(__m256d a, __mmask8 k, __m256d b, __m256d c);
VFMSUBxxxPD __m256d _mm256_maskz_fmsub_pd(__mmask8 k, __m256d a, __m256d b, __m256d c);
VFMSUBxxxPD __m256d _mm256_mask3_fmsub_pd(__m256d a, __m256d b, __m256d c, __mmask8 k);
VFMSUBxxxPD __m128d _mm_mask_fmsub_pd(__m128d a, __mmask8 k, __m128d b, __m128d c);
VFMSUBxxxPD __m128d _mm_maskz_fmsub_pd(__mmask8 k, __m128d a, __m128d b, __m128d c);
VFMSUBxxxPD __m128d _mm_mask3_fmsub_pd(__m128d a, __m128d b, __m128d c, __mmask8 k);
VFMSUBxxxPD __m128d _mm_fmsub_pd (__m128d a, __m128d b, __m128d c);
VFMSUBxxxPD __m256d _mm256_fmsub_pd (__m256d a, __m256d b, __m256d c);

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal

Other Exceptions

VEX-encoded instructions, see Exceptions Type 2.
EVEX-encoded instructions, see Exceptions Type E2.

VFMSUB132PS/VFMSUB213PS/VFMSUB231PS—Fused Multiply-Subtract of Packed Single-Precision Floating-Point Values

Opcode/ Instruction	Op/En	64/32 bit Mode Support	CPUID Feature Flag	Description
VEX.NDS.128.66.0F38.W0 9A /r VFMSUB132PS xmm1, xmm2, xmm3/m128	A	V/V	FMA	Multiply packed single-precision floating-point values from xmm1 and xmm3/mem, subtract xmm2 and put result in xmm1.
VEX.NDS.128.66.0F38.W0 AA /r VFMSUB213PS xmm1, xmm2, xmm3/m128	A	V/V	FMA	Multiply packed single-precision floating-point values from xmm1 and xmm2, subtract xmm3/mem and put result in xmm1.
VEX.NDS.128.66.0F38.W0 BA /r VFMSUB231PS xmm1, xmm2, xmm3/m128	A	V/V	FMA	Multiply packed single-precision floating-point values from xmm2 and xmm3/mem, subtract xmm1 and put result in xmm1.
VEX.NDS.256.66.0F38.W0 9A /r VFMSUB132PS ymm1, ymm2, ymm3/m256	A	V/V	FMA	Multiply packed single-precision floating-point values from ymm1 and ymm3/mem, subtract ymm2 and put result in ymm1.
VEX.NDS.256.66.0F38.W0 AA /r VFMSUB213PS ymm1, ymm2, ymm3/m256	A	V/V	FMA	Multiply packed single-precision floating-point values from ymm1 and ymm2, subtract ymm3/mem and put result in ymm1.
VEX.NDS.256.66.0F38.0 BA /r VFMSUB231PS ymm1, ymm2, ymm3/m256	A	V/V	FMA	Multiply packed single-precision floating-point values from ymm2 and ymm3/mem, subtract ymm1 and put result in ymm1.
EVEX.NDS.128.66.0F38.W0 9A /r VFMSUB132PS xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst	B	V/V	AVX512VL AVX512F	Multiply packed single-precision floating-point values from xmm1 and xmm3/m128/m32bcst, subtract xmm2 and put result in xmm1.
EVEX.NDS.128.66.0F38.W0 AA /r VFMSUB213PS xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst	B	V/V	AVX512VL AVX512F	Multiply packed single-precision floating-point values from xmm1 and xmm2, subtract xmm3/m128/m32bcst and put result in xmm1.
EVEX.NDS.128.66.0F38.W0 BA /r VFMSUB231PS xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst	B	V/V	AVX512VL AVX512F	Multiply packed single-precision floating-point values from xmm2 and xmm3/m128/m32bcst, subtract xmm1 and put result in xmm1.
EVEX.NDS.256.66.0F38.W0 9A /r VFMSUB132PS ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst	B	V/V	AVX512VL AVX512F	Multiply packed single-precision floating-point values from ymm1 and ymm3/m256/m32bcst, subtract ymm2 and put result in ymm1.
EVEX.NDS.256.66.0F38.W0 AA /r VFMSUB213PS ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst	B	V/V	AVX512VL AVX512F	Multiply packed single-precision floating-point values from ymm1 and ymm2, subtract ymm3/m256/m32bcst and put result in ymm1.
EVEX.NDS.256.66.0F38.W0 BA /r VFMSUB231PS ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst	B	V/V	AVX512VL AVX512F	Multiply packed single-precision floating-point values from ymm2 and ymm3/m256/m32bcst, subtract ymm1 and put result in ymm1.
EVEX.NDS.512.66.0F38.W0 9A /r VFMSUB132PS zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst{er}	B	V/V	AVX512F	Multiply packed single-precision floating-point values from zmm1 and zmm3/m512/m32bcst, subtract zmm2 and put result in zmm1.
EVEX.NDS.512.66.0F38.W0 AA /r VFMSUB213PS zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst{er}	B	V/V	AVX512F	Multiply packed single-precision floating-point values from zmm1 and zmm2, subtract zmm3/m512/m32bcst and put result in zmm1.
EVEX.NDS.512.66.0F38.W0 BA /r VFMSUB231PS zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst{er}	B	V/V	AVX512F	Multiply packed single-precision floating-point values from zmm2 and zmm3/m512/m32bcst, subtract zmm1 and put result in zmm1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	NA	ModRM:reg (r, w)	VEX.vvvv (r)	ModRM:r/m (r)	NA
B	Full Vector	ModRM:reg (r, w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA

Description

Performs a set of SIMD multiply-subtract computation on packed single-precision floating-point values using three source operands and writes the multiply-subtract results in the destination operand. The destination operand is also the first source operand. The second operand must be a SIMD register. The third source operand can be a SIMD register or a memory location.

VFMSUB132PS: Multiplies the four, eight or sixteen packed single-precision floating-point values from the first source operand to the four, eight or sixteen packed single-precision floating-point values in the third source operand. From the infinite precision intermediate result, subtracts the four, eight or sixteen packed single-precision floating-point values in the second source operand, performs rounding and stores the resulting four, eight or sixteen packed single-precision floating-point values to the destination operand (first source operand).

VFMSUB213PS: Multiplies the four, eight or sixteen packed single-precision floating-point values from the second source operand to the four, eight or sixteen packed single-precision floating-point values in the first source operand. From the infinite precision intermediate result, subtracts the four, eight or sixteen packed single-precision floating-point values in the third source operand, performs rounding and stores the resulting four, eight or sixteen packed single-precision floating-point values to the destination operand (first source operand).

VFMSUB231PS: Multiplies the four, eight or sixteen packed single-precision floating-point values from the second source to the four, eight or sixteen packed single-precision floating-point values in the third source operand. From the infinite precision intermediate result, subtracts the four, eight or sixteen packed single-precision floating-point values in the first source operand, performs rounding and stores the resulting four, eight or sixteen packed single-precision floating-point values to the destination operand (first source operand).

EVEX encoded versions: The destination operand (also first source operand) and the second source operand are ZMM/YMM/XMM register. The third source operand is a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 32-bit memory location. The destination operand is conditionally updated with write mask k1.

VEX.256 encoded version: The destination operand (also first source operand) is a YMM register and encoded in reg_field. The second source operand is a YMM register and encoded in VEX.vvvv. The third source operand is a YMM register or a 256-bit memory location and encoded in rm_field.

VEX.128 encoded version: The destination operand (also first source operand) is a XMM register and encoded in reg_field. The second source operand is a XMM register and encoded in VEX.vvvv. The third source operand is a XMM register or a 128-bit memory location and encoded in rm_field. The upper 128 bits of the YMM destination register are zeroed.

Operation

In the operations below, "*" and "-" symbols represent multiplication and subtraction with infinite precision inputs and outputs (no rounding).

VFMSUB132PS DEST, SRC2, SRC3 (VEX encoded version)

```
IF (VEX.128) THEN
    MAXNUM ←2
ELSEIF (VEX.256)
    MAXNUM ← 4
FI
For i = 0 to MAXNUM-1 {
    n ← 32*i;
    DEST[n+31:n] ← RoundFPControl_MXCSR(DEST[n+31:n]*SRC3[n+31:n] - SRC2[n+31:n])
}
IF (VEX.128) THEN
    DEST[MAXVL-1:128] ← 0
ELSEIF (VEX.256)
    DEST[MAXVL-1:256] ← 0
FI
```

VFMSUB213PS DEST, SRC2, SRC3 (VEX encoded version)

```
IF (VEX.128) THEN
    MAXNUM ←2
ELSEIF (VEX.256)
    MAXNUM ← 4
FI
For i = 0 to MAXNUM-1 {
    n ← 32*i;
    DEST[n+31:n] ← RoundFPControl_MXCSR(SRC2[n+31:n]*DEST[n+31:n] - SRC3[n+31:n])
}
IF (VEX.128) THEN
    DEST[MAXVL-1:128] ← 0
ELSEIF (VEX.256)
    DEST[MAXVL-1:256] ← 0
FI
```

VFMSUB231PS DEST, SRC2, SRC3 (VEX encoded version)

```
IF (VEX.128) THEN
    MAXNUM ←2
ELSEIF (VEX.256)
    MAXNUM ← 4
FI
For i = 0 to MAXNUM-1 {
    n ← 32*i;
    DEST[n+31:n] ← RoundFPControl_MXCSR(SRC2[n+31:n]*SRC3[n+31:n] - DEST[n+31:n])
}
IF (VEX.128) THEN
    DEST[MAXVL-1:128] ← 0
ELSEIF (VEX.256)
    DEST[MAXVL-1:256] ← 0
FI
```

VFMSUB132PS DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a register)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
IF (VL = 512) AND (EVEX.b = 1)
    THEN
        SET_RM(EVEX.RC);
    ELSE
        SET_RM(MXCSR.RM);
FI;
FOR j ← 0 TO KL-1
    i ← j * 32
    IF k1[j] OR *no writemask*
        THEN DEST[i+31:i] ←
            RoundFPControl(DEST[i+31:i]*SRC3[i+31:i] - SRC2[i+31:i])
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE                            ; zeroing-masking
                    DEST[i+31:i] ← 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0
```

VFMSUB132PS DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a memory source)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)

FOR j ← 0 TO KL-1
    i ← j * 32
    IF k1[j] OR *no writemask*
        THEN
            IF (EVEX.b = 1)
                THEN
                    DEST[i+31:i] ←
            RoundFPControl_MXCSR(DEST[i+31:i]*SRC3[31:0] - SRC2[i+31:i])
                ELSE
                    DEST[i+31:i] ←
            RoundFPControl_MXCSR(DEST[i+31:i]*SRC3[i+31:i] - SRC2[i+31:i])
            FI;
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE                            ; zeroing-masking
                    DEST[i+31:i] ← 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0
```

VFMSUB213PS DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a register)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
IF (VL = 512) AND (EVEX.b = 1)
    THEN
        SET_RM(EVEX.RC);
    ELSE
        SET_RM(MXCSR.RM);
FI;
FOR j ← 0 TO KL-1
    i ← j * 32
    IF k1[j] OR *no writemask*
        THEN DEST[i+31:i] ←
            RoundFPControl_MXCSR(SRC2[i+31:i]*DEST[i+31:i] - SRC3[i+31:i])
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE                            ; zeroing-masking
                    DEST[i+31:i] ← 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0
```

VFMSUB213PS DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a memory source)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)

FOR j ← 0 TO KL-1
    i ← j * 32
    IF k1[j] OR *no writemask*
        THEN
            IF (EVEX.b = 1)
                THEN
                    DEST[i+31:i] ←
            RoundFPControl_MXCSR(SRC2[i+31:i]*DEST[i+31:i] - SRC3[31:0])
                ELSE
                    DEST[i+31:i] ←
            RoundFPControl_MXCSR(SRC2[i+31:i]*DEST[i+31:i] - SRC3[i+31:i])
            FI;
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE                            ; zeroing-masking
                    DEST[i+31:i] ← 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0
```

VFMSUB231PS DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a register)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
IF (VL = 512) AND (EVEX.b = 1)
    THEN
        SET_RM(EVEX.RC);
    ELSE
        SET_RM(MXCSR.RM);
FI;
FOR j ← 0 TO KL-1
    i ← j * 32
    IF k1[j] OR *no writemask*
        THEN DEST[i+31:i] ←
            RoundFPControl_MXCSR(SRC2[i+31:i]*SRC3[i+31:i] - DEST[i+31:i])
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE                            ; zeroing-masking
                    DEST[i+31:i] ← 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0
```

VFMSUB231PS DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a memory source)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)

FOR j ← 0 TO KL-1
    i ← j * 32
    IF k1[j] OR *no writemask*
        THEN
            IF (EVEX.b = 1)
                THEN
                    DEST[i+31:i] ←
            RoundFPControl_MXCSR(SRC2[i+31:i]*SRC3[31:0] - DEST[i+31:i])
                ELSE
                    DEST[i+31:i] ←
            RoundFPControl_MXCSR(SRC2[i+31:i]*SRC3[i+31:i] - DEST[i+31:i])
            FI;
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE                            ; zeroing-masking
                    DEST[i+31:i] ← 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VFMSUBxxxPS __m512 _mm512_fmsub_ps(__m512 a, __m512 b, __m512 c);
VFMSUBxxxPS __m512 _mm512_fmsub_round_ps(__m512 a, __m512 b, __m512 c, int r);
VFMSUBxxxPS __m512 _mm512_mask_fmsub_ps(__m512 a, __mmask16 k, __m512 b, __m512 c);
VFMSUBxxxPS __m512 _mm512_maskz_fmsub_ps(__mmask16 k, __m512 a, __m512 b, __m512 c);
VFMSUBxxxPS __m512 _mm512_mask3_fmsub_ps(__m512 a, __m512 b, __m512 c, __mmask16 k);
VFMSUBxxxPS __m512 _mm512_mask_fmsub_round_ps(__m512 a, __mmask16 k, __m512 b, __m512 c, int r);
VFMSUBxxxPS __m512 _mm512_maskz_fmsub_round_ps(__mmask16 k, __m512 a, __m512 b, __m512 c, int r);
VFMSUBxxxPS __m512 _mm512_mask3_fmsub_round_ps(__m512 a, __m512 b, __m512 c, __mmask16 k, int r);
VFMSUBxxxPS __m256 _mm256_mask_fmsub_ps(__m256 a, __mmask8 k, __m256 b, __m256 c);
VFMSUBxxxPS __m256 _mm256_maskz_fmsub_ps(__mmask8 k, __m256 a, __m256 b, __m256 c);
VFMSUBxxxPS __m256 _mm256_mask3_fmsub_ps(__m256 a, __m256 b, __m256 c, __mmask8 k);
VFMSUBxxxPS __m128 _mm_mask_fmsub_ps(__m128 a, __mmask8 k, __m128 b, __m128 c);
VFMSUBxxxPS __m128 _mm_maskz_fmsub_ps(__mmask8 k, __m128 a, __m128 b, __m128 c);
VFMSUBxxxPS __m128 _mm_mask3_fmsub_ps(__m128 a, __m128 b, __m128 c, __mmask8 k);
VFMSUBxxxPS __m128 _mm_fmsub_ps (__m128 a, __m128 b, __m128 c);
VFMSUBxxxPS __m256 _mm256_fmsub_ps (__m256 a, __m256 b, __m256 c);

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal

Other Exceptions

VEX-encoded instructions, see Exceptions Type 2.
EVEX-encoded instructions, see Exceptions Type E2.

VFMSUB132SD/VFMSUB213SD/VFMSUB231SD—Fused Multiply-Subtract of Scalar Double-Precision Floating-Point Values

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
VEX.DDS.LIG.66.0F38.W1 9B /r VFMSUB132SD xmm1, xmm2, xmm3/m64	A	V/V	FMA	Multiply scalar double-precision floating-point value from xmm1 and xmm3/m64, subtract xmm2 and put result in xmm1.
VEX.DDS.LIG.66.0F38.W1 AB /r VFMSUB213SD xmm1, xmm2, xmm3/m64	A	V/V	FMA	Multiply scalar double-precision floating-point value from xmm1 and xmm2, subtract xmm3/m64 and put result in xmm1.
VEX.DDS.LIG.66.0F38.W1 BB /r VFMSUB231SD xmm1, xmm2, xmm3/m64	A	V/V	FMA	Multiply scalar double-precision floating-point value from xmm2 and xmm3/m64, subtract xmm1 and put result in xmm1.
EVEX.DDS.LIG.66.0F38.W1 9B /r VFMSUB132SD xmm1 {k1}{z}, xmm2, xmm3/m64{er}	B	V/V	AVX512F	Multiply scalar double-precision floating-point value from xmm1 and xmm3/m64, subtract xmm2 and put result in xmm1.
EVEX.DDS.LIG.66.0F38.W1 AB /r VFMSUB213SD xmm1 {k1}{z}, xmm2, xmm3/m64{er}	B	V/V	AVX512F	Multiply scalar double-precision floating-point value from xmm1 and xmm2, subtract xmm3/m64 and put result in xmm1.
EVEX.DDS.LIG.66.0F38.W1 BB /r VFMSUB231SD xmm1 {k1}{z}, xmm2, xmm3/m64{er}	B	V/V	AVX512F	Multiply scalar double-precision floating-point value from xmm2 and xmm3/m64, subtract xmm1 and put result in xmm1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	NA	ModRM:reg (r, w)	VEX.vvvv (r)	ModRM:r/m (r)	NA
B	Tuple1 Scalar	ModRM:reg (r, w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA

Description

Performs a SIMD multiply-subtract computation on the low packed double-precision floating-point values using three source operands and writes the multiply-subtract result in the destination operand. The destination operand is also the first source operand. The second operand must be a XMM register. The third source operand can be a XMM register or a 64-bit memory location.

VFMSUB132SD: Multiplies the low packed double-precision floating-point value from the first source operand to the low packed double-precision floating-point value in the third source operand. From the infinite precision intermediate result, subtracts the low packed double-precision floating-point values in the second source operand, performs rounding and stores the resulting packed double-precision floating-point value to the destination operand (first source operand).

VFMSUB213SD: Multiplies the low packed double-precision floating-point value from the second source operand to the low packed double-precision floating-point value in the first source operand. From the infinite precision intermediate result, subtracts the low packed double-precision floating-point value in the third source operand, performs rounding and stores the resulting packed double-precision floating-point value to the destination operand (first source operand).

VFMSUB231SD: Multiplies the low packed double-precision floating-point value from the second source to the low packed double-precision floating-point value in the third source operand. From the infinite precision intermediate result, subtracts the low packed double-precision floating-point value in the first source operand, performs rounding and stores the resulting packed double-precision floating-point value to the destination operand (first source operand).

VEX.128 and EVEX encoded version: The destination operand (also first source operand) is encoded in reg_field. The second source operand is encoded in VEX.vvvv/EVEX.vvvv. The third source operand is encoded in rm_field. Bits 127:64 of the destination are unchanged. Bits MAXVL-1:128 of the destination register are zeroed.

EVEX encoded version: The low quadword element of the destination is updated according to the writemask.

Compiler tools may optionally support a complementary mnemonic for each instruction mnemonic listed in the opcode/instruction column of the summary table. The behavior of the complementary mnemonic in situations involving NANs are governed by the definition of the instruction mnemonic defined in the opcode/instruction column.

Operation

In the operations below, "*" and "-" symbols represent multiplication and subtraction with infinite precision inputs and outputs (no rounding).

VFMSUB132SD DEST, SRC2, SRC3 (EVEX encoded version)

```
IF (EVEX.b = 1) and SRC3 *is a register*
    THEN
        SET_RM(EVEX.RC);
    ELSE
        SET_RM(MXCSR.RM);
FI;
IF k1[0] or *no writemask*
    THEN    DEST[63:0] ← RoundFPControl(DEST[63:0]*SRC3[63:0] - SRC2[63:0])
    ELSE
        IF *merging-masking*                ; merging-masking
            THEN *DEST[63:0] remains unchanged*
            ELSE                            ; zeroing-masking
                THEN DEST[63:0] ← 0
        FI;
FI;
DEST[127:64] ← DEST[127:64]
DEST[MAXVL-1:128] ← 0
```

VFMSUB213SD DEST, SRC2, SRC3 (EVEX encoded version)

```
IF (EVEX.b = 1) and SRC3 *is a register*
    THEN
        SET_RM(EVEX.RC);
    ELSE
        SET_RM(MXCSR.RM);
FI;
IF k1[0] or *no writemask*
    THEN    DEST[63:0] ← RoundFPControl(SRC2[63:0]*DEST[63:0] - SRC3[63:0])
    ELSE
        IF *merging-masking*                ; merging-masking
            THEN *DEST[63:0] remains unchanged*
            ELSE                            ; zeroing-masking
                THEN DEST[63:0] ← 0
        FI;
FI;
DEST[127:64] ← DEST[127:64]
DEST[MAXVL-1:128] ← 0
```

VFMSUB231SD DEST, SRC2, SRC3 (EVEX encoded version)

```
IF (EVEX.b = 1) and SRC3 *is a register*
    THEN
        SET_RM(EVEX.RC);
    ELSE
        SET_RM(MXCSR.RM);
FI;
IF k1[0] or *no writemask*
    THEN    DEST[63:0] ← RoundFPControl(SRC2[63:0]*SRC3[63:0] - DEST[63:0])
    ELSE
        IF *merging-masking*                ; merging-masking
            THEN *DEST[63:0] remains unchanged*
            ELSE                            ; zeroing-masking
                THEN DEST[63:0] ← 0
        FI;
FI;
DEST[127:64] ← DEST[127:64]
DEST[MAXVL-1:128] ← 0
```

VFMSUB132SD DEST, SRC2, SRC3 (VEX encoded version)

```
DEST[63:0] ←RoundFPControl_MXCSR(DEST[63:0]*SRC3[63:0] - SRC2[63:0])
DEST[127:64] ←DEST[127:64]
DEST[MAXVL-1:128] ←0
```

VFMSUB213SD DEST, SRC2, SRC3 (VEX encoded version)

```
DEST[63:0] ←RoundFPControl_MXCSR(SRC2[63:0]*DEST[63:0] - SRC3[63:0])
DEST[127:64] ←DEST[127:64]
DEST[MAXVL-1:128] ←0
```

VFMSUB231SD DEST, SRC2, SRC3 (VEX encoded version)

```
DEST[63:0] ←RoundFPControl_MXCSR(SRC2[63:0]*SRC3[63:0] - DEST[63:0])
DEST[127:64] ←DEST[127:64]
DEST[MAXVL-1:128] ←0
```

Intel C/C++ Compiler Intrinsic Equivalent

VFMSUBxxxSD __m128d _mm_fmsub_round_sd(__m128d a, __m128d b, __m128d c, int r);
VFMSUBxxxSD __m128d _mm_mask_fmsub_sd(__m128d a, __mmask8 k, __m128d b, __m128d c);
VFMSUBxxxSD __m128d _mm_maskz_fmsub_sd(__mmask8 k, __m128d a, __m128d b, __m128d c);
VFMSUBxxxSD __m128d _mm_mask3_fmsub_sd(__m128d a, __m128d b, __m128d c, __mmask8 k);
VFMSUBxxxSD __m128d _mm_mask_fmsub_round_sd(__m128d a, __mmask8 k, __m128d b, __m128d c, int r);
VFMSUBxxxSD __m128d _mm_maskz_fmsub_round_sd(__mmask8 k, __m128d a, __m128d b, __m128d c, int r);
VFMSUBxxxSD __m128d _mm_mask3_fmsub_round_sd(__m128d a, __m128d b, __m128d c, __mmask8 k, int r);
VFMSUBxxxSD __m128d _mm_fmsub_sd (__m128d a, __m128d b, __m128d c);

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal

Other Exceptions

VEX-encoded instructions, see Exceptions Type 3.
EVEX-encoded instructions, see Exceptions Type E3.

VFMSUB132SS/VFMSUB213SS/VFMSUB231SS—Fused Multiply-Subtract of Scalar Single-Precision Floating-Point Values

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
VEX.DDS.LIG.66.0F38.W0 9B /r VFMSUB132SS xmm1, xmm2, xmm3/m32	A	V/V	FMA	Multiply scalar single-precision floating-point value from xmm1 and xmm3/m32, subtract xmm2 and put result in xmm1.
VEX.DDS.LIG.66.0F38.W0 AB /r VFMSUB213SS xmm1, xmm2, xmm3/m32	A	V/V	FMA	Multiply scalar single-precision floating-point value from xmm1 and xmm2, subtract xmm3/m32 and put result in xmm1.
VEX.DDS.LIG.66.0F38.W0 BB /r VFMSUB231SS xmm1, xmm2, xmm3/m32	A	V/V	FMA	Multiply scalar single-precision floating-point value from xmm2 and xmm3/m32, subtract xmm1 and put result in xmm1.
EVEX.DDS.LIG.66.0F38.W0 9B /r VFMSUB132SS xmm1 {k1}{z}, xmm2, xmm3/m32{er}	B	V/V	AVX512F	Multiply scalar single-precision floating-point value from xmm1 and xmm3/m32, subtract xmm2 and put result in xmm1.
EVEX.DDS.LIG.66.0F38.W0 AB /r VFMSUB213SS xmm1 {k1}{z}, xmm2, xmm3/m32{er}	B	V/V	AVX512F	Multiply scalar single-precision floating-point value from xmm1 and xmm2, subtract xmm3/m32 and put result in xmm1.
EVEX.DDS.LIG.66.0F38.W0 BB /r VFMSUB231SS xmm1 {k1}{z}, xmm2, xmm3/m32{er}	B	V/V	AVX512F	Multiply scalar single-precision floating-point value from xmm2 and xmm3/m32, subtract xmm1 and put result in xmm1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	NA	ModRM:reg (r, w)	VEX.vvvv (r)	ModRM:r/m (r)	NA
B	Tuple1 Scalar	ModRM:reg (r, w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA

Description

Performs a SIMD multiply-subtract computation on the low packed single-precision floating-point values using three source operands and writes the multiply-subtract result in the destination operand. The destination operand is also the first source operand. The second operand must be a XMM register. The third source operand can be a XMM register or a 32-bit memory location.

VFMSUB132SS: Multiplies the low packed single-precision floating-point value from the first source operand to the low packed single-precision floating-point value in the third source operand. From the infinite precision intermediate result, subtracts the low packed single-precision floating-point values in the second source operand, performs rounding and stores the resulting packed single-precision floating-point value to the destination operand (first source operand).

VFMSUB213SS: Multiplies the low packed single-precision floating-point value from the second source operand to the low packed single-precision floating-point value in the first source operand. From the infinite precision intermediate result, subtracts the low packed single-precision floating-point value in the third source operand, performs rounding and stores the resulting packed single-precision floating-point value to the destination operand (first source operand).

VFMSUB231SS: Multiplies the low packed single-precision floating-point value from the second source to the low packed single-precision floating-point value in the third source operand. From the infinite precision intermediate result, subtracts the low packed single-precision floating-point value in the first source operand, performs rounding and stores the resulting packed single-precision floating-point value to the destination operand (first source operand).

VEX.128 and EVEX encoded version: The destination operand (also first source operand) is encoded in reg_field. The second source operand is encoded in VEX.vvvv/EVEX.vvvv. The third source operand is encoded in rm_field. Bits 127:32 of the destination are unchanged. Bits MAXVL-1:128 of the destination register are zeroed.

EVEX encoded version: The low doubleword element of the destination is updated according to the writemask.

Compiler tools may optionally support a complementary mnemonic for each instruction mnemonic listed in the opcode/instruction column of the summary table. The behavior of the complementary mnemonic in situations involving NANs are governed by the definition of the instruction mnemonic defined in the opcode/instruction column.

Operation

In the operations below, "*" and "-" symbols represent multiplication and subtraction with infinite precision inputs and outputs (no rounding).

VFMSUB132SS DEST, SRC2, SRC3 (EVEX encoded version)

```
IF (EVEX.b = 1) and SRC3 *is a register*
    THEN
        SET_RM(EVEX.RC);
    ELSE
        SET_RM(MXCSR.RM);
FI;
IF k1[0] or *no writemask*
    THEN    DEST[31:0] ← RoundFPControl(DEST[31:0]*SRC3[31:0] - SRC2[31:0])
    ELSE
        IF *merging-masking*                ; merging-masking
            THEN *DEST[31:0] remains unchanged*
            ELSE                            ; zeroing-masking
                THEN DEST[31:0] ← 0
        FI;
FI;
DEST[127:32] ← DEST[127:32]
DEST[MAXVL-1:128] ← 0
```

VFMSUB213SS DEST, SRC2, SRC3 (EVEX encoded version)

```
IF (EVEX.b = 1) and SRC3 *is a register*
    THEN
        SET_RM(EVEX.RC);
    ELSE
        SET_RM(MXCSR.RM);
FI;
IF k1[0] or *no writemask*
    THEN    DEST[31:0] ← RoundFPControl(SRC2[31:0]*DEST[31:0] - SRC3[31:0])
    ELSE
        IF *merging-masking*                ; merging-masking
            THEN *DEST[31:0] remains unchanged*
            ELSE                            ; zeroing-masking
                THEN DEST[31:0] ← 0
        FI;
FI;
DEST[127:32] ← DEST[127:32]
DEST[MAXVL-1:128] ← 0
```

VFMSUB231SS DEST, SRC2, SRC3 (EVEX encoded version)

```
IF (EVEX.b = 1) and SRC3 *is a register*
    THEN
        SET_RM(EVEX.RC);
    ELSE
        SET_RM(MXCSR.RM);
FI;
IF k1[0] or *no writemask*
    THEN    DEST[31:0] ← RoundFPControl(SRC2[31:0]*SRC3[63:0] - DEST[31:0])
    ELSE
        IF *merging-masking*                    ; merging-masking
            THEN *DEST[31:0] remains unchanged*
            ELSE                                ; zeroing-masking
                THEN DEST[31:0] ← 0
        FI;
FI;
DEST[127:32] ← DEST[127:32]
DEST[MAXVL-1:128] ← 0
```

VFMSUB132SS DEST, SRC2, SRC3 (VEX encoded version)

```
DEST[31:0] ←RoundFPControl_MXCSR(DEST[31:0]*SRC3[31:0] - SRC2[31:0])
DEST[127:32] ←DEST[127:32]
DEST[MAXVL-1:128] ←0
```

VFMSUB213SS DEST, SRC2, SRC3 (VEX encoded version)

```
DEST[31:0] ←RoundFPControl_MXCSR(SRC2[31:0]*DEST[31:0] - SRC3[31:0])
DEST[127:32] ←DEST[127:32]
DEST[MAXVL-1:128] ←0
```

VFMSUB231SS DEST, SRC2, SRC3 (VEX encoded version)

```
DEST[31:0] ←RoundFPControl_MXCSR(SRC2[31:0]*SRC3[31:0] - DEST[31:0])
DEST[127:32] ←DEST[127:32]
DEST[MAXVL-1:128] ←0
```

Intel C/C++ Compiler Intrinsic Equivalent

VFMSUBxxxSS __m128 _mm_fmsub_round_ss(__m128 a, __m128 b, __m128 c, int r);
VFMSUBxxxSS __m128 _mm_mask_fmsub_ss(__m128 a, __mmask8 k, __m128 b, __m128 c);
VFMSUBxxxSS __m128 _mm_maskz_fmsub_ss(__mmask8 k, __m128 a, __m128 b, __m128 c);
VFMSUBxxxSS __m128 _mm_mask3_fmsub_ss(__m128 a, __m128 b, __m128 c, __mmask8 k);
VFMSUBxxxSS __m128 _mm_mask_fmsub_round_ss(__m128 a, __mmask8 k, __m128 b, __m128 c, int r);
VFMSUBxxxSS __m128 _mm_maskz_fmsub_round_ss(__mmask8 k, __m128 a, __m128 b, __m128 c, int r);
VFMSUBxxxSS __m128 _mm_mask3_fmsub_round_ss(__m128 a, __m128 b, __m128 c, __mmask8 k, int r);
VFMSUBxxxSS __m128 _mm_fmsub_ss (__m128 a, __m128 b, __m128 c);

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal

Other Exceptions

VEX-encoded instructions, see Exceptions Type 3.
EVEX-encoded instructions, see Exceptions Type E3.

VFNMADD132PD/VFNMADD213PD/VFNMADD231PD—Fused Negative Multiply-Add of Packed Double-Precision Floating-Point Values

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
VEX.NDS.128.66.0F38.W1 9C /r VFNMADD132PD xmm1, xmm2, xmm3/m128	A	V/V	FMA	Multiply packed double-precision floating-point values from xmm1 and xmm3/mem, negate the multiplication result and add to xmm2 and put result in xmm1.
VEX.NDS.128.66.0F38.W1 AC /r VFNMADD213PD xmm1, xmm2, xmm3/m128	A	V/V	FMA	Multiply packed double-precision floating-point values from xmm1 and xmm2, negate the multiplication result and add to xmm3/mem and put result in xmm1.
VEX.NDS.128.66.0F38.W1 BC /r VFNMADD231PD xmm1, xmm2, xmm3/m128	A	V/V	FMA	Multiply packed double-precision floating-point values from xmm2 and xmm3/mem, negate the multiplication result and add to xmm1 and put result in xmm1.
VEX.NDS.256.66.0F38.W1 9C /r VFNMADD132PD ymm1, ymm2, ymm3/m256	A	V/V	FMA	Multiply packed double-precision floating-point values from ymm1 and ymm3/mem, negate the multiplication result and add to ymm2 and put result in ymm1.
VEX.NDS.256.66.0F38.W1 AC /r VFNMADD213PD ymm1, ymm2, ymm3/m256	A	V/V	FMA	Multiply packed double-precision floating-point values from ymm1 and ymm2, negate the multiplication result and add to ymm3/mem and put result in ymm1.
VEX.NDS.256.66.0F38.W1 BC /r VFNMADD231PD ymm1, ymm2, ymm3/m256	A	V/V	FMA	Multiply packed double-precision floating-point values from ymm2 and ymm3/mem, negate the multiplication result and add to ymm1 and put result in ymm1.
EVEX.NDS.128.66.0F38.W1 9C /r VFNMADD132PD xmm0 {k1}{z}, xmm1, xmm2/m128/m64bcst	B	V/V	AVX512VL AVX512F	Multiply packed double-precision floating-point values from xmm1 and xmm3/m128/m64bcst, negate the multiplication result and add to xmm2 and put result in xmm1.
EVEX.NDS.128.66.0F38.W1 AC /r VFNMADD213PD xmm1 {k1}{z}, xmm2, xmm3/m128/m64bcst	B	V/V	AVX512VL AVX512F	Multiply packed double-precision floating-point values from xmm1 and xmm2, negate the multiplication result and add to xmm3/m128/m64bcst and put result in xmm1.
EVEX.NDS.128.66.0F38.W1 BC /r VFNMADD231PD xmm1 {k1}{z}, xmm2, xmm3/m128/m64bcst	B	V/V	AVX512VL AVX512F	Multiply packed double-precision floating-point values from xmm2 and xmm3/m128/m64bcst, negate the multiplication result and add to xmm1 and put result in xmm1.
EVEX.NDS.256.66.0F38.W1 9C /r VFNMADD132PD ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst	B	V/V	AVX512VL AVX512F	Multiply packed double-precision floating-point values from ymm1 and ymm3/m256/m64bcst, negate the multiplication result and add to ymm2 and put result in ymm1.
EVEX.NDS.256.66.0F38.W1 AC /r VFNMADD213PD ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst	B	V/V	AVX512VL AVX512F	Multiply packed double-precision floating-point values from ymm1 and ymm2, negate the multiplication result and add to ymm3/m256/m64bcst and put result in ymm1.
EVEX.NDS.256.66.0F38.W1 BC /r VFNMADD231PD ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst	B	V/V	AVX512VL AVX512F	Multiply packed double-precision floating-point values from ymm2 and ymm3/m256/m64bcst, negate the multiplication result and add to ymm1 and put result in ymm1.
EVEX.NDS.512.66.0F38.W1 9C /r VFNMADD132PD zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst{er}	B	V/V	AVX512F	Multiply packed double-precision floating-point values from zmm1 and zmm3/m512/m64bcst, negate the multiplication result and add to zmm2 and put result in zmm1.
EVEX.NDS.512.66.0F38.W1 AC /r VFNMADD213PD zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst{er}	B	V/V	AVX512F	Multiply packed double-precision floating-point values from zmm1 and zmm2, negate the multiplication result and add to zmm3/m512/m64bcst and put result in zmm1.
EVEX.NDS.512.66.0F38.W1 BC /r VFNMADD231PD zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst{er}	B	V/V	AVX512F	Multiply packed double-precision floating-point values from zmm2 and zmm3/m512/m64bcst, negate the multiplication result and add to zmm1 and put result in zmm1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	NA	ModRM:reg (r, w)	VEX.vvvv (r)	ModRM:r/m (r)	NA
B	Full Vector	ModRM:reg (r, w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA

Description

VFNMADD132PD: Multiplies the two, four or eight packed double-precision floating-point values from the first source operand to the two, four or eight packed double-precision floating-point values in the third source operand, adds the negated infinite precision intermediate result to the two, four or eight packed double-precision floating-point values in the second source operand, performs rounding and stores the resulting two, four or eight packed double-precision floating-point values to the destination operand (first source operand).

VFNMADD213PD: Multiplies the two, four or eight packed double-precision floating-point values from the second source operand to the two, four or eight packed double-precision floating-point values in the first source operand, adds the negated infinite precision intermediate result to the two, four or eight packed double-precision floating-point values in the third source operand, performs rounding and stores the resulting two, four or eight packed double-precision floating-point values to the destination operand (first source operand).

VFNMADD231PD: Multiplies the two, four or eight packed double-precision floating-point values from the second source to the two, four or eight packed double-precision floating-point values in the third source operand, the negated infinite precision intermediate result to the two, four or eight packed double-precision floating-point values in the first source operand, performs rounding and stores the resulting two, four or eight packed double-precision floating-point values to the destination operand (first source operand).

EVEX encoded versions: The destination operand (also first source operand) and the second source operand are ZMM/YMM/XMM register. The third source operand is a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 64-bit memory location. The destination operand is conditionally updated with write mask k1.

VEX.256 encoded version: The destination operand (also first source operand) is a YMM register and encoded in reg_field. The second source operand is a YMM register and encoded in VEX.vvvv. The third source operand is a YMM register or a 256-bit memory location and encoded in rm_field.

VEX.128 encoded version: The destination operand (also first source operand) is a XMM register and encoded in reg_field. The second source operand is a XMM register and encoded in VEX.vvvv. The third source operand is a XMM register or a 128-bit memory location and encoded in rm_field. The upper 128 bits of the YMM destination register are zeroed.

Operation

In the operations below, "*" and "-" symbols represent multiplication and subtraction with infinite precision inputs and outputs (no rounding).

VFNMADD132PD DEST, SRC2, SRC3 (VEX encoded version)

```
IF (VEX.128) THEN
    MAXNUM ←2
ELSEIF (VEX.256)
    MAXNUM ← 4
FI
For i = 0 to MAXNUM-1 {
    n ← 64*i;
    DEST[n+63:n] ← RoundFPControl_MXCSR(-(DEST[n+63:n]*SRC3[n+63:n]) + SRC2[n+63:n])
}
IF (VEX.128) THEN
    DEST[MAXVL-1:128] ← 0
ELSEIF (VEX.256)
    DEST[MAXVL-1:256] ← 0
FI
```

VFNMADD213PD DEST, SRC2, SRC3 (VEX encoded version)

```
IF (VEX.128) THEN
    MAXNUM ←2
ELSEIF (VEX.256)
    MAXNUM ← 4
FI
For i = 0 to MAXNUM-1 {
    n ← 64*i;
    DEST[n+63:n] ← RoundFPControl_MXCSR(-(SRC2[n+63:n]*DEST[n+63:n]) + SRC3[n+63:n])
}
IF (VEX.128) THEN
    DEST[MAXVL-1:128] ← 0
ELSEIF (VEX.256)
    DEST[MAXVL-1:256] ← 0
FI
```

VFNMADD231PD DEST, SRC2, SRC3 (VEX encoded version)

```
IF (VEX.128) THEN
    MAXNUM ←2
ELSEIF (VEX.256)
    MAXNUM ← 4
FI
For i = 0 to MAXNUM-1 {
    n ← 64*i;
    DEST[n+63:n] ← RoundFPControl_MXCSR(-(SRC2[n+63:n]*SRC3[n+63:n]) + DEST[n+63:n])
}
IF (VEX.128) THEN
    DEST[MAXVL-1:128] ← 0
ELSEIF (VEX.256)
    DEST[MAXVL-1:256] ← 0
FI
```

VFNMADD132PD DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a register)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
IF (VL = 512) AND (EVEX.b = 1)
    THEN
        SET_RM(EVEX.RC);
    ELSE
        SET_RM(MXCSR.RM);
FI;
FOR j ← 0 TO KL-1
    i ← j * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+63:i] ←
            RoundFPControl(-(DEST[i+63:i]*SRC3[i+63:i]) + SRC2[i+63:i])
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE                            ; zeroing-masking
                    DEST[i+63:i] ← 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0
```

VFNMADD132PD DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a memory source)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)

FOR j ← 0 TO KL-1
    i ← j * 64
    IF k1[j] OR *no writemask*
        THEN
            IF (EVEX.b = 1)
                THEN
                    DEST[i+63:i] ←
            RoundFPControl_MXCSR(-(DEST[i+63:i]*SRC3[63:0]) + SRC2[i+63:i])
                ELSE
                    DEST[i+63:i] ←
            RoundFPControl_MXCSR(-(DEST[i+63:i]*SRC3[i+63:i]) + SRC2[i+63:i])
            FI;
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE                            ; zeroing-masking
                    DEST[i+63:i] ← 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0
```

VFNMADD213PD DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a register)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
IF (VL = 512) AND (EVEX.b = 1)
    THEN
        SET_RM(EVEX.RC);
    ELSE
        SET_RM(MXCSR.RM);
FI;
FOR j ← 0 TO KL-1
    i ← j * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+63:i] ←
            RoundFPControl(-(SRC2[i+63:i]*DEST[i+63:i]) + SRC3[i+63:i])
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE                            ; zeroing-masking
                    DEST[i+63:i] ← 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0
```

VFNMADD213PD DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a memory source)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)

FOR j ← 0 TO KL-1
    i ← j * 64
    IF k1[j] OR *no writemask*
        THEN
            IF (EVEX.b = 1)
                THEN
                    DEST[i+63:i] ←
            RoundFPControl_MXCSR(-(SRC2[i+63:i]*DEST[i+63:i]) + SRC3[63:0])
                ELSE
                    DEST[i+63:i] ←
            RoundFPControl_MXCSR(-(SRC2[i+63:i]*DEST[i+63:i]) + SRC3[i+63:i])
            FI;
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE                            ; zeroing-masking
                    DEST[i+63:i] ← 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0
```

VFNMADD231PD DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a register)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
IF (VL = 512) AND (EVEX.b = 1)
    THEN
        SET_RM(EVEX.RC);
    ELSE
        SET_RM(MXCSR.RM);
FI;
FOR j ← 0 TO KL-1
    i ← j * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+63:i] ←
            RoundFPControl(-(SRC2[i+63:i]*SRC3[i+63:i]) + DEST[i+63:i])
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE                            ; zeroing-masking
                    DEST[i+63:i] ← 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0
```

VFNMADD231PD DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a memory source)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)

FOR j ← 0 TO KL-1
    i ← j * 64
    IF k1[j] OR *no writemask*
        THEN
            IF (EVEX.b = 1)
                THEN
                    DEST[i+63:i] ←
            RoundFPControl_MXCSR(-(SRC2[i+63:i]*SRC3[63:0]) + DEST[i+63:i])
                ELSE
                    DEST[i+63:i] ←
            RoundFPControl_MXCSR(-(SRC2[i+63:i]*SRC3[i+63:i]) + DEST[i+63:i])
            FI;
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE                            ; zeroing-masking
                    DEST[i+63:i] ← 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VFNMADDxxxPD __m512d _mm512_fnmadd_pd(__m512d a, __m512d b, __m512d c);
VFNMADDxxxPD __m512d _mm512_fnmadd_round_pd(__m512d a, __m512d b, __m512d c, int r);
VFNMADDxxxPD __m512d _mm512_mask_fnmadd_pd(__m512d a, __mmask8 k, __m512d b, __m512d c);
VFNMADDxxxPD __m512d _mm512_maskz_fnmadd_pd(__mmask8 k, __m512d a, __m512d b, __m512d c);
VFNMADDxxxPD __m512d _mm512_mask3_fnmadd_pd(__m512d a, __m512d b, __m512d c, __mmask8 k);
VFNMADDxxxPD __m512d _mm512_mask_fnmadd_round_pd(__m512d a, __mmask8 k, __m512d b, __m512d c, int r);
VFNMADDxxxPD __m512d _mm512_maskz_fnmadd_round_pd(__mmask8 k, __m512d a, __m512d b, __m512d c, int r);
VFNMADDxxxPD __m512d _mm512_mask3_fnmadd_round_pd(__m512d a, __m512d b, __m512d c, __mmask8 k, int r);
VFNMADDxxxPD __m256d _mm256_mask_fnmadd_pd(__m256d a, __mmask8 k, __m256d b, __m256d c);
VFNMADDxxxPD __m256d _mm256_maskz_fnmadd_pd(__mmask8 k, __m256d a, __m256d b, __m256d c);
VFNMADDxxxPD __m256d _mm256_mask3_fnmadd_pd(__m256d a, __m256d b, __m256d c, __mmask8 k);
VFNMADDxxxPD __m128d _mm_mask_fnmadd_pd(__m128d a, __mmask8 k, __m128d b, __m128d c);
VFNMADDxxxPD __m128d _mm_maskz_fnmadd_pd(__mmask8 k, __m128d a, __m128d b, __m128d c);
VFNMADDxxxPD __m128d _mm_mask3_fnmadd_pd(__m128d a, __m128d b, __m128d c, __mmask8 k);
VFNMADDxxxPD __m128d _mm_fnmadd_pd (__m128d a, __m128d b, __m128d c);
VFNMADDxxxPD __m256d _mm256_fnmadd_pd (__m256d a, __m256d b, __m256d c);

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal

Other Exceptions

VEX-encoded instructions, see Exceptions Type 2.
EVEX-encoded instructions, see Exceptions Type E2.

VFNMADD132PS/VFNMADD213PS/VFNMADD231PS—Fused Negative Multiply-Add of Packed Single-Precision Floating-Point Values

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
VEX.NDS.128.66.0F38.W0 9C /r VFNMADD132PS xmm1, xmm2, xmm3/m128	A	V/V	FMA	Multiply packed single-precision floating-point values from xmm1 and xmm3/mem, negate the multiplication result and add to xmm2 and put result in xmm1.
VEX.NDS.128.66.0F38.W0 AC /r VFNMADD213PS xmm1, xmm2, xmm3/m128	A	V/V	FMA	Multiply packed single-precision floating-point values from xmm1 and xmm2, negate the multiplication result and add to xmm3/mem and put result in xmm1.
VEX.NDS.128.66.0F38.W0 BC /r VFNMADD231PS xmm1, xmm2, xmm3/m128	A	V/V	FMA	Multiply packed single-precision floating-point values from xmm2 and xmm3/mem, negate the multiplication result and add to xmm1 and put result in xmm1.
VEX.NDS.256.66.0F38.W0 9C /r VFNMADD132PS ymm1, ymm2, ymm3/m256	A	V/V	FMA	Multiply packed single-precision floating-point values from ymm1 and ymm3/mem, negate the multiplication result and add to ymm2 and put result in ymm1.
VEX.NDS.256.66.0F38.W0 AC /r VFNMADD213PS ymm1, ymm2, ymm3/m256	A	V/V	FMA	Multiply packed single-precision floating-point values from ymm1 and ymm2, negate the multiplication result and add to ymm3/mem and put result in ymm1.
VEX.NDS.256.66.0F38.0 BC /r VFNMADD231PS ymm1, ymm2, ymm3/m256	A	V/V	FMA	Multiply packed single-precision floating-point values from ymm2 and ymm3/mem, negate the multiplication result and add to ymm1 and put result in ymm1.
EVEX.NDS.128.66.0F38.W0 9C /r VFNMADD132PS xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst	B	V/V	AVX512VL AVX512F	Multiply packed single-precision floating-point values from xmm1 and xmm3/m128/m32bcst, negate the multiplication result and add to xmm2 and put result in xmm1.
EVEX.NDS.128.66.0F38.W0 AC /r VFNMADD213PS xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst	B	V/V	AVX512VL AVX512F	Multiply packed single-precision floating-point values from xmm1 and xmm2, negate the multiplication result and add to xmm3/m128/m32bcst and put result in xmm1.
EVEX.NDS.128.66.0F38.W0 BC /r VFNMADD231PS xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst	B	V/V	AVX512VL AVX512F	Multiply packed single-precision floating-point values from xmm2 and xmm3/m128/m32bcst, negate the multiplication result and add to xmm1 and put result in xmm1.
EVEX.NDS.256.66.0F38.W0 9C /r VFNMADD132PS ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst	B	V/V	AVX512VL AVX512F	Multiply packed single-precision floating-point values from ymm1 and ymm3/m256/m32bcst, negate the multiplication result and add to ymm2 and put result in ymm1.
EVEX.NDS.256.66.0F38.W0 AC /r VFNMADD213PS ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst	B	V/V	AVX512VL AVX512F	Multiply packed single-precision floating-point values from ymm1 and ymm2, negate the multiplication result and add to ymm3/m256/m32bcst and put result in ymm1.
EVEX.NDS.256.66.0F38.W0 BC /r VFNMADD231PS ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst	B	V/V	AVX512VL AVX512F	Multiply packed single-precision floating-point values from ymm2 and ymm3/m256/m32bcst, negate the multiplication result and add to ymm1 and put result in ymm1.
EVEX.NDS.512.66.0F38.W0 9C /r VFNMADD132PS zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst{er}	B	V/V	AVX512VL AVX512F	Multiply packed single-precision floating-point values from zmm1 and zmm3/m512/m32bcst, negate the multiplication result and add to zmm2 and put result in zmm1.
EVEX.NDS.512.66.0F38.W0 AC /r VFNMADD213PS zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst{er}	B	V/V	AVX512F	Multiply packed single-precision floating-point values from zmm1 and zmm2, negate the multiplication result and add to zmm3/m512/m32bcst and put result in zmm1.
EVEX.NDS.512.66.0F38.W0 BC /r VFNMADD231PS zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst{er}	B	V/V	AVX512F	Multiply packed single-precision floating-point values from zmm2 and zmm3/m512/m32bcst, negate the multiplication result and add to zmm1 and put result in zmm1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	NA	ModRM:reg (r, w)	VEX.vvvv (r)	ModRM:r/m (r)	NA
B	Full Vector	ModRM:reg (r, w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA

Description

VFNMADD132PS: Multiplies the four, eight or sixteen packed single-precision floating-point values from the first source operand to the four, eight or sixteen packed single-precision floating-point values in the third source operand, adds the negated infinite precision intermediate result to the four, eight or sixteen packed single-precision floating-point values in the second source operand, performs rounding and stores the resulting four, eight or sixteen packed single-precision floating-point values to the destination operand (first source operand).

VFNMADD213PS: Multiplies the four, eight or sixteen packed single-precision floating-point values from the second source operand to the four, eight or sixteen packed single-precision floating-point values in the first source operand, adds the negated infinite precision intermediate result to the four, eight or sixteen packed single-precision floating-point values in the third source operand, performs rounding and stores the resulting the four, eight or sixteen packed single-precision floating-point values to the destination operand (first source operand).

VFNMADD231PS: Multiplies the four, eight or sixteen packed single-precision floating-point values from the second source operand to the four, eight or sixteen packed single-precision floating-point values in the third source operand, adds the negated infinite precision intermediate result to the four, eight or sixteen packed single-precision floating-point values in the first source operand, performs rounding and stores the resulting four, eight or sixteen packed single-precision floating-point values to the destination operand (first source operand).

EVEX encoded versions: The destination operand (also first source operand) and the second source operand are ZMM/YMM/XMM register. The third source operand is a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 32-bit memory location. The destination operand is conditionally updated with write mask k1.

VEX.256 encoded version: The destination operand (also first source operand) is a YMM register and encoded in reg_field. The second source operand is a YMM register and encoded in VEX.vvvv. The third source operand is a YMM register or a 256-bit memory location and encoded in rm_field.

VEX.128 encoded version: The destination operand (also first source operand) is a XMM register and encoded in reg_field. The second source operand is a XMM register and encoded in VEX.vvvv. The third source operand is a XMM register or a 128-bit memory location and encoded in rm_field. The upper 128 bits of the YMM destination register are zeroed.

Operation

In the operations below, "*" and "+" symbols represent multiplication and addition with infinite precision inputs and outputs (no rounding).

VFNMADD132PS DEST, SRC2, SRC3 (VEX encoded version)

```
IF (VEX.128) THEN
    MAXNUM ←2
ELSEIF (VEX.256)
    MAXNUM ← 4
FI
For i = 0 to MAXNUM-1 {
    n ← 32*i;
    DEST[n+31:n] ← RoundFPControl_MXCSR(- (DEST[n+31:n]*SRC3[n+31:n]) + SRC2[n+31:n])
}
IF (VEX.128) THEN
    DEST[MAXVL-1:128] ← 0
ELSEIF (VEX.256)
    DEST[MAXVL-1:256] ← 0
FI
```

VFNMADD213PS DEST, SRC2, SRC3 (VEX encoded version)

```
IF (VEX.128) THEN
    MAXNUM ←2
ELSEIF (VEX.256)
    MAXNUM ← 4
FI
For i = 0 to MAXNUM-1 {
    n ← 32*i;
    DEST[n+31:n] ← RoundFPControl_MXCSR(- (SRC2[n+31:n]*DEST[n+31:n]) + SRC3[n+31:n])
}
IF (VEX.128) THEN
    DEST[MAXVL-1:128] ← 0
ELSEIF (VEX.256)
    DEST[MAXVL-1:256] ← 0
FI
```

VFNMADD231PS DEST, SRC2, SRC3 (VEX encoded version)

```
IF (VEX.128) THEN
    MAXNUM ←2
ELSEIF (VEX.256)
    MAXNUM ← 4
FI
For i = 0 to MAXNUM-1 {
    n ← 32*i;
    DEST[n+31:n] ← RoundFPControl_MXCSR(- (SRC2[n+31:n]*SRC3[n+31:n]) + DEST[n+31:n])
}
IF (VEX.128) THEN
    DEST[MAXVL-1:128] ← 0
ELSEIF (VEX.256)
    DEST[MAXVL-1:256] ← 0
FI
```

VFNMADD132PS DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a register)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
IF (VL = 512) AND (EVEX.b = 1)
    THEN
        SET_RM(EVEX.RC);
    ELSE
        SET_RM(MXCSR.RM);
FI;
FOR j ← 0 TO KL-1
    i ← j * 32
    IF k1[j] OR *no writemask*
        THEN DEST[i+31:i] ←
            RoundFPControl(-(DEST[i+31:i]*SRC3[i+31:i]) + SRC2[i+31:i])
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE                            ; zeroing-masking
                    DEST[i+31:i] ← 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0
```

VFNMADD132PS DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a memory source)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)

FOR j ← 0 TO KL-1
    i ← j * 32
    IF k1[j] OR *no writemask*
        THEN
            IF (EVEX.b = 1)
                THEN
                    DEST[i+31:i] ←
            RoundFPControl_MXCSR(-(DEST[i+31:i]*SRC3[31:0]) + SRC2[i+31:i])
                ELSE
                    DEST[i+31:i] ←
            RoundFPControl_MXCSR(-(DEST[i+31:i]*SRC3[i+31:i]) + SRC2[i+31:i])
            FI;

        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE                            ; zeroing-masking
                    DEST[i+31:i] ← 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0
```

VFNMADD213PS DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a register)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
IF (VL = 512) AND (EVEX.b = 1)
    THEN
        SET_RM(EVEX.RC);
    ELSE
        SET_RM(MXCSR.RM);
FI;
FOR j ← 0 TO KL-1
    i ← j * 32
    IF k1[j] OR *no writemask*
        THEN DEST[i+31:i] ←
            RoundFPControl(-(SRC2[i+31:i]*DEST[i+31:i]) + SRC3[i+31:i])
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE                            ; zeroing-masking
                    DEST[i+31:i] ← 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0
```

VFNMADD213PS DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a memory source)
(KL, VL) = (4, 128), (8, 256), (16, 512)

```
FOR j ← 0 TO KL-1
    i ← j * 32
    IF k1[j] OR *no writemask*
        THEN
            IF (EVEX.b = 1)
                THEN
                    DEST[i+31:i] ←
            RoundFPControl_MXCSR(-(SRC2[i+31:i]*DEST[i+31:i]) + SRC3[31:0])

                ELSE
                    DEST[i+31:i] ←
            RoundFPControl_MXCSR(-(SRC2[i+31:i]*DEST[i+31:i]) + SRC3[i+31:i])
            FI;
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE                            ; zeroing-masking
                    DEST[i+31:i] ← 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0
```

VFNMADD231PS DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a register)
(KL, VL) = (4, 128), (8, 256), (16, 512)

```
IF (VL = 512) AND (EVEX.b = 1)
    THEN
        SET_RM(EVEX.RC);
    ELSE
        SET_RM(MXCSR.RM);
FI;
FOR j ← 0 TO KL-1
    i ← j * 32
    IF k1[j] OR *no writemask*
        THEN DEST[i+31:i] ←
            RoundFPControl(-(SRC2[i+31:i]*SRC3[i+31:i]) + DEST[i+31:i])
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE                            ; zeroing-masking
                    DEST[i+31:i] ← 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0
```

VFNMADD231PS DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a memory source)
(KL, VL) = (4, 128), (8, 256), (16, 512)

```
FOR j ← 0 TO KL-1
    i ← j * 32
    IF k1[j] OR *no writemask*
        THEN
            IF (EVEX.b = 1)
                THEN
                    DEST[i+31:i] ←
            RoundFPControl_MXCSR(-(SRC2[i+31:i]*SRC3[31:0]) + DEST[i+31:i])
                ELSE
                    DEST[i+31:i] ←
            RoundFPControl_MXCSR(-(SRC2[i+31:i]*SRC3[i+31:i]) + DEST[i+31:i])
            FI;
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE                                ; zeroing-masking
                    DEST[i+31:i] ← 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VFNMADDxxxPS __m512 _mm512_fnmadd_ps(__m512 a, __m512 b, __m512 c);
VFNMADDxxxPS __m512 _mm512_fnmadd_round_ps(__m512 a, __m512 b, __m512 c, int r);
VFNMADDxxxPS __m512 _mm512_mask_fnmadd_ps(__m512 a, __mmask16 k, __m512 b, __m512 c);
VFNMADDxxxPS __m512 _mm512_maskz_fnmadd_ps(__mmask16 k, __m512 a, __m512 b, __m512 c);
VFNMADDxxxPS __m512 _mm512_mask3_fnmadd_ps(__m512 a, __m512 b, __m512 c, __mmask16 k);
VFNMADDxxxPS __m512 _mm512_mask_fnmadd_round_ps(__m512 a, __mmask16 k, __m512 b, __m512 c, int r);
VFNMADDxxxPS __m512 _mm512_maskz_fnmadd_round_ps(__mmask16 k, __m512 a, __m512 b, __m512 c, int r);
VFNMADDxxxPS __m512 _mm512_mask3_fnmadd_round_ps(__m512 a, __m512 b, __m512 c, __mmask16 k, int r);
VFNMADDxxxPS __m256 _mm256_mask_fnmadd_ps(__m256 a, __mmask8 k, __m256 b, __m256 c);
VFNMADDxxxPS __m256 _mm256_maskz_fnmadd_ps(__mmask8 k, __m256 a, __m256 b, __m256 c);
VFNMADDxxxPS __m256 _mm256_mask3_fnmadd_ps(__m256 a, __m256 b, __m256 c, __mmask8 k);
VFNMADDxxxPS __m128 _mm_mask_fnmadd_ps(__m128 a, __mmask8 k, __m128 b, __m128 c);
VFNMADDxxxPS __m128 _mm_maskz_fnmadd_ps(__mmask8 k, __m128 a, __m128 b, __m128 c);
VFNMADDxxxPS __m128 _mm_mask3_fnmadd_ps(__m128 a, __m128 b, __m128 c, __mmask8 k);
VFNMADDxxxPS __m128 _mm_fnmadd_ps (__m128 a, __m128 b, __m128 c);
VFNMADDxxxPS __m256 _mm256_fnmadd_ps (__m256 a, __m256 b, __m256 c);

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal

Other Exceptions

VEX-encoded instructions, see Exceptions Type 2.
EVEX-encoded instructions, see Exceptions Type E2.

VFNMADD132SD/VFNMADD213SD/VFNMADD231SD—Fused Negative Multiply-Add of Scalar Double-Precision Floating-Point Values

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
VEX.DDS.LIG.66.0F38.W1 9D /r VFNMADD132SD xmm1, xmm2, xmm3/m64	A	V/V	FMA	Multiply scalar double-precision floating-point value from xmm1 and xmm3/mem, negate the multiplication result and add to xmm2 and put result in xmm1.
VEX.DDS.LIG.66.0F38.W1 AD /r VFNMADD213SD xmm1, xmm2, xmm3/m64	A	V/V	FMA	Multiply scalar double-precision floating-point value from xmm1 and xmm2, negate the multiplication result and add to xmm3/mem and put result in xmm1.
VEX.DDS.LIG.66.0F38.W1 BD /r VFNMADD231SD xmm1, xmm2, xmm3/m64	A	V/V	FMA	Multiply scalar double-precision floating-point value from xmm2 and xmm3/mem, negate the multiplication result and add to xmm1 and put result in xmm1.
EVEX.DDS.LIG.66.0F38.W1 9D /r VFNMADD132SD xmm1 {k1}{z}, xmm2, xmm3/m64{er}	B	V/V	AVX512F	Multiply scalar double-precision floating-point value from xmm1 and xmm3/m64, negate the multiplication result and add to xmm2 and put result in xmm1.
EVEX.DDS.LIG.66.0F38.W1 AD /r VFNMADD213SD xmm1 {k1}{z}, xmm2, xmm3/m64{er}	B	V/V	AVX512F	Multiply scalar double-precision floating-point value from xmm1 and xmm2, negate the multiplication result and add to xmm3/m64 and put result in xmm1.
EVEX.DDS.LIG.66.0F38.W1 BD /r VFNMADD231SD xmm1 {k1}{z}, xmm2, xmm3/m64{er}	B	V/V	AVX512F	Multiply scalar double-precision floating-point value from xmm2 and xmm3/m64, negate the multiplication result and add to xmm1 and put result in xmm1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	NA	ModRM:reg (r, w)	VEX.vvvv (r)	ModRM:r/m (r)	NA
B	Tuple1 Scalar	ModRM:reg (r, w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA

Description

VFNMADD132SD: Multiplies the low packed double-precision floating-point value from the first source operand to the low packed double-precision floating-point value in the third source operand, adds the negated infinite precision intermediate result to the low packed double-precision floating-point values in the second source operand, performs rounding and stores the resulting packed double-precision floating-point value to the destination operand (first source operand).

VFNMADD213SD: Multiplies the low packed double-precision floating-point value from the second source operand to the low packed double-precision floating-point value in the first source operand, adds the negated infinite precision intermediate result to the low packed double-precision floating-point value in the third source operand, performs rounding and stores the resulting packed double-precision floating-point value to the destination operand (first source operand).

VFNMADD231SD: Multiplies the low packed double-precision floating-point value from the second source to the low packed double-precision floating-point value in the third source operand, adds the negated infinite precision intermediate result to the low packed double-precision floating-point value in the first source operand, performs rounding and stores the resulting packed double-precision floating-point value to the destination operand (first source operand).

VEX.128 and EVEX encoded version: The destination operand (also first source operand) is encoded in reg_field. The second source operand is encoded in VEX.vvvv/EVEX.vvvv. The third source operand is encoded in rm_field. Bits 127:64 of the destination are unchanged. Bits MAXVL-1:128 of the destination register are zeroed.

EVEX encoded version: The low quadword element of the destination is updated according to the writemask.

Compiler tools may optionally support a complementary mnemonic for each instruction mnemonic listed in the opcode/instruction column of the summary table. The behavior of the complementary mnemonic in situations involving NANs are governed by the definition of the instruction mnemonic defined in the opcode/instruction column.

Operation

In the operations below, "*" and "+" symbols represent multiplication and addition with infinite precision inputs and outputs (no rounding).

VFNMADD132SD DEST, SRC2, SRC3 (EVEX encoded version)

```
IF (EVEX.b = 1) and SRC3 *is a register*
    THEN
        SET_RM(EVEX.RC);
    ELSE
        SET_RM(MXCSR.RM);
FI;
IF k1[0] or *no writemask*
    THEN    DEST[63:0] ← RoundFPControl(-(DEST[63:0]*SRC3[63:0]) + SRC2[63:0])
    ELSE
        IF *merging-masking*                ; merging-masking
            THEN *DEST[63:0] remains unchanged*
            ELSE                            ; zeroing-masking
                THEN DEST[63:0] ← 0
        FI;
FI;
DEST[127:64] ← DEST[127:64]
DEST[MAXVL-1:128] ← 0
```

VFNMADD213SD DEST, SRC2, SRC3 (EVEX encoded version)

```
IF (EVEX.b = 1) and SRC3 *is a register*
    THEN
        SET_RM(EVEX.RC);
    ELSE
        SET_RM(MXCSR.RM);
FI;
IF k1[0] or *no writemask*
    THEN    DEST[63:0] ← RoundFPControl(-(SRC2[63:0]*DEST[63:0]) + SRC3[63:0])
    ELSE
        IF *merging-masking*                ; merging-masking
            THEN *DEST[63:0] remains unchanged*
            ELSE                            ; zeroing-masking
                THEN DEST[63:0] ← 0
        FI;
FI;
DEST[127:64] ← DEST[127:64]
DEST[MAXVL-1:128] ← 0
```

VFNMADD231SD DEST, SRC2, SRC3 (EVEX encoded version)

```
IF (EVEX.b = 1) and SRC3 *is a register*
    THEN
        SET_RM(EVEX.RC);
    ELSE
        SET_RM(MXCSR.RM);
FI;
IF k1[0] or *no writemask*
    THEN    DEST[63:0] ← RoundFPControl(-(SRC2[63:0]*SRC3[63:0]) + DEST[63:0])
    ELSE
        IF *merging-masking*                    ; merging-masking
            THEN *DEST[63:0] remains unchanged*
            ELSE                                ; zeroing-masking
                THEN DEST[63:0] ← 0
        FI;
FI;
DEST[127:64] ← DEST[127:64]
DEST[MAXVL-1:128] ← 0
```

VFNMADD132SD DEST, SRC2, SRC3 (VEX encoded version)

```
DEST[63:0] ←RoundFPControl_MXCSR(- (DEST[63:0]*SRC3[63:0]) + SRC2[63:0])
DEST[127:64] ←DEST[127:64]
DEST[MAXVL-1:128] ←0
```

VFNMADD213SD DEST, SRC2, SRC3 (VEX encoded version)

```
DEST[63:0] ←RoundFPControl_MXCSR(- (SRC2[63:0]*DEST[63:0]) + SRC3[63:0])
DEST[127:64] ←DEST[127:64]
DEST[MAXVL-1:128] ←0
```

VFNMADD231SD DEST, SRC2, SRC3 (VEX encoded version)

```
DEST[63:0] ←RoundFPControl_MXCSR(- (SRC2[63:0]*SRC3[63:0]) + DEST[63:0])
DEST[127:64] ←DEST[127:64]
DEST[MAXVL-1:128] ←0
```

Intel C/C++ Compiler Intrinsic Equivalent

VFNMADDxxxSD __m128d _mm_fnmadd_round_sd(__m128d a, __m128d b, __m128d c, int r);
VFNMADDxxxSD __m128d _mm_mask_fnmadd_sd(__m128d a, __mmask8 k, __m128d b, __m128d c);
VFNMADDxxxSD __m128d _mm_maskz_fnmadd_sd(__mmask8 k, __m128d a, __m128d b, __m128d c);
VFNMADDxxxSD __m128d _mm_mask3_fnmadd_sd(__m128d a, __m128d b, __m128d c, __mmask8 k);
VFNMADDxxxSD __m128d _mm_mask_fnmadd_round_sd(__m128d a, __mmask8 k, __m128d b, __m128d c, int r);
VFNMADDxxxSD __m128d _mm_maskz_fnmadd_round_sd(__mmask8 k, __m128d a, __m128d b, __m128d c, int r);
VFNMADDxxxSD __m128d _mm_mask3_fnmadd_round_sd(__m128d a, __m128d b, __m128d c, __mmask8 k, int r);
VFNMADDxxxSD __m128d _mm_fnmadd_sd (__m128d a, __m128d b, __m128d c);

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal

Other Exceptions

VEX-encoded instructions, see Exceptions Type 3.
EVEX-encoded instructions, see Exceptions Type E3.

VFNMADD132SS/VFNMADD213SS/VFNMADD231SS—Fused Negative Multiply-Add of Scalar Single-Precision Floating-Point Values

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
VEX.DDS.LIG.66.0F38.W0 9D /r VFNMADD132SS xmm1, xmm2, xmm3/m32	A	V/V	FMA	Multiply scalar single-precision floating-point value from xmm1 and xmm3/m32, negate the multiplication result and add to xmm2 and put result in xmm1.
VEX.DDS.LIG.66.0F38.W0 AD /r VFNMADD213SS xmm1, xmm2, xmm3/m32	A	V/V	FMA	Multiply scalar single-precision floating-point value from xmm1 and xmm2, negate the multiplication result and add to xmm3/m32 and put result in xmm1.
VEX.DDS.LIG.66.0F38.W0 BD /r VFNMADD231SS xmm1, xmm2, xmm3/m32	A	V/V	FMA	Multiply scalar single-precision floating-point value from xmm2 and xmm3/m32, negate the multiplication result and add to xmm1 and put result in xmm1.
EVEX.DDS.LIG.66.0F38.W0 9D /r VFNMADD132SS xmm1 {k1}{z}, xmm2, xmm3/m32{er}	B	V/V	AVX512F	Multiply scalar single-precision floating-point value from xmm1 and xmm3/m32, negate the multiplication result and add to xmm2 and put result in xmm1.
EVEX.DDS.LIG.66.0F38.W0 AD /r VFNMADD213SS xmm1 {k1}{z}, xmm2, xmm3/m32{er}	B	V/V	AVX512F	Multiply scalar single-precision floating-point value from xmm1 and xmm2, negate the multiplication result and add to xmm3/m32 and put result in xmm1.
EVEX.DDS.LIG.66.0F38.W0 BD /r VFNMADD231SS xmm1 {k1}{z}, xmm2, xmm3/m32{er}	B	V/V	AVX512F	Multiply scalar single-precision floating-point value from xmm2 and xmm3/m32, negate the multiplication result and add to xmm1 and put result in xmm1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	NA	ModRM:reg (r, w)	VEX.vvvv (r)	ModRM:r/m (r)	NA
B	Tuple1 Scalar	ModRM:reg (r, w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA

Description

VFNMADD132SS: Multiplies the low packed single-precision floating-point value from the first source operand to the low packed single-precision floating-point value in the third source operand, adds the negated infinite precision intermediate result to the low packed single-precision floating-point value in the second source operand, performs rounding and stores the resulting packed single-precision floating-point value to the destination operand (first source operand).

VFNMADD213SS: Multiplies the low packed single-precision floating-point value from the second source operand to the low packed single-precision floating-point value in the first source operand, adds the negated infinite precision intermediate result to the low packed single-precision floating-point value in the third source operand, performs rounding and stores the resulting packed single-precision floating-point value to the destination operand (first source operand).

VFNMADD231SS: Multiplies the low packed single-precision floating-point value from the second source operand to the low packed single-precision floating-point value in the third source operand, adds the negated infinite precision intermediate result to the low packed single-precision floating-point value in the first source operand, performs rounding and stores the resulting packed single-precision floating-point value to the destination operand (first source operand).

VEX.128 and EVEX encoded version: The destination operand (also first source operand) is encoded in reg_field. The second source operand is encoded in VEX.vvvv/EVEX.vvvv. The third source operand is encoded in rm_field. Bits 127:32 of the destination are unchanged. Bits MAXVL-1:128 of the destination register are zeroed.

EVEX encoded version: The low doubleword element of the destination is updated according to the writemask.

Compiler tools may optionally support a complementary mnemonic for each instruction mnemonic listed in the opcode/instruction column of the summary table. The behavior of the complementary mnemonic in situations involving NANs are governed by the definition of the instruction mnemonic defined in the opcode/instruction column.

Operation

In the operations below, "*" and "+" symbols represent multiplication and addition with infinite precision inputs and outputs (no rounding).

VFNMADD132SS DEST, SRC2, SRC3 (EVEX encoded version)

```
IF (EVEX.b = 1) and SRC3 *is a register*
    THEN
        SET_RM(EVEX.RC);
    ELSE
        SET_RM(MXCSR.RM);
FI;
IF k1[0] or *no writemask*
    THEN    DEST[31:0] ← RoundFPControl(-(DEST[31:0]*SRC3[31:0]) + SRC2[31:0])
    ELSE
        IF *merging-masking*                ; merging-masking
            THEN *DEST[31:0] remains unchanged*
            ELSE                            ; zeroing-masking
                THEN DEST[31:0] ← 0
        FI;
FI;
DEST[127:32] ← DEST[127:32]
DEST[MAXVL-1:128] ← 0
```

VFNMADD213SS DEST, SRC2, SRC3 (EVEX encoded version)

```
IF (EVEX.b = 1) and SRC3 *is a register*
    THEN
        SET_RM(EVEX.RC);
    ELSE
        SET_RM(MXCSR.RM);
FI;
IF k1[0] or *no writemask*
    THEN    DEST[31:0] ← RoundFPControl(-(SRC2[31:0]*DEST[31:0]) + SRC3[31:0])
    ELSE
        IF *merging-masking*                ; merging-masking
            THEN *DEST[31:0] remains unchanged*
            ELSE                            ; zeroing-masking
                THEN DEST[31:0] ← 0
        FI;
FI;
DEST[127:32] ← DEST[127:32]
DEST[MAXVL-1:128] ← 0
```

VFNMADD231SS DEST, SRC2, SRC3 (EVEX encoded version)

```
IF (EVEX.b = 1) and SRC3 *is a register*
    THEN
        SET_RM(EVEX.RC);
    ELSE
        SET_RM(MXCSR.RM);
FI;
IF k1[0] or *no writemask*
    THEN    DEST[31:0] ← RoundFPControl(-(SRC2[31:0]*SRC3[63:0]) + DEST[31:0])
    ELSE
        IF *merging-masking*                ; merging-masking
            THEN *DEST[31:0] remains unchanged*
            ELSE                            ; zeroing-masking
                THEN DEST[31:0] ← 0
        FI;
FI;
DEST[127:32] ← DEST[127:32]
DEST[MAXVL-1:128] ← 0
```

VFNMADD132SS DEST, SRC2, SRC3 (VEX encoded version)

```
DEST[31:0] ←RoundFPControl_MXCSR(- (DEST[31:0]*SRC3[31:0]) + SRC2[31:0])
DEST[127:32] ←DEST[127:32]
DEST[MAXVL-1:128] ←0
```

VFNMADD213SS DEST, SRC2, SRC3 (VEX encoded version)

```
DEST[31:0] ←RoundFPControl_MXCSR(- (SRC2[31:0]*DEST[31:0]) + SRC3[31:0])
DEST[127:32] ←DEST[127:32]
DEST[MAXVL-1:128] ←0
```

VFNMADD231SS DEST, SRC2, SRC3 (VEX encoded version)

```
DEST[31:0] ←RoundFPControl_MXCSR(- (SRC2[31:0]*SRC3[31:0]) + DEST[31:0])
DEST[127:32] ←DEST[127:32]
DEST[MAXVL-1:128] ←0
```

Intel C/C++ Compiler Intrinsic Equivalent

VFNMADDxxxSS __m128 _mm_fnmadd_round_ss(__m128 a, __m128 b, __m128 c, int r);
VFNMADDxxxSS __m128 _mm_mask_fnmadd_ss(__m128 a, __mmask8 k, __m128 b, __m128 c);
VFNMADDxxxSS __m128 _mm_maskz_fnmadd_ss(__mmask8 k, __m128 a, __m128 b, __m128 c);
VFNMADDxxxSS __m128 _mm_mask3_fnmadd_ss(__m128 a, __m128 b, __m128 c, __mmask8 k);
VFNMADDxxxSS __m128 _mm_mask_fnmadd_round_ss(__m128 a, __mmask8 k, __m128 b, __m128 c, int r);
VFNMADDxxxSS __m128 _mm_maskz_fnmadd_round_ss(__mmask8 k, __m128 a, __m128 b, __m128 c, int r);
VFNMADDxxxSS __m128 _mm_mask3_fnmadd_round_ss(__m128 a, __m128 b, __m128 c, __mmask8 k, int r);
VFNMADDxxxSS __m128 _mm_fnmadd_ss (__m128 a, __m128 b, __m128 c);

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal

Other Exceptions

VEX-encoded instructions, see Exceptions Type 3.
EVEX-encoded instructions, see Exceptions Type E3.

VFNMSUB132PD/VFNMSUB213PD/VFNMSUB231PD—Fused Negative Multiply-Subtract of Packed Double-Precision Floating-Point Values

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
VEX.NDS.128.66.0F38.W1 9E /r VFNMSUB132PD xmm1, xmm2, xmm3/m128	A	V/V	FMA	Multiply packed double-precision floating-point values from xmm1 and xmm3/mem, negate the multiplication result and subtract xmm2 and put result in xmm1.
VEX.NDS.128.66.0F38.W1 AE /r VFNMSUB213PD xmm1, xmm2, xmm3/m128	A	V/V	FMA	Multiply packed double-precision floating-point values from xmm1 and xmm2, negate the multiplication result and subtract xmm3/mem and put result in xmm1.
VEX.NDS.128.66.0F38.W1 BE /r VFNMSUB231PD xmm1, xmm2, xmm3/m128	A	V/V	FMA	Multiply packed double-precision floating-point values from xmm2 and xmm3/mem, negate the multiplication result and subtract xmm1 and put result in xmm1.
VEX.NDS.256.66.0F38.W1 9E /r VFNMSUB132PD ymm1, ymm2, ymm3/m256	A	V/V	FMA	Multiply packed double-precision floating-point values from ymm1 and ymm3/mem, negate the multiplication result and subtract ymm2 and put result in ymm1.
VEX.NDS.256.66.0F38.W1 AE /r VFNMSUB213PD ymm1, ymm2, ymm3/m256	A	V/V	FMA	Multiply packed double-precision floating-point values from ymm1 and ymm2, negate the multiplication result and subtract ymm3/mem and put result in ymm1.
VEX.NDS.256.66.0F38.W1 BE /r VFNMSUB231PD ymm1, ymm2, ymm3/m256	A	V/V	FMA	Multiply packed double-precision floating-point values from ymm2 and ymm3/mem, negate the multiplication result and subtract ymm1 and put result in ymm1.
EVEX.NDS.128.66.0F38.W1 9E /r VFNMSUB132PD xmm1 {k1}{z}, xmm2, xmm3/m128/m64bcst	B	V/V	AVX512VL AVX512F	Multiply packed double-precision floating-point values from xmm1 and xmm3/m128/m64bcst, negate the multiplication result and subtract xmm2 and put result in xmm1.
EVEX.NDS.128.66.0F38.W1 AE /r VFNMSUB213PD xmm1 {k1}{z}, xmm2, xmm3/m128/m64bcst	B	V/V	AVX512VL AVX512F	Multiply packed double-precision floating-point values from xmm1 and xmm2, negate the multiplication result and subtract xmm3/m128/m64bcst and put result in xmm1.
EVEX.NDS.128.66.0F38.W1 BE /r VFNMSUB231PD xmm1 {k1}{z}, xmm2, xmm3/m128/m64bcst	B	V/V	AVX512VL AVX512F	Multiply packed double-precision floating-point values from xmm2 and xmm3/m128/m64bcst, negate the multiplication result and subtract xmm1 and put result in xmm1.
EVEX.NDS.256.66.0F38.W1 9E /r VFNMSUB132PD ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst	B	V/V	AVX512VL AVX512F	Multiply packed double-precision floating-point values from ymm1 and ymm3/m256/m64bcst, negate the multiplication result and subtract ymm2 and put result in ymm1.
EVEX.NDS.256.66.0F38.W1 AE /r VFNMSUB213PD ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst	B	V/V	AVX512VL AVX512F	Multiply packed double-precision floating-point values from ymm1 and ymm2, negate the multiplication result and subtract ymm3/m256/m64bcst and put result in ymm1.
EVEX.NDS.256.66.0F38.W1 BE /r VFNMSUB231PD ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst	B	V/V	AVX512VL AVX512F	Multiply packed double-precision floating-point values from ymm2 and ymm3/m256/m64bcst, negate the multiplication result and subtract ymm1 and put result in ymm1.
EVEX.NDS.512.66.0F38.W1 9E /r VFNMSUB132PD zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst{er}	B	V/V	AVX512F	Multiply packed double-precision floating-point values from zmm1 and zmm3/m512/m64bcst, negate the multiplication result and subtract zmm2 and put result in zmm1.
EVEX.NDS.512.66.0F38.W1 AE /r VFNMSUB213PD zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst{er}	B	V/V	AVX512F	Multiply packed double-precision floating-point values from zmm1 and zmm2, negate the multiplication result and subtract zmm3/m512/m64bcst and put result in zmm1.
EVEX.NDS.512.66.0F38.W1 BE /r VFNMSUB231PD zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst{er}	B	V/V	AVX512F	Multiply packed double-precision floating-point values from zmm2 and zmm3/m512/m64bcst, negate the multiplication result and subtract zmm1 and put result in zmm1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	NA	ModRM:reg (r, w)	VEX.vvvv (r)	ModRM:r/m (r)	NA
B	Full Vector	ModRM:reg (r, w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA

Description

VFNMSUB132PD: Multiplies the two, four or eight packed double-precision floating-point values from the first source operand to the two, four or eight packed double-precision floating-point values in the third source operand. From negated infinite precision intermediate results, subtracts the two, four or eight packed double-precision floating-point values in the second source operand, performs rounding and stores the resulting two, four or eight packed double-precision floating-point values to the destination operand (first source operand).

VFNMSUB213PD: Multiplies the two, four or eight packed double-precision floating-point values from the second source operand to the two, four or eight packed double-precision floating-point values in the first source operand. From negated infinite precision intermediate results, subtracts the two, four or eight packed double-precision floating-point values in the third source operand, performs rounding and stores the resulting two, four or eight packed double-precision floating-point values to the destination operand (first source operand).

VFNMSUB231PD: Multiplies the two, four or eight packed double-precision floating-point values from the second source to the two, four or eight packed double-precision floating-point values in the third source operand. From negated infinite precision intermediate results, subtracts the two, four or eight packed double-precision floating-point values in the first source operand, performs rounding and stores the resulting two, four or eight packed double-precision floating-point values to the destination operand (first source operand).

EVEX encoded versions: The destination operand (also first source operand) and the second source operand are ZMM/YMM/XMM register. The third source operand is a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 64-bit memory location. The destination operand is conditionally updated with write mask k1.

VEX.256 encoded version: The destination operand (also first source operand) is a YMM register and encoded in reg_field. The second source operand is a YMM register and encoded in VEX.vvvv. The third source operand is a YMM register or a 256-bit memory location and encoded in rm_field.

VEX.128 encoded version: The destination operand (also first source operand) is a XMM register and encoded in reg_field. The second source operand is a XMM register and encoded in VEX.vvvv. The third source operand is a XMM register or a 128-bit memory location and encoded in rm_field. The upper 128 bits of the YMM destination register are zeroed.

Operation

In the operations below, "*" and "-" symbols represent multiplication and subtraction with infinite precision inputs and outputs (no rounding).

VFNMSUB132PD DEST, SRC2, SRC3 (VEX encoded version)

```
IF (VEX.128) THEN
    MAXNUM ←2
ELSEIF (VEX.256)
    MAXNUM ← 4
FI
For i = 0 to MAXNUM-1 {
    n ← 64*i;
    DEST[n+63:n] ← RoundFPControl_MXCSR( - (DEST[n+63:n]*SRC3[n+63:n]) - SRC2[n+63:n])
}
IF (VEX.128) THEN
    DEST[MAXVL-1:128] ← 0
ELSEIF (VEX.256)
    DEST[MAXVL-1:256] ← 0
FI
```

VFNMSUB213PD DEST, SRC2, SRC3 (VEX encoded version)

```
IF (VEX.128) THEN
    MAXNUM ←2
ELSEIF (VEX.256)
    MAXNUM ← 4
FI
For i = 0 to MAXNUM-1 {
    n ← 64*i;
    DEST[n+63:n] ← RoundFPControl_MXCSR( - (SRC2[n+63:n]*DEST[n+63:n]) - SRC3[n+63:n])
}
IF (VEX.128) THEN
    DEST[MAXVL-1:128] ← 0
ELSEIF (VEX.256)
    DEST[MAXVL-1:256] ← 0
FI
```

VFNMSUB231PD DEST, SRC2, SRC3 (VEX encoded version)

```
IF (VEX.128) THEN
    MAXNUM ←2
ELSEIF (VEX.256)
    MAXNUM ← 4
FI
For i = 0 to MAXNUM-1 {
    n ← 64*i;
    DEST[n+63:n] ← RoundFPControl_MXCSR( - (SRC2[n+63:n]*SRC3[n+63:n]) - DEST[n+63:n])
}
IF (VEX.128) THEN
    DEST[MAXVL-1:128] ← 0
ELSEIF (VEX.256)
    DEST[MAXVL-1:256] ← 0
FI
```

VFNMSUB132PD DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a register)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
IF (VL = 512) AND (EVEX.b = 1)
    THEN
        SET_RM(EVEX.RC);
    ELSE
        SET_RM(MXCSR.RM);
FI;
FOR j ← 0 TO KL-1
    i ← j * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+63:i] ←
            RoundFPControl(-(DEST[i+63:i]*SRC3[i+63:i]) - SRC2[i+63:i])
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE                            ; zeroing-masking
                    DEST[i+63:i] ← 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0
```

VFNMSUB132PD DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a memory source)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)

FOR j ← 0 TO KL-1
    i ← j * 64
    IF k1[j] OR *no writemask*
        THEN
            IF (EVEX.b = 1)
                THEN
                    DEST[i+63:i] ←
            RoundFPControl_MXCSR(-(DEST[i+63:i]*SRC3[63:0]) - SRC2[i+63:i])
                ELSE
                    DEST[i+63:i] ←
            RoundFPControl_MXCSR(-(DEST[i+63:i]*SRC3[i+63:i]) - SRC2[i+63:i])
            FI;
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE                                ; zeroing-masking
                    DEST[i+63:i] ← 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0
```

VFNMSUB213PD DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a register)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
IF (VL = 512) AND (EVEX.b = 1)
    THEN
        SET_RM(EVEX.RC);
    ELSE
        SET_RM(MXCSR.RM);
FI;
FOR j ← 0 TO KL-1
    i ← j * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+63:i] ←
            RoundFPControl(-(SRC2[i+63:i]*DEST[i+63:i]) - SRC3[i+63:i])
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE                                ; zeroing-masking
                    DEST[i+63:i] ← 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0
```

VFNMSUB213PD DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a memory source)
(KL, VL) = (2, 128), (4, 256), (8, 512)

```
FOR j ← 0 TO KL-1
    i ← j * 64
    IF k1[j] OR *no writemask*
        THEN
            IF (EVEX.b = 1)
                THEN
                    DEST[i+63:i] ←
            RoundFPControl_MXCSR(-(SRC2[i+63:i]*DEST[i+63:i]) - SRC3[63:0])
                ELSE
                    DEST[i+63:i] ←
            RoundFPControl_MXCSR(-(SRC2[i+63:i]*DEST[i+63:i]) - SRC3[i+63:i])
            FI;
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE                                ; zeroing-masking
                    DEST[i+63:i] ← 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0
```

VFNMSUB231PD DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a register)
(KL, VL) = (2, 128), (4, 256), (8, 512)

```
IF (VL = 512) AND (EVEX.b = 1)
    THEN
        SET_RM(EVEX.RC);
    ELSE
        SET_RM(MXCSR.RM);
FI;
FOR j ← 0 TO KL-1
    i ← j * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+63:i] ←
            RoundFPControl(-(SRC2[i+63:i]*SRC3[i+63:i]) - DEST[i+63:i])
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE                                ; zeroing-masking
                    DEST[i+63:i] ← 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0
```

VFNMSUB231PD DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a memory source)
(KL, VL) = (2, 128), (4, 256), (8, 512)

```
FOR j ← 0 TO KL-1
    i ← j * 64
    IF k1[j] OR *no writemask*
        THEN
            IF (EVEX.b = 1)
                THEN
                    DEST[i+63:i] ←
            RoundFPControl_MXCSR(-(SRC2[i+63:i]*SRC3[63:0]) - DEST[i+63:i])
                ELSE
                    DEST[i+63:i] ←
            RoundFPControl_MXCSR(-(SRC2[i+63:i]*SRC3[i+63:i]) - DEST[i+63:i])
            FI;
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE                                ; zeroing-masking
                    DEST[i+63:i] ← 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VFNMSUBxxxPD __m512d _mm512_fnmsub_pd(__m512d a, __m512d b, __m512d c);
VFNMSUBxxxPD __m512d _mm512_fnmsub_round_pd(__m512d a, __m512d b, __m512d c, int r);
VFNMSUBxxxPD __m512d _mm512_mask_fnmsub_pd(__m512d a, __mmask8 k, __m512d b, __m512d c);
VFNMSUBxxxPD __m512d _mm512_maskz_fnmsub_pd(__mmask8 k, __m512d a, __m512d b, __m512d c);
VFNMSUBxxxPD __m512d _mm512_mask3_fnmsub_pd(__m512d a, __m512d b, __m512d c, __mmask8 k);
VFNMSUBxxxPD __m512d _mm512_mask_fnmsub_round_pd(__m512d a, __mmask8 k, __m512d b, __m512d c, int r);
VFNMSUBxxxPD __m512d _mm512_maskz_fnmsub_round_pd(__mmask8 k, __m512d a, __m512d b, __m512d c, int r);
VFNMSUBxxxPD __m512d _mm512_mask3_fnmsub_round_pd(__m512d a, __m512d b, __m512d c, __mmask8 k, int r);
VFNMSUBxxxPD __m256d _mm256_mask_fnmsub_pd(__m256d a, __mmask8 k, __m256d b, __m256d c);
VFNMSUBxxxPD __m256d _mm256_maskz_fnmsub_pd(__mmask8 k, __m256d a, __m256d b, __m256d c);
VFNMSUBxxxPD __m256d _mm256_mask3_fnmsub_pd(__m256d a, __m256d b, __m256d c, __mmask8 k);
VFNMSUBxxxPD __m128d _mm_mask_fnmsub_pd(__m128d a, __mmask8 k, __m128d b, __m128d c);
VFNMSUBxxxPD __m128d _mm_maskz_fnmsub_pd(__mmask8 k, __m128d a, __m128d b, __m128d c);
VFNMSUBxxxPD __m128d _mm_mask3_fnmsub_pd(__m128d a, __m128d b, __m128d c, __mmask8 k);
VFNMSUBxxxPD __m128d _mm_fnmsub_pd (__m128d a, __m128d b, __m128d c);
VFNMSUBxxxPD __m256d _mm256_fnmsub_pd (__m256d a, __m256d b, __m256d c);

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal

Other Exceptions

VEX-encoded instructions, see Exceptions Type 2.
EVEX-encoded instructions, see Exceptions Type E2.

VFNMSUB132PS/VFNMSUB213PS/VFNMSUB231PS—Fused Negative Multiply-Subtract of Packed Single-Precision Floating-Point Values

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
VEX.NDS.128.66.0F38.W0 9E /r VFNMSUB132PS xmm1, xmm2, xmm3/m128	A	V/V	FMA	Multiply packed single-precision floating-point values from xmm1 and xmm3/mem, negate the multiplication result and subtract xmm2 and put result in xmm1.
VEX.NDS.128.66.0F38.W0 AE /r VFNMSUB213PS xmm1, xmm2, xmm3/m128	A	V/V	FMA	Multiply packed single-precision floating-point values from xmm1 and xmm2, negate the multiplication result and subtract xmm3/mem and put result in xmm1.
VEX.NDS.128.66.0F38.W0 BE /r VFNMSUB231PS xmm1, xmm2, xmm3/m128	A	V/V	FMA	Multiply packed single-precision floating-point values from xmm2 and xmm3/mem, negate the multiplication result and subtract xmm1 and put result in xmm1.
VEX.NDS.256.66.0F38.W0 9E /r VFNMSUB132PS ymm1, ymm2, ymm3/m256	A	V/V	FMA	Multiply packed single-precision floating-point values from ymm1 and ymm3/mem, negate the multiplication result and subtract ymm2 and put result in ymm1.
VEX.NDS.256.66.0F38.W0 AE /r VFNMSUB213PS ymm1, ymm2, ymm3/m256	A	V/V	FMA	Multiply packed single-precision floating-point values from ymm1 and ymm2, negate the multiplication result and subtract ymm3/mem and put result in ymm1.
VEX.NDS.256.66.0F38.0 BE /r VFNMSUB231PS ymm1, ymm2, ymm3/m256	A	V/V	FMA	Multiply packed single-precision floating-point values from ymm2 and ymm3/mem, negate the multiplication result and subtract ymm1 and put result in ymm1.
EVEX.NDS.128.66.0F38.W0 9E /r VFNMSUB132PS xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst	B	V/V	AVX512VL AVX512F	Multiply packed single-precision floating-point values from xmm1 and xmm3/m128/m32bcst, negate the multiplication result and subtract xmm2 and put result in xmm1.
EVEX.NDS.128.66.0F38.W0 AE /r VFNMSUB213PS xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst	B	V/V	AVX512VL AVX512F	Multiply packed single-precision floating-point values from xmm1 and xmm2, negate the multiplication result and subtract xmm3/m128/m32bcst and put result in xmm1.
EVEX.NDS.128.66.0F38.W0 BE /r VFNMSUB231PS xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst	B	V/V	AVX512VL AVX512F	Multiply packed single-precision floating-point values from xmm2 and xmm3/m128/m32bcst, negate the multiplication result subtract add to xmm1 and put result in xmm1.
EVEX.NDS.256.66.0F38.W0 9E /r VFNMSUB132PS ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst	B	V/V	AVX512VL AVX512F	Multiply packed single-precision floating-point values from ymm1 and ymm3/m256/m32bcst, negate the multiplication result and subtract ymm2 and put result in ymm1.
EVEX.NDS.256.66.0F38.W0 AE /r VFNMSUB213PS ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst	B	V/V	AVX512VL AVX512F	Multiply packed single-precision floating-point values from ymm1 and ymm2, negate the multiplication result and subtract ymm3/m256/m32bcst and put result in ymm1.
EVEX.NDS.256.66.0F38.W0 BE /r VFNMSUB231PS ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst	B	V/V	AVX512VL AVX512F	Multiply packed single-precision floating-point values from ymm2 and ymm3/m256/m32bcst, negate the multiplication result subtract add to ymm1 and put result in ymm1.
EVEX.NDS.512.66.0F38.W0 9E /r VFNMSUB132PS zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst{er}	B	V/V	AVX512F	Multiply packed single-precision floating-point values from zmm1 and zmm3/m512/m32bcst, negate the multiplication result and subtract zmm2 and put result in zmm1.
EVEX.NDS.512.66.0F38.W0 AE /r VFNMSUB213PS zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst{er}	B	V/V	AVX512F	Multiply packed single-precision floating-point values from zmm1 and zmm2, negate the multiplication result and subtract zmm3/m512/m32bcst and put result in zmm1.
EVEX.NDS.512.66.0F38.W0 BE /r VFNMSUB231PS zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst{er}	B	V/V	AVX512F	Multiply packed single-precision floating-point values from zmm2 and zmm3/m512/m32bcst, negate the multiplication result subtract add to zmm1 and put result in zmm1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	NA	ModRM:reg (r, w)	VEX.vvvv (r)	ModRM:r/m (r)	NA
B	Full Vector	ModRM:reg (r, w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA

Description

VFNMSUB132PS: Multiplies the four, eight or sixteen packed single-precision floating-point values from the first source operand to the four, eight or sixteen packed single-precision floating-point values in the third source operand. From negated infinite precision intermediate results, subtracts the four, eight or sixteen packed single-precision floating-point values in the second source operand, performs rounding and stores the resulting four, eight or sixteen packed single-precision floating-point values to the destination operand (first source operand).

VFNMSUB213PS: Multiplies the four, eight or sixteen packed single-precision floating-point values from the second source operand to the four, eight or sixteen packed single-precision floating-point values in the first source operand. From negated infinite precision intermediate results, subtracts the four, eight or sixteen packed single-precision floating-point values in the third source operand, performs rounding and stores the resulting four, eight or sixteen packed single-precision floating-point values to the destination operand (first source operand).

VFNMSUB231PS: Multiplies the four, eight or sixteen packed single-precision floating-point values from the second source to the four, eight or sixteen packed single-precision floating-point values in the third source operand. From negated infinite precision intermediate results, subtracts the four, eight or sixteen packed single-precision floating-point values in the first source operand, performs rounding and stores the resulting four, eight or sixteen packed single-precision floating-point values to the destination operand (first source operand).

EVEX encoded versions: The destination operand (also first source operand) and the second source operand are ZMM/YMM/XMM register. The third source operand is a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 32-bit memory location. The destination operand is conditionally updated with write mask k1.

VEX.256 encoded version: The destination operand (also first source operand) is a YMM register and encoded in reg_field. The second source operand is a YMM register and encoded in VEX.vvvv. The third source operand is a YMM register or a 256-bit memory location and encoded in rm_field.

VEX.128 encoded version: The destination operand (also first source operand) is a XMM register and encoded in reg_field. The second source operand is a XMM register and encoded in VEX.vvvv. The third source operand is a XMM register or a 128-bit memory location and encoded in rm_field. The upper 128 bits of the YMM destination register are zeroed.

Operation

In the operations below, "*" and "-" symbols represent multiplication and subtraction with infinite precision inputs and outputs (no rounding).

VFNMSUB132PS DEST, SRC2, SRC3 (VEX encoded version)

```
IF (VEX.128) THEN
    MAXNUM ←2
ELSEIF (VEX.256)
    MAXNUM ← 4
FI
For i = 0 to MAXNUM-1 {
    n ← 32*i;
    DEST[n+31:n] ← RoundFPControl_MXCSR( - (DEST[n+31:n]*SRC3[n+31:n]) - SRC2[n+31:n])
}
IF (VEX.128) THEN
    DEST[MAXVL-1:128] ← 0
ELSEIF (VEX.256)
    DEST[MAXVL-1:256] ← 0
FI
```

VFNMSUB213PS DEST, SRC2, SRC3 (VEX encoded version)

```
IF (VEX.128) THEN
    MAXNUM ←2
ELSEIF (VEX.256)
    MAXNUM ← 4
FI
For i = 0 to MAXNUM-1 {
    n ← 32*i;
    DEST[n+31:n] ← RoundFPControl_MXCSR( - (SRC2[n+31:n]*DEST[n+31:n]) - SRC3[n+31:n])
}
IF (VEX.128) THEN
    DEST[MAXVL-1:128] ← 0
ELSEIF (VEX.256)
    DEST[MAXVL-1:256] ← 0
FI
```

VFNMSUB231PS DEST, SRC2, SRC3 (VEX encoded version)

```
IF (VEX.128) THEN
    MAXNUM ←2
ELSEIF (VEX.256)
    MAXNUM ← 4
FI
For i = 0 to MAXNUM-1 {
    n ← 32*i;
    DEST[n+31:n] ← RoundFPControl_MXCSR( - (SRC2[n+31:n]*SRC3[n+31:n]) - DEST[n+31:n])
}
IF (VEX.128) THEN
    DEST[MAXVL-1:128] ← 0
ELSEIF (VEX.256)
    DEST[MAXVL-1:256] ← 0
FI
```

VFNMSUB132PS DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a register)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
IF (VL = 512) AND (EVEX.b = 1)
    THEN
        SET_RM(EVEX.RC);
    ELSE
        SET_RM(MXCSR.RM);
FI;
FOR j ← 0 TO KL-1
    i ← j * 32
    IF k1[j] OR *no writemask*
        THEN DEST[i+31:i] ←
            RoundFPControl(-(DEST[i+31:i]*SRC3[i+31:i]) - SRC2[i+31:i])
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE                            ; zeroing-masking
                    DEST[i+31:i] ← 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0
```

VFNMSUB132PS DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a memory source)

(KL, VL) = (4, 128), (8, 256), (16, 512)

```
FOR j ← 0 TO KL-1
    i ← j * 32
    IF k1[j] OR *no writemask*
        THEN
            IF (EVEX.b = 1)
                THEN
                    DEST[i+31:i] ←
            RoundFPControl_MXCSR(-(DEST[i+31:i]*SRC3[31:0]) - SRC2[i+31:i])
                ELSE
                    DEST[i+31:i] ←
            RoundFPControl_MXCSR(-(DEST[i+31:i]*SRC3[i+31:i]) - SRC2[i+31:i])
            FI;
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE                                ; zeroing-masking
                    DEST[i+31:i] ← 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0
```

VFNMSUB213PS DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a register)

(KL, VL) = (4, 128), (8, 256), (16, 512)

```
IF (VL = 512) AND (EVEX.b = 1)
    THEN
        SET_RM(EVEX.RC);
    ELSE
        SET_RM(MXCSR.RM);
FI;
FOR j ← 0 TO KL-1
    i ← j * 32
    IF k1[j] OR *no writemask*
        THEN DEST[i+31:i] ←
            RoundFPControl_MXCSR(-(SRC2[i+31:i]*DEST[i+31:i]) - SRC3[i+31:i])
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE                                ; zeroing-masking
                    DEST[i+31:i] ← 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0
```

VFNMSUB213PS DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a memory source)
(KL, VL) = (4, 128), (8, 256), (16, 512)

```
FOR j ← 0 TO KL-1
    i ← j * 32
    IF k1[j] OR *no writemask*
        THEN
            IF (EVEX.b = 1)
                THEN
                    DEST[i+31:i] ←
            RoundFPControl_MXCSR(-(SRC2[i+31:i]*DEST[i+31:i]) - SRC3[31:0])
                ELSE
                    DEST[i+31:i] ←
            RoundFPControl_MXCSR(-(SRC2[i+31:i]*DEST[i+31:i]) - SRC3[i+31:i])
            FI;
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE                                ; zeroing-masking
                    DEST[i+31:i] ← 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0
```

VFNMSUB231PS DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a register)
(KL, VL) = (4, 128), (8, 256), (16, 512)

```
IF (VL = 512) AND (EVEX.b = 1)
    THEN
        SET_RM(EVEX.RC);
    ELSE
        SET_RM(MXCSR.RM);
FI;
FOR j ← 0 TO KL-1
    i ← j * 32
    IF k1[j] OR *no writemask*
        THEN DEST[i+31:i] ←
            RoundFPControl_MXCSR(-(SRC2[i+31:i]*SRC3[i+31:i]) - DEST[i+31:i])
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE                                ; zeroing-masking
                    DEST[i+31:i] ← 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0
```

VFNMSUB231PS DEST, SRC2, SRC3 (EVEX encoded version, when src3 operand is a memory source)
(KL, VL) = (4, 128), (8, 256), (16, 512)

```
FOR j ← 0 TO KL-1
    i ← j * 32
    IF k1[j] OR *no writemask*
        THEN
            IF (EVEX.b = 1)
                THEN
                    DEST[i+31:i] ←
            RoundFPControl_MXCSR(-(SRC2[i+31:i]*SRC3[31:0]) - DEST[i+31:i])
                ELSE
                    DEST[i+31:i] ←
            RoundFPControl_MXCSR(-(SRC2[i+31:i]*SRC3[i+31:i]) - DEST[i+31:i])
            FI;
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE                                ; zeroing-masking
                    DEST[i+31:i] ← 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VFNMSUBxxxPS __m512 _mm512_fnmsub_ps(__m512 a, __m512 b, __m512 c);
VFNMSUBxxxPS __m512 _mm512_fnmsub_round_ps(__m512 a, __m512 b, __m512 c, int r);
VFNMSUBxxxPS __m512 _mm512_mask_fnmsub_ps(__m512 a, __mmask16 k, __m512 b, __m512 c);
VFNMSUBxxxPS __m512 _mm512_maskz_fnmsub_ps(__mmask16 k, __m512 a, __m512 b, __m512 c);
VFNMSUBxxxPS __m512 _mm512_mask3_fnmsub_ps(__m512 a, __m512 b, __m512 c, __mmask16 k);
VFNMSUBxxxPS __m512 _mm512_mask_fnmsub_round_ps(__m512 a, __mmask16 k, __m512 b, __m512 c, int r);
VFNMSUBxxxPS __m512 _mm512_maskz_fnmsub_round_ps(__mmask16 k, __m512 a, __m512 b, __m512 c, int r);
VFNMSUBxxxPS __m512 _mm512_mask3_fnmsub_round_ps(__m512 a, __m512 b, __m512 c, __mmask16 k, int r);
VFNMSUBxxxPS __m256 _mm256_mask_fnmsub_ps(__m256 a, __mmask8 k, __m256 b, __m256 c);
VFNMSUBxxxPS __m256 _mm256_maskz_fnmsub_ps(__mmask8 k, __m256 a, __m256 b, __m256 c);
VFNMSUBxxxPS __m256 _mm256_mask3_fnmsub_ps(__m256 a, __m256 b, __m256 c, __mmask8 k);
VFNMSUBxxxPS __m128 _mm_mask_fnmsub_ps(__m128 a, __mmask8 k, __m128 b, __m128 c);
VFNMSUBxxxPS __m128 _mm_maskz_fnmsub_ps(__mmask8 k, __m128 a, __m128 b, __m128 c);
VFNMSUBxxxPS __m128 _mm_mask3_fnmsub_ps(__m128 a, __m128 b, __m128 c, __mmask8 k);
VFNMSUBxxxPS __m128 _mm_fnmsub_ps (__m128 a, __m128 b, __m128 c);
VFNMSUBxxxPS __m256 _mm256_fnmsub_ps (__m256 a, __m256 b, __m256 c);

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal

Other Exceptions

VEX-encoded instructions, see Exceptions Type 2.
EVEX-encoded instructions, see Exceptions Type E2.

VFNMSUB132SD/VFNMSUB213SD/VFNMSUB231SD—Fused Negative Multiply-Subtract of Scalar Double-Precision Floating-Point Values

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
VEX.DDS.LIG.66.0F38.W1 9F /r VFNMSUB132SD xmm1, xmm2, xmm3/m64	A	V/V	FMA	Multiply scalar double-precision floating-point value from xmm1 and xmm3/mem, negate the multiplication result and subtract xmm2 and put result in xmm1.
VEX.DDS.LIG.66.0F38.W1 AF /r VFNMSUB213SD xmm1, xmm2, xmm3/m64	A	V/V	FMA	Multiply scalar double-precision floating-point value from xmm1 and xmm2, negate the multiplication result and subtract xmm3/mem and put result in xmm1.
VEX.DDS.LIG.66.0F38.W1 BF /r VFNMSUB231SD xmm1, xmm2, xmm3/m64	A	V/V	FMA	Multiply scalar double-precision floating-point value from xmm2 and xmm3/mem, negate the multiplication result and subtract xmm1 and put result in xmm1.
EVEX.DDS.LIG.66.0F38.W1 9F /r VFNMSUB132SD xmm1 {k1}{z}, xmm2, xmm3/m64{er}	B	V/V	AVX512F	Multiply scalar double-precision floating-point value from xmm1 and xmm3/m64, negate the multiplication result and subtract xmm2 and put result in xmm1.
EVEX.DDS.LIG.66.0F38.W1 AF /r VFNMSUB213SD xmm1 {k1}{z}, xmm2, xmm3/m64{er}	B	V/V	AVX512F	Multiply scalar double-precision floating-point value from xmm1 and xmm2, negate the multiplication result and subtract xmm3/m64 and put result in xmm1.
EVEX.DDS.LIG.66.0F38.W1 BF /r VFNMSUB231SD xmm1 {k1}{z}, xmm2, xmm3/m64{er}	B	V/V	AVX512F	Multiply scalar double-precision floating-point value from xmm2 and xmm3/m64, negate the multiplication result and subtract xmm1 and put result in xmm1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	NA	ModRM:reg (r, w)	VEX.vvvv (r)	ModRM:r/m (r)	NA
B	Tuple1 Scalar	ModRM:reg (r, w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA

Description

VFNMSUB132SD: Multiplies the low packed double-precision floating-point value from the first source operand to the low packed double-precision floating-point value in the third source operand. From negated infinite precision intermediate result, subtracts the low double-precision floating-point value in the second source operand, performs rounding and stores the resulting packed double-precision floating-point value to the destination operand (first source operand).

VFNMSUB213SD: Multiplies the low packed double-precision floating-point value from the second source operand to the low packed double-precision floating-point value in the first source operand. From negated infinite precision intermediate result, subtracts the low double-precision floating-point value in the third source operand, performs rounding and stores the resulting packed double-precision floating-point value to the destination operand (first source operand).

VFNMSUB231SD: Multiplies the low packed double-precision floating-point value from the second source to the low packed double-precision floating-point value in the third source operand. From negated infinite precision intermediate result, subtracts the low double-precision floating-point value in the first source operand, performs rounding and stores the resulting packed double-precision floating-point value to the destination operand (first source operand).

VEX.128 and EVEX encoded version: The destination operand (also first source operand) is encoded in reg_field. The second source operand is encoded in VEX.vvvv/EVEX.vvvv. The third source operand is encoded in rm_field. Bits 127:64 of the destination are unchanged. Bits MAXVL-1:128 of the destination register are zeroed.

EVEX encoded version: The low quadword element of the destination is updated according to the writemask.

Compiler tools may optionally support a complementary mnemonic for each instruction mnemonic listed in the opcode/instruction column of the summary table. The behavior of the complementary mnemonic in situations involving NANs are governed by the definition of the instruction mnemonic defined in the opcode/instruction column.

Operation

In the operations below, "*" and "-" symbols represent multiplication and subtraction with infinite precision inputs and outputs (no rounding).

VFNMSUB132SD DEST, SRC2, SRC3 (EVEX encoded version)

```
IF (EVEX.b = 1) and SRC3 *is a register*
    THEN
        SET_RM(EVEX.RC);
    ELSE
        SET_RM(MXCSR.RM);
FI;
IF k1[0] or *no writemask*
    THEN    DEST[63:0] ← RoundFPControl(-(DEST[63:0]*SRC3[63:0]) - SRC2[63:0])
    ELSE
        IF *merging-masking*                ; merging-masking
            THEN *DEST[63:0] remains unchanged*
            ELSE                            ; zeroing-masking
                THEN DEST[63:0] ← 0
        FI;
FI;
DEST[127:64] ← DEST[127:64]
DEST[MAXVL-1:128] ← 0
```

VFNMSUB213SD DEST, SRC2, SRC3 (EVEX encoded version)

```
IF (EVEX.b = 1) and SRC3 *is a register*
    THEN
        SET_RM(EVEX.RC);
    ELSE
        SET_RM(MXCSR.RM);
FI;
IF k1[0] or *no writemask*
    THEN    DEST[63:0] ← RoundFPControl(-(SRC2[63:0]*DEST[63:0]) - SRC3[63:0])
    ELSE
        IF *merging-masking*                ; merging-masking
            THEN *DEST[63:0] remains unchanged*
            ELSE                            ; zeroing-masking
                THEN DEST[63:0] ← 0
        FI;
FI;
DEST[127:64] ← DEST[127:64]
DEST[MAXVL-1:128] ← 0
```

VFNMSUB231SD DEST, SRC2, SRC3 (EVEX encoded version)

```
IF (EVEX.b = 1) and SRC3 *is a register*
    THEN
        SET_RM(EVEX.RC);
    ELSE
        SET_RM(MXCSR.RM);
FI;
IF k1[0] or *no writemask*
    THEN    DEST[63:0] ← RoundFPControl(-(SRC2[63:0]*SRC3[63:0]) - DEST[63:0])
    ELSE
        IF *merging-masking*                    ; merging-masking
            THEN *DEST[63:0] remains unchanged*
            ELSE                                ; zeroing-masking
                THEN DEST[63:0] ← 0
        FI;
FI;
DEST[127:64] ← DEST[127:64]
DEST[MAXVL-1:128] ← 0
```

VFNMSUB132SD DEST, SRC2, SRC3 (VEX encoded version)

```
DEST[63:0] ←RoundFPControl_MXCSR(- (DEST[63:0]*SRC3[63:0]) - SRC2[63:0])
DEST[127:64] ←DEST[127:64]
DEST[MAXVL-1:128] ←0
```

VFNMSUB213SD DEST, SRC2, SRC3 (VEX encoded version)

```
DEST[63:0] ←RoundFPControl_MXCSR(- (SRC2[63:0]*DEST[63:0]) - SRC3[63:0])
DEST[127:64] ←DEST[127:64]
DEST[MAXVL-1:128] ←0
```

VFNMSUB231SD DEST, SRC2, SRC3 (VEX encoded version)

```
DEST[63:0] ←RoundFPControl_MXCSR(- (SRC2[63:0]*SRC3[63:0]) - DEST[63:0])
DEST[127:64] ←DEST[127:64]
DEST[MAXVL-1:128] ←0
```

Intel C/C++ Compiler Intrinsic Equivalent

VFNMSUBxxxSD __m128d _mm_fnmsub_round_sd(__m128d a, __m128d b, __m128d c, int r);
VFNMSUBxxxSD __m128d _mm_mask_fnmsub_sd(__m128d a, __mmask8 k, __m128d b, __m128d c);
VFNMSUBxxxSD __m128d _mm_maskz_fnmsub_sd(__mmask8 k, __m128d a, __m128d b, __m128d c);
VFNMSUBxxxSD __m128d _mm_mask3_fnmsub_sd(__m128d a, __m128d b, __m128d c, __mmask8 k);
VFNMSUBxxxSD __m128d _mm_mask_fnmsub_round_sd(__m128d a, __mmask8 k, __m128d b, __m128d c, int r);
VFNMSUBxxxSD __m128d _mm_maskz_fnmsub_round_sd(__mmask8 k, __m128d a, __m128d b, __m128d c, int r);
VFNMSUBxxxSD __m128d _mm_mask3_fnmsub_round_sd(__m128d a, __m128d b, __m128d c, __mmask8 k, int r);
VFNMSUBxxxSD __m128d _mm_fnmsub_sd (__m128d a, __m128d b, __m128d c);

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal

Other Exceptions

VEX-encoded instructions, see Exceptions Type 3.
EVEX-encoded instructions, see Exceptions Type E3.

VFNMSUB132SS/VFNMSUB213SS/VFNMSUB231SS—Fused Negative Multiply-Subtract of Scalar Single-Precision Floating-Point Values

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
VEX.DDS.LIG.66.0F38.W0 9F /r VFNMSUB132SS xmm1, xmm2, xmm3/m32	A	V/V	FMA	Multiply scalar single-precision floating-point value from xmm1 and xmm3/m32, negate the multiplication result and subtract xmm2 and put result in xmm1.
VEX.DDS.LIG.66.0F38.W0 AF /r VFNMSUB213SS xmm1, xmm2, xmm3/m32	A	V/V	FMA	Multiply scalar single-precision floating-point value from xmm1 and xmm2, negate the multiplication result and subtract xmm3/m32 and put result in xmm1.
VEX.DDS.LIG.66.0F38.W0 BF /r VFNMSUB231SS xmm1, xmm2, xmm3/m32	A	V/V	FMA	Multiply scalar single-precision floating-point value from xmm2 and xmm3/m32, negate the multiplication result and subtract xmm1 and put result in xmm1.
EVEX.DDS.LIG.66.0F38.W0 9F /r VFNMSUB132SS xmm1 {k1}{z}, xmm2, xmm3/m32{er}	B	V/V	AVX512F	Multiply scalar single-precision floating-point value from xmm1 and xmm3/m32, negate the multiplication result and subtract xmm2 and put result in xmm1.
EVEX.DDS.LIG.66.0F38.W0 AF /r VFNMSUB213SS xmm1 {k1}{z}, xmm2, xmm3/m32{er}	B	V/V	AVX512F	Multiply scalar single-precision floating-point value from xmm1 and xmm2, negate the multiplication result and subtract xmm3/m32 and put result in xmm1.
EVEX.DDS.LIG.66.0F38.W0 BF /r VFNMSUB231SS xmm1 {k1}{z}, xmm2, xmm3/m32{er}	B	V/V	AVX512F	Multiply scalar single-precision floating-point value from xmm2 and xmm3/m32, negate the multiplication result and subtract xmm1 and put result in xmm1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	NA	ModRM:reg (r, w)	VEX.vvvv (r)	ModRM:r/m (r)	NA
B	Tuple1 Scalar	ModRM:reg (r, w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA

Description

VFNMSUB132SS: Multiplies the low packed single-precision floating-point value from the first source operand to the low packed single-precision floating-point value in the third source operand. From negated infinite precision intermediate result, the low single-precision floating-point value in the second source operand, performs rounding and stores the resulting packed single-precision floating-point value to the destination operand (first source operand).

VFNMSUB213SS: Multiplies the low packed single-precision floating-point value from the second source operand to the low packed single-precision floating-point value in the first source operand. From negated infinite precision intermediate result, the low single-precision floating-point value in the third source operand, performs rounding and stores the resulting packed single-precision floating-point value to the destination operand (first source operand).

VFNMSUB231SS: Multiplies the low packed single-precision floating-point value from the second source to the low packed single-precision floating-point value in the third source operand. From negated infinite precision intermediate result, the low single-precision floating-point value in the first source operand, performs rounding and stores the resulting packed single-precision floating-point value to the destination operand (first source operand).

VEX.128 and EVEX encoded version: The destination operand (also first source operand) is encoded in reg_field. The second source operand is encoded in VEX.vvvv/EVEX.vvvv. The third source operand is encoded in rm_field. Bits 127:32 of the destination are unchanged. Bits MAXVL-1:128 of the destination register are zeroed.

EVEX encoded version: The low doubleword element of the destination is updated according to the writemask.

Compiler tools may optionally support a complementary mnemonic for each instruction mnemonic listed in the opcode/instruction column of the summary table. The behavior of the complementary mnemonic in situations involving NANs are governed by the definition of the instruction mnemonic defined in the opcode/instruction column.

Operation

In the operations below, "*" and "-" symbols represent multiplication and subtraction with infinite precision inputs and outputs (no rounding).

VFNMSUB132SS DEST, SRC2, SRC3 (EVEX encoded version)

```
IF (EVEX.b = 1) and SRC3 *is a register*
    THEN
        SET_RM(EVEX.RC);
    ELSE
        SET_RM(MXCSR.RM);
FI;
IF k1[0] or *no writemask*
    THEN    DEST[31:0] ← RoundFPControl(-(DEST[31:0]*SRC3[31:0]) - SRC2[31:0])
    ELSE
        IF *merging-masking*                ; merging-masking
            THEN *DEST[31:0] remains unchanged*
            ELSE                            ; zeroing-masking
                THEN DEST[31:0] ← 0
        FI;
FI;
DEST[127:32] ← DEST[127:32]
DEST[MAXVL-1:128] ← 0
```

VFNMSUB213SS DEST, SRC2, SRC3 (EVEX encoded version)

```
IF (EVEX.b = 1) and SRC3 *is a register*
    THEN
        SET_RM(EVEX.RC);
    ELSE
        SET_RM(MXCSR.RM);
FI;
IF k1[0] or *no writemask*
    THEN    DEST[31:0] ← RoundFPControl(-(SRC2[31:0]*DEST[31:0]) - SRC3[31:0])
    ELSE
        IF *merging-masking*                ; merging-masking
            THEN *DEST[31:0] remains unchanged*
            ELSE                            ; zeroing-masking
                THEN DEST[31:0] ← 0
        FI;
FI;
DEST[127:32] ← DEST[127:32]
DEST[MAXVL-1:128] ← 0
```

VFNMSUB231SS DEST, SRC2, SRC3 (EVEX encoded version)

```
IF (EVEX.b = 1) and SRC3 *is a register*
    THEN
        SET_RM(EVEX.RC);
    ELSE
        SET_RM(MXCSR.RM);
FI;
IF k1[0] or *no writemask*
    THEN    DEST[31:0] ← RoundFPControl(-(SRC2[31:0]*SRC3[63:0]) - DEST[31:0])
    ELSE
        IF *merging-masking*                ; merging-masking
            THEN *DEST[31:0] remains unchanged*
            ELSE                            ; zeroing-masking
                THEN DEST[31:0] ← 0
        FI;
FI;
DEST[127:32] ← DEST[127:32]
DEST[MAXVL-1:128] ← 0
```

VFNMSUB132SS DEST, SRC2, SRC3 (VEX encoded version)

```
DEST[31:0] ←RoundFPControl_MXCSR(- (DEST[31:0]*SRC3[31:0]) - SRC2[31:0])
DEST[127:32] ←DEST[127:32]
DEST[MAXVL-1:128] ←0
```

VFNMSUB213SS DEST, SRC2, SRC3 (VEX encoded version)

```
DEST[31:0] ←RoundFPControl_MXCSR(- (SRC2[31:0]*DEST[31:0]) - SRC3[31:0])
DEST[127:32] ←DEST[127:32]
DEST[MAXVL-1:128] ←0
```

VFNMSUB231SS DEST, SRC2, SRC3 (VEX encoded version)

```
DEST[31:0] ←RoundFPControl_MXCSR(- (SRC2[31:0]*SRC3[31:0]) - DEST[31:0])
DEST[127:32] ←DEST[127:32]
DEST[MAXVL-1:128] ←0
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VFNMSUBxxxSS __m128 _mm_fnmsub_round_ss(__m128 a, __m128 b, __m128 c, int r);
VFNMSUBxxxSS __m128 _mm_mask_fnmsub_ss(__m128 a, __mmask8 k, __m128 b, __m128 c);
VFNMSUBxxxSS __m128 _mm_maskz_fnmsub_ss(__mmask8 k, __m128 a, __m128 b, __m128 c);
VFNMSUBxxxSS __m128 _mm_mask3_fnmsub_ss(__m128 a, __m128 b, __m128 c, __mmask8 k);
VFNMSUBxxxSS __m128 _mm_mask_fnmsub_round_ss(__m128 a, __mmask8 k, __m128 b, __m128 c, int r);
VFNMSUBxxxSS __m128 _mm_maskz_fnmsub_round_ss(__mmask8 k, __m128 a, __m128 b, __m128 c, int r);
VFNMSUBxxxSS __m128 _mm_mask3_fnmsub_round_ss(__m128 a, __m128 b, __m128 c, __mmask8 k, int r);
VFNMSUBxxxSS __m128 _mm_fnmsub_ss (__m128 a, __m128 b, __m128 c);
```

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal

Other Exceptions

VEX-encoded instructions, see Exceptions Type 3.

EVEX-encoded instructions, see Exceptions Type E3.

VFPCLASSPD—Tests Types Of a Packed Float64 Values

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.128.66.0F3A.W1 66 /r ib VFPCLASSPD k2 {k1}, xmm2/m128/m64bcst, imm8	A	V/V	AVX512VL AVX512DQ	Tests the input for the following categories: NaN, +0, -0, +Infinity, -Infinity, denormal, finite negative. The immediate field provides a mask bit for each of these category tests. The masked test results are OR-ed together to form a mask result.
EVEX.256.66.0F3A.W1 66 /r ib VFPCLASSPD k2 {k1}, ymm2/m256/m64bcst, imm8	A	V/V	AVX512VL AVX512DQ	Tests the input for the following categories: NaN, +0, -0, +Infinity, -Infinity, denormal, finite negative. The immediate field provides a mask bit for each of these category tests. The masked test results are OR-ed together to form a mask result.
EVEX.512.66.0F3A.W1 66 /r ib VFPCLASSPD k2 {k1}, zmm2/m512/m64bcst, imm8	A	V/V	AVX512DQ	Tests the input for the following categories: NaN, +0, -0, +Infinity, -Infinity, denormal, finite negative. The immediate field provides a mask bit for each of these category tests. The masked test results are OR-ed together to form a mask result.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Full Vector	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

Description

The FPCLASSPD instruction checks the packed double precision floating point values for special categories, specified by the set bits in the imm8 byte. Each set bit in imm8 specifies a category of floating-point values that the input data element is classified against. The classified results of all specified categories of an input value are ORed together to form the final boolean result for the input element. The result of each element is written to the corresponding bit in a mask register k2 according to the writemask k1. Bits [MAX_KL-1:8/4/2] of the destination are cleared.

The classification categories specified by imm8 are shown in Figure 5-13. The classification test for each category is listed in Table 5-6.

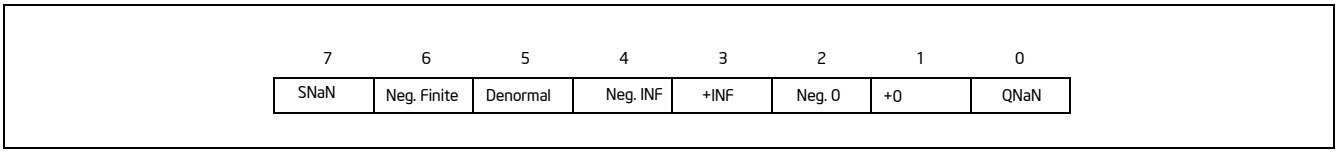


Figure 5-13. Imm8 Byte Specifier of Special Case FP Values for VFPCLASSPD/SD/PS/SS

Table 5-6. Classifier Operations for VFPCLASSPD/SD/PS/SS

Bits	Imm8[0]	Imm8[1]	Imm8[2]	Imm8[3]	Imm8[4]	Imm8[5]	Imm8[6]	Imm8[7]
Category	QNAN	PosZero	NegZero	PosINF	NegINF	Denormal	Negative	SNAN
Classifier	Checks for QNaN	Checks for +0	Checks for -0	Checks for +INF	Checks for -INF	Checks for Denormal	Checks for Negative finite	Checks for SNaN

The source operand is a ZMM/YMM/XMM register, a 512/256/128-bit memory location, or a 512/256/128-bit vector broadcasted from a 64-bit memory location.

EVEX.vvvv is reserved and must be 1111b otherwise instructions will #UD.

Operation

```
CheckFPClassDP (tsrc[63:0], imm8[7:0]){

    //* Start checking the source operand for special type *//
    NegNum  ← tsrc[63];
    IF (tsrc[62:52]=07FFh) Then ExpAllOnes ← 1; FI;
    IF (tsrc[62:52]=0h) Then ExpAllZeros ← 1;
    IF (ExpAllZeros AND MXCSR.DAZ) Then
        MantAllZeros ← 1;
    ELSIF (tsrc[51:0]=0h) Then
        MantAllZeros ← 1;
    FI;
    ZeroNumber ← ExpAllZeros AND MantAllZeros
    SignalingBit ← tsrc[51];

    sNaN_res ← ExpAllOnes AND  NOT(MantAllZeros) AND NOT(SignalingBit); // sNaN
    qNaN_res ← ExpAllOnes AND  NOT(MantAllZeros) AND SignalingBit; // qNaN
    Pzero_res ← NOT(NegNum) AND ExpAllZeros AND MantAllZeros; // +0
    Nzero_res ← NegNum AND ExpAllZeros AND MantAllZeros; // -0
    PInf_res ← NOT(NegNum) AND ExpAllOnes AND MantAllZeros; // +Inf
    NInf_res ← NegNum AND ExpAllOnes AND MantAllZeros; // -Inf
    Denorm_res ← ExpAllZeros AND  NOT(MantAllZeros); // denorm
    FinNeg_res ← NegNum AND NOT(ExpAllOnes) AND NOT(ZeroNumber); // -finite

    bResult = ( imm8[0] AND qNaN_res ) OR (imm8[1] AND Pzero_res ) OR
              ( imm8[2] AND Nzero_res ) OR ( imm8[3] AND PInf_res ) OR
              ( imm8[4] AND NInf_res ) OR ( imm8[5] AND Denorm_res ) OR
              ( imm8[6] AND FinNeg_res ) OR ( imm8[7] AND sNaN_res );
    Return bResult;
} //* end of CheckFPClassDP() *//
```

VFPCLASSPD (EVEX Encoded versions)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j ← 0 TO KL-1
    i ← j * 64
    IF k1[j] OR *no writemask*
        THEN
            IF (EVEX.b == 1) AND (SRC *is memory*)
                THEN
                    DEST[j] ← CheckFPClassDP(SRC1[63:0], imm8[7:0]);
                ELSE
                    DEST[j] ← CheckFPClassDP(SRC1[i+63:i], imm8[7:0]);
            FI;
        ELSE DEST[j] ← 0                    ; zeroing-masking only
    FI;
ENDFOR
DEST[MAX_KL-1:KL] ← 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VFPCLASSPD __mmask8 _mm512_fpclass_pd_mask(__m512d a, int c);
VFPCLASSPD __mmask8 _mm512_mask_fpclass_pd_mask(__mmask8 m, __m512d a, int c)
VFPCLASSPD __mmask8 _mm256_fpclass_pd_mask(__m256d a, int c)
VFPCLASSPD __mmask8 _mm256_mask_fpclass_pd_mask(__mmask8 m, __m256d a, int c)
VFPCLASSPD __mmask8 _mm_fpclass_pd_mask(__m128d a, int c)
VFPCLASSPD __mmask8 _mm_mask_fpclass_pd_mask(__mmask8 m, __m128d a, int c)

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type E4

#UD	If EVEX.vvvv != 1111B.

VFPCLASSPS—Tests Types Of a Packed Float32 Values

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.128.66.0F3A.W0 66 /r ib VFPCLASSPS k2 {k1}, xmm2/m128/m32bcst, imm8	A	V/V	AVX512VL AVX512DQ	Tests the input for the following categories: NaN, +0, -0, +Infinity, -Infinity, denormal, finite negative. The immediate field provides a mask bit for each of these category tests. The masked test results are OR-ed together to form a mask result.
EVEX.256.66.0F3A.W0 66 /r ib VFPCLASSPS k2 {k1}, ymm2/m256/m32bcst, imm8	A	V/V	AVX512VL AVX512DQ	Tests the input for the following categories: NaN, +0, -0, +Infinity, -Infinity, denormal, finite negative. The immediate field provides a mask bit for each of these category tests. The masked test results are OR-ed together to form a mask result.
EVEX.512.66.0F3A.W0 66 /r ib VFPCLASSPS k2 {k1}, zmm2/m512/m32bcst, imm8	A	V/V	AVX512DQ	Tests the input for the following categories: NaN, +0, -0, +Infinity, -Infinity, denormal, finite negative. The immediate field provides a mask bit for each of these category tests. The masked test results are OR-ed together to form a mask result.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Full Vector	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

Description

The FPCLASSPS instruction checks the packed single-precision floating point values for special categories, specified by the set bits in the imm8 byte. Each set bit in imm8 specifies a category of floating-point values that the input data element is classified against. The classified results of all specified categories of an input value are ORed together to form the final boolean result for the input element. The result of each element is written to the corresponding bit in a mask register k2 according to the writemask k1. Bits [MAX_KL-1:16/8/4] of the destination are cleared.

The classification categories specified by imm8 are shown in Figure 5-13. The classification test for each category is listed in Table 5-6.

The source operand is a ZMM/YMM/XMM register, a 512/256/128-bit memory location, or a 512/256/128-bit vector broadcasted from a 32-bit memory location.

EVEX.vvvv is reserved and must be 1111b otherwise instructions will #UD.

Operation

```
CheckFPClassSP (tsrc[31:0], imm8[7:0]){

    //* Start checking the source operand for special type *//
    NegNum  ← tsrc[31];
    IF (tsrc[30:23]=0FFh) Then ExpAllOnes ← 1; FI;
    IF (tsrc[30:23]=0h) Then ExpAllZeros ← 1;
    IF (ExpAllZeros AND MXCSR.DAZ) Then
        MantAllZeros ← 1;
    ELSIF (tsrc[22:0]=0h) Then
        MantAllZeros ← 1;
    FI;
    ZeroNumber= ExpAllZeros AND MantAllZeros
    SignalingBit= tsrc[22];

    sNaN_res ← ExpAllOnes AND  NOT(MantAllZeros) AND NOT(SignalingBit); // sNaN
    qNaN_res ← ExpAllOnes AND  NOT(MantAllZeros) AND SignalingBit; // qNaN
    Pzero_res ← NOT(NegNum) AND ExpAllZeros AND MantAllZeros; // +0
```

```
    Nzero_res ← NegNum AND ExpAllZeros AND MantAllZeros; // -0
    PInf_res ← NOT(NegNum) AND ExpAllOnes AND MantAllZeros; // +Inf
    NInf_res ← NegNum AND ExpAllOnes AND MantAllZeros; // -Inf
    Denorm_res ← ExpAllZeros AND  NOT(MantAllZeros); // denorm
    FinNeg_res ← NegNum AND NOT(ExpAllOnes) AND NOT(ZeroNumber); // -finite

    bResult = ( imm8[0] AND qNaN_res ) OR (imm8[1] AND Pzero_res ) OR
              ( imm8[2] AND Nzero_res ) OR ( imm8[3] AND PInf_res ) OR
              ( imm8[4] AND NInf_res ) OR ( imm8[5] AND Denorm_res ) OR
              ( imm8[6] AND FinNeg_res ) OR ( imm8[7] AND sNaN_res );
    Return bResult;
} //* end of CheckSPClassSP() *//
```

VFPCLASSPS (EVEX encoded versions)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j ← 0 TO KL-1
    i ← j * 32
    IF k1[j] OR *no writemask*
        THEN
            IF (EVEX.b == 1) AND (SRC *is memory*)
                THEN
                    DEST[j] ← CheckFPClassDP(SRC1[31:0], imm8[7:0]);
                ELSE
                    DEST[j] ← CheckFPClassDP(SRC1[i+31:i], imm8[7:0]);
            FI;
        ELSE  DEST[j] ← 0                    ; zeroing-masking only
    FI;
ENDFOR
DEST[MAX_KL-1:KL] ← 0
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VFPCLASSPS __mmask16 _mm512_fpclass_ps_mask( __m512 a, int c);
VFPCLASSPS __mmask16 _mm512_mask_fpclass_ps_mask( __mmask16 m, __m512 a, int c)
VFPCLASSPS __mmask8 _mm256_fpclass_ps_mask( __m256 a, int c)
VFPCLASSPS __mmask8 _mm256_mask_fpclass_ps_mask( __mmask8 m, __m256 a, int c)
VFPCLASSPS __mmask8 _mm_fpclass_ps_mask( __m128 a, int c)
VFPCLASSPS __mmask8 _mm_mask_fpclass_ps_mask( __mmask8 m, __m128 a, int c)
```

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type E4

#UD	If EVEX.vvvv != 1111B.

VFPCLASSSD—Tests Types Of a Scalar Float64 Values

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.LIG.66.0F3A.W1 67 /r ib VFPCLASSSD k2 {k1}, xmm2/m64, imm8	A	V/V	AVX512DQ	Tests the input for the following categories: NaN, +0, -0, +Infinity, -Infinity, denormal, finite negative. The immediate field provides a mask bit for each of these category tests. The masked test results are OR-ed together to form a mask result.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Tuple1 Scalar	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

Description

The FPCLASSSD instruction checks the low double precision floating point value in the source operand for special categories, specified by the set bits in the imm8 byte. Each set bit in imm8 specifies a category of floating-point values that the input data element is classified against. The classified results of all specified categories of an input value are ORed together to form the final boolean result for the input element. The result is written to the low bit in a mask register k2 according to the writemask k1. Bits MAX_KL-1: 1 of the destination are cleared.

The classification categories specified by imm8 are shown in Figure 5-13. The classification test for each category is listed in Table 5-6.

EVEX.vvvv is reserved and must be 1111b otherwise instructions will #UD.

Operation

```
CheckFPClassDP (tsrc[63:0], imm8[7:0]){

    NegNum  ← tsrc[63];
    IF (tsrc[62:52]=07FFh) Then ExpAllOnes ← 1; FI;
    IF (tsrc[62:52]=0h) Then ExpAllZeros ← 1;
    IF (ExpAllZeros AND MXCSR.DAZ) Then
          MantAllZeros ← 1;
    ELSIF (tsrc[51:0]=0h) Then
          MantAllZeros ← 1;
    FI;
    ZeroNumber ← ExpAllZeros AND MantAllZeros
    SignalingBit ← tsrc[51];

    sNaN_res ← ExpAllOnes AND  NOT(MantAllZeros) AND NOT(SignalingBit); // sNaN
    qNaN_res ← ExpAllOnes AND  NOT(MantAllZeros) AND SignalingBit; // qNaN
    Pzero_res ← NOT(NegNum) AND ExpAllZeros AND MantAllZeros; // +0
    Nzero_res ← NegNum AND ExpAllZeros AND MantAllZeros; // -0
    PInf_res ← NOT(NegNum) AND ExpAllOnes AND MantAllZeros; // +Inf
    NInf_res ← NegNum AND ExpAllOnes AND MantAllZeros; // -Inf
    Denorm_res ← ExpAllZeros AND  NOT(MantAllZeros); // denorm
    FinNeg_res ← NegNum AND NOT(ExpAllOnes) AND NOT(ZeroNumber); // -finite

    bResult = ( imm8[0] AND qNaN_res ) OR (imm8[1] AND Pzero_res ) OR
              ( imm8[2] AND Nzero_res ) OR ( imm8[3] AND PInf_res ) OR
              ( imm8[4] AND NInf_res ) OR ( imm8[5] AND Denorm_res ) OR
              ( imm8[6] AND FinNeg_res ) OR ( imm8[7] AND sNaN_res );
    Return bResult;
} //* end of CheckFPClassDP() *//
```

VFPCLASSSD (EVEX encoded version)

```
IF k1[0] OR *no writemask*
    THEN DEST[0] ←
            CheckFPClassDP(SRC1[63:0], imm8[7:0])
    ELSE  DEST[0] ← 0                ; zeroing-masking only
FI;
DEST[MAX_KL-1:1] ← 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VFPCLASSSD __mmask8 _mm_fpclass_sd_mask(__m128d a, int c)
VFPCLASSSD __mmask8 _mm_mask_fpclass_sd_mask(__mmask8 m, __m128d a, int c)

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type E6

#UD	If EVEX.vvvv != 1111B.

VFPCLASSSS—Tests Types Of a Scalar Float32 Values

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.LIG.66.0F3A.W0 67 /r VFPCLASSSS k2 {k1}, xmm2/m32, imm8	A	V/V	AVX512DQ	Tests the input for the following categories: NaN, +0, -0, +Infinity, -Infinity, denormal, finite negative. The immediate field provides a mask bit for each of these category tests. The masked test results are OR-ed together to form a mask result.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Tuple1 Scalar	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

Description

The FPCLASSSS instruction checks the low single-precision floating point value in the source operand for special categories, specified by the set bits in the imm8 byte. Each set bit in imm8 specifies a category of floating-point values that the input data element is classified against. The classified results of all specified categories of an input value are ORed together to form the final boolean result for the input element. The result is written to the low bit in a mask register k2 according to the writemask k1. Bits MAX_KL-1: 1 of the destination are cleared.

The classification categories specified by imm8 are shown in Figure 5-13. The classification test for each category is listed in Table 5-6.

EVEX.vvvv is reserved and must be 1111b otherwise instructions will #UD.

Operation

```
CheckFPClassSP (tsrc[31:0], imm8[7:0]){

    //* Start checking the source operand for special type *//
    NegNum  ← tsrc[31];
    IF (tsrc[30:23]=0FFh) Then ExpAllOnes ← 1; FI;
    IF (tsrc[30:23]=0h) Then ExpAllZeros ← 1;
    IF (ExpAllZeros AND MXCSR.DAZ) Then
          MantAllZeros ← 1;
    ELSIF (tsrc[22:0]=0h) Then
          MantAllZeros ← 1;
    FI;
    ZeroNumber= ExpAllZeros AND MantAllZeros
    SignalingBit= tsrc[22];

    sNaN_res ← ExpAllOnes AND  NOT(MantAllZeros) AND NOT(SignalingBit); // sNaN
    qNaN_res ← ExpAllOnes AND  NOT(MantAllZeros) AND SignalingBit; // qNaN
    Pzero_res ← NOT(NegNum) AND ExpAllZeros AND MantAllZeros; // +0
    Nzero_res ← NegNum AND ExpAllZeros AND MantAllZeros; // -0
    PInf_res ← NOT(NegNum) AND ExpAllOnes AND MantAllZeros; // +Inf
    NInf_res ← NegNum AND ExpAllOnes AND MantAllZeros; // -Inf
    Denorm_res ← ExpAllZeros AND  NOT(MantAllZeros); // denorm
    FinNeg_res ← NegNum AND NOT(ExpAllOnes) AND NOT(ZeroNumber); // -finite

    bResult = ( imm8[0] AND qNaN_res ) OR (imm8[1] AND Pzero_res ) OR
              ( imm8[2] AND Nzero_res ) OR ( imm8[3] AND PInf_res ) OR
              ( imm8[4] AND NInf_res ) OR ( imm8[5] AND Denorm_res ) OR
              ( imm8[6] AND FinNeg_res ) OR ( imm8[7] AND sNaN_res );
    Return bResult;
```

```
} //* end of CheckSPClassSP() *//
```

VFPCLASSSS (EVEX encoded version)

```
IF k1[0] OR *no writemask*
    THEN DEST[0] ←
                CheckFPClassSP(SRC1[31:0], imm8[7:0])
    ELSE  DEST[0] ← 0                ; zeroing-masking only
FI;
DEST[MAX_KL-1:1] ← 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VFPCLASSSS __mmask8 _mm_fpclass_ss_mask(__m128 a, int c)
VFPCLASSSS __mmask8 _mm_mask_fpclass_ss_mask(__mmask8 m, __m128 a, int c)

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type E6

#UD	If EVEX.vvvv != 1111B.

VGATHERDPD/VGATHERQPD — Gather Packed DP FP Values Using Signed Dword/Qword Indices

Opcode/ Instruction	Op/ En	64/3 2-bit Mode	CPUID Feature Flag	Description
VEX.DDS.128.66.0F38.W1 92 /r VGATHERDPD *xmm1, vm32x, xmm2*	RMV	V/V	AVX2	Using dword indices specified in *vm32x*, gather double-precision FP values from memory conditioned on mask specified by *xmm2*. Conditionally gathered elements are merged into *xmm1*.
VEX.DDS.128.66.0F38.W1 93 /r VGATHERQPD *xmm1, vm64x, xmm2*	RMV	V/V	AVX2	Using qword indices specified in *vm64x*, gather double-precision FP values from memory conditioned on mask specified by *xmm2*. Conditionally gathered elements are merged into *xmm1*.
VEX.DDS.256.66.0F38.W1 92 /r VGATHERDPD *ymm1, vm32x, ymm2*	RMV	V/V	AVX2	Using dword indices specified in *vm32x*, gather double-precision FP values from memory conditioned on mask specified by *ymm2*. Conditionally gathered elements are merged into *ymm1*.
VEX.DDS.256.66.0F38.W1 93 /r VGATHERQPD *ymm1, vm64y, ymm2*	RMV	V/V	AVX2	Using qword indices specified in *vm64y*, gather double-precision FP values from memory conditioned on mask specified by *ymm2*. Conditionally gathered elements are merged into *ymm1*.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RMV	ModRM:reg (r,w)	BaseReg (R): VSIB:base, VectorReg(R): VSIB:index	VEX.vvvv (r, w)	NA

Description

The instruction conditionally loads up to 2 or 4 double-precision floating-point values from memory addresses specified by the memory operand (the second operand) and using qword indices. The memory operand uses the VSIB form of the SIB byte to specify a general purpose register operand as the common base, a vector register for an array of indices relative to the base and a constant scale factor.

The mask operand (the third operand) specifies the conditional load operation from each memory address and the corresponding update of each data element of the destination operand (the first operand). Conditionality is specified by the most significant bit of each data element of the mask register. If an element's mask bit is not set, the corresponding element of the destination register is left unchanged. The width of data element in the destination register and mask register are identical. The entire mask register will be set to zero by this instruction unless the instruction causes an exception.

Using dword indices in the lower half of the mask register, the instruction conditionally loads up to 2 or 4 double-precision floating-point values from the VSIB addressing memory operand, and updates the destination register.

This instruction can be suspended by an exception if at least one element is already gathered (i.e., if the exception is triggered by an element other than the rightmost one with its mask bit set). When this happens, the destination register and the mask operand are partially updated; those elements that have been gathered are placed into the destination register and have their mask bits set to zero. If any traps or interrupts are pending from already gathered elements, they will be delivered in lieu of the exception; in this case, EFLAG.RF is set to one so an instruction breakpoint is not re-triggered when the instruction is continued.

If the data size and index size are different, part of the destination register and part of the mask register do not correspond to any elements being gathered. This instruction sets those parts to zero. It may do this to one or both of those registers even if the instruction triggers an exception, and even if the instruction triggers the exception before gathering any elements.

VEX.128 version: The instruction will gather two double-precision floating-point values. For dword indices, only the lower two indices in the vector index register are used.

VEX.256 version: The instruction will gather four double-precision floating-point values. For dword indices, only the lower four indices in the vector index register are used.

Note that:

- If any pair of the index, mask, or destination registers are the same, this instruction results a #UD fault.
- The values may be read from memory in any order. Memory ordering with other instructions follows the Intel-64 memory-ordering model.
- Faults are delivered in a right-to-left manner. That is, if a fault is triggered by an element and delivered, all elements closer to the LSB of the destination will be completed (and non-faulting). Individual elements closer to the MSB may or may not be completed. If a given element triggers multiple faults, they are delivered in the conventional order.
- Elements may be gathered in any order, but faults must be delivered in a right-to-left order; thus, elements to the left of a faulting one may be gathered before the fault is delivered. A given implementation of this instruction is repeatable - given the same input values and architectural state, the same set of elements to the left of the faulting one will be gathered.
- This instruction does not perform AC checks, and so will never deliver an AC fault.
- This instruction will cause a #UD if the address size attribute is 16-bit.
- This instruction will cause a #UD if the memory operand is encoded without the SIB byte.
- This instruction should not be used to access memory mapped I/O as the ordering of the individual loads it does is implementation specific, and some implementations may use loads larger than the data element size or load elements an indeterminate number of times.
- The scaled index may require more bits to represent than the address bits used by the processor (e.g., in 32-bit mode, if the scale is greater than one). In this case, the most significant bits beyond the number of address bits are ignored.

Operation

```
DEST ← SRC1;
BASE_ADDR: base register encoded in VSIB addressing;
VINDEX: the vector index register encoded by VSIB addressing;
SCALE: scale factor encoded by SIB:[7:6];
DISP: optional 1, 4 byte displacement;
MASK ← SRC3;
```

VGATHERDPD (VEX.128 version)

```
FOR j← 0 to 1
    i ← j * 64;
    IF MASK[63+i] THEN
        MASK[i +63:i] ← FFFFFFFF_FFFFFFFFH; // extend from most significant bit
    ELSE
        MASK[i +63:i] ← 0;
    FI;
ENDFOR
FOR j← 0 to 1
    k ← j * 32;
    i ← j * 64;
    DATA_ADDR ← BASE_ADDR + (SignExtend(VINDEX[k+31:k])*SCALE + DISP;
    IF MASK[63+i] THEN
        DEST[i +63:i] ← FETCH_64BITS(DATA_ADDR); // a fault exits the instruction
    FI;
    MASK[i +63: i] ← 0;
ENDFOR
MASK[MAXVL-1:128] ← 0;
DEST[MAXVL-1:128] ← 0;
```

(non-masked elements of the mask register have the content of respective element cleared)

VGATHERQPD (VEX.128 version)

```
FOR j← 0 to 1
    i ← j * 64;
    IF MASK[63+i] THEN
        MASK[i +63:i] ← FFFFFFFF_FFFFFFFFH; // extend from most significant bit
    ELSE
        MASK[i +63:i] ← 0;
    FI;
ENDFOR
FOR j← 0 to 1
    i ← j * 64;
    DATA_ADDR ← BASE_ADDR + (SignExtend(VINDEX1[i+63:i])*SCALE + DISP;
    IF MASK[63+i] THEN
        DEST[i +63:i] ← FETCH_64BITS(DATA_ADDR); // a fault exits this instruction
    FI;
    MASK[i +63: i] ← 0;
ENDFOR
MASK[MAXVL-1:128] ← 0;
DEST[MAXVL-1:128] ← 0;
```

(non-masked elements of the mask register have the content of respective element cleared)

VGATHERQPD (VEX.256 version)

```
FOR j← 0 to 3
    i ← j * 64;
    IF MASK[63+i] THEN
        MASK[i +63:i] ← FFFFFFFF_FFFFFFFFH; // extend from most significant bit
    ELSE
        MASK[i +63:i] ← 0;
    FI;
ENDFOR
FOR j← 0 to 3
    i ← j * 64;
    DATA_ADDR ← BASE_ADDR + (SignExtend(VINDEX1[i+63:i])*SCALE + DISP;
    IF MASK[63+i] THEN
        DEST[i +63:i] ← FETCH_64BITS(DATA_ADDR); // a fault exits the instruction
    FI;
    MASK[i +63: i] ← 0;
ENDFOR
```

(non-masked elements of the mask register have the content of respective element cleared)

VGATHERDPD (VEX.256 version)

```
FOR j← 0 to 3
    i ← j * 64;
    IF MASK[63+i] THEN
        MASK[i +63:i] ← FFFFFFFF_FFFFFFFFH; // extend from most significant bit
    ELSE
        MASK[i +63:i] ← 0;
    FI;
ENDFOR
FOR j← 0 to 3
    k ← j * 32;
    i ← j * 64;
    DATA_ADDR ← BASE_ADDR + (SignExtend(VINDEX1[k+31:k])*SCALE + DISP;
    IF MASK[63+i] THEN
        DEST[i +63:i] ← FETCH_64BITS(DATA_ADDR); // a fault exits the instruction
    FI;
    MASK[i +63:i] ← 0;
ENDFOR
```

(non-masked elements of the mask register have the content of respective element cleared)

Intel C/C++ Compiler Intrinsic Equivalent

VGATHERDPD: __m128d _mm_i32gather_pd (double const * base, __m128i index, const int scale);

VGATHERDPD: __m128d _mm_mask_i32gather_pd (__m128d src, double const * base, __m128i index, __m128d mask, const int scale);

VGATHERDPD: __m256d _mm256_i32gather_pd (double const * base, __m128i index, const int scale);

VGATHERDPD: __m256d _mm256_mask_i32gather_pd (__m256d src, double const * base, __m128i index, __m256d mask, const int scale);

VGATHERQPD: __m128d _mm_i64gather_pd (double const * base, __m128i index, const int scale);

VGATHERQPD: __m128d _mm_mask_i64gather_pd (__m128d src, double const * base, __m128i index, __m128d mask, const int scale);

VGATHERQPD: __m256d _mm256_i64gather_pd (double const * base, __m256i index, const int scale);

VGATHERQPD: __m256d _mm256_mask_i64gather_pd (__m256d src, double const * base, __m256i index, __m256d mask, const int scale);

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type 12.

VGATHERDPS/VGATHERQPS — Gather Packed SP FP values Using Signed Dword/Qword Indices

Opcode/ Instruction	Op/ En	64/32 -bit Mode	CPUID Feature Flag	Description
VEX.DDS.128.66.0F38.W0 92 /r VGATHERDPS *xmm1, vm32x, xmm2*	A	V/V	AVX2	Using dword indices specified in *vm32x*, gather single-precision FP values from memory conditioned on mask specified by *xmm2*. Conditionally gathered elements are merged into *xmm1*.
VEX.DDS.128.66.0F38.W0 93 /r VGATHERQPS *xmm1, vm64x, xmm2*	A	V/V	AVX2	Using qword indices specified in *vm64x*, gather single-precision FP values from memory conditioned on mask specified by *xmm2*. Conditionally gathered elements are merged into *xmm1*.
VEX.DDS.256.66.0F38.W0 92 /r VGATHERDPS *ymm1, vm32y, ymm2*	A	V/V	AVX2	Using dword indices specified in *vm32y*, gather single-precision FP values from memory conditioned on mask specified by *ymm2*. Conditionally gathered elements are merged into *ymm1*.
VEX.DDS.256.66.0F38.W0 93 /r VGATHERQPS *xmm1, vm64y, xmm2*	A	V/V	AVX2	Using qword indices specified in *vm64y*, gather single-precision FP values from memory conditioned on mask specified by *xmm2*. Conditionally gathered elements are merged into *xmm1*.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
A	ModRM:reg (r,w)	BaseReg (R): VSIB:base, VectorReg(R): VSIB:index	VEX.vvvv (r, w)	NA

Description

The instruction conditionally loads up to 4 or 8 single-precision floating-point values from memory addresses specified by the memory operand (the second operand) and using dword indices. The memory operand uses the VSIB form of the SIB byte to specify a general purpose register operand as the common base, a vector register for an array of indices relative to the base and a constant scale factor.

The mask operand (the third operand) specifies the conditional load operation from each memory address and the corresponding update of each data element of the destination operand (the first operand). Conditionality is specified by the most significant bit of each data element of the mask register. If an element's mask bit is not set, the corresponding element of the destination register is left unchanged. The width of data element in the destination register and mask register are identical. The entire mask register will be set to zero by this instruction unless the instruction causes an exception.

Using qword indices, the instruction conditionally loads up to 2 or 4 single-precision floating-point values from the VSIB addressing memory operand, and updates the lower half of the destination register. The upper 128 or 256 bits of the destination register are zero'ed with qword indices.

This instruction can be suspended by an exception if at least one element is already gathered (i.e., if the exception is triggered by an element other than the rightmost one with its mask bit set). When this happens, the destination register and the mask operand are partially updated; those elements that have been gathered are placed into the destination register and have their mask bits set to zero. If any traps or interrupts are pending from already gathered elements, they will be delivered in lieu of the exception; in this case, EFLAG.RF is set to one so an instruction breakpoint is not re-triggered when the instruction is continued.

If the data size and index size are different, part of the destination register and part of the mask register do not correspond to any elements being gathered. This instruction sets those parts to zero. It may do this to one or both of those registers even if the instruction triggers an exception, and even if the instruction triggers the exception before gathering any elements.

VEX.128 version: For dword indices, the instruction will gather four single-precision floating-point values. For qword indices, the instruction will gather two values and zeroes the upper 64 bits of the destination.

VEX.256 version: For dword indices, the instruction will gather eight single-precision floating-point values. For qword indices, the instruction will gather four values and zeroes the upper 128 bits of the destination.

Note that:

- If any pair of the index, mask, or destination registers are the same, this instruction results a UD fault.
- The values may be read from memory in any order. Memory ordering with other instructions follows the Intel-64 memory-ordering model.
- Faults are delivered in a right-to-left manner. That is, if a fault is triggered by an element and delivered, all elements closer to the LSB of the destination will be completed (and non-faulting). Individual elements closer to the MSB may or may not be completed. If a given element triggers multiple faults, they are delivered in the conventional order.
- Elements may be gathered in any order, but faults must be delivered in a right-to-left order; thus, elements to the left of a faulting one may be gathered before the fault is delivered. A given implementation of this instruction is repeatable - given the same input values and architectural state, the same set of elements to the left of the faulting one will be gathered.
- This instruction does not perform AC checks, and so will never deliver an AC fault.
- This instruction will cause a #UD if the address size attribute is 16-bit.
- This instruction will cause a #UD if the memory operand is encoded without the SIB byte.
- This instruction should not be used to access memory mapped I/O as the ordering of the individual loads it does is implementation specific, and some implementations may use loads larger than the data element size or load elements an indeterminate number of times.
- The scaled index may require more bits to represent than the address bits used by the processor (e.g., in 32-bit mode, if the scale is greater than one). In this case, the most significant bits beyond the number of address bits are ignored.

Operation

```
DEST ← SRC1;
BASE_ADDR: base register encoded in VSIB addressing;
VINDEX: the vector index register encoded by VSIB addressing;
SCALE: scale factor encoded by SIB:[7:6];
DISP: optional 1, 4 byte displacement;
MASK ← SRC3;
```

VGATHERDPS (VEX.128 version)

```
FOR j← 0 to 3
    i ← j * 32;
    IF MASK[31+i] THEN
        MASK[i +31:i] ← FFFFFFFFH; // extend from most significant bit
    ELSE
        MASK[i +31:i] ← 0;
    FI;
ENDFOR
MASK[MAXVL-1:128] ← 0;
FOR j← 0 to 3
    i ← j * 32;
    DATA_ADDR ← BASE_ADDR + (SignExtend(VINDEX[i+31:i])*SCALE + DISP;
    IF MASK[31+i] THEN
        DEST[i +31:i] ← FETCH_32BITS(DATA_ADDR); // a fault exits the instruction
    FI;
    MASK[i +31:i] ← 0;
ENDFOR
DEST[MAXVL-1:128] ← 0;
```

(non-masked elements of the mask register have the content of respective element cleared)

VGATHERQPS (VEX.128 version)

```
FOR j← 0 to 3
    i ← j * 32;
    IF MASK[31+i] THEN
        MASK[i +31:i] ← FFFFFFFFH; // extend from most significant bit
    ELSE
        MASK[i +31:i] ← 0;
    FI;
ENDFOR
MASK[MAXVL-1:128] ← 0;
FOR j← 0 to 1
    k ← j * 64;
    i ← j * 32;
    DATA_ADDR ← BASE_ADDR + (SignExtend(VINDEX1[k+63:k])*SCALE + DISP;
    IF MASK[31+i] THEN
        DEST[i +31:i] ← FETCH_32BITS(DATA_ADDR); // a fault exits the instruction
    FI;
    MASK[i +31:i] ← 0;
ENDFOR
MASK[127:64] ← 0;
DEST[MAXVL-1:64] ← 0;
```

(non-masked elements of the mask register have the content of respective element cleared)

VGATHERDPS (VEX.256 version)

```
FOR j← 0 to 7
    i ← j * 32;
    IF MASK[31+i] THEN
        MASK[i +31:i] ← FFFFFFFFH; // extend from most significant bit
    ELSE
        MASK[i +31:i] ← 0;
    FI;
ENDFOR
FOR j← 0 to 7
    i ← j * 32;
    DATA_ADDR ← BASE_ADDR + (SignExtend(VINDEX1[i+31:i])*SCALE + DISP;
    IF MASK[31+i] THEN
        DEST[i +31:i] ← FETCH_32BITS(DATA_ADDR); // a fault exits the instruction
    FI;
    MASK[i +31:i] ← 0;
ENDFOR
```

(non-masked elements of the mask register have the content of respective element cleared)

VGATHERQPS (VEX.256 version)

```
FOR j← 0 to 7
    i ← j * 32;
    IF MASK[31+i] THEN
        MASK[i +31:i] ← FFFFFFFFH; // extend from most significant bit
    ELSE
        MASK[i +31:i] ← 0;
    FI;
ENDFOR
FOR j← 0 to 3
    k ← j * 64;
    i ← j * 32;
    DATA_ADDR ← BASE_ADDR + (SignExtend(VINDEX1[k+63:k])*SCALE + DISP;
    IF MASK[31+i] THEN
        DEST[i +31:i] ← FETCH_32BITS(DATA_ADDR); // a fault exits the instruction
    FI;
    MASK[i +31:i] ← 0;
ENDFOR
MASK[MAXVL-1:128] ← 0;
DEST[MAXVL-1:128] ← 0;
```

(non-masked elements of the mask register have the content of respective element cleared)

Intel C/C++ Compiler Intrinsic Equivalent

VGATHERDPS: __m128 _mm_i32gather_ps (float const * base, __m128i index, const int scale);

VGATHERDPS: __m128 _mm_mask_i32gather_ps (__m128 src, float const * base, __m128i index, __m128 mask, const int scale);

VGATHERDPS: __m256 _mm256_i32gather_ps (float const * base, __m256i index, const int scale);

VGATHERDPS: __m256 _mm256_mask_i32gather_ps (__m256 src, float const * base, __m256i index, __m256 mask, const int scale);

VGATHERQPS: __m128 _mm_i64gather_ps (float const * base, __m128i index, const int scale);

VGATHERQPS: __m128 _mm_mask_i64gather_ps (__m128 src, float const * base, __m128i index, __m128 mask, const int scale);

VGATHERQPS: __m128 _mm256_i64gather_ps (float const * base, __m256i index, const int scale);

VGATHERQPS: __m128 _mm256_mask_i64gather_ps (__m128 src, float const * base, __m256i index, __m128 mask, const int scale);

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type 12.

VGATHERDPS/VGATHERDPD—Gather Packed Single, Packed Double with Signed Dword

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.128.66.0F38.W0 92 /vsib VGATHERDPS xmm1 {k1}, vm32x	A	V/V	AVX512VL AVX512F	Using signed dword indices, gather single-precision floating-point values from memory using k1 as completion mask.
EVEX.256.66.0F38.W0 92 /vsib VGATHERDPS ymm1 {k1}, vm32y	A	V/V	AVX512VL AVX512F	Using signed dword indices, gather single-precision floating-point values from memory using k1 as completion mask.
EVEX.512.66.0F38.W0 92 /vsib VGATHERDPS zmm1 {k1}, vm32z	A	V/V	AVX512F	Using signed dword indices, gather single-precision floating-point values from memory using k1 as completion mask.
EVEX.128.66.0F38.W1 92 /vsib VGATHERDPD xmm1 {k1}, vm32x	A	V/V	AVX512VL AVX512F	Using signed dword indices, gather float64 vector into float64 vector xmm1 using k1 as completion mask.
EVEX.256.66.0F38.W1 92 /vsib VGATHERDPD ymm1 {k1}, vm32x	A	V/V	AVX512VL AVX512F	Using signed dword indices, gather float64 vector into float64 vector ymm1 using k1 as completion mask.
EVEX.512.66.0F38.W1 92 /vsib VGATHERDPD zmm1 {k1}, vm32y	A	V/V	AVX512F	Using signed dword indices, gather float64 vector into float64 vector zmm1 using k1 as completion mask.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Tuple1 Scalar	ModRM:reg (w)	BaseReg (R): VSIB:base, VectorReg(R): VSIB:index	NA	NA

Description

A set of single-precision/double-precision faulting-point memory locations pointed by base address BASE_ADDR and index vector V_INDEX with scale SCALE are gathered. The result is written into a vector register. The elements are specified via the VSIB (i.e., the index register is a vector register, holding packed indices). Elements will only be loaded if their corresponding mask bit is one. If an element's mask bit is not set, the corresponding element of the destination register is left unchanged. The entire mask register will be set to zero by this instruction unless it triggers an exception.

This instruction can be suspended by an exception if at least one element is already gathered (i.e., if the exception is triggered by an element other than the right most one with its mask bit set). When this happens, the destination register and the mask register (k1) are partially updated; those elements that have been gathered are placed into the destination register and have their mask bits set to zero. If any traps or interrupts are pending from already gathered elements, they will be delivered in lieu of the exception; in this case, EFLAG.RF is set to one so an instruction breakpoint is not re-triggered when the instruction is continued.

If the data element size is less than the index element size, the higher part of the destination register and the mask register do not correspond to any elements being gathered. This instruction sets those higher parts to zero. It may update these unused elements to one or both of those registers even if the instruction triggers an exception, and even if the instruction triggers the exception before gathering any elements.

Note that:

- The values may be read from memory in any order. Memory ordering with other instructions follows the Intel-64 memory-ordering model.
- Faults are delivered in a right-to-left manner. That is, if a fault is triggered by an element and delivered, all elements closer to the LSB of the destination zmm will be completed (and non-faulting). Individual elements closer to the MSB may or may not be completed. If a given element triggers multiple faults, they are delivered in the conventional order.
- Elements may be gathered in any order, but faults must be delivered in a right-to left order; thus, elements to the left of a faulting one may be gathered before the fault is delivered. A given implementation of this instruction is repeatable - given the same input values and architectural state, the same set of elements to the left of the faulting one will be gathered.

- This instruction does not perform AC checks, and so will never deliver an AC fault.
- Not valid with 16-bit effective addresses. Will deliver a #UD fault.

Note that the presence of VSIB byte is enforced in this instruction. Hence, the instruction will #UD fault if ModRM.rm is different than 100b.

This instruction has special disp8*N and alignment rules. N is considered to be the size of a single vector element.

The scaled index may require more bits to represent than the address bits used by the processor (e.g., in 32-bit mode, if the scale is greater than one). In this case, the most significant bits beyond the number of address bits are ignored.

The instruction will #UD fault if the destination vector zmm1 is the same as index vector VINDEX. The instruction will #UD fault if the k0 mask register is specified.

Operation

BASE_ADDR stands for the memory operand base address (a GPR); may not exist
VINDEX stands for the memory operand vector of indices (a vector register)
SCALE stands for the memory operand scalar (1, 2, 4 or 8)
DISP is the optional 1, 2 or 4 byte displacement

VGATHERDPS (EVEX encoded version)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j ← 0 TO KL-1
    i ← j * 32
    IF k1[j]
        THEN DEST[i+31:i] ←
            MEM[BASE_ADDR +
                    SignExtend(VINDEX[i+31:i]) * SCALE + DISP]
            k1[j] ← 0
        ELSE *DEST[i+31:i] ← remains unchanged*
    FI;
ENDFOR
k1[MAX_KL-1:KL] ← 0
DEST[MAXVL-1:VL] ← 0
```

VGATHERDPD (EVEX encoded version)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j ← 0 TO KL-1
    i ← j * 64
    k ← j * 32
    IF k1[j]
        THEN DEST[i+63:i] ← MEM[BASE_ADDR +
                    SignExtend(VINDEX[k+31:k]) * SCALE + DISP]
            k1[j] ← 0
        ELSE *DEST[i+63:i] ← remains unchanged*
    FI;
ENDFOR
k1[MAX_KL-1:KL] ← 0
DEST[MAXVL-1:VL] ← 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VGATHERDPD __m512d _mm512_i32gather_pd(__m256i vdx, void * base, int scale);
VGATHERDPD __m512d _mm512_mask_i32gather_pd(__m512d s, __mmask8 k, __m256i vdx, void * base, int scale);
VGATHERDPD __m256d _mm256_mmask_i32gather_pd(__m256d s, __mmask8 k, __m128i vdx, void * base, int scale);
VGATHERDPD __m128d _mm_mmask_i32gather_pd(__m128d s, __mmask8 k, __m128i vdx, void * base, int scale);
VGATHERDPS __m512 _mm512_i32gather_ps(__m512i vdx, void * base, int scale);
VGATHERDPS __m512 _mm512_mask_i32gather_ps(__m512 s, __mmask16 k, __m512i vdx, void * base, int scale);
VGATHERDPS __m256 _mm256_mmask_i32gather_ps(__m256 s, __mmask8 k, __m256i vdx, void * base, int scale);
GATHERDPS __m128 _mm_mmask_i32gather_ps(__m128 s, __mmask8 k, __m128i vdx, void * base, int scale);

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type E12.

VGATHERPF0DPS/VGATHERPF0QPS/VGATHERPF0DPD/VGATHERPF0QPD—Sparse Prefetch Packed SP/DP Data Values with Signed Dword, Signed Qword Indices Using T0 Hint

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.512.66.0F38.W0 C6 /1 /vsib VGATHERPF0DPS vm32z {k1}	A	V/V	AVX512PF	Using signed dword indices, prefetch sparse byte memory locations containing single-precision data using opmask k1 and T0 hint.
EVEX.512.66.0F38.W0 C7 /1 /vsib VGATHERPF0QPS vm64z {k1}	A	V/V	AVX512PF	Using signed qword indices, prefetch sparse byte memory locations containing single-precision data using opmask k1 and T0 hint.
EVEX.512.66.0F38.W1 C6 /1 /vsib VGATHERPF0DPD vm32y {k1}	A	V/V	AVX512PF	Using signed dword indices, prefetch sparse byte memory locations containing double-precision data using opmask k1 and T0 hint.
EVEX.512.66.0F38.W1 C7 /1 /vsib VGATHERPF0QPD vm64z {k1}	A	V/V	AVX512PF	Using signed qword indices, prefetch sparse byte memory locations containing double-precision data using opmask k1 and T0 hint.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Tuple1 Scalar	BaseReg (R): VSIB:base, VectorReg(R): VSIB:index	NA	NA	NA

Description

The instruction conditionally prefetches up to sixteen 32-bit or eight 64-bit integer byte data elements. The elements are specified via the VSIB (i.e., the index register is an zmm, holding packed indices). Elements will only be prefetched if their corresponding mask bit is one.

Lines prefetched are loaded into to a location in the cache hierarchy specified by a locality hint (T0):

• T0 (temporal data)—prefetch data into the first level cache.

[PS data] For dword indices, the instruction will prefetch sixteen memory locations. For qword indices, the instruction will prefetch eight values.

[PD data] For dword and qword indices, the instruction will prefetch eight memory locations.

Note that:

(1) The prefetches may happen in any order (or not at all). The instruction is a hint.

(2) The mask is left unchanged.

(3) Not valid with 16-bit effective addresses. Will deliver a #UD fault.

(4) No FP nor memory faults may be produced by this instruction.

(5) Prefetches do not handle cache line splits

(6) A #UD is signaled if the memory operand is encoded without the SIB byte.

Operation

BASE_ADDR stands for the memory operand base address (a GPR); may not exist
VINDEX stands for the memory operand vector of indices (a vector register)
SCALE stands for the memory operand scalar (1, 2, 4 or 8)
DISP is the optional 1, 2 or 4 byte displacement
PREFETCH(mem, Level, State) Prefetches a byte memory location pointed by 'mem' into the cache level specified by 'Level'; a request for exclusive/ownership is done if 'State' is 1. Note that the memory location ignore cache line splits. This operation is considered a hint for the processor and may be skipped depending on implementation.

VGATHERPF0DPS (EVEX encoded version)

```
(KL, VL) = (16, 512)
FOR j ← 0 TO KL-1
    i ← j * 32
    IF k1[j]
        Prefetch( [BASE_ADDR + SignExtend(VINDEX[i+31:i]) * SCALE + DISP], Level=0, RFO = 0)
    FI;
ENDFOR
```

VGATHERPF0DPD (EVEX encoded version)

```
(KL, VL) = (8, 512)
FOR j ← 0 TO KL-1
    i ← j * 64
    k ← j * 32
    IF k1[j]
        Prefetch( [BASE_ADDR + SignExtend(VINDEX[k+31:k]) * SCALE + DISP], Level=0, RFO = 0)
    FI;
ENDFOR
```

VGATHERPF0QPS (EVEX encoded version)

```
(KL, VL) = (8, 256)
FOR j ← 0 TO KL-1
    i ← j * 64
    IF k1[j]
        Prefetch( [BASE_ADDR + SignExtend(VINDEX[i+63:i]) * SCALE + DISP], Level=0, RFO = 0)
    FI;
ENDFOR
```

VGATHERPF0QPD (EVEX encoded version)

```
(KL, VL) = (8, 512)
FOR j ← 0 TO KL-1
    i ← j * 64
    k ← j * 64
    IF k1[j]
        Prefetch( [BASE_ADDR + SignExtend(VINDEX[k+63:k]) * SCALE + DISP], Level=0, RFO = 0)
    FI;
ENDFOR
```

Intel C/C++ Compiler Intrinsic Equivalent

VGATHERPF0DPD void _mm512_mask_prefetch_i32gather_pd(__m256i vdx, __mmask8 m, void * base, int scale, int hint);
VGATHERPF0DPS void _mm512_mask_prefetch_i32gather_ps(__m512i vdx, __mmask16 m, void * base, int scale, int hint);
VGATHERPF0QPD void _mm512_mask_prefetch_i64gather_pd(__m512i vdx, __mmask8 m, void * base, int scale, int hint);
VGATHERPF0QPS void _mm512_mask_prefetch_i64gather_ps(__m512i vdx, __mmask8 m, void * base, int scale, int hint);

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type E12NP.

VGATHERPF1DPS/VGATHERPF1QPS/VGATHERPF1DPD/VGATHERPF1QPD—Sparse Prefetch Packed SP/DP Data Values with Signed Dword, Signed Qword Indices Using T1 Hint

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.512.66.0F38.W0 C6 /2 /vsib VGATHERPF1DPS vm32z {k1}	A	V/V	AVX512PF	Using signed dword indices, prefetch sparse byte memory locations containing single-precision data using opmask k1 and T1 hint.
EVEX.512.66.0F38.W0 C7 /2 /vsib VGATHERPF1QPS vm64z {k1}	A	V/V	AVX512PF	Using signed qword indices, prefetch sparse byte memory locations containing single-precision data using opmask k1 and T1 hint.
EVEX.512.66.0F38.W1 C6 /2 /vsib VGATHERPF1DPD vm32y {k1}	A	V/V	AVX512PF	Using signed dword indices, prefetch sparse byte memory locations containing double-precision data using opmask k1 and T1 hint.
EVEX.512.66.0F38.W1 C7 /2 /vsib VGATHERPF1QPD vm64z {k1}	A	V/V	AVX512PF	Using signed qword indices, prefetch sparse byte memory locations containing double-precision data using opmask k1 and T1 hint.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Tuple1 Scalar	BaseReg (R): VSIB:base, VectorReg(R): VSIB:index	NA	NA	NA

Description

The instruction conditionally prefetches up to sixteen 32-bit or eight 64-bit integer byte data elements. The elements are specified via the VSIB (i.e., the index register is an zmm, holding packed indices). Elements will only be prefetched if their corresponding mask bit is one.

Lines prefetched are loaded into to a location in the cache hierarchy specified by a locality hint (T1):

• T1 (temporal data)—prefetch data into the second level cache.

[PS data] For dword indices, the instruction will prefetch sixteen memory locations. For qword indices, the instruction will prefetch eight values.

[PD data] For dword and qword indices, the instruction will prefetch eight memory locations.

Note that:

(1) The prefetches may happen in any order (or not at all). The instruction is a hint.

(2) The mask is left unchanged.

(3) Not valid with 16-bit effective addresses. Will deliver a #UD fault.

(4) No FP nor memory faults may be produced by this instruction.

(5) Prefetches do not handle cache line splits

(6) A #UD is signaled if the memory operand is encoded without the SIB byte.

Operation

BASE_ADDR stands for the memory operand base address (a GPR); may not exist
VINDEX stands for the memory operand vector of indices (a vector register)
SCALE stands for the memory operand scalar (1, 2, 4 or 8)
DISP is the optional 1, 2 or 4 byte displacement
PREFETCH(mem, Level, State) Prefetches a byte memory location pointed by 'mem' into the cache level specified by 'Level'; a request for exclusive/ownership is done if 'State' is 1. Note that the memory location ignore cache line splits. This operation is considered a hint for the processor and may be skipped depending on implementation.

VGATHERPF1DPS (EVEX encoded version)

```
(KL, VL) = (16, 512)
FOR j ← 0 TO KL-1
    i ← j * 32
    IF k1[j]
        Prefetch( [BASE_ADDR + SignExtend(VINDEX[i+31:i]) * SCALE + DISP], Level=1, RFO = 0)
    FI;
ENDFOR
```

VGATHERPF1DPD (EVEX encoded version)

```
(KL, VL) = (8, 512)
FOR j ← 0 TO KL-1
    i ← j * 64
    k ← j * 32
    IF k1[j]
        Prefetch( [BASE_ADDR + SignExtend(VINDEX[k+31:k]) * SCALE + DISP], Level=1, RFO = 0)
    FI;
ENDFOR
```

VGATHERPF1QPS (EVEX encoded version)

```
(KL, VL) = (8, 256)
FOR j ← 0 TO KL-1
    i ← j * 64
    IF k1[j]
        Prefetch( [BASE_ADDR + SignExtend(VINDEX[i+63:i]) * SCALE + DISP], Level=1, RFO = 0)
    FI;
ENDFOR
```

VGATHERPF1QPD (EVEX encoded version)

```
(KL, VL) = (8, 512)
FOR j ← 0 TO KL-1
    i ← j * 64
    k ← j * 64
    IF k1[j]
        Prefetch( [BASE_ADDR + SignExtend(VINDEX[k+63:k]) * SCALE + DISP], Level=1, RFO = 0)
    FI;
ENDFOR
```

Intel C/C++ Compiler Intrinsic Equivalent

VGATHERPF1DPD void _mm512_mask_prefetch_i32gather_pd(__m256i vdx, __mmask8 m, void * base, int scale, int hint);
VGATHERPF1DPS void _mm512_mask_prefetch_i32gather_ps(__m512i vdx, __mmask16 m, void * base, int scale, int hint);
VGATHERPF1QPD void _mm512_mask_prefetch_i64gather_pd(__m512i vdx, __mmask8 m, void * base, int scale, int hint);
VGATHERPF1QPS void _mm512_mask_prefetch_i64gather_ps(__m512i vdx, __mmask8 m, void * base, int scale, int hint);

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type E12NP.

VGATHERQPS/VGATHERQPD—Gather Packed Single, Packed Double with Signed Qword Indices

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.128.66.0F38.W0 93 /vsib VGATHERQPS xmm1 {k1}, vm64x	A	V/V	AVX512VL AVX512F	Using signed qword indices, gather single-precision floating-point values from memory using k1 as completion mask.
EVEX.256.66.0F38.W0 93 /vsib VGATHERQPS xmm1 {k1}, vm64y	A	V/V	AVX512VL AVX512F	Using signed qword indices, gather single-precision floating-point values from memory using k1 as completion mask.
EVEX.512.66.0F38.W0 93 /vsib VGATHERQPS ymm1 {k1}, vm64z	A	V/V	AVX512F	Using signed qword indices, gather single-precision floating-point values from memory using k1 as completion mask.
EVEX.128.66.0F38.W1 93 /vsib VGATHERQPD xmm1 {k1}, vm64x	A	V/V	AVX512VL AVX512F	Using signed qword indices, gather float64 vector into float64 vector xmm1 using k1 as completion mask.
EVEX.256.66.0F38.W1 93 /vsib VGATHERQPD ymm1 {k1}, vm64y	A	V/V	AVX512VL AVX512F	Using signed qword indices, gather float64 vector into float64 vector ymm1 using k1 as completion mask.
EVEX.512.66.0F38.W1 93 /vsib VGATHERQPD zmm1 {k1}, vm64z	A	V/V	AVX512F	Using signed qword indices, gather float64 vector into float64 vector zmm1 using k1 as completion mask.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Tuple1 Scalar	ModRM:reg (w)	BaseReg (R): VSIB:base, VectorReg(R): VSIB:index	NA	NA

Description

A set of 8 single-precision/double-precision faulting-point memory locations pointed by base address BASE_ADDR and index vector V_INDEX with scale SCALE are gathered. The result is written into vector a register. The elements are specified via the VSIB (i.e., the index register is a vector register, holding packed indices). Elements will only be loaded if their corresponding mask bit is one. If an element's mask bit is not set, the corresponding element of the destination register is left unchanged. The entire mask register will be set to zero by this instruction unless it triggers an exception.

This instruction can be suspended by an exception if at least one element is already gathered (i.e., if the exception is triggered by an element other than the rightmost one with its mask bit set). When this happens, the destination register and the mask register (k1) are partially updated; those elements that have been gathered are placed into the destination register and have their mask bits set to zero. If any traps or interrupts are pending from already gathered elements, they will be delivered in lieu of the exception; in this case, EFLAG.RF is set to one so an instruction breakpoint is not re-triggered when the instruction is continued.

If the data element size is less than the index element size, the higher part of the destination register and the mask register do not correspond to any elements being gathered. This instruction sets those higher parts to zero. It may update these unused elements to one or both of those registers even if the instruction triggers an exception, and even if the instruction triggers the exception before gathering any elements.

Note that:

- The values may be read from memory in any order. Memory ordering with other instructions follows the Intel-64 memory-ordering model.
- Faults are delivered in a right-to-left manner. That is, if a fault is triggered by an element and delivered, all elements closer to the LSB of the destination zmm will be completed (and non-faulting). Individual elements closer to the MSB may or may not be completed. If a given element triggers multiple faults, they are delivered in the conventional order.

- Elements may be gathered in any order, but faults must be delivered in a right-to left order; thus, elements to the left of a faulting one may be gathered before the fault is delivered. A given implementation of this instruction is repeatable - given the same input values and architectural state, the same set of elements to the left of the faulting one will be gathered.
- This instruction does not perform AC checks, and so will never deliver an AC fault.
- Not valid with 16-bit effective addresses. Will deliver a #UD fault.

Note that the presence of VSIB byte is enforced in this instruction. Hence, the instruction will #UD fault if ModRM.rm is different than 100b.

This instruction has special disp8*N and alignment rules. N is considered to be the size of a single vector element.

The scaled index may require more bits to represent than the address bits used by the processor (e.g., in 32-bit mode, if the scale is greater than one). In this case, the most significant bits beyond the number of address bits are ignored.

The instruction will #UD fault if the destination vector zmm1 is the same as index vector VINDEX. The instruction will #UD fault if the k0 mask register is specified.

Operation

BASE_ADDR stands for the memory operand base address (a GPR); may not exist
VINDEX stands for the memory operand vector of indices (a ZMM register)
SCALE stands for the memory operand scalar (1, 2, 4 or 8)
DISP is the optional 1, 2 or 4 byte displacement

VGATHERQPS (EVEX encoded version)

```
(KL, VL) = (2, 64), (4, 128), (8, 256)
FOR j ← 0 TO KL-1

    i ← j * 32
    k ← j * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+31:i] ←
            MEM[BASE_ADDR + (VINDEX[k+63:k]) * SCALE + DISP]
            k1[j] ← 0
        ELSE *DEST[i+31:i] ← remains unchanged*
    FI;
ENDFOR
k1[MAX_KL-1:KL] ← 0
DEST[MAXVL-1:VL/2] ← 0
```

VGATHERQPD (EVEX encoded version)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j ← 0 TO KL-1
    i ← j * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+63:i] ← MEM[BASE_ADDR + (VINDEX[i+63:i]) * SCALE + DISP]
            k1[j] ← 0
        ELSE *DEST[i+63:i] ← remains unchanged*
    FI;
ENDFOR
k1[MAX_KL-1:KL] ← 0
DEST[MAXVL-1:VL] ← 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VGATHERQPD __m512d _mm512_i64gather_pd(__m512i vdx, void * base, int scale);
VGATHERQPD __m512d _mm512_mask_i64gather_pd(__m512d s, __mmask8 k, __m512i vdx, void * base, int scale);
VGATHERQPD __m256d _mm256_mask_i64gather_pd(__m256d s, __mmask8 k, __m256i vdx, void * base, int scale);
VGATHERQPD __m128d _mm_mask_i64gather_pd(__m128d s, __mmask8 k, __m128i vdx, void * base, int scale);
VGATHERQPS __m256 _mm512_i64gather_ps(__m512i vdx, void * base, int scale);
VGATHERQPS __m256 _mm512_mask_i64gather_ps(__m256 s, __mmask16 k, __m512i vdx, void * base, int scale);
VGATHERQPS __m128 _mm256_mask_i64gather_ps(__m128 s, __mmask8 k, __m256i vdx, void * base, int scale);
VGATHERQPS __m128 _mm_mask_i64gather_ps(__m128 s, __mmask8 k, __m128i vdx, void * base, int scale);

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type E12.

VPGATHERDD/VPGATHERQD — Gather Packed Dword Values Using Signed Dword/Qword Indices

Opcode/ Instruction	Op/ En	64/32 -bit Mode	CPUID Feature Flag	Description
VEX.DDS.128.66.0F38.W0 90 /r VPGATHERDD *xmm1, vm32x, xmm2*	RMV	V/V	AVX2	Using dword indices specified in *vm32x*, gather dword values from memory conditioned on mask specified by *xmm2*. Conditionally gathered elements are merged into *xmm1*.
VEX.DDS.128.66.0F38.W0 91 /r VPGATHERQD *xmm1, vm64x, xmm2*	RMV	V/V	AVX2	Using qword indices specified in *vm64x*, gather dword values from memory conditioned on mask specified by *xmm2*. Conditionally gathered elements are merged into *xmm1*.
VEX.DDS.256.66.0F38.W0 90 /r VPGATHERDD *ymm1, vm32y, ymm2*	RMV	V/V	AVX2	Using dword indices specified in *vm32y*, gather dword from memory conditioned on mask specified by *ymm2*. Conditionally gathered elements are merged into *ymm1*.
VEX.DDS.256.66.0F38.W0 91 /r VPGATHERQD *xmm1, vm64y, xmm2*	RMV	V/V	AVX2	Using qword indices specified in *vm64y*, gather dword values from memory conditioned on mask specified by *xmm2*. Conditionally gathered elements are merged into *xmm1*.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RMV	ModRM:reg (r,w)	BaseReg (R): VSIB:base, VectorReg(R): VSIB:index	VEX.vvvv (r, w)	NA

Description

The instruction conditionally loads up to 4 or 8 dword values from memory addresses specified by the memory operand (the second operand) and using dword indices. The memory operand uses the VSIB form of the SIB byte to specify a general purpose register operand as the common base, a vector register for an array of indices relative to the base and a constant scale factor.

The mask operand (the third operand) specifies the conditional load operation from each memory address and the corresponding update of each data element of the destination operand (the first operand). Conditionality is specified by the most significant bit of each data element of the mask register. If an element's mask bit is not set, the corresponding element of the destination register is left unchanged. The width of data element in the destination register and mask register are identical. The entire mask register will be set to zero by this instruction unless the instruction causes an exception.

Using qword indices, the instruction conditionally loads up to 2 or 4 qword values from the VSIB addressing memory operand, and updates the lower half of the destination register. The upper 128 or 256 bits of the destination register are zero'ed with qword indices.

This instruction can be suspended by an exception if at least one element is already gathered (i.e., if the exception is triggered by an element other than the rightmost one with its mask bit set). When this happens, the destination register and the mask operand are partially updated; those elements that have been gathered are placed into the destination register and have their mask bits set to zero. If any traps or interrupts are pending from already gathered elements, they will be delivered in lieu of the exception; in this case, EFLAG.RF is set to one so an instruction breakpoint is not re-triggered when the instruction is continued.

If the data size and index size are different, part of the destination register and part of the mask register do not correspond to any elements being gathered. This instruction sets those parts to zero. It may do this to one or both of those registers even if the instruction triggers an exception, and even if the instruction triggers the exception before gathering any elements.

VEX.128 version: For dword indices, the instruction will gather four dword values. For qword indices, the instruction will gather two values and zeroes the upper 64 bits of the destination.

VEX.256 version: For dword indices, the instruction will gather eight dword values. For qword indices, the instruction will gather four values and zeroes the upper 128 bits of the destination.

Note that:

- If any pair of the index, mask, or destination registers are the same, this instruction results a UD fault.
- The values may be read from memory in any order. Memory ordering with other instructions follows the Intel-64 memory-ordering model.
- Faults are delivered in a right-to-left manner. That is, if a fault is triggered by an element and delivered, all elements closer to the LSB of the destination will be completed (and non-faulting). Individual elements closer to the MSB may or may not be completed. If a given element triggers multiple faults, they are delivered in the conventional order.
- Elements may be gathered in any order, but faults must be delivered in a right-to-left order; thus, elements to the left of a faulting one may be gathered before the fault is delivered. A given implementation of this instruction is repeatable - given the same input values and architectural state, the same set of elements to the left of the faulting one will be gathered.
- This instruction does not perform AC checks, and so will never deliver an AC fault.
- This instruction will cause a #UD if the address size attribute is 16-bit.
- This instruction will cause a #UD if the memory operand is encoded without the SIB byte.
- This instruction should not be used to access memory mapped I/O as the ordering of the individual loads it does is implementation specific, and some implementations may use loads larger than the data element size or load elements an indeterminate number of times.
- The scaled index may require more bits to represent than the address bits used by the processor (e.g., in 32-bit mode, if the scale is greater than one). In this case, the most significant bits beyond the number of address bits are ignored.

Operation

```
DEST ← SRC1;
BASE_ADDR: base register encoded in VSIB addressing;
VINDEX: the vector index register encoded by VSIB addressing;
SCALE: scale factor encoded by SIB:[7:6];
DISP: optional 1, 4 byte displacement;
MASK ← SRC3;

VPGATHERDD (VEX.128 version)
FOR j← 0 to 3
    i ← j * 32;
    IF MASK[31+i] THEN
        MASK[i +31:i] ← FFFFFFFFH; // extend from most significant bit
    ELSE
        MASK[i +31:i] ← 0;
    FI;
ENDFOR
MASK[MAXVL-1:128] ← 0;
FOR j← 0 to 3
    i ← j * 32;
    DATA_ADDR ← BASE_ADDR + (SignExtend(VINDEX[i+31:i])*SCALE + DISP;
    IF MASK[31+i] THEN
        DEST[i +31:i] ← FETCH_32BITS(DATA_ADDR); // a fault exits the instruction
    FI;
    MASK[i +31:i] ← 0;
ENDFOR
DEST[MAXVL-1:128] ← 0;
(non-masked elements of the mask register have the content of respective element  cleared)
```

VPGATHERQD (VEX.128 version)

```
FOR j← 0 to 3
    i ← j * 32;
    IF MASK[31+i] THEN
        MASK[i +31:i] ← FFFFFFFFH; // extend from most significant bit
    ELSE
        MASK[i +31:i] ← 0;
    FI;
ENDFOR
MASK[MAXVL-1:128] ← 0;
FOR j← 0 to 1
    k ← j * 64;
    i ← j * 32;
    DATA_ADDR ← BASE_ADDR + (SignExtend(VINDEX1[k+63:k])*SCALE + DISP;
    IF MASK[31+i] THEN
        DEST[i +31:i] ← FETCH_32BITS(DATA_ADDR); // a fault exits the instruction
    FI;
    MASK[i +31:i] ← 0;
ENDFOR
MASK[127:64] ← 0;
DEST[MAXVL-1:64] ← 0;
```

(non-masked elements of the mask register have the content of respective element cleared)

VPGATHERDD (VEX.256 version)

```
FOR j← 0 to 7
    i ← j * 32;
    IF MASK[31+i] THEN
        MASK[i +31:i] ← FFFFFFFFH; // extend from most significant bit
    ELSE
        MASK[i +31:i] ← 0;
    FI;
ENDFOR
FOR j← 0 to 7
    i ← j * 32;
    DATA_ADDR ← BASE_ADDR + (SignExtend(VINDEX1[i+31:i])*SCALE + DISP;
    IF MASK[31+i] THEN
        DEST[i +31:i] ← FETCH_32BITS(DATA_ADDR); // a fault exits the instruction
    FI;
    MASK[i +31:i] ← 0;
ENDFOR
```

(non-masked elements of the mask register have the content of respective element cleared)

VPGATHERQD (VEX.256 version)

```
FOR j← 0 to 7
    i ← j * 32;
    IF MASK[31+i] THEN
        MASK[i +31:i] ← FFFFFFFFH; // extend from most significant bit
    ELSE
        MASK[i +31:i] ← 0;
    FI;
ENDFOR
FOR j← 0 to 3
    k ← j * 64;
    i ← j * 32;
    DATA_ADDR ← BASE_ADDR + (SignExtend(VINDEX1[k+63:k])*SCALE + DISP;
    IF MASK[31+i] THEN
        DEST[i +31:i] ← FETCH_32BITS(DATA_ADDR); // a fault exits the instruction
    FI;
    MASK[i +31:i] ← 0;
ENDFOR
MASK[MAXVL-1:128] ← 0;
DEST[MAXVL-1:128] ← 0;
```

(non-masked elements of the mask register have the content of respective element cleared)

Intel C/C++ Compiler Intrinsic Equivalent

VPGATHERDD: __m128i _mm_i32gather_epi32 (int const * base, __m128i index, const int scale);

VPGATHERDD: __m128i _mm_mask_i32gather_epi32 (__m128i src, int const * base, __m128i index, __m128i mask, const int scale);

VPGATHERDD: __m256i _mm256_i32gather_epi32 (int const * base, __m256i index, const int scale);

VPGATHERDD: __m256i _mm256_mask_i32gather_epi32 (__m256i src, int const * base, __m256i index, __m256i mask, const int scale);

VPGATHERQD: __m128i _mm_i64gather_epi32 (int const * base, __m128i index, const int scale);

VPGATHERQD: __m128i _mm_mask_i64gather_epi32 (__m128i src, int const * base, __m128i index, __m128i mask, const int scale);

VPGATHERQD: __m128i _mm256_i64gather_epi32 (int const * base, __m256i index, const int scale);

VPGATHERQD: __m128i _mm256_mask_i64gather_epi32 (__m128i src, int const * base, __m256i index, __m128i mask, const int scale);

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type 12.

VPGATHERDD/VPGATHERDQ—Gather Packed Dword, Packed Qword with Signed Dword Indices

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.128.66.0F38.W0 90 /vsib VPGATHERDD xmm1 {k1}, vm32x	A	V/V	AVX512VL AVX512F	Using signed dword indices, gather dword values from memory using writemask k1 for merging-masking.
EVEX.256.66.0F38.W0 90 /vsib VPGATHERDD ymm1 {k1}, vm32y	A	V/V	AVX512VL AVX512F	Using signed dword indices, gather dword values from memory using writemask k1 for merging-masking.
EVEX.512.66.0F38.W0 90 /vsib VPGATHERDD zmm1 {k1}, vm32z	A	V/V	AVX512F	Using signed dword indices, gather dword values from memory using writemask k1 for merging-masking.
EVEX.128.66.0F38.W1 90 /vsib VPGATHERDQ xmm1 {k1}, vm32x	A	V/V	AVX512VL AVX512F	Using signed dword indices, gather quadword values from memory using writemask k1 for merging-masking.
EVEX.256.66.0F38.W1 90 /vsib VPGATHERDQ ymm1 {k1}, vm32x	A	V/V	AVX512VL AVX512F	Using signed dword indices, gather quadword values from memory using writemask k1 for merging-masking.
EVEX.512.66.0F38.W1 90 /vsib VPGATHERDQ zmm1 {k1}, vm32y	A	V/V	AVX512F	Using signed dword indices, gather quadword values from memory using writemask k1 for merging-masking.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Tuple1 Scalar	ModRM:reg (w)	BaseReg (R): VSIB:base, VectorReg(R): VSIB:index	NA	NA

Description

A set of 16 or 8 doubleword/quadword memory locations pointed to by base address BASE_ADDR and index vector VINDEX with scale SCALE are gathered. The result is written into vector zmm1. The elements are specified via the VSIB (i.e., the index register is a zmm, holding packed indices). Elements will only be loaded if their corresponding mask bit is one. If an element's mask bit is not set, the corresponding element of the destination register (zmm1) is left unchanged. The entire mask register will be set to zero by this instruction unless it triggers an exception.

This instruction can be suspended by an exception if at least one element is already gathered (i.e., if the exception is triggered by an element other than the rightmost one with its mask bit set). When this happens, the destination register and the mask register (k1) are partially updated; those elements that have been gathered are placed into the destination register and have their mask bits set to zero. If any traps or interrupts are pending from already gathered elements, they will be delivered in lieu of the exception; in this case, EFLAG.RF is set to one so an instruction breakpoint is not re-triggered when the instruction is continued.

If the data element size is less than the index element size, the higher part of the destination register and the mask register do not correspond to any elements being gathered. This instruction sets those higher parts to zero. It may update these unused elements to one or both of those registers even if the instruction triggers an exception, and even if the instruction triggers the exception before gathering any elements.

Note that:

- The values may be read from memory in any order. Memory ordering with other instructions follows the Intel-64 memory-ordering model.
- Faults are delivered in a right-to-left manner. That is, if a fault is triggered by an element and delivered, all elements closer to the LSB of the destination zmm will be completed (and non-faulting). Individual elements closer to the MSB may or may not be completed. If a given element triggers multiple faults, they are delivered in the conventional order.
- Elements may be gathered in any order, but faults must be delivered in a right-to-left order; thus, elements to the left of a faulting one may be gathered before the fault is delivered. A given implementation of this instruction is repeatable - given the same input values and architectural state, the same set of elements to the left of the faulting one will be gathered.
- This instruction does not perform AC checks, and so will never deliver an AC fault.
- Not valid with 16-bit effective addresses. Will deliver a #UD fault.
- These instructions do not accept zeroing-masking since the 0 values in k1 are used to determine completion.

Note that the presence of VSIB byte is enforced in this instruction. Hence, the instruction will #UD fault if ModRM.rm is different than 100b.

This instruction has the same disp8*N and alignment rules as for scalar instructions (Tuple 1).

The instruction will #UD fault if the destination vector zmm1 is the same as index vector VINDEX. The instruction will #UD fault if the k0 mask register is specified.

The scaled index may require more bits to represent than the address bits used by the processor (e.g., in 32-bit mode, if the scale is greater than one). In this case, the most significant bits beyond the number of address bits are ignored.

Operation

BASE_ADDR stands for the memory operand base address (a GPR); may not exist
VINDEX stands for the memory operand vector of indices (a ZMM register)
SCALE stands for the memory operand scalar (1, 2, 4 or 8)
DISP is the optional 1, 2 or 4 byte displacement

VPGATHERDD (EVEX encoded version)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j ← 0 TO KL-1
    i ← j * 32
    IF k1[j]
        THEN DEST[i+31:i] ← MEM[BASE_ADDR +
                        SignExtend(VINDEX[i+31:i]) * SCALE + DISP]), 1)
             k1[j] ← 0
        ELSE *DEST[i+31:i] ← remains unchanged*             ; Only merging masking is allowed
    FI;
ENDFOR
k1[MAX_KL-1:KL] ← 0
DEST[MAXVL-1:VL] ← 0
```

VPGATHERDQ (EVEX encoded version)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j ← 0 TO KL-1
    i ← j * 64
    k ← j * 32
    IF k1[j]
        THEN DEST[i+63:i] ←
             MEM[BASE_ADDR + SignExtend(VINDEX[k+31:k]) * SCALE + DISP])
             k1[j] ← 0
        ELSE *DEST[i+63:i] ← remains unchanged*             ; Only merging masking is allowed
    FI;
ENDFOR
k1[MAX_KL-1:KL] ← 0
DEST[MAXVL-1:VL] ← 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VPGATHERDD __m512i _mm512_i32gather_epi32(__m512i vdx, void * base, int scale);
VPGATHERDD __m512i _mm512_mask_i32gather_epi32(__m512i s, __mmask16 k, __m512i vdx, void * base, int scale);
VPGATHERDD __m256i _mm256_mmask_i32gather_epi32(__m256i s, __mmask8 k, __m256i vdx, void * base, int scale);
VPGATHERDD __m128i _mm_mmask_i32gather_epi32(__m128i s, __mmask8 k, __m128i vdx, void * base, int scale);
VPGATHERDQ __m512i _mm512_i32logather_epi64(__m256i vdx, void * base, int scale);
VPGATHERDQ __m512i _mm512_mask_i32logather_epi64(__m512i s, __mmask8 k, __m256i vdx, void * base, int scale);
VPGATHERDQ __m256i _mm256_mmask_i32logather_epi64(__m256i s, __mmask8 k, __m128i vdx, void * base, int scale);
VPGATHERDQ __m128i _mm_mmask_i32gather_epi64(__m128i s, __mmask8 k, __m128i vdx, void * base, int scale);

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type E12.

VPGATHERDQ/VPGATHERQQ — Gather Packed Qword Values Using Signed Dword/Qword Indices

Opcode/ Instruction	Op/ En	64/32 -bit Mode	CPUID Feature Flag	Description
VEX.DDS.128.66.0F38.W1 90 /r VPGATHERDQ *xmm1, vm32x, xmm2*	A	V/V	AVX2	Using dword indices specified in *vm32x*, gather qword values from memory conditioned on mask specified by *xmm2*. Conditionally gathered elements are merged into *xmm1*.
VEX.DDS.128.66.0F38.W1 91 /r VPGATHERQQ *xmm1, vm64x, xmm2*	A	V/V	AVX2	Using qword indices specified in *vm64x*, gather qword values from memory conditioned on mask specified by *xmm2*. Conditionally gathered elements are merged into *xmm1*.
VEX.DDS.256.66.0F38.W1 90 /r VPGATHERDQ *ymm1, vm32x, ymm2*	A	V/V	AVX2	Using dword indices specified in *vm32x*, gather qword values from memory conditioned on mask specified by *ymm2*. Conditionally gathered elements are merged into *ymm1*.
VEX.DDS.256.66.0F38.W1 91 /r VPGATHERQQ *ymm1, vm64y, ymm2*	A	V/V	AVX2	Using qword indices specified in *vm64y*, gather qword values from memory conditioned on mask specified by *ymm2*. Conditionally gathered elements are merged into *ymm1*.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
A	ModRM:reg (r,w)	BaseReg (R): VSIB:base, VectorReg(R): VSIB:index	VEX.vvvv (r, w)	NA

Description

The instruction conditionally loads up to 2 or 4 qword values from memory addresses specified by the memory operand (the second operand) and using qword indices. The memory operand uses the VSIB form of the SIB byte to specify a general purpose register operand as the common base, a vector register for an array of indices relative to the base and a constant scale factor.

The mask operand (the third operand) specifies the conditional load operation from each memory address and the corresponding update of each data element of the destination operand (the first operand). Conditionality is specified by the most significant bit of each data element of the mask register. If an element's mask bit is not set, the corresponding element of the destination register is left unchanged. The width of data element in the destination register and mask register are identical. The entire mask register will be set to zero by this instruction unless the instruction causes an exception.

Using dword indices in the lower half of the mask register, the instruction conditionally loads up to 2 or 4 qword values from the VSIB addressing memory operand, and updates the destination register.

This instruction can be suspended by an exception if at least one element is already gathered (i.e., if the exception is triggered by an element other than the rightmost one with its mask bit set). When this happens, the destination register and the mask operand are partially updated; those elements that have been gathered are placed into the destination register and have their mask bits set to zero. If any traps or interrupts are pending from already gathered elements, they will be delivered in lieu of the exception; in this case, EFLAG.RF is set to one so an instruction breakpoint is not re-triggered when the instruction is continued.

If the data size and index size are different, part of the destination register and part of the mask register do not correspond to any elements being gathered. This instruction sets those parts to zero. It may do this to one or both of those registers even if the instruction triggers an exception, and even if the instruction triggers the exception before gathering any elements.

VEX.128 version: The instruction will gather two qword values. For dword indices, only the lower two indices in the vector index register are used.

VEX.256 version: The instruction will gather four qword values. For dword indices, only the lower four indices in the vector index register are used.

Note that:

- If any pair of the index, mask, or destination registers are the same, this instruction results a UD fault.
- The values may be read from memory in any order. Memory ordering with other instructions follows the Intel-64 memory-ordering model.
- Faults are delivered in a right-to-left manner. That is, if a fault is triggered by an element and delivered, all elements closer to the LSB of the destination will be completed (and non-faulting). Individual elements closer to the MSB may or may not be completed. If a given element triggers multiple faults, they are delivered in the conventional order.
- Elements may be gathered in any order, but faults must be delivered in a right-to-left order; thus, elements to the left of a faulting one may be gathered before the fault is delivered. A given implementation of this instruction is repeatable - given the same input values and architectural state, the same set of elements to the left of the faulting one will be gathered.
- This instruction does not perform AC checks, and so will never deliver an AC fault.
- This instruction will cause a #UD if the address size attribute is 16-bit.
- This instruction will cause a #UD if the memory operand is encoded without the SIB byte.
- This instruction should not be used to access memory mapped I/O as the ordering of the individual loads it does is implementation specific, and some implementations may use loads larger than the data element size or load elements an indeterminate number of times.
- The scaled index may require more bits to represent than the address bits used by the processor (e.g., in 32-bit mode, if the scale is greater than one). In this case, the most significant bits beyond the number of address bits are ignored.

Operation

```
DEST ← SRC1;
BASE_ADDR: base register encoded in VSIB addressing;
VINDEX: the vector index register encoded by VSIB addressing;
SCALE: scale factor encoded by SIB:[7:6];
DISP: optional 1, 4 byte displacement;
MASK ← SRC3;

VPGATHERDQ (VEX.128 version)
FOR j← 0 to 1
    i ← j * 64;
    IF MASK[63+i] THEN
        MASK[i +63:i] ← FFFFFFFF_FFFFFFFFH; // extend from most significant bit
    ELSE
        MASK[i +63:i] ← 0;
    FI;
ENDFOR
FOR j← 0 to 1
    k ← j * 32;
    i ← j * 64;
    DATA_ADDR ← BASE_ADDR + (SignExtend(VINDEX[k+31:k])*SCALE + DISP;
    IF MASK[63+i] THEN
        DEST[i +63:i] ← FETCH_64BITS(DATA_ADDR); // a fault exits the instruction
    FI;
    MASK[i +63:i] ← 0;
ENDFOR
MASK[MAXVL-1:128] ← 0;
DEST[MAXVL-1:128] ← 0;
```

(non-masked elements of the mask register have the content of respective element cleared)

VPGATHERQQ (VEX.128 version)

```
FOR j← 0 to 1
    i ← j * 64;
    IF MASK[63+i] THEN
        MASK[i +63:i] ← FFFFFFFF_FFFFFFFFH; // extend from most significant bit
    ELSE
        MASK[i +63:i] ← 0;
    FI;
ENDFOR
FOR j← 0 to 1
    i ←j * 64;
    DATA_ADDR ← BASE_ADDR + (SignExtend(VINDEX1[i+63:i])*SCALE + DISP;
    IF MASK[63+i] THEN
        DEST[i +63:i] ← FETCH_64BITS(DATA_ADDR); // a fault exits the instruction
    FI;
    MASK[i +63:i] ← 0;
ENDFOR
MASK[MAXVL-1:128] ← 0;
DEST[MAXVL-1:128] ← 0;
```

(non-masked elements of the mask register have the content of respective element cleared)

VPGATHERQQ (VEX.256 version)

```
FOR j← 0 to 3
    i ← j * 64;
    IF MASK[63+i] THEN
        MASK[i +63:i] ← FFFFFFFF_FFFFFFFFH; // extend from most significant bit
    ELSE
        MASK[i +63:i] ← 0;
    FI;
ENDFOR
FOR j← 0 to 3
    i ← j * 64;
    DATA_ADDR ← BASE_ADDR + (SignExtend(VINDEX1[i+63:i])*SCALE + DISP;
    IF MASK[63+i] THEN
        DEST[i +63:i] ← FETCH_64BITS(DATA_ADDR); // a fault exits the instruction
    FI;
    MASK[i +63:i] ← 0;
ENDFOR
```

(non-masked elements of the mask register have the content of respective element cleared)

VPGATHERDQ (VEX.256 version)

```
FOR j← 0 to 3
    i ← j * 64;
    IF MASK[63+i] THEN
        MASK[i +63:i] ← FFFFFFFF_FFFFFFFFH; // extend from most significant bit
    ELSE
        MASK[i +63:i] ← 0;
    FI;
ENDFOR
FOR j← 0 to 3
    k ← j * 32;
    i ← j * 64;
```

```
    DATA_ADDR ← BASE_ADDR + (SignExtend(VINDEX1[k+31:k])*SCALE + DISP;
    IF MASK[63+i] THEN
        DEST[i +63:i] ← FETCH_64BITS(DATA_ADDR); // a fault exits the instruction
    FI;
    MASK[i +63:i] ← 0;
ENDFOR
```

(non-masked elements of the mask register have the content of respective element cleared)

Intel C/C++ Compiler Intrinsic Equivalent

VPGATHERDQ: __m128i _mm_i32gather_epi64 (__int64 const * base, __m128i index, const int scale);

VPGATHERDQ: __m128i _mm_mask_i32gather_epi64 (__m128i src, __int64 const * base, __m128i index, __m128i mask, const int scale);

VPGATHERDQ: __m256i _mm256_i32gather_epi64 (__int64 const * base, __m128i index, const int scale);

VPGATHERDQ: __m256i _mm256_mask_i32gather_epi64 (__m256i src, __int64 const * base, __m128i index, __m256i mask, const int scale);

VPGATHERQQ: __m128i _mm_i64gather_epi64 (__int64 const * base, __m128i index, const int scale);

VPGATHERQQ: __m128i _mm_mask_i64gather_epi64 (__m128i src, __int64 const * base, __m128i index, __m128i mask, const int scale);

VPGATHERQQ: __m256i _mm256_i64gather_epi64 __(int64 const * base, __m256i index, const int scale);

VPGATHERQQ: __m256i _mm256_mask_i64gather_epi64 (__m256i src, __int64 const * base, __m256i index, __m256i mask, const int scale);

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type 12.

VPGATHERQD/VPGATHERQQ—Gather Packed Dword, Packed Qword with Signed Qword Indices

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.128.66.0F38.W0 91 /vsib VPGATHERQD xmm1 {k1}, vm64x	A	V/V	AVX512VL AVX512F	Using signed qword indices, gather dword values from memory using writemask k1 for merging-masking.
EVEX.256.66.0F38.W0 91 /vsib VPGATHERQD xmm1 {k1}, vm64y	A	V/V	AVX512VL AVX512F	Using signed qword indices, gather dword values from memory using writemask k1 for merging-masking.
EVEX.512.66.0F38.W0 91 /vsib VPGATHERQD ymm1 {k1}, vm64z	A	V/V	AVX512F	Using signed qword indices, gather dword values from memory using writemask k1 for merging-masking.
EVEX.128.66.0F38.W1 91 /vsib VPGATHERQQ xmm1 {k1}, vm64x	A	V/V	AVX512VL AVX512F	Using signed qword indices, gather quadword values from memory using writemask k1 for merging-masking.
EVEX.256.66.0F38.W1 91 /vsib VPGATHERQQ ymm1 {k1}, vm64y	A	V/V	AVX512VL AVX512F	Using signed qword indices, gather quadword values from memory using writemask k1 for merging-masking.
EVEX.512.66.0F38.W1 91 /vsib VPGATHERQQ zmm1 {k1}, vm64z	A	V/V	AVX512F	Using signed qword indices, gather quadword values from memory using writemask k1 for merging-masking.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Tuple1 Scalar	ModRM:reg (w)	BaseReg (R): VSIB:base, VectorReg(R): VSIB:index	NA	NA

Description

A set of 8 doubleword/quadword memory locations pointed to by base address BASE_ADDR and index vector VINDEX with scale SCALE are gathered. The result is written into a vector register. The elements are specified via the VSIB (i.e., the index register is a vector register, holding packed indices). Elements will only be loaded if their corresponding mask bit is one. If an element's mask bit is not set, the corresponding element of the destination register is left unchanged. The entire mask register will be set to zero by this instruction unless it triggers an exception.

This instruction can be suspended by an exception if at least one element is already gathered (i.e., if the exception is triggered by an element other than the rightmost one with its mask bit set). When this happens, the destination register and the mask register (k1) are partially updated; those elements that have been gathered are placed into the destination register and have their mask bits set to zero. If any traps or interrupts are pending from already gathered elements, they will be delivered in lieu of the exception; in this case, EFLAG.RF is set to one so an instruction breakpoint is not re-triggered when the instruction is continued.

If the data element size is less than the index element size, the higher part of the destination register and the mask register do not correspond to any elements being gathered. This instruction sets those higher parts to zero. It may update these unused elements to one or both of those registers even if the instruction triggers an exception, and even if the instruction triggers the exception before gathering any elements.

Note that:

- The values may be read from memory in any order. Memory ordering with other instructions follows the Intel-64 memory-ordering model.
- Faults are delivered in a right-to-left manner. That is, if a fault is triggered by an element and delivered, all elements closer to the LSB of the destination zmm will be completed (and non-faulting). Individual elements closer to the MSB may or may not be completed. If a given element triggers multiple faults, they are delivered in the conventional order.
- Elements may be gathered in any order, but faults must be delivered in a right-to-left order; thus, elements to the left of a faulting one may be gathered before the fault is delivered. A given implementation of this instruction is repeatable - given the same input values and architectural state, the same set of elements to the left of the faulting one will be gathered.

- This instruction does not perform AC checks, and so will never deliver an AC fault.
- Not valid with 16-bit effective addresses. Will deliver a #UD fault.
- These instructions do not accept zeroing-masking since the 0 values in k1 are used to determine completion.

Note that the presence of VSIB byte is enforced in this instruction. Hence, the instruction will #UD fault if ModRM.rm is different than 100b.

This instruction has the same disp8*N and alignment rules as for scalar instructions (Tuple 1).

The instruction will #UD fault if the destination vector zmm1 is the same as index vector VINDEX. The instruction will #UD fault if the k0 mask register is specified.

The scaled index may require more bits to represent than the address bits used by the processor (e.g., in 32-bit mode, if the scale is greater than one). In this case, the most significant bits beyond the number of address bits are ignored.

Operation

BASE_ADDR stands for the memory operand base address (a GPR); may not exist
VINDEX stands for the memory operand vector of indices (a ZMM register)
SCALE stands for the memory operand scalar (1, 2, 4 or 8)
DISP is the optional 1, 2 or 4 byte displacement

VPGATHERQD (EVEX encoded version)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j ← 0 TO KL-1
    i ← j * 32
    k ← j * 64
    IF k1[j]
        THEN DEST[i+31:i] ← MEM[BASE_ADDR + (VINDEX[k+63:k]) * SCALE + DISP]), 1)
            k1[j] ← 0
        ELSE *DEST[i+31:i] ← remains unchanged*          ; Only merging masking is allowed
    FI;
ENDFOR
k1[MAX_KL-1:KL] ← 0
DEST[MAXVL-1:VL/2] ← 0
```

VPGATHERQQ (EVEX encoded version)

```
(KL, VL) = (2, 64), (4, 128), (8, 256)
FOR j ← 0 TO KL-1
    i ← j * 64
    IF k1[j]
        THEN DEST[i+63:i] ←
            MEM[BASE_ADDR + (VINDEX[i+63:i]) * SCALE + DISP])
            k1[j] ← 0
        ELSE *DEST[i+63:i] ← remains unchanged*          ; Only merging masking is allowed
    FI;
ENDFOR
k1[MAX_KL-1:KL] ← 0
DEST[MAXVL-1:VL] ← 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VPGATHERQD __m256i _mm512_i64gather_epi32(__m512i vdx, void * base, int scale);
VPGATHERQD __m256i _mm512_mask_i64gather_epi32lo(__m256i s, __mmask8 k, __m512i vdx, void * base, int scale);
VPGATHERQD __m128i _mm256_mask_i64gather_epi32lo(__m128i s, __mmask8 k, __m256i vdx, void * base, int scale);
VPGATHERQD __m128i _mm_mask_i64gather_epi32(__m128i s, __mmask8 k, __m128i vdx, void * base, int scale);
VPGATHERQQ __m512i _mm512_i64gather_epi64(__m512i vdx, void * base, int scale);
VPGATHERQQ __m512i _mm512_mask_i64gather_epi64(__m512i s, __mmask8 k, __m512i vdx, void * base, int scale);
VPGATHERQQ __m256i _mm256_mask_i64gather_epi64(__m256i s, __mmask8 k, __m256i vdx, void * base, int scale);
VPGATHERQQ __m128i _mm_mask_i64gather_epi64(__m128i s, __mmask8 k, __m128i vdx, void * base, int scale);

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type E12.

VGETEXPPD—Convert Exponents of Packed DP FP Values to DP FP Values

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.128.66.0F38.W1 42 /r VGETEXPPD xmm1 {k1}{z}, xmm2/m128/m64bcst	A	V/V	AVX512VL AVX512F	Convert the exponent of packed double-precision floating-point values in the source operand to DP FP results representing unbiased integer exponents and stores the results in the destination register.
EVEX.256.66.0F38.W1 42 /r VGETEXPPD ymm1 {k1}{z}, ymm2/m256/m64bcst	A	V/V	AVX512VL AVX512F	Convert the exponent of packed double-precision floating-point values in the source operand to DP FP results representing unbiased integer exponents and stores the results in the destination register.
EVEX.512.66.0F38.W1 42 /r VGETEXPPD zmm1 {k1}{z}, zmm2/m512/m64bcst{sae}	A	V/V	AVX512F	Convert the exponent of packed double-precision floating-point values in the source operand to DP FP results representing unbiased integer exponents and stores the results in the destination under writemask k1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Full Vector	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

Description

Extracts the biased exponents from the normalized DP FP representation of each qword data element of the source operand (the second operand) as unbiased signed integer value, or convert the denormal representation of input data to unbiased negative integer values. Each integer value of the unbiased exponent is converted to double-precision FP value and written to the corresponding qword elements of the destination operand (the first operand) as DP FP numbers.

The destination operand is a ZMM/YMM/XMM register and updated under the writemask. The source operand can be a ZMM/YMM/XMM register, a 512/256/128-bit memory location, or a 512/256/128-bit vector broadcasted from a 64-bit memory location.

EVEX.vvvv is reserved and must be 1111b, otherwise instructions will #UD.

Each GETEXP operation converts the exponent value into a FP number (permitting input value in denormal representation). Special cases of input values are listed in Table 5-7.

The formula is:

GETEXP(x) = floor($\log_2$(|x|))

Notation **floor(x)** stands for the greatest integer not exceeding real number x.

Table 5-7. VGETEXPPD/SD Special Cases

Input Operand	Result	Comments
src1 = NaN	QNaN(src1)	No Exceptions
0 < |src1| < INF	floor($\log_2$(|src1|))	
| src1| = +INF	+INF	
| src1| = 0	-INF	

Operation

```
NormalizeExpTinyDPFP(SRC[63:0])
{
    // Jbit is the hidden integral bit of a FP number. In case of denormal number it has the value of ZERO.
    Src.Jbit ← 0;
    Dst.exp ← 1;
    Dst.fraction ← SRC[51:0];
    WHILE(Src.Jbit = 0)
    {
        Src.Jbit ← Dst.fraction[51];            // Get the fraction MSB
        Dst.fraction ← Dst.fraction << 1 ;      // One bit shift left
        Dst.exp-- ;                 // Decrement the exponent
    }
    Dst.fraction ← 0;               // zero out fraction bits
    Dst.sign ← 1;                   // Return negative sign
    TMP[63:0] ← MXCSR.DAZ? 0 : (Dst.sign << 63) OR (Dst.exp << 52) OR (Dst.fraction) ;
    Return (TMP[63:0]);
}

ConvertExpDPFP(SRC[63:0])
{
    Src.sign ← 0;                   // Zero out sign bit
    Src.exp ← SRC[62:52];
    Src.fraction ← SRC[51:0];
    // Check for NaN
    IF (SRC = NaN)
    {
        IF ( SRC = SNAN ) SET IE;
        Return QNAN(SRC);
    }
    // Check for +INF
    IF (SRC = +INF) Return (SRC);

    // check if zero operand
    IF ((Src.exp = 0) AND ((Src.fraction = 0) OR (MXCSR.DAZ = 1))) Return (-INF);
    }
    ELSE        // check if denormal operand (notice that MXCSR.DAZ = 0)
    {
        IF ((Src.exp = 0) AND (Src.fraction != 0))
        {
            TMP[63:0] ← NormalizeExpTinyDPFP(SRC[63:0]) ;    // Get Normalized Exponent
            Set #DE
        }
        ELSE        // exponent value is correct
        {
            TMP[63:0] ← (Src.sign << 63) OR (Src.exp << 52) OR (Src.fraction) ;
        }
        TMP ← SAR(TMP, 52) ;        // Shift Arithmetic Right
        TMP ← TMP – 1023;           // Subtract Bias
        Return Cvtl2D(TMP);         // Convert INT to Double-Precision FP number
    }
}
```

VGETEXPPD (EVEX encoded versions)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j ← 0 TO KL-1
    i ← j * 64
    IF k1[j] OR *no writemask*
        THEN
            IF (EVEX.b = 1) AND (SRC *is memory*)
                THEN
                    DEST[i+63:i] ←
            ConvertExpDPFP(SRC[63:0])
                ELSE
                    DEST[i+63:i] ←
            ConvertExpDPFP(SRC[i+63:i])
            FI;
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE                                ; zeroing-masking
                    DEST[i+63:i] ← 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VGETEXPPD __m512d _mm512_getexp_pd(__m512d a);
VGETEXPPD __m512d _mm512_mask_getexp_pd(__m512d s, __mmask8 k, __m512d a);
VGETEXPPD __m512d _mm512_maskz_getexp_pd(__mmask8 k, __m512d a);
VGETEXPPD __m512d _mm512_getexp_round_pd(__m512d a, int sae);
VGETEXPPD __m512d _mm512_mask_getexp_round_pd(__m512d s, __mmask8 k, __m512d a, int sae);
VGETEXPPD __m512d _mm512_maskz_getexp_round_pd(__mmask8 k, __m512d a, int sae);
VGETEXPPD __m256d _mm256_getexp_pd(__m256d a);
VGETEXPPD __m256d _mm256_mask_getexp_pd(__m256d s, __mmask8 k, __m256d a);
VGETEXPPD __m256d _mm256_maskz_getexp_pd(__mmask8 k, __m256d a);
VGETEXPPD __m128d _mm_getexp_pd(__m128d a);
VGETEXPPD __m128d _mm_mask_getexp_pd(__m128d s, __mmask8 k, __m128d a);
VGETEXPPD __m128d _mm_maskz_getexp_pd(__mmask8 k, __m128d a);

SIMD Floating-Point Exceptions

Invalid, Denormal

Other Exceptions

See Exceptions Type E2.

#UD	If EVEX.vvvv != 1111B.

VGETEXPPS—Convert Exponents of Packed SP FP Values to SP FP Values

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.128.66.0F38.W0 42 /r VGETEXPPS xmm1 {k1}{z}, xmm2/m128/m32bcst	A	V/V	AVX512VL AVX512F	Convert the exponent of packed single-precision floating-point values in the source operand to SP FP results representing unbiased integer exponents and stores the results in the destination register.
EVEX.256.66.0F38.W0 42 /r VGETEXPPS ymm1 {k1}{z}, ymm2/m256/m32bcst	A	V/V	AVX512VL AVX512F	Convert the exponent of packed single-precision floating-point values in the source operand to SP FP results representing unbiased integer exponents and stores the results in the destination register.
EVEX.512.66.0F38.W0 42 /r VGETEXPPS zmm1 {k1}{z}, zmm2/m512/m32bcst{sae}	A	V/V	AVX512F	Convert the exponent of packed single-precision floating-point values in the source operand to SP FP results representing unbiased integer exponents and stores the results in the destination register.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Full Vector	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

Description

Extracts the biased exponents from the normalized SP FP representation of each dword element of the source operand (the second operand) as unbiased signed integer value, or convert the denormal representation of input data to unbiased negative integer values. Each integer value of the unbiased exponent is converted to single-precision FP value and written to the corresponding dword elements of the destination operand (the first operand) as SP FP numbers.

The destination operand is a ZMM/YMM/XMM register and updated under the writemask. The source operand can be a ZMM/YMM/XMM register, a 512/256/128-bit memory location, or a 512/256/128-bit vector broadcasted from a 32-bit memory location.

EVEX.vvvv is reserved and must be 1111b, otherwise instructions will #UD.

Each GETEXP operation converts the exponent value into a FP number (permitting input value in denormal representation). Special cases of input values are listed in Table 5-8.

The formula is:

GETEXP(x) = floor($\log_2$(|x|))

Notation **floor(x)** stands for maximal integer not exceeding real number x.

Software usage of VGETEXPxx and VGETMANTxx instructions generally involve a combination of GETEXP operation and GETMANT operation (see VGETMANTPD). Thus VGETEXPxx instruction do not require software to handle SIMD FP exceptions.

Table 5-8. VGETEXPPS/SS Special Cases

Input Operand	Result	Comments
src1 = NaN	QNaN(src1)	No Exceptions
0 < |src1| < INF	floor($\log_2$(|src1|))	
| src1| = +INF	+INF	
| src1| = 0	-INF	

Figure 5-14 illustrates the VGETEXPPS functionality on input values with normalized representation.

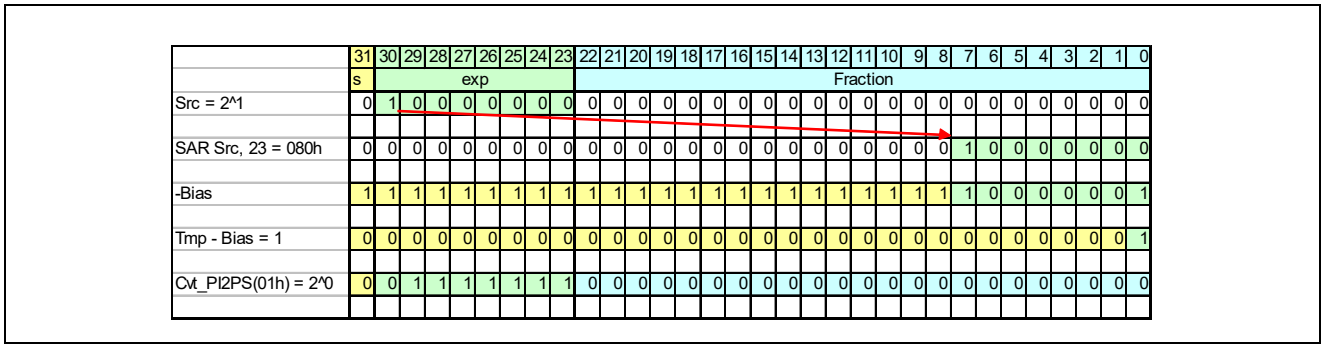


Figure 5-14. VGETEXPPS Functionality On Normal Input values

Operation

```
NormalizeExpTinySPFP(SRC[31:0])
{
    // Jbit is the hidden integral bit of a FP number. In case of denormal number it has the value of ZERO.
    Src.Jbit ← 0;
    Dst.exp ← 1;
    Dst.fraction ← SRC[22:0];
    WHILE(Src.Jbit = 0)
    {
        Src.Jbit ← Dst.fraction[22];          // Get the fraction MSB
        Dst.fraction ← Dst.fraction << 1 ;    // One bit shift left
        Dst.exp-- ;                           // Decrement the exponent
    }
    Dst.fraction ← 0;                         // zero out fraction bits
    Dst.sign ← 1;                             // Return negative sign
    TMP[31:0] ← MXCSR.DAZ? 0 : (Dst.sign << 31) OR (Dst.exp << 23) OR (Dst.fraction) ;
    Return (TMP[31:0]);
}
ConvertExpSPFP(SRC[31:0])
{
    Src.sign ← 0;                             // Zero out sign bit
    Src.exp ← SRC[30:23];
    Src.fraction ← SRC[22:0];
    // Check for NaN
    IF (SRC = NaN)
    {
        IF ( SRC = SNAN ) SET IE;
        Return QNAN(SRC);
    }
    // Check for +INF
    IF (SRC = +INF) Return (SRC);

    // check if zero operand
    IF ((Src.exp = 0) AND ((Src.fraction = 0) OR (MXCSR.DAZ = 1))) Return (-INF);
    }
    ELSE          // check if denormal operand (notice that MXCSR.DAZ = 0)
    {
```

```
        IF ((Src.exp = 0) AND (Src.fraction != 0))
        {
            TMP[31:0] ← NormalizeExpTinySPFP(SRC[31:0]) ;       // Get Normalized Exponent
            Set #DE
        }
        ELSE            // exponent value is correct
        {
            TMP[31:0] ← (Src.sign << 31) OR (Src.exp << 23) OR (Src.fraction) ;
        }
        TMP ← SAR(TMP, 23) ;            // Shift Arithmetic Right
        TMP ← TMP – 127;                // Subtract Bias
        Return CvtI2D(TMP);             // Convert INT to Single-Precision FP number
    }
}
```

VGETEXPPS (EVEX encoded versions)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j ← 0 TO KL-1
    i ← j * 32
    IF k1[j] OR *no writemask*
        THEN
            IF (EVEX.b = 1) AND (SRC *is memory*)
                THEN
                    DEST[i+31:i] ←
            ConvertExpSPFP(SRC[31:0])
                ELSE
                    DEST[i+31:i] ←
            ConvertExpSPFP(SRC[i+31:i])
            FI;
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE                                ; zeroing-masking
                    DEST[i+31:i] ← 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VGETEXPPS __m512 _mm512_getexp_ps(__m512 a);
VGETEXPPS __m512 _mm512_mask_getexp_ps(__m512 s, __mmask16 k, __m512 a);
VGETEXPPS __m512 _mm512_maskz_getexp_ps(__mmask16 k, __m512 a);
VGETEXPPS __m512 _mm512_getexp_round_ps(__m512 a, int sae);
VGETEXPPS __m512 _mm512_mask_getexp_round_ps(__m512 s, __mmask16 k, __m512 a, int sae);
VGETEXPPS __m512 _mm512_maskz_getexp_round_ps(__mmask16 k, __m512 a, int sae);
VGETEXPPS __m256 _mm256_getexp_ps(__m256 a);
VGETEXPPS __m256 _mm256_mask_getexp_ps(__m256 s, __mmask8 k, __m256 a);
VGETEXPPS __m256 _mm256_maskz_getexp_ps(__mmask8 k, __m256 a);
VGETEXPPS __m128 _mm_getexp_ps(__m128 a);
VGETEXPPS __m128 _mm_mask_getexp_ps(__m128 s, __mmask8 k, __m128 a);
VGETEXPPS __m128 _mm_maskz_getexp_ps(__mmask8 k, __m128 a);

SIMD Floating-Point Exceptions

Invalid, Denormal

Other Exceptions

See Exceptions Type E2.

#UD	If EVEX.vvvv != 1111B.

VGETEXPSD—Convert Exponents of Scalar DP FP Values to DP FP Value

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.NDS.LIG.66.0F38.W1 43 /r VGETEXPSD xmm1 {k1}{z}, xmm2, xmm3/m64{sae}	A	V/V	AVX512F	Convert the biased exponent (bits 62:52) of the low double-precision floating-point value in xmm3/m64 to a DP FP value representing unbiased integer exponent. Stores the result to the low 64-bit of xmm1 under the writemask k1 and merge with the other elements of xmm2.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Tuple1 Scalar	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA

Description

Extracts the biased exponent from the normalized DP FP representation of the low qword data element of the source operand (the third operand) as unbiased signed integer value, or convert the denormal representation of input data to unbiased negative integer values. The integer value of the unbiased exponent is converted to double-precision FP value and written to the destination operand (the first operand) as DP FP numbers. Bits (127:64) of the XMM register destination are copied from corresponding bits in the first source operand.

The destination must be a XMM register, the source operand can be a XMM register or a float64 memory location. The low quadword element of the destination operand is conditionally updated with writemask k1.

Each GETEXP operation converts the exponent value into a FP number (permitting input value in denormal representation). Special cases of input values are listed in Table 5-7.

The formula is:

$GETEXP(x) = floor(\log_2(|x|))$

Notation **floor(x)** stands for maximal integer not exceeding real number x.

Operation

```
// NormalizeExpTinyDPFP(SRC[63:0]) is defined in the Operation section of VGETEXPPD

// ConvertExpDPFP(SRC[63:0]) is defined in the Operation section of VGETEXPPD

VGETEXPSD (EVEX encoded version)
IF k1[0] OR *no writemask*
    THEN DEST[63:0] ←
            ConvertExpDPFP(SRC2[63:0])
    ELSE
        IF *merging-masking*                ; merging-masking
            THEN *DEST[63:0] remains unchanged*
            ELSE                            ; zeroing-masking
                DEST[63:0] ← 0
        FI
FI;
DEST[127:64] ← SRC1[127:64]
DEST[MAXVL-1:128] ← 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VGETEXPSD __m128d _mm_getexp_sd(__m128d a, __m128d b);
VGETEXPSD __m128d _mm_mask_getexp_sd(__m128d s, __mmask8 k, __m128d a, __m128d b);
VGETEXPSD __m128d _mm_maskz_getexp_sd(__mmask8 k, __m128d a, __m128d b);
VGETEXPSD __m128d _mm_getexp_round_sd(__m128d a, __m128d b, int sae);
VGETEXPSD __m128d _mm_mask_getexp_round_sd(__m128d s, __mmask8 k, __m128d a, __m128d b, int sae);
VGETEXPSD __m128d _mm_maskz_getexp_round_sd(__mmask8 k, __m128d a, __m128d b, int sae);

SIMD Floating-Point Exceptions

Invalid, Denormal

Other Exceptions

See Exceptions Type E3.

VGETEXPSS—Convert Exponents of Scalar SP FP Values to SP FP Value

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.NDS.LIG.66.0F38.W0 43 /r VGETEXPSS xmm1 {k1}{z}, xmm2, xmm3/m32{sae}	A	V/V	AVX512F	Convert the biased exponent (bits 30:23) of the low single-precision floating-point value in xmm3/m32 to a SP FP value representing unbiased integer exponent. Stores the result to xmm1 under the writemask k1 and merge with the other elements of xmm2.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Tuple1 Scalar	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA

Description

Extracts the biased exponent from the normalized SP FP representation of the low doubleword data element of the source operand (the third operand) as unbiased signed integer value, or convert the denormal representation of input data to unbiased negative integer values. The integer value of the unbiased exponent is converted to single-precision FP value and written to the destination operand (the first operand) as SP FP numbers. Bits (127:32) of the XMM register destination are copied from corresponding bits in the first source operand.

The destination must be a XMM register, the source operand can be a XMM register or a float32 memory location. The the low doubleword element of the destination operand is conditionally updated with writemask k1.

Each GETEXP operation converts the exponent value into a FP number (permitting input value in denormal representation). Special cases of input values are listed in Table 5-8.

The formula is:

GETEXP(x) = floor($\log_2$(|x|))

Notation **floor(x)** stands for maximal integer not exceeding real number x.

Software usage of VGETEXPxx and VGETMANTxx instructions generally involve a combination of GETEXP operation and GETMANT operation (see VGETMANTPD). Thus VGETEXPxx instruction do not require software to handle SIMD FP exceptions.

Operation

```
// NormalizeExpTinySPFP(SRC[31:0]) is defined in the Operation section of VGETEXPPS

// ConvertExpSPFP(SRC[31:0]) is defined in the Operation section of VGETEXPPS

VGETEXPSS (EVEX encoded version)
IF k1[0] OR *no writemask*
    THEN DEST[31:0] ←
            ConvertExpDPFP(SRC2[31:0])
    ELSE
        IF *merging-masking*                ; merging-masking
            THEN *DEST[31:0] remains unchanged*
            ELSE                            ; zeroing-masking
                DEST[31:0]← 0
            FI
    FI;
ENDFOR
DEST[127:32] ← SRC1[127:32]
DEST[MAXVL-1:128] ← 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VGETEXPSS __m128 _mm_getexp_ss(__m128 a, __m128 b);
VGETEXPSS __m128 _mm_mask_getexp_ss(__m128 s, __mmask8 k, __m128 a, __m128 b);
VGETEXPSS __m128 _mm_maskz_getexp_ss(__mmask8 k, __m128 a, __m128 b);
VGETEXPSS __m128 _mm_getexp_round_ss(__m128 a, __m128 b, int sae);
VGETEXPSS __m128 _mm_mask_getexp_round_ss(__m128 s, __mmask8 k, __m128 a, __m128 b, int sae);
VGETEXPSS __m128 _mm_maskz_getexp_round_ss(__mmask8 k, __m128 a, __m128 b, int sae);

SIMD Floating-Point Exceptions

Invalid, Denormal

Other Exceptions

See Exceptions Type E3.

VGETMANTPD—Extract Float64 Vector of Normalized Mantissas from Float64 Vector

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.128.66.0F3A.W1 26 /r ib VGETMANTPD xmm1 {k1}{z}, xmm2/m128/m64bcst, imm8	A	V/V	AVX512VL AVX512F	Get Normalized Mantissa from float64 vector xmm2/m128/m64bcst and store the result in xmm1, using *imm*8 for sign control and mantissa interval normalization, under writemask.
EVEX.256.66.0F3A.W1 26 /r ib VGETMANTPD ymm1 {k1}{z}, ymm2/m256/m64bcst, imm8	A	V/V	AVX512VL AVX512F	Get Normalized Mantissa from float64 vector ymm2/m256/m64bcst and store the result in ymm1, using *imm*8 for sign control and mantissa interval normalization, under writemask.
EVEX.512.66.0F3A.W1 26 /r ib VGETMANTPD zmm1 {k1}{z}, zmm2/m512/m64bcst{sae}, imm8	A	V/V	AVX512F	Get Normalized Mantissa from float64 vector zmm2/m512/m64bcst and store the result in zmm1, using *imm*8 for sign control and mantissa interval normalization, under writemask.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Full Vector	ModRM:reg (w)	ModRM:r/m (r)	Imm8	NA

Description

Convert double-precision floating values in the source operand (the second operand) to DP FP values with the mantissa normalization and sign control specified by the imm8 byte, see Figure 5-15. The converted results are written to the destination operand (the first operand) using writemask k1. The normalized mantissa is specified by interv (imm8[1:0]) and the sign control (sc) is specified by bits 3:2 of the immediate byte.

The destination operand is a ZMM/YMM/XMM register updated under the writemask. The source operand can be a ZMM/YMM/XMM register, a 512/256/128-bit memory location, or a 512/256/128-bit vector broadcasted from a 64-bit memory location.

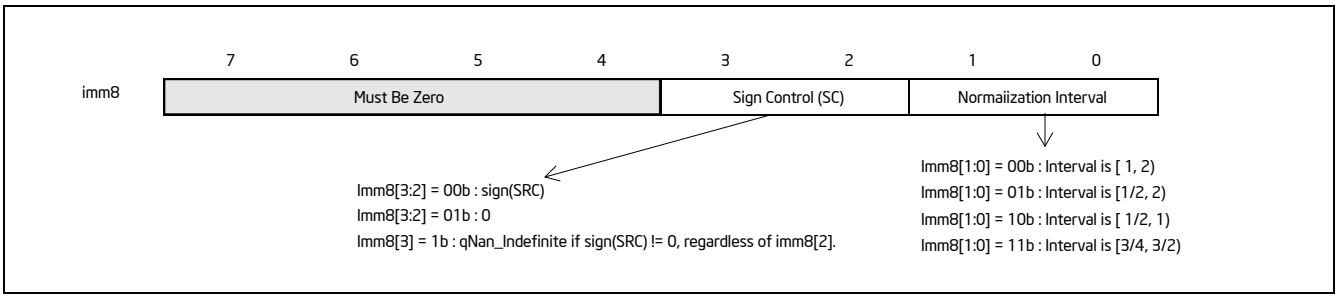


Figure 5-15. Imm8 Controls for VGETMANTPD/SD/PS/SS

For each input DP FP value x, The conversion operation is:

$$GetMant(x) = \pm 2^{k}|x.significand|$$

where:

$$1 <= |x.significand| < 2$$

Unbiased exponent k depends on the interval range defined by interv and whether the exponent of the source is even or odd. The sign of the final result is determined by sc and the source sign.

If interv != 0 then k = -1, otherwise K = 0. The encoded value of imm8[1:0] and sign control are shown in Figure 5-15.

Each converted DP FP result is encoded according to the sign control, the unbiased exponent k (adding bias) and a mantissa normalized to the range specified by interv.

The GetMant() function follows Table 5-9 when dealing with floating-point special numbers.

This instruction is writemasked, so only those elements with the corresponding bit set in vector mask register k1 are computed and stored into the destination. Elements in zmm1 with the corresponding bit clear in k1 retain their previous values.

Note: EVEX.vvvv is reserved and must be 1111b; otherwise instructions will #UD.

Table 5-9. GetMant() Special Float Values Behavior

Input	Result	Exceptions / Comments
NaN	QNaN(SRC)	Ignore *interv* If (SRC = SNaN) then #IE
+∞	1.0	Ignore *interv*
+0	1.0	Ignore *interv*
-0	IF (SC[0]) THEN +1.0 ELSE -1.0	Ignore *interv*

-∞	IF (SC[1]) THEN {QNaN_Indefinite}
	ELSE {
	IF (SC[0]) THEN +1.0
	ELSE -1.0
		Ignore *interv*
		If (SC[1]) then #IE

negative	SC[1] ? QNaN_Indefinite : Getmant(SRC)	If (SC[1]) then #IE

Operation

```
GetNormalizeMantissaDP(SRC[63:0], SignCtrl[1:0], Interv[1:0])
{
    // Extracting the SRC sign, exponent and mantissa fields
    Dst.sign ← SignCtrl[0] ? 0 : Src[63];                    // Get sign bit
    Dst.exp ← SRC[62:52]; ; Get original exponent value
    Dst.fraction ← SRC[51:0];; Get original fraction value
    ZeroOperand ← (Dst.exp = 0) AND (Dst.fraction = 0);
    DenormOperand ← (Dst.exp = 0h) AND (Dst.fraction != 0);
    InfiniteOperand ← (Dst.exp = 07FFh) AND (Dst.fraction = 0);
    NaNOperand ← (Dst.exp = 07FFh) AND (Dst.fraction != 0);
    // Check for NAN operand
    IF (NaNOperand)
    {    IF (SRC = SNaN) {Set #IE;}
         Return QNAN(SRC);
    }
    // Check for Zero and Infinite operands
    IF ((ZeroOperand) OR (InfiniteOperand)
    {    Dst.exp ← 03FFh;                    // Override exponent with BIAS
         Return ((Dst.sign<<63) | (Dst.exp<<52) | (Dst.fraction));
    }
    // Check for negative operand (including -0.0)
    IF ((Src[63] = 1) AND SignCtrl[1])
    {    Set #IE;
         Return QNaN_Indefinite;
    }
```

```
    // Checking for denormal operands
    IF (DenormOperand)
    {     IF (MXCSR.DAZ=1) Dst.fraction ← 0;// Zero out fraction
          ELSE
          {     // Jbit is the hidden integral bit. Zero in case of denormal operand.
                Src.Jbit ← 0;                    // Zero Src Jbit
                Dst.exp ← 03FFh;                 // Override exponent with BIAS
                WHILE (Src.Jbit = 0) {           // normalize mantissa
                      Src.Jbit ← Dst.fraction[51];        // Get the fraction MSB
                      Dst.fraction ← (Dst.fraction << 1);          // Start normalizing the mantissa
                      Dst.exp--;       // Adjust the exponent
                }
                SET #DE;              // Set DE bit
          }
    }           // At this point, Dst.fraction is normalized.
    // Checking for exponent response
    Unbiased.exp ← Dst.exp – 03FFh;            // subtract the bias from exponent
    IsOddExp ← Unbiased.exp[0];                // recognized unbiased ODD exponent
    SignalingBit ← Dst.fraction[51];
    CASE (interv[1:0])
          00: Dst.exp ← 03FFh;                              // This is the bias
          01: Dst.exp ← (IsOddExp) ? 03FEh : 03FFh;         // either bias-1, or bias
          10: Dst.exp ← 03FEh;                              // bias-1
          11: Dst.exp ← (SignalingBit) ? 03FEh : 03FFh;     // either bias-1, or bias
    ESAC
    // At this point Dst.exp has the correct result. Form the final destination
    DEST[63:0] ← (Dst.sign << 63) OR (Dst.exp << 52) OR (Dst.fraction);
    Return (DEST);
}
```

VGETMANTPD (EVEX encoded versions)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
SignCtrl[1:0] ← IMM8[3:2];
Interv[1:0] ← IMM8[1:0];
FOR j ← 0 TO KL-1
    i ← j * 64
    IF k1[j] OR *no writemask*
          THEN
                IF (EVEX.b = 1) AND (SRC *is memory*)
                      THEN
                            DEST[i+63:i] ←   GetNormalizedMantissaDP(SRC[63:0], SignCtrl, Interv)
                      ELSE
                            DEST[i+63:i] ←   GetNormalizedMantissaDP(SRC[i+63:i], SignCtrl, Interv)
                FI;
          ELSE
                IF *merging-masking*                ; merging-masking
                      THEN *DEST[i+63:i] remains unchanged*
                      ELSE                          ; zeroing-masking
                            DEST[i+63:i] ← 0
                FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VGETMANTPD __m512d _mm512_getmant_pd(__m512d a, enum intv, enum sgn);
VGETMANTPD __m512d _mm512_mask_getmant_pd(__m512d s, __mmask8 k, __m512d a, enum intv, enum sgn);
VGETMANTPD __m512d _mm512_maskz_getmant_pd(__mmask8 k, __m512d a, enum intv, enum sgn);
VGETMANTPD __m512d _mm512_getmant_round_pd(__m512d a, enum intv, enum sgn, int r);
VGETMANTPD __m512d _mm512_mask_getmant_round_pd(__m512d s, __mmask8 k, __m512d a, enum intv, enum sgn, int r);
VGETMANTPD __m512d _mm512_maskz_getmant_round_pd(__mmask8 k, __m512d a, enum intv, enum sgn, int r);
VGETMANTPD __m256d _mm256_getmant_pd(__m256d a, enum intv, enum sgn);
VGETMANTPD __m256d _mm256_mask_getmant_pd(__m256d s, __mmask8 k, __m256d a, enum intv, enum sgn);
VGETMANTPD __m256d _mm256_maskz_getmant_pd(__mmask8 k, __m256d a, enum intv, enum sgn);
VGETMANTPD __m128d _mm_getmant_pd(__m128d a, enum intv, enum sgn);
VGETMANTPD __m128d _mm_mask_getmant_pd(__m128d s, __mmask8 k, __m128d a, enum intv, enum sgn);
VGETMANTPD __m128d _mm_maskz_getmant_pd(__mmask8 k, __m128d a, enum intv, enum sgn);

SIMD Floating-Point Exceptions

Denormal, Invalid

Other Exceptions

See Exceptions Type E2.

#UD	If EVEX.vvvv != 1111B.

VGETMANTPS—Extract Float32 Vector of Normalized Mantissas from Float32 Vector

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.128.66.0F3A.W0 26 /r ib VGETMANTPS xmm1 {k1}{z}, xmm2/m128/m32bcst, imm8	A	V/V	AVX512VL AVX512F	Get normalized mantissa from float32 vector xmm2/m128/m32bcst and store the result in xmm1, using imm8 for sign control and mantissa interval normalization, under writemask.
EVEX.256.66.0F3A.W0 26 /r ib VGETMANTPS ymm1 {k1}{z}, ymm2/m256/m32bcst, imm8	A	V/V	AVX512VL AVX512F	Get normalized mantissa from float32 vector ymm2/m256/m32bcst and store the result in ymm1, using imm8 for sign control and mantissa interval normalization, under writemask.
EVEX.512.66.0F3A.W0 26 /r ib VGETMANTPS zmm1 {k1}{z}, zmm2/m512/m32bcst{sae}, imm8	A	V/V	AVX512F	Get normalized mantissa from float32 vector zmm2/m512/m32bcst and store the result in zmm1, using imm8 for sign control and mantissa interval normalization, under writemask.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Full Vector	ModRM:reg (w)	ModRM:r/m (r)	Imm8	NA

Description

Convert single-precision floating values in the source operand (the second operand) to SP FP values with the mantissa normalization and sign control specified by the imm8 byte, see Figure 5-15. The converted results are written to the destination operand (the first operand) using writemask k1. The normalized mantissa is specified by interv (imm8[1:0]) and the sign control (sc) is specified by bits 3:2 of the immediate byte.

The destination operand is a ZMM/YMM/XMM register updated under the writemask. The source operand can be a ZMM/YMM/XMM register, a 512/256/128-bit memory location, or a 512/256/128-bit vector broadcasted from a 32-bit memory location.

For each input SP FP value x, The conversion operation is:

$$GetMant(x) = \pm 2^{k}|x.significand|$$

where:

$$1 <= |x.significand| < 2$$

Unbiased exponent k depends on the interval range defined by interv and whether the exponent of the source is even or odd. The sign of the final result is determined by sc and the source sign.

if interv != 0 then k = -1, otherwise K = 0. The encoded value of imm8[1:0] and sign control are shown in Figure 5-15.

Each converted SP FP result is encoded according to the sign control, the unbiased exponent k (adding bias) and a mantissa normalized to the range specified by interv.

The GetMant() function follows Table 5-9 when dealing with floating-point special numbers.

This instruction is writemasked, so only those elements with the corresponding bit set in vector mask register k1 are computed and stored into the destination. Elements in zmm1 with the corresponding bit clear in k1 retain their previous values.

Note: EVEX.vvvv is reserved and must be 1111b, VEX.L must be 0; otherwise instructions will #UD.

Operation

```
GetNormalizeMantissaSP(SRC[31:0] , SignCtrl[1:0], Interv[1:0])
{
    // Extracting the SRC sign, exponent and mantissa fields
    Dst.sign ← SignCtrl[0] ? 0 : Src[31];                    // Get sign bit
    Dst.exp ← SRC[30:23]; ; Get original exponent value
    Dst.fraction ← SRC[22:0];; Get original fraction value
    ZeroOperand ← (Dst.exp = 0) AND (Dst.fraction = 0);
    DenormOperand ← (Dst.exp = 0h) AND (Dst.fraction != 0);
    InfiniteOperand ← (Dst.exp = 0FFh) AND (Dst.fraction = 0);
    NaNOperand ← (Dst.exp = 0FFh) AND (Dst.fraction != 0);
    // Check for NAN operand
    IF (NaNOperand)
    {    IF (SRC = SNaN) {Set #IE;}
         Return QNAN(SRC);
    }
    // Check for Zero and Infinite operands
    IF ((ZeroOperand) OR (InfiniteOperand)
    {    Dst.exp ← 07Fh;                    // Override exponent with BIAS
         Return ((Dst.sign<<31) | (Dst.exp<<23) | (Dst.fraction));
    }
    // Check for negative operand (including -0.0)
    IF ((Src[31] = 1) AND SignCtrl[1])
    {    Set #IE;
         Return QNaN_Indefinite;
    }
    // Checking for denormal operands
    IF (DenormOperand)
    {    IF (MXCSR.DAZ=1) Dst.fraction ← 0;// Zero out fraction
         ELSE
         {    // Jbit is the hidden integral bit. Zero in case of denormal operand.
              Src.Jbit ← 0;              // Zero Src Jbit
              Dst.exp ← 07Fh;            // Override exponent with BIAS
              WHILE (Src.Jbit = 0) {     // normalize mantissa
                   Src.Jbit ← Dst.fraction[22];     // Get the fraction MSB
                   Dst.fraction ← (Dst.fraction << 1);       // Start normalizing the mantissa
                   Dst.exp--;     // Adjust the exponent
              }
              SET #DE;          // Set DE bit
         }
    }         // At this point, Dst.fraction is normalized.
    // Checking for exponent response
    Unbiased.exp ← Dst.exp – 07Fh;          // subtract the bias from exponent
    IsOddExp ← Unbiased.exp[0];             // recognized unbiased ODD exponent
    SignalingBit ← Dst.fraction[22];
    CASE (interv[1:0])
         00: Dst.exp ← 07Fh;                          // This is the bias
         01: Dst.exp ← (IsOddExp) ? 07Eh : 07Fh;          // either bias-1, or bias
         10: Dst.exp ← 07Eh;                          // bias-1
         11: Dst.exp ← (SignalingBit) ? 07Eh : 07Fh;      // either bias-1, or bias
    ESAC

    // Form the final destination
    DEST[31:0] ← (Dst.sign << 31) OR (Dst.exp << 23) OR (Dst.fraction);
```

```
    Return (DEST);
}
```

VGETMANTPS (EVEX encoded versions)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
SignCtrl[1:0] ← IMM8[3:2];
Interv[1:0] ← IMM8[1:0];
FOR j ← 0 TO KL-1
    i ← j * 32
    IF k1[j] OR *no writemask*
        THEN
            IF (EVEX.b = 1) AND (SRC *is memory*)
                THEN
                    DEST[i+31:i] ←   GetNormalizedMantissaSP(SRC[31:0], SignCtrl, Interv)
                ELSE
                    DEST[i+31:i] ←   GetNormalizedMantissaSP(SRC[i+31:i], SignCtrl, Interv)
            FI;
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE                                ; zeroing-masking
                    DEST[i+31:i] ← 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VGETMANTPS __m512 _mm512_getmant_ps( __m512 a, enum intv, enum sgn);
VGETMANTPS __m512 _mm512_mask_getmant_ps(__m512 s, __mmask16 k, __m512 a, enum intv, enum sgn;
VGETMANTPS __m512 _mm512_maskz_getmant_ps(__mmask16 k, __m512 a, enum intv, enum sgn);
VGETMANTPS __m512 _mm512_getmant_round_ps( __m512 a, enum intv, enum sgn, int r);
VGETMANTPS __m512 _mm512_mask_getmant_round_ps(__m512 s, __mmask16 k, __m512 a, enum intv, enum sgn, int r);
VGETMANTPS __m512 _mm512_maskz_getmant_round_ps(__mmask16 k, __m512 a, enum intv, enum sgn, int r);
VGETMANTPS __m256 _mm256_getmant_ps( __m256 a, enum intv, enum sgn);
VGETMANTPS __m256 _mm256_mask_getmant_ps(__m256 s, __mmask8 k, __m256 a, enum intv, enum sgn);
VGETMANTPS __m256 _mm256_maskz_getmant_ps( __mmask8 k, __m256 a, enum intv, enum sgn);
VGETMANTPS __m128 _mm_getmant_ps( __m128 a, enum intv, enum sgn);
VGETMANTPS __m128 _mm_mask_getmant_ps(__m128 s, __mmask8 k, __m128 a, enum intv, enum sgn);
VGETMANTPS __m128 _mm_maskz_getmant_ps( __mmask8 k, __m128 a, enum intv, enum sgn);
```

SIMD Floating-Point Exceptions

Denormal, Invalid

Other Exceptions

See Exceptions Type E2.

#UD	If EVEX.vvvv != 1111B.

VGETMANTSD—Extract Float64 of Normalized Mantissas from Float64 Scalar

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.NDS.LIG.66.0F3A.W1 27 /r ib VGETMANTSD xmm1 {k1}{z}, xmm2, xmm3/m64{sae}, imm8	A	V/V	AVX512F	Extract the normalized mantissa of the low float64 element in xmm3/m64 using *imm*8 for sign control and mantissa interval normalization. Store the mantissa to xmm1 under the writemask k1 and merge with the other elements of xmm2.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Tuple1 Scalar	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA

Description

Convert the double-precision floating values in the low quadword element of the second source operand (the third operand) to DP FP value with the mantissa normalization and sign control specified by the imm8 byte, see Figure 5-15. The converted result is written to the low quadword element of the destination operand (the first operand) using writemask k1. Bits (127:64) of the XMM register destination are copied from corresponding bits in the first source operand. The normalized mantissa is specified by interv (imm8[1:0]) and the sign control (sc) is specified by bits 3:2 of the immediate byte.

The conversion operation is:

$$GetMant(x) = \pm 2^{k}|x.significand|$$

where:

$$1 <= |x.significand| < 2$$

Unbiased exponent k depends on the interval range defined by interv and whether the exponent of the source is even or odd. The sign of the final result is determined by sc and the source sign.

If interv != 0 then k = -1, otherwise K = 0. The encoded value of imm8[1:0] and sign control are shown in Figure 5-15.

The converted DP FP result is encoded according to the sign control, the unbiased exponent k (adding bias) and a mantissa normalized to the range specified by interv.

The GetMant() function follows Table 5-9 when dealing with floating-point special numbers.

This instruction is writemasked, so only those elements with the corresponding bit set in vector mask register k1 are computed and stored into zmm1. Elements in zmm1 with the corresponding bit clear in k1 retain their previous values.

Operation

```
// GetNormalizeMantissaDP(SRC[63:0], SignCtrl[1:0], Interv[1:0]) is defined in the operation section of VGETMANTPD

VGETMANTSD (EVEX encoded version)
SignCtrl[1:0] ← IMM8[3:2];
Interv[1:0] ← IMM8[1:0];
IF k1[0] OR *no writemask*
    THEN DEST[63:0] ←
            GetNormalizedMantissaDP(SRC2[63:0], SignCtrl, Interv)
    ELSE
        IF *merging-masking*                    ; merging-masking
            THEN *DEST[63:0] remains unchanged*
            ELSE                                ; zeroing-masking
                DEST[63:0] ← 0
        FI
FI;
DEST[127:64] ← SRC1[127:64]
DEST[MAXVL-1:128] ← 0
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VGETMANTSD __m128d _mm_getmant_sd( __m128d a, __m128 b, enum intv, enum sgn);
VGETMANTSD __m128d _mm_mask_getmant_sd(__m128d s, __mmask8 k, __m128d a, __m128d b, enum intv, enum sgn);
VGETMANTSD __m128d _mm_maskz_getmant_sd( __mmask8 k, __m128 a, __m128d b, enum intv, enum sgn);
VGETMANTSD __m128d _mm_getmant_round_sd( __m128d a, __m128 b, enum intv, enum sgn, int r);
VGETMANTSD __m128d _mm_mask_getmant_round_sd(__m128d s, __mmask8 k, __m128d a, __m128d b, enum intv, enum sgn, int r);
VGETMANTSD __m128d _mm_maskz_getmant_round_sd( __mmask8 k, __m128d a, __m128d b, enum intv, enum sgn, int r);
```

SIMD Floating-Point Exceptions

Denormal, Invalid

Other Exceptions

See Exceptions Type E3.

VGETMANTSS—Extract Float32 Vector of Normalized Mantissa from Float32 Vector

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.NDS.LIG.66.0F3A.W0 27 /r ib VGETMANTSS xmm1 {k1}{z}, xmm2, xmm3/m32{sae}, imm8	A	V/V	AVX512F	Extract the normalized mantissa from the low float32 element of xmm3/m32 using imm8 for sign control and mantissa interval normalization, store the mantissa to xmm1 under the writemask k1 and merge with the other elements of xmm2.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Tuple1 Scalar	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA

Description

Convert the single-precision floating values in the low doubleword element of the second source operand (the third operand) to SP FP value with the mantissa normalization and sign control specified by the imm8 byte, see Figure 5-15. The converted result is written to the low doubleword element of the destination operand (the first operand) using writemask k1. Bits (127:32) of the XMM register destination are copied from corresponding bits in the first source operand. The normalized mantissa is specified by interv (imm8[1:0]) and the sign control (sc) is specified by bits 3:2 of the immediate byte.

The conversion operation is:

$$GetMant(x) = \pm 2^{k}|x.significand|$$

where:

$$1 <= |x.significand| < 2$$

Unbiased exponent k depends on the interval range defined by interv and whether the exponent of the source is even or odd. The sign of the final result is determined by sc and the source sign.

if interv != 0 then k = -1, otherwise K = 0. The encoded value of imm8[1:0] and sign control are shown in Figure 5-15.

The converted SP FP result is encoded according to the sign control, the unbiased exponent k (adding bias) and a mantissa normalized to the range specified by interv.

The GetMant() function follows Table 5-9 when dealing with floating-point special numbers.

This instruction is writemasked, so only those elements with the corresponding bit set in vector mask register k1 are computed and stored into zmm1. Elements in zmm1 with the corresponding bit clear in k1 retain their previous values.

Operation

```
// GetNormalizeMantissaSP(SRC[31:0], SignCtrl[1:0], Interv[1:0]) is defined in the operation section of VGETMANTPD

VGETMANTSS (EVEX encoded version)
SignCtrl[1:0] ← IMM8[3:2];
Interv[1:0] ← IMM8[1:0];
IF k1[0] OR *no writemask*
    THEN DEST[31:0] ←
            GetNormalizedMantissaSP(SRC2[31:0], SignCtrl, Interv)
    ELSE
        IF *merging-masking*                    ; merging-masking
            THEN *DEST[31:0] remains unchanged*
            ELSE                                ; zeroing-masking
                DEST[31:0] ← 0
        FI
FI;
DEST[127:32] ← SRC1[127:64]
DEST[MAXVL-1:128] ← 0
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VGETMANTSS __m128 _mm_getmant_ss( __m128 a, __m128 b, enum intv, enum sgn);
VGETMANTSS __m128 _mm_mask_getmant_ss(__m128 s, __mmask8 k, __m128 a, __m128 b, enum intv, enum sgn);
VGETMANTSS __m128 _mm_maskz_getmant_ss( __mmask8 k, __m128 a, __m128 b, enum intv, enum sgn);
VGETMANTSS __m128 _mm_getmant_round_ss( __m128 a, __m128 b, enum intv, enum sgn, int r);
VGETMANTSS __m128 _mm_mask_getmant_round_ss(__m128 s, __mmask8 k, __m128 a, __m128 b, enum intv, enum sgn, int r);
VGETMANTSS __m128 _mm_maskz_getmant_round_ss( __mmask8 k, __m128 a, __m128 b, enum intv, enum sgn, int r);
```

SIMD Floating-Point Exceptions

Denormal, Invalid

Other Exceptions

See Exceptions Type E3.

VINSERTF128/VINSERTF32x4/VINSERTF64x2/VINSERTF32x8/VINSERTF64x4—Insert Packed Floating-Point Values

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
VEX.NDS.256.66.0F3A.W0 18 /r ib VINSERTF128 ymm1, ymm2, xmm3/m128, imm8	A	V/V	AVX	Insert 128 bits of packed floating-point values from xmm3/m128 and the remaining values from ymm2 into ymm1.
EVEX.NDS.256.66.0F3A.W0 18 /r ib VINSERTF32X4 ymm1 {k1}{z}, ymm2, xmm3/m128, imm8	C	V/V	AVX512VL AVX512F	Insert 128 bits of packed single-precision floating-point values from xmm3/m128 and the remaining values from ymm2 into ymm1 under writemask k1.
EVEX.NDS.512.66.0F3A.W0 18 /r ib VINSERTF32X4 zmm1 {k1}{z}, zmm2, xmm3/m128, imm8	C	V/V	AVX512F	Insert 128 bits of packed single-precision floating-point values from xmm3/m128 and the remaining values from zmm2 into zmm1 under writemask k1.
EVEX.NDS.256.66.0F3A.W1 18 /r ib VINSERTF64X2 ymm1 {k1}{z}, ymm2, xmm3/m128, imm8	B	V/V	AVX512VL AVX512DQ	Insert 128 bits of packed double-precision floating-point values from xmm3/m128 and the remaining values from ymm2 into ymm1 under writemask k1.
EVEX.NDS.512.66.0F3A.W1 18 /r ib VINSERTF64X2 zmm1 {k1}{z}, zmm2, xmm3/m128, imm8	B	V/V	AVX512DQ	Insert 128 bits of packed double-precision floating-point values from xmm3/m128 and the remaining values from zmm2 into zmm1 under writemask k1.
EVEX.NDS.512.66.0F3A.W0 1A /r ib VINSERTF32X8 zmm1 {k1}{z}, zmm2, ymm3/m256, imm8	D	V/V	AVX512DQ	Insert 256 bits of packed single-precision floating-point values from ymm3/m256 and the remaining values from zmm2 into zmm1 under writemask k1.
EVEX.NDS.512.66.0F3A.W1 1A /r ib VINSERTF64X4 zmm1 {k1}{z}, zmm2, ymm3/m256, imm8	C	V/V	AVX512F	Insert 256 bits of packed double-precision floating-point values from ymm3/m256 and the remaining values from zmm2 into zmm1 under writemask k1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	NA	ModRM:reg (w)	VEX.vvvv	ModRM:r/m (r)	Imm8
B	Tuple2	ModRM:reg (w)	EVEX.vvvv	ModRM:r/m (r)	Imm8
C	Tuple4	ModRM:reg (w)	EVEX.vvvv	ModRM:r/m (r)	Imm8
D	Tuple8	ModRM:reg (w)	EVEX.vvvv	ModRM:r/m (r)	Imm8

Description

VINSERTF128/VINSERTF32x4 and VINSERTF64x2 insert 128-bits of packed floating-point values from the second source operand (the third operand) into the destination operand (the first operand) at an 128-bit granularity offset multiplied by imm8[0] (256-bit) or imm8[1:0]. The remaining portions of the destination operand are copied from the corresponding fields of the first source operand (the second operand). The second source operand can be either an XMM register or a 128-bit memory location. The destination and first source operands are vector registers.

VINSERTF32x4: The destination operand is a ZMM/YMM register and updated at 32-bit granularity according to the writemask. The high 6/7 bits of the immediate are ignored.

VINSERTF64x2: The destination operand is a ZMM/YMM register and updated at 64-bit granularity according to the writemask. The high 6/7 bits of the immediate are ignored.

VINSERTF32x8 and VINSERTF64x4 inserts 256-bits of packed floating-point values from the second source operand (the third operand) into the destination operand (the first operand) at a 256-bit granular offset multiplied by imm8[0]. The remaining portions of the destination are copied from the corresponding fields of the first source operand (the second operand). The second source operand can be either an YMM register or a 256-bit memory location. The high 7 bits of the immediate are ignored. The destination operand is a ZMM register and updated at 32/64-bit granularity according to the writemask.

Operation

VINSERTF32x4 (EVEX encoded versions)

```
(KL, VL) = (8, 256), (16, 512)
TEMP_DEST[VL-1:0] ← SRC1[VL-1:0]
IF VL = 256
    CASE (imm8[0]) OF
        0:    TMP_DEST[127:0] ← SRC2[127:0]
        1:    TMP_DEST[255:128] ← SRC2[127:0]
    ESAC.
FI;
IF VL = 512
    CASE (imm8[1:0]) OF
        00:  TMP_DEST[127:0] ← SRC2[127:0]
        01:  TMP_DEST[255:128] ← SRC2[127:0]
        10:  TMP_DEST[383:256] ← SRC2[127:0]
        11:  TMP_DEST[511:384] ← SRC2[127:0]
    ESAC.
FI;
FOR j ← 0 TO KL-1
    i ← j * 32
    IF k1[j] OR *no writemask*
        THEN DEST[i+31:i] ← TMP_DEST[i+31:i]
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE                                ; zeroing-masking
                    DEST[i+31:i] ← 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0
```

VINSERTF64x2 (EVEX encoded versions)

```
(KL, VL) = (4, 256), (8, 512)
TEMP_DEST[VL-1:0] ← SRC1[VL-1:0]
IF VL = 256
    CASE (imm8[0]) OF
        0:    TMP_DEST[127:0] ← SRC2[127:0]
        1:    TMP_DEST[255:128] ← SRC2[127:0]
    ESAC.
FI;
IF VL = 512
    CASE (imm8[1:0]) OF
        00:  TMP_DEST[127:0] ← SRC2[127:0]
        01:  TMP_DEST[255:128] ← SRC2[127:0]
        10:  TMP_DEST[383:256] ← SRC2[127:0]
        11:  TMP_DEST[511:384] ← SRC2[127:0]
    ESAC.
FI;
FOR j ← 0 TO KL-1
    i ← j * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+63:i] ← TMP_DEST[i+63:i]
        ELSE
```

```
                IF *merging-masking*                    ; merging-masking
                    THEN *DEST[i+63:i] remains unchanged*
                    ELSE                                ; zeroing-masking
                        DEST[i+63:i] ← 0
                FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0
```

VINSERTF32x8 (EVEX.U1.512 encoded version)

```
TEMP_DEST[VL-1:0] ← SRC1[VL-1:0]
CASE (imm8[0]) OF
    0: TMP_DEST[255:0] ← SRC2[255:0]
    1: TMP_DEST[511:256] ← SRC2[255:0]
ESAC.

FOR j ← 0 TO 15
    i ← j * 32
    IF k1[j] OR *no writemask*
        THEN DEST[i+31:i] ← TMP_DEST[i+31:i]
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE                                ; zeroing-masking
                    DEST[i+31:i] ← 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0
```

VINSERTF64x4 (EVEX.512 encoded version)

```
VL = 512
TEMP_DEST[VL-1:0] ← SRC1[VL-1:0]
CASE (imm8[0]) OF
    0: TMP_DEST[255:0] ← SRC2[255:0]
    1: TMP_DEST[511:256] ← SRC2[255:0]
ESAC.

FOR j ← 0 TO 7
    i ← j * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+63:i] ← TMP_DEST[i+63:i]
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE                                ; zeroing-masking
                    DEST[i+63:i] ← 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0
```

VINSERTF128 (VEX encoded version)

```
TEMP[255:0] ←SRC1[255:0]
CASE (imm8[0]) OF
    0: TEMP[127:0] ←SRC2[127:0]
    1: TEMP[255:128] ←SRC2[127:0]
ESAC
DEST ←TEMP
```

Intel C/C++ Compiler Intrinsic Equivalent

VINSERTF32x4 __m512 _mm512_insertf32x4(__m512 a, __m128 b, int imm);
VINSERTF32x4 __m512 _mm512_mask_insertf32x4(__m512 s, __mmask16 k, __m512 a, __m128 b, int imm);
VINSERTF32x4 __m512 _mm512_maskz_insertf32x4(__mmask16 k, __m512 a, __m128 b, int imm);
VINSERTF32x4 __m256 _mm256_insertf32x4(__m256 a, __m128 b, int imm);
VINSERTF32x4 __m256 _mm256_mask_insertf32x4(__m256 s, __mmask8 k, __m256 a, __m128 b, int imm);
VINSERTF32x4 __m256 _mm256_maskz_insertf32x4(__mmask8 k, __m256 a, __m128 b, int imm);
VINSERTF32x8 __m512 _mm512_insertf32x8(__m512 a, __m256 b, int imm);
VINSERTF32x8 __m512 _mm512_mask_insertf32x8(__m512 s, __mmask16 k, __m512 a, __m256 b, int imm);
VINSERTF32x8 __m512 _mm512_maskz_insertf32x8(__mmask16 k, __m512 a, __m256 b, int imm);
VINSERTF64x2 __m512d _mm512_insertf64x2(__m512d a, __m128d b, int imm);
VINSERTF64x2 __m512d _mm512_mask_insertf64x2(__m512d s, __mmask8 k, __m512d a, __m128d b, int imm);
VINSERTF64x2 __m512d _mm512_maskz_insertf64x2(__mmask8 k, __m512d a, __m128d b, int imm);
VINSERTF64x2 __m256d _mm256_insertf64x2(__m256d a, __m128d b, int imm);
VINSERTF64x2 __m256d _mm256_mask_insertf64x2(__m256d s, __mmask8 k, __m256d a, __m128d b, int imm);
VINSERTF64x2 __m256d _mm256_maskz_insertf64x2(__mmask8 k, __m256d a, __m128d b, int imm);
VINSERTF64x4 __m512d _mm512_insertf64x4(__m512d a, __m256d b, int imm);
VINSERTF64x4 __m512d _mm512_mask_insertf64x4(__m512d s, __mmask8 k, __m512d a, __m256d b, int imm);
VINSERTF64x4 __m512d _mm512_maskz_insertf64x4(__mmask8 k, __m512d a, __m256d b, int imm);
VINSERTF128 __m256 _mm256_insertf128_ps (__m256 a, __m128 b, int offset);
VINSERTF128 __m256d _mm256_insertf128_pd (__m256d a, __m128d b, int offset);
VINSERTF128 __m256i _mm256_insertf128_si256 (__m256i a, __m128i b, int offset);

SIMD Floating-Point Exceptions

None

Other Exceptions

VEX-encoded instruction, see Exceptions Type 6; additionally

#UD	If VEX.L = 0.

EVEX-encoded instruction, see Exceptions Type E6NF.

VINSERTI128/VINSERTI32x4/VINSERTI64x2/VINSERTI32x8/VINSERTI64x4—Insert Packed Integer Values

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
VEX.NDS.256.66.0F3A.W0 38 /r ib VINSERTI128 ymm1, ymm2, xmm3/m128, imm8	A	V/V	AVX2	Insert 128 bits of integer data from xmm3/m128 and the remaining values from ymm2 into ymm1.
EVEX.NDS.256.66.0F3A.W0 38 /r ib VINSERTI32X4 ymm1 {k1}{z}, ymm2, xmm3/m128, imm8	C	V/V	AVX512VL AVX512F	Insert 128 bits of packed doubleword integer values from xmm3/m128 and the remaining values from ymm2 into ymm1 under writemask k1.
EVEX.NDS.512.66.0F3A.W0 38 /r ib VINSERTI32X4 zmm1 {k1}{z}, zmm2, xmm3/m128, imm8	C	V/V	AVX512F	Insert 128 bits of packed doubleword integer values from xmm3/m128 and the remaining values from zmm2 into zmm1 under writemask k1.
EVEX.NDS.256.66.0F3A.W1 38 /r ib VINSERTI64X2 ymm1 {k1}{z}, ymm2, xmm3/m128, imm8	B	V/V	AVX512VL AVX512DQ	Insert 128 bits of packed quadword integer values from xmm3/m128 and the remaining values from ymm2 into ymm1 under writemask k1.
EVEX.NDS.512.66.0F3A.W1 38 /r ib VINSERTI64X2 zmm1 {k1}{z}, zmm2, xmm3/m128, imm8	B	V/V	AVX512DQ	Insert 128 bits of packed quadword integer values from xmm3/m128 and the remaining values from zmm2 into zmm1 under writemask k1.
EVEX.NDS.512.66.0F3A.W0 3A /r ib VINSERTI32X8 zmm1 {k1}{z}, zmm2, ymm3/m256, imm8	D	V/V	AVX512DQ	Insert 256 bits of packed doubleword integer values from ymm3/m256 and the remaining values from zmm2 into zmm1 under writemask k1.
EVEX.NDS.512.66.0F3A.W1 3A /r ib VINSERTI64X4 zmm1 {k1}{z}, zmm2, ymm3/m256, imm8	C	V/V	AVX512F	Insert 256 bits of packed quadword integer values from ymm3/m256 and the remaining values from zmm2 into zmm1 under writemask k1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	NA	ModRM:reg (w)	VEX.vvvv	ModRM:r/m (r)	Imm8
B	Tuple2	ModRM:reg (w)	EVEX.vvvv	ModRM:r/m (r)	Imm8
C	Tuple4	ModRM:reg (w)	EVEX.vvvv	ModRM:r/m (r)	Imm8
D	Tuple8	ModRM:reg (w)	EVEX.vvvv	ModRM:r/m (r)	Imm8

Description

VINSERTI32x4 and VINSERTI64x2 inserts 128-bits of packed integer values from the second source operand (the third operand) into the destination operand (the first operand) at an 128-bit granular offset multiplied by imm8[0] (256-bit) or imm8[1:0]. The remaining portions of the destination are copied from the corresponding fields of the first source operand (the second operand). The second source operand can be either an XMM register or a 128-bit memory location. The high 6/7bits of the immediate are ignored. The destination operand is a ZMM/YMM register and updated at 32 and 64-bit granularity according to the writemask.

VINSERTI32x8 and VINSERTI64x4 inserts 256-bits of packed integer values from the second source operand (the third operand) into the destination operand (the first operand) at a 256-bit granular offset multiplied by imm8[0]. The remaining portions of the destination are copied from the corresponding fields of the first source operand (the second operand). The second source operand can be either an YMM register or a 256-bit memory location. The upper bits of the immediate are ignored. The destination operand is a ZMM register and updated at 32 and 64-bit granularity according to the writemask.

VINSERTI128 inserts 128-bits of packed integer data from the second source operand (the third operand) into the destination operand (the first operand) at a 128-bit granular offset multiplied by imm8[0]. The remaining portions of the destination are copied from the corresponding fields of the first source operand (the second operand). The second source operand can be either an XMM register or a 128-bit memory location. The high 7 bits of the immediate are ignored. VEX.L must be 1, otherwise attempt to execute this instruction with VEX.L=0 will cause #UD.

Operation

VINSERTI32x4 (EVEX encoded versions)

```
(KL, VL) = (8, 256), (16, 512)
TEMP_DEST[VL-1:0] ← SRC1[VL-1:0]
IF VL = 256
    CASE (imm8[0]) OF
        0:    TMP_DEST[127:0] ← SRC2[127:0]
        1:    TMP_DEST[255:128] ← SRC2[127:0]
    ESAC.
FI;
IF VL = 512
    CASE (imm8[1:0]) OF
        00:  TMP_DEST[127:0] ← SRC2[127:0]
        01:  TMP_DEST[255:128] ← SRC2[127:0]
        10:  TMP_DEST[383:256] ← SRC2[127:0]
        11:  TMP_DEST[511:384] ← SRC2[127:0]
    ESAC.
FI;
FOR j ← 0 TO KL-1
    i ← j * 32
    IF k1[j] OR *no writemask*
        THEN DEST[i+31:i] ← TMP_DEST[i+31:i]
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE                                ; zeroing-masking
                    DEST[i+31:i] ← 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0
```

VINSERTI64x2 (EVEX encoded versions)

```
(KL, VL) = (4, 256), (8, 512)
TEMP_DEST[VL-1:0] ← SRC1[VL-1:0]
IF VL = 256
    CASE (imm8[0]) OF
        0:    TMP_DEST[127:0] ← SRC2[127:0]
        1:    TMP_DEST[255:128] ← SRC2[127:0]
    ESAC.
FI;
IF VL = 512
    CASE (imm8[1:0]) OF
        00:  TMP_DEST[127:0] ← SRC2[127:0]
        01:  TMP_DEST[255:128] ← SRC2[127:0]
        10:  TMP_DEST[383:256] ← SRC2[127:0]
        11:  TMP_DEST[511:384] ← SRC2[127:0]
    ESAC.
FI;
FOR j ← 0 TO KL-1
    i ← j * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+63:i] ← TMP_DEST[i+63:i]
        ELSE
```

```
                IF *merging-masking*                    ; merging-masking
                    THEN *DEST[i+63:i] remains unchanged*
                    ELSE                                ; zeroing-masking
                        DEST[i+63:i] ← 0
                FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0
```

VINSERTI32x8 (EVEX.U1.512 encoded version)

```
TEMP_DEST[VL-1:0] ← SRC1[VL-1:0]
CASE (imm8[0]) OF
    0: TMP_DEST[255:0] ← SRC2[255:0]
    1: TMP_DEST[511:256] ← SRC2[255:0]
ESAC.

FOR j ← 0 TO 15
    i ← j * 32
    IF k1[j] OR *no writemask*
        THEN DEST[i+31:i] ← TMP_DEST[i+31:i]
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE                                ; zeroing-masking
                    DEST[i+31:i] ← 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0
```

VINSERTI64x4 (EVEX.512 encoded version)

```
VL = 512
TEMP_DEST[VL-1:0] ← SRC1[VL-1:0]
CASE (imm8[0]) OF
    0: TMP_DEST[255:0] ← SRC2[255:0]
    1: TMP_DEST[511:256] ← SRC2[255:0]
ESAC.

FOR j ← 0 TO 7
    i ← j * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+63:i] ← TMP_DEST[i+63:i]
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE                                ; zeroing-masking
                    DEST[i+63:i] ← 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0
```

VINSERTI128

```
TEMP[255:0] ←SRC1[255:0]
CASE (imm8[0]) OF
    0: TEMP[127:0] ←SRC2[127:0]
    1: TEMP[255:128] ←SRC2[127:0]
ESAC
DEST ←TEMP
```

Intel C/C++ Compiler Intrinsic Equivalent

VINSERTI32x4 _mm512i _inserti32x4(__m512i a, __m128i b, int imm);
VINSERTI32x4 _mm512i _mask_inserti32x4(__m512i s, __mmask16 k, __m512i a, __m128i b, int imm);
VINSERTI32x4 _mm512i _maskz_inserti32x4(__mmask16 k, __m512i a, __m128i b, int imm);
VINSERTI32x4 __m256i _mm256_inserti32x4(__m256i a, __m128i b, int imm);
VINSERTI32x4 __m256i _mm256_mask_inserti32x4(__m256i s, __mmask8 k, __m256i a, __m128i b, int imm);
VINSERTI32x4 __m256i _mm256_maskz_inserti32x4(__mmask8 k, __m256i a, __m128i b, int imm);
VINSERTI32x8 __m512i _mm512_inserti32x8(__m512i a, __m256i b, int imm);
VINSERTI32x8 __m512i _mm512_mask_inserti32x8(__m512i s, __mmask16 k, __m512i a, __m256i b, int imm);
VINSERTI32x8 __m512i _mm512_maskz_inserti32x8(__mmask16 k, __m512i a, __m256i b, int imm);
VINSERTI64x2 __m512i _mm512_inserti64x2(__m512i a, __m128i b, int imm);
VINSERTI64x2 __m512i _mm512_mask_inserti64x2(__m512i s, __mmask8 k, __m512i a, __m128i b, int imm);
VINSERTI64x2 __m512i _mm512_maskz_inserti64x2(__mmask8 k, __m512i a, __m128i b, int imm);
VINSERTI64x2 __m256i _mm256_inserti64x2(__m256i a, __m128i b, int imm);
VINSERTI64x2 __m256i _mm256_mask_inserti64x2(__m256i s, __mmask8 k, __m256i a, __m128i b, int imm);
VINSERTI64x2 __m256i _mm256_maskz_inserti64x2(__mmask8 k, __m256i a, __m128i b, int imm);
VINSERTI64x4 _mm512_inserti64x4(__m512i a, __m256i b, int imm);
VINSERTI64x4 _mm512_mask_inserti64x4(__m512i s, __mmask8 k, __m512i a, __m256i b, int imm);
VINSERTI64x4 _mm512_maskz_inserti64x4(__mmask m, __m512i a, __m256i b, int imm);
VINSERTI128 __m256i _mm256_insertf128_si256 (__m256i a, __m128i b, int offset);

SIMD Floating-Point Exceptions

None

Other Exceptions

VEX-encoded instruction, see Exceptions Type 6; additionally

#UD	If VEX.L = 0.

EVEX-encoded instruction, see Exceptions Type E6NF.

VMASKMOV—Conditional SIMD Packed Loads and Stores

Opcode/ Instruction	Op/ En	64/32-bit Mode	CPUID Feature Flag	Description
VEX.NDS.128.66.0F38.W0 2C /r VMASKMOVPS *xmm1, xmm2, m128*	RVM	V/V	AVX	Conditionally load packed single-precision values from *m128* using mask in *xmm2* and store in *xmm1*.
VEX.NDS.256.66.0F38.W0 2C /r VMASKMOVPS *ymm1, ymm2, m256*	RVM	V/V	AVX	Conditionally load packed single-precision values from *m256* using mask in *ymm2* and store in *ymm1*.
VEX.NDS.128.66.0F38.W0 2D /r VMASKMOVPD *xmm1, xmm2, m128*	RVM	V/V	AVX	Conditionally load packed double-precision values from *m128* using mask in *xmm2* and store in *xmm1*.
VEX.NDS.256.66.0F38.W0 2D /r VMASKMOVPD *ymm1, ymm2, m256*	RVM	V/V	AVX	Conditionally load packed double-precision values from *m256* using mask in *ymm2* and store in *ymm1*.
VEX.NDS.128.66.0F38.W0 2E /r VMASKMOVPS *m128, xmm1, xmm2*	MVR	V/V	AVX	Conditionally store packed single-precision values from *xmm2* using mask in *xmm1*.
VEX.NDS.256.66.0F38.W0 2E /r VMASKMOVPS *m256, ymm1, ymm2*	MVR	V/V	AVX	Conditionally store packed single-precision values from *ymm2* using mask in *ymm1*.
VEX.NDS.128.66.0F38.W0 2F /r VMASKMOVPD *m128, xmm1, xmm2*	MVR	V/V	AVX	Conditionally store packed double-precision values from *xmm2* using mask in *xmm1*.
VEX.NDS.256.66.0F38.W0 2F /r VMASKMOVPD *m256, ymm1, ymm2*	MVR	V/V	AVX	Conditionally store packed double-precision values from *ymm2* using mask in *ymm1*.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RVM	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA
MVR	ModRM:r/m (w)	VEX.vvvv (r)	ModRM:reg (r)	NA

Description

Conditionally moves packed data elements from the second source operand into the corresponding data element of the destination operand, depending on the mask bits associated with each data element. The mask bits are specified in the first source operand.

The mask bit for each data element is the most significant bit of that element in the first source operand. If a mask is 1, the corresponding data element is copied from the second source operand to the destination operand. If the mask is 0, the corresponding data element is set to zero in the load form of these instructions, and unmodified in the store form.

The second source operand is a memory address for the load form of these instruction. The destination operand is a memory address for the store form of these instructions. The other operands are both XMM registers (for VEX.128 version) or YMM registers (for VEX.256 version).

Faults occur only due to mask-bit required memory accesses that caused the faults. Faults will not occur due to referencing any memory location if the corresponding mask bit for that memory location is 0. For example, no faults will be detected if the mask bits are all zero.

Unlike previous MASKMOV instructions (MASKMOVQ and MASKMOVDQU), a nontemporal hint is not applied to these instructions.

Instruction behavior on alignment check reporting with mask bits of less than all 1s are the same as with mask bits of all 1s.

VMASKMOV should not be used to access memory mapped I/O and un-cached memory as the access and the ordering of the individual loads or stores it does is implementation specific.

In cases where mask bits indicate data should not be loaded or stored paging A and D bits will be set in an implementation dependent way. However, A and D bits are always set for pages where data is actually loaded/stored.

Note: for load forms, the first source (the mask) is encoded in VEX.vvvv; the second source is encoded in rm_field, and the destination register is encoded in reg_field.

Note: for store forms, the first source (the mask) is encoded in VEX.vvvv; the second source register is encoded in reg_field, and the destination memory location is encoded in rm_field.

Operation

VMASKMOVPS -128-bit load

```
DEST[31:0] ← IF (SRC1[31]) Load_32(mem) ELSE 0
DEST[63:32] ← IF (SRC1[63]) Load_32(mem + 4) ELSE 0
DEST[95:64] ← IF (SRC1[95]) Load_32(mem + 8) ELSE 0
DEST[127:97] ← IF (SRC1[127]) Load_32(mem + 12) ELSE 0
DEST[MAXVL-1:128] ← 0
```

VMASKMOVPS - 256-bit load

```
DEST[31:0] ← IF (SRC1[31]) Load_32(mem) ELSE 0
DEST[63:32] ← IF (SRC1[63]) Load_32(mem + 4) ELSE 0
DEST[95:64] ← IF (SRC1[95]) Load_32(mem + 8) ELSE 0
DEST[127:96] ← IF (SRC1[127]) Load_32(mem + 12) ELSE 0
DEST[159:128] ← IF (SRC1[159]) Load_32(mem + 16) ELSE 0
DEST[191:160] ← IF (SRC1[191]) Load_32(mem + 20) ELSE 0
DEST[223:192] ← IF (SRC1[223]) Load_32(mem + 24) ELSE 0
DEST[255:224] ← IF (SRC1[255]) Load_32(mem + 28) ELSE 0
```

VMASKMOVPD - 128-bit load

```
DEST[63:0] ← IF (SRC1[63]) Load_64(mem) ELSE 0
DEST[127:64] ← IF (SRC1[127]) Load_64(mem + 16) ELSE 0
DEST[MAXVL-1:128] ← 0
```

VMASKMOVPD - 256-bit load

```
DEST[63:0] ← IF (SRC1[63]) Load_64(mem) ELSE 0
DEST[127:64] ← IF (SRC1[127]) Load_64(mem + 8) ELSE 0
DEST[195:128] ← IF (SRC1[191]) Load_64(mem + 16) ELSE 0
DEST[255:196] ← IF (SRC1[255]) Load_64(mem + 24) ELSE 0
```

VMASKMOVPS - 128-bit store

```
IF (SRC1[31]) DEST[31:0] ← SRC2[31:0]
IF (SRC1[63]) DEST[63:32] ← SRC2[63:32]
IF (SRC1[95]) DEST[95:64] ← SRC2[95:64]
IF (SRC1[127]) DEST[127:96] ← SRC2[127:96]
```

VMASKMOVPS - 256-bit store

```
IF (SRC1[31]) DEST[31:0] ← SRC2[31:0]
IF (SRC1[63]) DEST[63:32] ← SRC2[63:32]
IF (SRC1[95]) DEST[95:64] ← SRC2[95:64]
IF (SRC1[127]) DEST[127:96] ← SRC2[127:96]
IF (SRC1[159]) DEST[159:128] ←SRC2[159:128]
IF (SRC1[191]) DEST[191:160] ← SRC2[191:160]
IF (SRC1[223]) DEST[223:192] ← SRC2[223:192]
IF (SRC1[255]) DEST[255:224] ← SRC2[255:224]
```

VMASKMOVPD - 128-bit store
IF (SRC1[63]) DEST[63:0] ← SRC2[63:0]
IF (SRC1[127]) DEST[127:64] ←SRC2[127:64]

VMASKMOVPD - 256-bit store
IF (SRC1[63]) DEST[63:0] ← SRC2[63:0]
IF (SRC1[127]) DEST[127:64] ←SRC2[127:64]
IF (SRC1[191]) DEST[191:128] ← SRC2[191:128]
IF (SRC1[255]) DEST[255:192] ← SRC2[255:192]

Intel C/C++ Compiler Intrinsic Equivalent

__m256 _mm256_maskload_ps(float const *a, __m256i mask)

void _mm256_maskstore_ps(float *a, __m256i mask, __m256 b)

__m256d _mm256_maskload_pd(double *a, __m256i mask);

void _mm256_maskstore_pd(double *a, __m256i mask, __m256d b);

__m128 _mm128_maskload_ps(float const *a, __m128i mask)

void _mm128_maskstore_ps(float *a, __m128i mask, __m128 b)

__m128d _mm128_maskload_pd(double *a, __m128i mask);

void _mm128_maskstore_pd(double *a, __m128i mask, __m128d b);

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type 6 (No AC# reported for any mask bit combinations); additionally

#UD	If VEX.W = 1.

VPBLENDD — Blend Packed Dwords

Opcode/ Instruction	Op/ En	64/32 -bit Mode	CPUID Feature Flag	Description
VEX.NDS.128.66.0F3A.W0 02 /r ib VPBLENDD *xmm1, xmm2, xmm3/m128, imm8*	RVMI	V/V	AVX2	Select dwords from *xmm2* and *xmm3/m128* from mask specified in *imm8* and store the values into *xmm1*.
VEX.NDS.256.66.0F3A.W0 02 /r ib VPBLENDD *ymm1, ymm2, ymm3/m256, imm8*	RVMI	V/V	AVX2	Select dwords from *ymm2* and *ymm3/m256* from mask specified in *imm8* and store the values into *ymm1*.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RVMI	ModRM:reg (w)	VEX.vvvv	ModRM:r/m (r)	Imm8

Description

Dword elements from the source operand (second operand) are conditionally written to the destination operand (first operand) depending on bits in the immediate operand (third operand). The immediate bits (bits 7:0) form a mask that determines whether the corresponding word in the destination is copied from the source. If a bit in the mask, corresponding to a word, is "1", then the word is copied, else the word is unchanged.

VEX.128 encoded version: The second source operand can be an XMM register or a 128-bit memory location. The first source and destination operands are XMM registers. Bits (MAXVL-1:128) of the corresponding YMM register are zeroed.

VEX.256 encoded version: The first source operand is a YMM register. The second source operand is a YMM register or a 256-bit memory location. The destination operand is a YMM register.

Operation

VPBLENDD (VEX.256 encoded version)

```
IF (imm8[0] == 1) THEN DEST[31:0] ← SRC2[31:0]
ELSE DEST[31:0] ← SRC1[31:0]
IF (imm8[1] == 1) THEN DEST[63:32] ← SRC2[63:32]
ELSE DEST[63:32] ← SRC1[63:32]
IF (imm8[2] == 1) THEN DEST[95:64] ← SRC2[95:64]
ELSE DEST[95:64] ← SRC1[95:64]
IF (imm8[3] == 1) THEN DEST[127:96] ← SRC2[127:96]
ELSE DEST[127:96] ← SRC1[127:96]
IF (imm8[4] == 1) THEN DEST[159:128] ← SRC2[159:128]
ELSE DEST[159:128] ← SRC1[159:128]
IF (imm8[5] == 1) THEN DEST[191:160] ← SRC2[191:160]
ELSE DEST[191:160] ← SRC1[191:160]
IF (imm8[6] == 1) THEN DEST[223:192] ← SRC2[223:192]
ELSE DEST[223:192] ← SRC1[223:192]
IF (imm8[7] == 1) THEN DEST[255:224] ← SRC2[255:224]
ELSE DEST[255:224] ← SRC1[255:224]
```

VPBLENDD (VEX.128 encoded version)

```
IF (imm8[0] == 1) THEN DEST[31:0] ← SRC2[31:0]
ELSE DEST[31:0] ← SRC1[31:0]
IF (imm8[1] == 1) THEN DEST[63:32] ← SRC2[63:32]
ELSE DEST[63:32] ← SRC1[63:32]
IF (imm8[2] == 1) THEN DEST[95:64] ← SRC2[95:64]
ELSE DEST[95:64] ← SRC1[95:64]
IF (imm8[3] == 1) THEN DEST[127:96] ← SRC2[127:96]
ELSE DEST[127:96] ← SRC1[127:96]
DEST[MAXVL-1:128] ← 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VPBLENDD: __m128i _mm_blend_epi32 (__m128i v1, __m128i v2, const int mask)

VPBLENDD: __m256i _mm256_blend_epi32 (__m256i v1, __m256i v2, const int mask)

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type 4; additionally

#UD	If VEX.W = 1.

VPBLENDMB/VPBLENDMW—Blend Byte/Word Vectors Using an Opmask Control

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.NDS.128.66.0F38.W0 66 /r VPBLENDMB xmm1 {k1}{z}, xmm2, xmm3/m128	A	V/V	AVX512VL AVX512BW	Blend byte integer vector xmm2 and byte vector xmm3/m128 and store the result in xmm1, under control mask.
EVEX.NDS.256.66.0F38.W0 66 /r VPBLENDMB ymm1 {k1}{z}, ymm2, ymm3/m256	A	V/V	AVX512VL AVX512BW	Blend byte integer vector ymm2 and byte vector ymm3/m256 and store the result in ymm1, under control mask.
EVEX.NDS.512.66.0F38.W0 66 /r VPBLENDMB zmm1 {k1}{z}, zmm2, zmm3/m512	A	V/V	AVX512BW	Blend byte integer vector zmm2 and byte vector zmm3/m512 and store the result in zmm1, under control mask.
EVEX.NDS.128.66.0F38.W1 66 /r VPBLENDMW xmm1 {k1}{z}, xmm2, xmm3/m128	A	V/V	AVX512VL AVX512BW	Blend word integer vector xmm2 and word vector xmm3/m128 and store the result in xmm1, under control mask.
EVEX.NDS.256.66.0F38.W1 66 /r VPBLENDMW ymm1 {k1}{z}, ymm2, ymm3/m256	A	V/V	AVX512VL AVX512BW	Blend word integer vector ymm2 and word vector ymm3/m256 and store the result in ymm1, under control mask.
EVEX.NDS.512.66.0F38.W1 66 /r VPBLENDMW zmm1 {k1}{z}, zmm2, zmm3/m512	A	V/V	AVX512BW	Blend word integer vector zmm2 and word vector zmm3/m512 and store the result in zmm1, under control mask.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Full Vector Mem	ModRM:reg (w)	EVEX.vvvv	ModRM:r/m (r)	NA

Description

Performs an element-by-element blending of byte/word elements between the first source operand byte vector register and the second source operand byte vector from memory or register, using the instruction mask as selector. The result is written into the destination byte vector register.

The destination and first source operands are ZMM/YMM/XMM registers. The second source operand can be a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit memory location.

The mask is not used as a writemask for this instruction. Instead, the mask is used as an element selector: every element of the destination is conditionally selected between first source or second source using the value of the related mask bit (0 for first source, 1 for second source).

Operation

VPBLENDMB (EVEX encoded versions)
(KL, VL) = (16, 128), (32, 256), (64, 512)

```
FOR j ← 0 TO KL-1
    i ← j * 8
    IF k1[j] OR *no writemask*
        THEN DEST[i+7:i] ← SRC2[i+7:i]
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN DEST[i+7:i] ← SRC1[i+7:i]
                ELSE                            ; zeroing-masking
                    DEST[i+7:i] ← 0
            FI;
    FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0;
```

VPBLENDMW (EVEX encoded versions)
(KL, VL) = (8, 128), (16, 256), (32, 512)

```
FOR j ← 0 TO KL-1
    i ← j * 16
    IF k1[j] OR *no writemask*
        THEN DEST[i+15:i] ← SRC2[i+15:i]
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN DEST[i+15:i] ← SRC1[i+15:i]
                ELSE                            ; zeroing-masking
                    DEST[i+15:i] ← 0
            FI;
    FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VPBLENDMB __m512i _mm512_mask_blend_epi8(__mmask64 m, __m512i a, __m512i b);
VPBLENDMB __m256i _mm256_mask_blend_epi8(__mmask32 m, __m256i a, __m256i b);
VPBLENDMB __m128i _mm_mask_blend_epi8(__mmask16 m, __m128i a, __m128i b);
VPBLENDMW __m512i _mm512_mask_blend_epi16(__mmask32 m, __m512i a, __m512i b);
VPBLENDMW __m256i _mm256_mask_blend_epi16(__mmask16 m, __m256i a, __m256i b);
VPBLENDMW __m128i _mm_mask_blend_epi16(__mmask8 m, __m128i a, __m128i b);

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type E4.

VPBLENDMD/VPBLENDMQ—Blend Int32/Int64 Vectors Using an OpMask Control

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.NDS.128.66.0F38.W0 64 /r VPBLENDMD xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst	A	V/V	AVX512VL AVX512F	Blend doubleword integer vector xmm2 and doubleword vector xmm3/m128/m32bcst and store the result in xmm1, under control mask.
EVEX.NDS.256.66.0F38.W0 64 /r VPBLENDMD ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst	A	V/V	AVX512VL AVX512F	Blend doubleword integer vector ymm2 and doubleword vector ymm3/m256/m32bcst and store the result in ymm1, under control mask.
EVEX.NDS.512.66.0F38.W0 64 /r VPBLENDMD zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst	A	V/V	AVX512F	Blend doubleword integer vector zmm2 and doubleword vector zmm3/m512/m32bcst and store the result in zmm1, under control mask.
EVEX.NDS.128.66.0F38.W1 64 /r VPBLENDMQ xmm1 {k1}{z}, xmm2, xmm3/m128/m64bcst	A	V/V	AVX512VL AVX512F	Blend quadword integer vector xmm2 and quadword vector xmm3/m128/m64bcst and store the result in xmm1, under control mask.
EVEX.NDS.256.66.0F38.W1 64 /r VPBLENDMQ ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst	A	V/V	AVX512VL AVX512F	Blend quadword integer vector ymm2 and quadword vector ymm3/m256/m64bcst and store the result in ymm1, under control mask.
EVEX.NDS.512.66.0F38.W1 64 /r VPBLENDMQ zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst	A	V/V	AVX512F	Blend quadword integer vector zmm2 and quadword vector zmm3/m512/m64bcst and store the result in zmm1, under control mask.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Full Vector	ModRM:reg (w)	EVEX.vvvv	ModRM:r/m (r)	NA

Description

Performs an element-by-element blending of dword/qword elements between the first source operand (the second operand) and the elements of the second source operand (the third operand) using an opmask register as select control. The blended result is written into the destination.

The destination and first source operands are ZMM registers. The second source operand can be a ZMM register, a 512-bit memory location or a 512-bit vector broadcasted from a 32-bit memory location.

The opmask register is not used as a writemask for this instruction. Instead, the mask is used as an element selector: every element of the destination is conditionally selected between first source or second source using the value of the related mask bit (0 for the first source operand, 1 for the second source operand).

If EVEX.z is set, the elements with corresponding mask bit value of 0 in the destination operand are zeroed.

Operation

VPBLENDMD (EVEX encoded versions)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j ← 0 TO KL-1
    i ← j * 32
    IF k1[j] OR *no controlmask*
        THEN
            IF (EVEX.b = 1) AND (SRC2 *is memory*)
                THEN
                    DEST[i+31:i] ← SRC2[31:0]
                ELSE
                    DEST[i+31:i] ← SRC2[i+31:i]
            FI;
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN DEST[i+31:i] ← SRC1[i+31:i]
                ELSE                                ; zeroing-masking
                    DEST[i+31:i] ← 0
            FI;
    FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0;
```

VPBLENDMD (EVEX encoded versions)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j ← 0 TO KL-1
    i ← j * 32
    IF k1[j] OR *no controlmask*
        THEN
            IF (EVEX.b = 1) AND (SRC2 *is memory*)
                THEN
                    DEST[i+31:i] ← SRC2[31:0]
                ELSE
                    DEST[i+31:i] ← SRC2[i+31:i]
            FI;
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN DEST[i+31:i] ← SRC1[i+31:i]
                ELSE                                ; zeroing-masking
                    DEST[i+31:i] ← 0
            FI;
    FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VPBLENDMD __m512i _mm512_mask_blend_epi32(__mmask16 k, __m512i a, __m512i b);
VPBLENDMD __m256i _mm256_mask_blend_epi32(__mmask8 m, __m256i a, __m256i b);
VPBLENDMD __m128i _mm_mask_blend_epi32(__mmask8 m, __m128i a, __m128i b);
VPBLENDMQ __m512i _mm512_mask_blend_epi64(__mmask8 k, __m512i a, __m512i b);
VPBLENDMQ __m256i _mm256_mask_blend_epi64(__mmask8 m, __m256i a, __m256i b);
VPBLENDMQ __m128i _mm_mask_blend_epi64(__mmask8 m, __m128i a, __m128i b);

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type E4.

VPBROADCASTB/W/D/Q—Load with Broadcast Integer Data from General Purpose Register

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.128.66.0F38.W0 7A /r VPBROADCASTB xmm1 {k1}{z}, reg	A	V/V	AVX512VL AVX512BW	Broadcast an 8-bit value from a GPR to all bytes in the 128-bit destination subject to writemask k1.
EVEX.256.66.0F38.W0 7A /r VPBROADCASTB ymm1 {k1}{z}, reg	A	V/V	AVX512VL AVX512BW	Broadcast an 8-bit value from a GPR to all bytes in the 256-bit destination subject to writemask k1.
EVEX.512.66.0F38.W0 7A /r VPBROADCASTB zmm1 {k1}{z}, reg	A	V/V	AVX512BW	Broadcast an 8-bit value from a GPR to all bytes in the 512-bit destination subject to writemask k1.
EVEX.128.66.0F38.W0 7B /r VPBROADCASTW xmm1 {k1}{z}, reg	A	V/V	AVX512VL AVX512BW	Broadcast a 16-bit value from a GPR to all words in the 128-bit destination subject to writemask k1.
EVEX.256.66.0F38.W0 7B /r VPBROADCASTW ymm1 {k1}{z}, reg	A	V/V	AVX512VL AVX512BW	Broadcast a 16-bit value from a GPR to all words in the 256-bit destination subject to writemask k1.
EVEX.512.66.0F38.W0 7B /r VPBROADCASTW zmm1 {k1}{z}, reg	A	V/V	AVX512BW	Broadcast a 16-bit value from a GPR to all words in the 512-bit destination subject to writemask k1.
EVEX.128.66.0F38.W0 7C /r VPBROADCASTD xmm1 {k1}{z}, r32	A	V/V	AVX512VL AVX512F	Broadcast a 32-bit value from a GPR to all double-words in the 128-bit destination subject to writemask k1.
EVEX.256.66.0F38.W0 7C /r VPBROADCASTD ymm1 {k1}{z}, r32	A	V/V	AVX512VL AVX512F	Broadcast a 32-bit value from a GPR to all double-words in the 256-bit destination subject to writemask k1.
EVEX.512.66.0F38.W0 7C /r VPBROADCASTD zmm1 {k1}{z}, r32	A	V/V	AVX512F	Broadcast a 32-bit value from a GPR to all double-words in the 512-bit destination subject to writemask k1.
EVEX.128.66.0F38.W1 7C /r VPBROADCASTQ xmm1 {k1}{z}, r64	A	V/N.E.[1]	AVX512VL AVX512F	Broadcast a 64-bit value from a GPR to all quad-words in the 128-bit destination subject to writemask k1.
EVEX.256.66.0F38.W1 7C /r VPBROADCASTQ ymm1 {k1}{z}, r64	A	V/N.E.[1]	AVX512VL AVX512F	Broadcast a 64-bit value from a GPR to all quad-words in the 256-bit destination subject to writemask k1.
EVEX.512.66.0F38.W1 7C /r VPBROADCASTQ zmm1 {k1}{z}, r64	A	V/N.E.[1]	AVX512F	Broadcast a 64-bit value from a GPR to all quad-words in the 512-bit destination subject to writemask k1.

NOTES:

1. EVEX.W in non-64 bit is ignored; the instructions behaves as if the W0 version is used.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Tuple1 Scalar	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

Description

Broadcasts a 8-bit, 16-bit, 32-bit or 64-bit value from a general-purpose register (the second operand) to all the locations in the destination vector register (the first operand) using the writemask k1.

EVEX.vvvv is reserved and must be 1111b otherwise instructions will #UD.

Operation

VPBROADCASTB (EVEX encoded versions)

```
(KL, VL) = (16, 128), (32, 256), (64, 512)
FOR j ← 0 TO KL-1
    i ←j * 8
    IF k1[j] OR *no writemask*
        THEN DEST[i+7:i] ← SRC[7:0]
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+7:i] remains unchanged*
                ELSE                            ; zeroing-masking
                    DEST[i+7:i] ← 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0
```

VPBROADCASTW (EVEX encoded versions)

```
(KL, VL) = (8, 128), (16, 256), (32, 512)
FOR j ← 0 TO KL-1
    i ←j * 16
    IF k1[j] OR *no writemask*
        THEN DEST[i+15:i] ← SRC[15:0]
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+15:i] remains unchanged*
                ELSE                            ; zeroing-masking
                    DEST[i+15:i] ← 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0
```

VPBROADCASTD (EVEX encoded versions)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j ← 0 TO KL-1
    i ←j * 32
    IF k1[j] OR *no writemask*
        THEN DEST[i+31:i] ← SRC[31:0]
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE                            ; zeroing-masking
                    DEST[i+31:i] ← 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0
```

VPBROADCASTQ (EVEX encoded versions)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j ← 0 TO KL-1
    i ←j * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+63:i] ← SRC[63:0]
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE                            ; zeroing-masking
                    DEST[i+63:i] ← 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VPBROADCASTB __m512i _mm512_mask_set1_epi8(__m512i s, __mmask64 k, int a);
VPBROADCASTB __m512i _mm512_maskz_set1_epi8(__mmask64 k, int a);
VPBROADCASTB __m256i _mm256_mask_set1_epi8(__m256i s, __mmask32 k, int a);
VPBROADCASTB __m256i _mm256_maskz_set1_epi8(__mmask32 k, int a);
VPBROADCASTB __m128i _mm_mask_set1_epi8(__m128i s, __mmask16 k, int a);
VPBROADCASTB __m128i _mm_maskz_set1_epi8(__mmask16 k, int a);
VPBROADCASTD __m512i _mm512_mask_set1_epi32(__m512i s, __mmask16 k, int a);
VPBROADCASTD __m512i _mm512_maskz_set1_epi32(__mmask16 k, int a);
VPBROADCASTD __m256i _mm256_mask_set1_epi32(__m256i s, __mmask8 k, int a);
VPBROADCASTD __m256i _mm256_maskz_set1_epi32(__mmask8 k, int a);
VPBROADCASTD __m128i _mm_mask_set1_epi32(__m128i s, __mmask8 k, int a);
VPBROADCASTD __m128i _mm_maskz_set1_epi32(__mmask8 k, int a);
VPBROADCASTQ __m512i _mm512_mask_set1_epi64(__m512i s, __mmask8 k, __int64 a);
VPBROADCASTQ __m512i _mm512_maskz_set1_epi64(__mmask8 k, __int64 a);
VPBROADCASTQ __m256i _mm256_mask_set1_epi64(__m256i s, __mmask8 k, __int64 a);
VPBROADCASTQ __m256i _mm256_maskz_set1_epi64(__mmask8 k, __int64 a);
VPBROADCASTQ __m128i _mm_mask_set1_epi64(__m128i s, __mmask8 k, __int64 a);
VPBROADCASTQ __m128i _mm_maskz_set1_epi64(__mmask8 k, __int64 a);
VPBROADCASTW __m512i _mm512_mask_set1_epi16(__m512i s, __mmask32 k, int a);
VPBROADCASTW __m512i _mm512_maskz_set1_epi16(__mmask32 k, int a);
VPBROADCASTW __m256i _mm256_mask_set1_epi16(__m256i s, __mmask16 k, int a);
VPBROADCASTW __m256i _mm256_maskz_set1_epi16(__mmask16 k, int a);
VPBROADCASTW __m128i _mm_mask_set1_epi16(__m128i s, __mmask8 k, int a);
VPBROADCASTW __m128i _mm_maskz_set1_epi16(__mmask8 k, int a);

Exceptions

EVEX-encoded instructions, see Exceptions Type E7NM.

#UD	If EVEX.vvvv != 1111B.

VPBROADCAST—Load Integer and Broadcast

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
VEX.128.66.0F38.W0 78 /r VPBROADCASTB xmm1, xmm2/m8	A	V/V	AVX2	Broadcast a byte integer in the source operand to sixteen locations in xmm1.
VEX.256.66.0F38.W0 78 /r VPBROADCASTB ymm1, xmm2/m8	A	V/V	AVX2	Broadcast a byte integer in the source operand to thirty-two locations in ymm1.
EVEX.128.66.0F38.W0 78 /r VPBROADCASTB xmm1{k1}{z}, xmm2/m8	B	V/V	AVX512VL AVX512BW	Broadcast a byte integer in the source operand to locations in xmm1 subject to writemask k1.
EVEX.256.66.0F38.W0 78 /r VPBROADCASTB ymm1{k1}{z}, xmm2/m8	B	V/V	AVX512VL AVX512BW	Broadcast a byte integer in the source operand to locations in ymm1 subject to writemask k1.
EVEX.512.66.0F38.W0 78 /r VPBROADCASTB zmm1{k1}{z}, xmm2/m8	B	V/V	AVX512BW	Broadcast a byte integer in the source operand to 64 locations in zmm1 subject to writemask k1.
VEX.128.66.0F38.W0 79 /r VPBROADCASTW xmm1, xmm2/m16	A	V/V	AVX2	Broadcast a word integer in the source operand to eight locations in xmm1.
VEX.256.66.0F38.W0 79 /r VPBROADCASTW ymm1, xmm2/m16	A	V/V	AVX2	Broadcast a word integer in the source operand to sixteen locations in ymm1.
EVEX.128.66.0F38.W0 79 /r VPBROADCASTW xmm1{k1}{z}, xmm2/m16	B	V/V	AVX512VL AVX512BW	Broadcast a word integer in the source operand to locations in xmm1 subject to writemask k1.
EVEX.256.66.0F38.W0 79 /r VPBROADCASTW ymm1{k1}{z}, xmm2/m16	B	V/V	AVX512VL AVX512BW	Broadcast a word integer in the source operand to locations in ymm1 subject to writemask k1.
EVEX.512.66.0F38.W0 79 /r VPBROADCASTW zmm1{k1}{z}, xmm2/m16	B	V/V	AVX512BW	Broadcast a word integer in the source operand to 32 locations in zmm1 subject to writemask k1.
VEX.128.66.0F38.W0 58 /r VPBROADCASTD xmm1, xmm2/m32	A	V/V	AVX2	Broadcast a dword integer in the source operand to four locations in xmm1.
VEX.256.66.0F38.W0 58 /r VPBROADCASTD ymm1, xmm2/m32	A	V/V	AVX2	Broadcast a dword integer in the source operand to eight locations in ymm1.
EVEX.128.66.0F38.W0 58 /r VPBROADCASTD xmm1 {k1}{z}, xmm2/m32	B	V/V	AVX512VL AVX512F	Broadcast a dword integer in the source operand to locations in xmm1 subject to writemask k1.
EVEX.256.66.0F38.W0 58 /r VPBROADCASTD ymm1 {k1}{z}, xmm2/m32	B	V/V	AVX512VL AVX512F	Broadcast a dword integer in the source operand to locations in ymm1 subject to writemask k1.
EVEX.512.66.0F38.W0 58 /r VPBROADCASTD zmm1 {k1}{z}, xmm2/m32	B	V/V	AVX512F	Broadcast a dword integer in the source operand to locations in zmm1 subject to writemask k1.
VEX.128.66.0F38.W0 59 /r VPBROADCASTQ xmm1, xmm2/m64	A	V/V	AVX2	Broadcast a qword element in source operand to two locations in xmm1.
VEX.256.66.0F38.W0 59 /r VPBROADCASTQ ymm1, xmm2/m64	A	V/V	AVX2	Broadcast a qword element in source operand to four locations in ymm1.
EVEX.128.66.0F38.W1 59 /r VPBROADCASTQ xmm1 {k1}{z}, xmm2/m64	B	V/V	AVX512VL AVX512F	Broadcast a qword element in source operand to locations in xmm1 subject to writemask k1.
EVEX.256.66.0F38.W1 59 /r VPBROADCASTQ ymm1 {k1}{z}, xmm2/m64	B	V/V	AVX512VL AVX512F	Broadcast a qword element in source operand to locations in ymm1 subject to writemask k1.
EVEX.512.66.0F38.W1 59 /r VPBROADCASTQ zmm1 {k1}{z}, xmm2/m64	B	V/V	AVX512F	Broadcast a qword element in source operand to locations in zmm1 subject to writemask k1.
EVEX.128.66.0F38.W0 59 /r VBROADCASTI32x2 xmm1 {k1}{z}, xmm2/m64	C	V/V	AVX512VL AVX512DQ	Broadcast two dword elements in source operand to locations in xmm1 subject to writemask k1.

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.256.66.0F38.W0 59 /r VBROADCASTI32x2 ymm1 {k1}{z}, xmm2/m64	C	V/V	AVX512VL AVX512DQ	Broadcast two dword elements in source operand to locations in ymm1 subject to writemask k1.
EVEX.512.66.0F38.W0 59 /r VBROADCASTI32x2 zmm1 {k1}{z}, xmm2/m64	C	V/V	AVX512DQ	Broadcast two dword elements in source operand to locations in zmm1 subject to writemask k1.
VEX.256.66.0F38.W0 5A /r VBROADCASTI128 ymm1, m128	A	V/V	AVX2	Broadcast 128 bits of integer data in mem to low and high 128-bits in ymm1.
EVEX.256.66.0F38.W0 5A /r VBROADCASTI32X4 ymm1 {k1}{z}, m128	D	V/V	AVX512VL AVX512F	Broadcast 128 bits of 4 doubleword integer data in mem to locations in ymm1 using writemask k1.
EVEX.512.66.0F38.W0 5A /r VBROADCASTI32X4 zmm1 {k1}{z}, m128	D	V/V	AVX512F	Broadcast 128 bits of 4 doubleword integer data in mem to locations in zmm1 using writemask k1.
EVEX.256.66.0F38.W1 5A /r VBROADCASTI64X2 ymm1 {k1}{z}, m128	C	V/V	AVX512VL AVX512DQ	Broadcast 128 bits of 2 quadword integer data in mem to locations in ymm1 using writemask k1.
EVEX.512.66.0F38.W1 5A /r VBROADCASTI64X2 zmm1 {k1}{z}, m128	C	V/V	AVX512DQ	Broadcast 128 bits of 2 quadword integer data in mem to locations in zmm1 using writemask k1.
EVEX.512.66.0F38.W0 5B /r VBROADCASTI32X8 zmm1 {k1}{z}, m256	E	V/V	AVX512DQ	Broadcast 256 bits of 8 doubleword integer data in mem to locations in zmm1 using writemask k1.
EVEX.512.66.0F38.W1 5B /r VBROADCASTI64X4 zmm1 {k1}{z}, m256	D	V/V	AVX512F	Broadcast 256 bits of 4 quadword integer data in mem to locations in zmm1 using writemask k1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	NA	ModRM:reg (w)	ModRM:r/m (r)	NA	NA
B	Tuple1 Scalar	ModRM:reg (w)	ModRM:r/m (r)	NA	NA
C	Tuple2	ModRM:reg (w)	ModRM:r/m (r)	NA	NA
D	Tuple4	ModRM:reg (w)	ModRM:r/m (r)	NA	NA
E	Tuple8	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

Description

Load integer data from the source operand (the second operand) and broadcast to all elements of the destination operand (the first operand).

VEX256-encoded VPBROADCASTB/W/D/Q: The source operand is 8-bit, 16-bit, 32-bit, 64-bit memory location or the low 8-bit, 16-bit 32-bit, 64-bit data in an XMM register. The destination operand is a YMM register. VPBROADCASTI128 support the source operand of 128-bit memory location. Register source encodings for VPBROADCASTI128 is reserved and will #UD. Bits (MAXVL-1:256) of the destination register are zeroed.

EVEX-encoded VPBROADCASTD/Q: The source operand is a 32-bit, 64-bit memory location or the low 32-bit, 64-bit data in an XMM register. The destination operand is a ZMM/YMM/XMM register and updated according to the writemask k1.

VPBROADCASTI32X4 and VPBROADCASTI64X4: The destination operand is a ZMM register and updated according to the writemask k1. The source operand is 128-bit or 256-bit memory location. Register source encodings for VBROADCASTI32X4 and VBROADCASTI64X4 are reserved and will #UD.

Note: VEX.vvvv and EVEX.vvvv are reserved and must be 1111b otherwise instructions will #UD.

If VPBROADCASTI128 is encoded with VEX.L= 0, an attempt to execute the instruction encoded with VEX.L= 0 will cause an #UD exception.

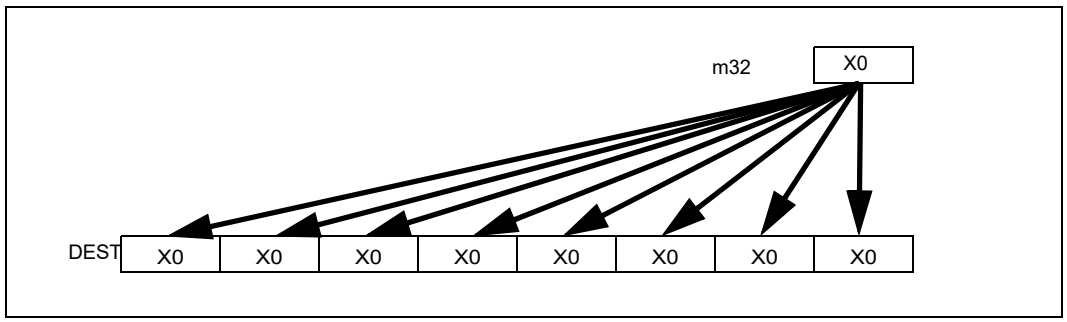


Figure 5-16. VPBROADCASTD Operation (VEX.256 encoded version)

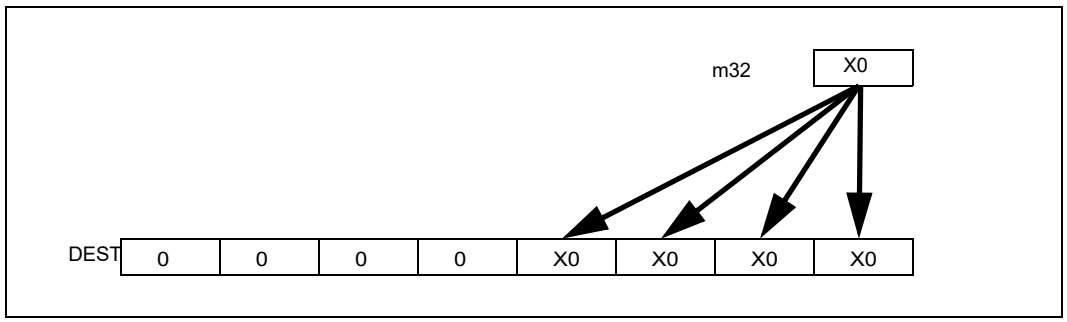


Figure 5-17. VPBROADCASTD Operation (128-bit version)

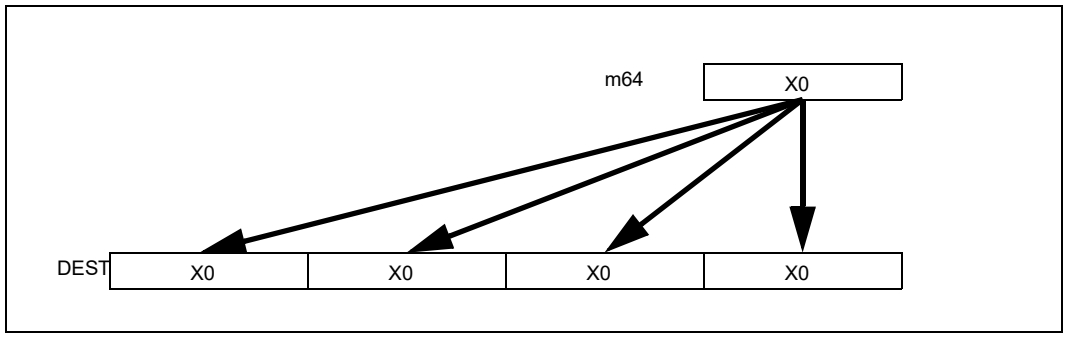


Figure 5-18. VPBROADCASTQ Operation (256-bit version)

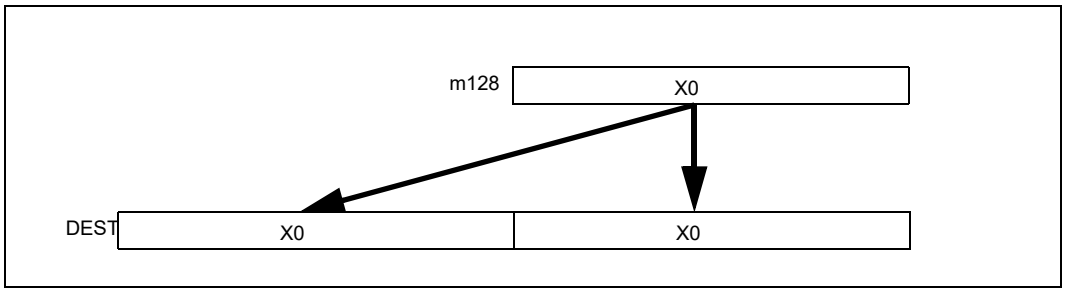


Figure 5-19. VBROADCASTI128 Operation (256-bit version)

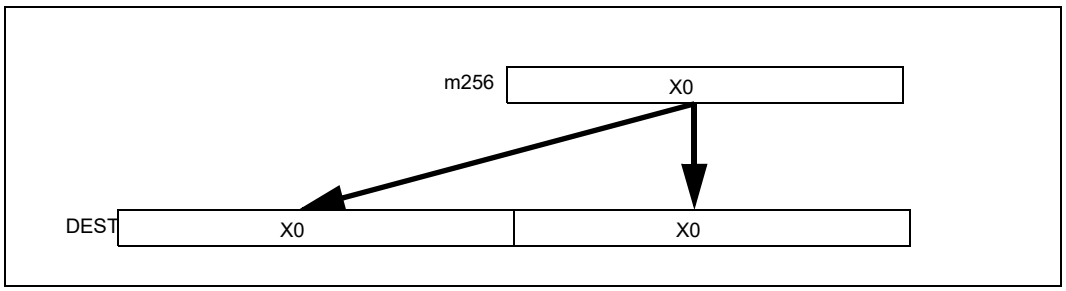


Figure 5-20. VBROADCASTI256 Operation (512-bit version)

Operation

VPBROADCASTB (EVEX encoded versions)

```
(KL, VL) = (16, 128), (32, 256), (64, 512)
FOR j ← 0 TO KL-1
    i ←j * 8
    IF k1[j] OR *no writemask*
        THEN DEST[i+7:i] ← SRC[7:0]
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+7:i] remains unchanged*
                ELSE                            ; zeroing-masking
                    DEST[i+7:i] ← 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0
```

VPBROADCASTW (EVEX encoded versions)

```
(KL, VL) = (8, 128), (16, 256), (32, 512)
FOR j ← 0 TO KL-1
    i ←j * 16
    IF k1[j] OR *no writemask*
        THEN DEST[i+15:i] ← SRC[15:0]
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+15:i] remains unchanged*
                ELSE                            ; zeroing-masking
                    DEST[i+15:i] ← 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0
```

VPBROADCASTD (128 bit version)

```
temp ← SRC[31:0]
DEST[31:0] ← temp
DEST[63:32] ← temp
DEST[95:64] ← temp
DEST[127:96] ← temp
DEST[MAXVL-1:128] ← 0
```

VPBROADCASTD (VEX.256 encoded version)

```
temp ← SRC[31:0]
DEST[31:0] ← temp
DEST[63:32] ← temp
DEST[95:64] ← temp
DEST[127:96] ← temp
DEST[159:128] ← temp
DEST[191:160] ← temp
DEST[223:192] ← temp
DEST[255:224] ← temp
DEST[MAXVL-1:256] ← 0
```

VPBROADCASTD (EVEX encoded versions)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j ← 0 TO KL-1
    i ←j * 32
    IF k1[j] OR *no writemask*
        THEN DEST[i+31:i] ← SRC[31:0]
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE                            ; zeroing-masking
                    DEST[i+31:i] ← 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0
```

VPBROADCASTQ (VEX.256 encoded version)

```
temp ← SRC[63:0]
DEST[63:0] ← temp
DEST[127:64] ← temp
DEST[191:128] ← temp
DEST[255:192] ← temp
DEST[MAXVL-1:256] ← 0
```

VPBROADCASTQ (EVEX encoded versions)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j ← 0 TO KL-1
    i ←j * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+63:i] ← SRC[63:0]
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE                            ; zeroing-masking
                    DEST[i+63:i] ← 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0
```

VBROADCASTI32x2 (EVEX encoded versions)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j ← 0 TO KL-1
    i ←   j * 32
    n ←   (j mod 2) * 32
    IF k1[j] OR *no writemask*
        THEN DEST[i+31:i] ← SRC[n+31:n]
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE                            ; zeroing-masking
                    DEST[i+31:i] ← 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0
```

VBROADCASTI128 (VEX.256 encoded version)

```
temp ← SRC[127:0]
DEST[127:0] ← temp
DEST[255:128] ← temp
DEST[MAXVL-1:256] ← 0
```

VBROADCASTI32X4 (EVEX encoded versions)

```
(KL, VL) = (8, 256), (16, 512)
FOR j ← 0 TO KL-1
    i ←   j* 32
    n ←   (j modulo 4) * 32
    IF k1[j] OR *no writemask*
        THEN DEST[i+31:i] ← SRC[n+31:n]
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE                            ; zeroing-masking
                    DEST[i+31:i] ← 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0
```

VBROADCASTI64X2 (EVEX encoded versions)

```
(KL, VL) = (8, 256), (16, 512)
FOR j ← 0 TO KL-1
    i ← j * 64
    n ←(j modulo 2) * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+63:i] ← SRC[n+63:n]
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE                            ; zeroing-masking
                    DEST[i+63:i] = 0
            FI
    FI;
ENDFOR;
```

VBROADCASTI32X8 (EVEX.U1.512 encoded version)

```
FOR j ← 0 TO 15
    i ← j * 32
    n ←(j modulo 8) * 32
    IF k1[j] OR *no writemask*
        THEN DEST[i+31:i] ← SRC[n+31:n]
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE                            ; zeroing-masking
                    DEST[i+31:i] ← 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0
```

VBROADCASTI64X4 (EVEX.512 encoded version)

```
FOR j ← 0 TO 7
    i ← j * 64
    n ←   (j modulo 4) * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+63:i] ← SRC[n+63:n]
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE                                ; zeroing-masking
                    DEST[i+63:i] ← 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VPBROADCASTB __m512i _mm512_broadcastb_epi8(__m128i a);
VPBROADCASTB __m512i _mm512_mask_broadcastb_epi8(__m512i s, __mmask64 k, __m128i a);
VPBROADCASTB __m512i _mm512_maskz_broadcastb_epi8(__mmask64 k, __m128i a);
VPBROADCASTB __m256i _mm256_broadcastb_epi8(__m128i a);
VPBROADCASTB __m256i _mm256_mask_broadcastb_epi8(__m256i s, __mmask32 k, __m128i a);
VPBROADCASTB __m256i _mm256_maskz_broadcastb_epi8(__mmask32 k, __m128i a);
VPBROADCASTB __m128i _mm_mask_broadcastb_epi8(__m128i s, __mmask16 k, __m128i a);
VPBROADCASTB __m128i _mm_maskz_broadcastb_epi8(__mmask16 k, __m128i a);
VPBROADCASTB __m128i _mm_broadcastb_epi8(__m128i a);
VPBROADCASTD __m512i _mm512_broadcastd_epi32(__m128i a);
VPBROADCASTD __m512i _mm512_mask_broadcastd_epi32(__m512i s, __mmask16 k, __m128i a);
VPBROADCASTD __m512i _mm512_maskz_broadcastd_epi32(__mmask16 k, __m128i a);
VPBROADCASTD __m256i _mm256_broadcastd_epi32(__m128i a);
VPBROADCASTD __m256i _mm256_mask_broadcastd_epi32(__m256i s, __mmask8 k, __m128i a);
VPBROADCASTD __m256i _mm256_maskz_broadcastd_epi32(__mmask8 k, __m128i a);
VPBROADCASTD __m128i _mm_broadcastd_epi32(__m128i a);
VPBROADCASTD __m128i _mm_mask_broadcastd_epi32(__m128i s, __mmask8 k, __m128i a);
VPBROADCASTD __m128i _mm_maskz_broadcastd_epi32(__mmask8 k, __m128i a);
VPBROADCASTQ __m512i _mm512_broadcastq_epi64(__m128i a);
VPBROADCASTQ __m512i _mm512_mask_broadcastq_epi64(__m512i s, __mmask8 k, __m128i a);
VPBROADCASTQ __m512i _mm512_maskz_broadcastq_epi64(__mmask8 k, __m128i a);
VPBROADCASTQ __m256i _mm256_broadcastq_epi64(__m128i a);
VPBROADCASTQ __m256i _mm256_mask_broadcastq_epi64(__m256i s, __mmask8 k, __m128i a);
VPBROADCASTQ __m256i _mm256_maskz_broadcastq_epi64(__mmask8 k, __m128i a);
VPBROADCASTQ __m128i _mm_broadcastq_epi64(__m128i a);
VPBROADCASTQ __m128i _mm_mask_broadcastq_epi64(__m128i s, __mmask8 k, __m128i a);
VPBROADCASTQ __m128i _mm_maskz_broadcastq_epi64(__mmask8 k, __m128i a);
VPBROADCASTW __m512i _mm512_broadcastw_epi16(__m128i a);
VPBROADCASTW __m512i _mm512_mask_broadcastw_epi16(__m512i s, __mmask32 k, __m128i a);
VPBROADCASTW __m512i _mm512_maskz_broadcastw_epi16(__mmask32 k, __m128i a);
VPBROADCASTW __m256i _mm256_broadcastw_epi16(__m128i a);
VPBROADCASTW __m256i _mm256_mask_broadcastw_epi16(__m256i s, __mmask16 k, __m128i a);
VPBROADCASTW __m256i _mm256_maskz_broadcastw_epi16(__mmask16 k, __m128i a);
VPBROADCASTW __m128i _mm_broadcastw_epi16(__m128i a);
VPBROADCASTW __m128i _mm_mask_broadcastw_epi16(__m128i s, __mmask8 k, __m128i a);
VPBROADCASTW __m128i _mm_maskz_broadcastw_epi16(__mmask8 k, __m128i a);
VBROADCASTI32x2 __m512i _mm512_broadcast_i32x2(__m128i a);

VBROADCASTI32x2 __m512i _mm512_mask_broadcast_i32x2(__m512i s, __mmask16 k, __m128i a);
VBROADCASTI32x2 __m512i _mm512_maskz_broadcast_i32x2(__mmask16 k, __m128i a);
VBROADCASTI32x2 __m256i _mm256_broadcast_i32x2(__m128i a);
VBROADCASTI32x2 __m256i _mm256_mask_broadcast_i32x2(__m256i s, __mmask8 k, __m128i a);
VBROADCASTI32x2 __m256i _mm256_maskz_broadcast_i32x2(__mmask8 k, __m128i a);
VBROADCASTI32x2 __m128i _mm_broadcastq_i32x2(__m128i a);
VBROADCASTI32x2 __m128i _mm_mask_broadcastq_i32x2(__m128i s, __mmask8 k, __m128i a);
VBROADCASTI32x2 __m128i _mm_maskz_broadcastq_i32x2(__mmask8 k, __m128i a);
VBROADCASTI32x4 __m512i _mm512_broadcast_i32x4(__m128i a);
VBROADCASTI32x4 __m512i _mm512_mask_broadcast_i32x4(__m512i s, __mmask16 k, __m128i a);
VBROADCASTI32x4 __m512i _mm512_maskz_broadcast_i32x4(__mmask16 k, __m128i a);
VBROADCASTI32x4 __m256i _mm256_broadcast_i32x4(__m128i a);
VBROADCASTI32x4 __m256i _mm256_mask_broadcast_i32x4(__m256i s, __mmask8 k, __m128i a);
VBROADCASTI32x4 __m256i _mm256_maskz_broadcast_i32x4(__mmask8 k, __m128i a);
VBROADCASTI32x8 __m512i _mm512_broadcast_i32x8(__m256i a);
VBROADCASTI32x8 __m512i _mm512_mask_broadcast_i32x8(__m512i s, __mmask16 k, __m256i a);
VBROADCASTI32x8 __m512i _mm512_maskz_broadcast_i32x8(__mmask16 k, __m256i a);
VBROADCASTI64x2 __m512i _mm512_broadcast_i64x2(__m128i a);
VBROADCASTI64x2 __m512i _mm512_mask_broadcast_i64x2(__m512i s, __mmask8 k, __m128i a);
VBROADCASTI64x2 __m512i _mm512_maskz_broadcast_i64x2(__mmask8 k, __m128i a);
VBROADCASTI64x2 __m256i _mm256_broadcast_i64x2(__m128i a);
VBROADCASTI64x2 __m256i _mm256_mask_broadcast_i64x2(__m256i s, __mmask8 k, __m128i a);
VBROADCASTI64x2 __m256i _mm256_maskz_broadcast_i64x2(__mmask8 k, __m128i a);
VBROADCASTI64x4 __m512i _mm512_broadcast_i64x4(__m256i a);
VBROADCASTI64x4 __m512i _mm512_mask_broadcast_i64x4(__m512i s, __mmask8 k, __m256i a);
VBROADCASTI64x4 __m512i _mm512_maskz_broadcast_i64x4(__mmask8 k, __m256i a);

SIMD Floating-Point Exceptions

None

Other Exceptions

EVEX-encoded instructions, see Exceptions Type 6;

EVEX-encoded instructions, syntax with reg/mem operand, see Exceptions Type E6.

#UD	If VEX.L = 0 for VPBROADCASTQ, VPBROADCASTI128.
	If EVEX.L'L = 0 for VBROADCASTI32X4/VBROADCASTI64X2.
	If EVEX.L'L < 10b for VBROADCASTI32X8/VBROADCASTI64X4.

VPBROADCASTM—Broadcast Mask to Vector Register

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.128.F3.0F38.W1 2A /r VPBROADCASTMB2Q xmm1, k1	RM	V/V	AVX512VL AVX512CD	Broadcast low byte value in k1 to two locations in xmm1.
EVEX.256.F3.0F38.W1 2A /r VPBROADCASTMB2Q ymm1, k1	RM	V/V	AVX512VL AVX512CD	Broadcast low byte value in k1 to four locations in ymm1.
EVEX.512.F3.0F38.W1 2A /r VPBROADCASTMB2Q zmm1, k1	RM	V/V	AVX512CD	Broadcast low byte value in k1 to eight locations in zmm1.
EVEX.128.F3.0F38.W0 3A /r VPBROADCASTMW2D xmm1, k1	RM	V/V	AVX512VL AVX512CD	Broadcast low word value in k1 to four locations in xmm1.
EVEX.256.F3.0F38.W0 3A /r VPBROADCASTMW2D ymm1, k1	RM	V/V	AVX512VL AVX512CD	Broadcast low word value in k1 to eight locations in ymm1.
EVEX.512.F3.0F38.W0 3A /r VPBROADCASTMW2D zmm1, k1	RM	V/V	AVX512CD	Broadcast low word value in k1 to sixteen locations in zmm1.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

Description

Broadcasts the zero-extended 64/32 bit value of the low byte/word of the source operand (the second operand) to each 64/32 bit element of the destination operand (the first operand). The source operand is an opmask register. The destination operand is a ZMM register (EVEX.512), YMM register (EVEX.256), or XMM register (EVEX.128).

EVEX.vvvv is reserved and must be 1111b otherwise instructions will #UD.

Operation

VPBROADCASTMB2Q

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j ← 0 TO KL-1
    i ← j*64
    DEST[i+63:i] ← ZeroExtend(SRC[7:0])
ENDFOR
DEST[MAXVL-1:VL] ← 0
```

VPBROADCASTMW2D

```
(KL, VL) = (4, 128), (8, 256),(16, 512)
FOR j ← 0 TO KL-1
    i ← j*32
    DEST[i+31:i] ← ZeroExtend(SRC[15:0])
ENDFOR
DEST[MAXVL-1:VL] ← 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VPBROADCASTMB2Q __m512i _mm512_broadcastmb_epi64(__mmask8);
VPBROADCASTMW2D __m512i _mm512_broadcastmw_epi32(__mmask16);
VPBROADCASTMB2Q __m256i _mm256_broadcastmb_epi64(__mmask8);
VPBROADCASTMW2D __m256i _mm256_broadcastmw_epi32(__mmask8);
VPBROADCASTMB2Q __m128i _mm_broadcastmb_epi64(__mmask8);
VPBROADCASTMW2D __m128i _mm_broadcastmw_epi32(__mmask8);

SIMD Floating-Point Exceptions

None

Other Exceptions

EVEX-encoded instruction, see Exceptions Type E6NF.

VPCMPB/VPCMPUB—Compare Packed Byte Values Into Mask

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.NDS.128.66.0F3A.W0 3F /r ib VPCMPB k1 {k2}, xmm2, xmm3/m128, imm8	A	V/V	AVX512VL AVX512BW	Compare packed signed byte values in xmm3/m128 and xmm2 using bits 2:0 of imm8 as a comparison predicate with writemask k2 and leave the result in mask register k1.
EVEX.NDS.256.66.0F3A.W0 3F /r ib VPCMPB k1 {k2}, ymm2, ymm3/m256, imm8	A	V/V	AVX512VL AVX512BW	Compare packed signed byte values in ymm3/m256 and ymm2 using bits 2:0 of imm8 as a comparison predicate with writemask k2 and leave the result in mask register k1.
EVEX.NDS.512.66.0F3A.W0 3F /r ib VPCMPB k1 {k2}, zmm2, zmm3/m512, imm8	A	V/V	AVX512BW	Compare packed signed byte values in zmm3/m512 and zmm2 using bits 2:0 of imm8 as a comparison predicate with writemask k2 and leave the result in mask register k1.
EVEX.NDS.128.66.0F3A.W0 3E /r ib VPCMPUB k1 {k2}, xmm2, xmm3/m128, imm8	A	V/V	AVX512VL AVX512BW	Compare packed unsigned byte values in xmm3/m128 and xmm2 using bits 2:0 of imm8 as a comparison predicate with writemask k2 and leave the result in mask register k1.
EVEX.NDS.256.66.0F3A.W0 3E /r ib VPCMPUB k1 {k2}, ymm2, ymm3/m256, imm8	A	V/V	AVX512VL AVX512BW	Compare packed unsigned byte values in ymm3/m256 and ymm2 using bits 2:0 of imm8 as a comparison predicate with writemask k2 and leave the result in mask register k1.
EVEX.NDS.512.66.0F3A.W0 3E /r ib VPCMPUB k1 {k2}, zmm2, zmm3/m512, imm8	A	V/V	AVX512BW	Compare packed unsigned byte values in zmm3/m512 and zmm2 using bits 2:0 of imm8 as a comparison predicate with writemask k2 and leave the result in mask register k1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Full Vector Mem	ModRM:reg (w)	vvvv (r)	ModRM:r/m (r)	NA

Description

Performs a SIMD compare of the packed byte values in the second source operand and the first source operand and returns the results of the comparison to the mask destination operand. The comparison predicate operand (immediate byte) specifies the type of comparison performed on each pair of packed values in the two source operands. The result of each comparison is a single mask bit result of 1 (comparison true) or 0 (comparison false).

VPCMPB performs a comparison between pairs of signed byte values.

VPCMPUB performs a comparison between pairs of unsigned byte values.

The first source operand (second operand) is a ZMM/YMM/XMM register. The second source operand can be a ZMM/YMM/XMM register or a 512/256/128-bit memory location. The destination operand (first operand) is a mask register k1. Up to 64/32/16 comparisons are performed with results written to the destination operand under the writemask k2.

The comparison predicate operand is an 8-bit immediate: bits 2:0 define the type of comparison to be performed. Bits 3 through 7 of the immediate are reserved. Compiler can implement the pseudo-op mnemonic listed in Table 5-10.

Table 5-10. Pseudo-Op and VPCMP* Implementation

Pseudo-Op	PCMPM Implementation
VPCMPEQ* *reg1, reg2, reg3*	VPCMP* *reg1, reg2, reg3, 0*
VPCMPLT* *reg1, reg2, reg3*	VPCMP**reg1, reg2, reg3, 1*
VPCMPLE* *reg1, reg2, reg3*	VPCMP* *reg1, reg2, reg3, 2*
VPCMPNEQ* *reg1, reg2, reg3*	VPCMP* *reg1, reg2, reg3, 4*
VPPCMPNLT* *reg1, reg2, reg3*	VPCMP* *reg1, reg2, reg3, 5*
VPCMPNLE* *reg1, reg2, reg3*	VPCMP* *reg1, reg2, reg3, 6*

Operation

```
CASE (COMPARISON PREDICATE) OF
    0: OP ← EQ;
    1: OP ← LT;
    2: OP ← LE;
    3: OP ← FALSE;
    4: OP ← NEQ;
    5: OP ← NLT;
    6: OP ← NLE;
    7: OP ← TRUE;
ESAC;
```

VPCMPB (EVEX encoded versions)

```
(KL, VL) = (16, 128), (32, 256), (64, 512)
FOR j ← 0 TO KL-1
    i ← j * 8
    IF k2[j] OR *no writemask*
        THEN
            CMP ← SRC1[i+7:i] OP SRC2[i+7:i];
            IF CMP = TRUE
                THEN DEST[j] ← 1;
                ELSE DEST[j] ← 0; FI;
        ELSE    DEST[j] = 0                     ; zeroing-masking onlyFI;
    FI;
ENDFOR
DEST[MAX_KL-1:KL] ← 0
```

VPCMPUB (EVEX encoded versions)

```
(KL, VL) = (16, 128), (32, 256), (64, 512)
FOR j ← 0 TO KL-1
    i ← j * 8
    IF k2[j] OR *no writemask*
        THEN
            CMP ← SRC1[i+7:i] OP SRC2[i+7:i];
            IF CMP = TRUE
                THEN DEST[j] ← 1;
                ELSE DEST[j] ← 0; FI;
        ELSE    DEST[j] = 0                     ; zeroing-masking onlyFI;
    FI;
ENDFOR
DEST[MAX_KL-1:KL] ← 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VPCMPB __mmask64 _mm512_cmp_epi8_mask(__m512i a, __m512i b, int cmp);
VPCMPB __mmask64 _mm512_mask_cmp_epi8_mask(__mmask64 m, __m512i a, __m512i b, int cmp);
VPCMPB __mmask32 _mm256_cmp_epi8_mask(__m256i a, __m256i b, int cmp);
VPCMPB __mmask32 _mm256_mask_cmp_epi8_mask(__mmask32 m, __m256i a, __m256i b, int cmp);
VPCMPB __mmask16 _mm_cmp_epi8_mask(__m128i a, __m128i b, int cmp);
VPCMPB __mmask16 _mm_mask_cmp_epi8_mask(__mmask16 m, __m128i a, __m128i b, int cmp);
VPCMPB __mmask64 _mm512_cmp[eq|ge|gt|le|lt|neq]_epi8_mask(__m512i a, __m512i b);
VPCMPB __mmask64 _mm512_mask_cmp[eq|ge|gt|le|lt|neq]_epi8_mask(__mmask64 m, __m512i a, __m512i b);
VPCMPB __mmask32 _mm256_cmp[eq|ge|gt|le|lt|neq]_epi8_mask(__m256i a, __m256i b);
VPCMPB __mmask32 _mm256_mask_cmp[eq|ge|gt|le|lt|neq]_epi8_mask(__mmask32 m, __m256i a, __m256i b);
VPCMPB __mmask16 _mm_cmp[eq|ge|gt|le|lt|neq]_epi8_mask(__m128i a, __m128i b);
VPCMPB __mmask16 _mm_mask_cmp[eq|ge|gt|le|lt|neq]_epi8_mask(__mmask16 m, __m128i a, __m128i b);
VPCMPUB __mmask64 _mm512_cmp_epu8_mask(__m512i a, __m512i b, int cmp);
VPCMPUB __mmask64 _mm512_mask_cmp_epu8_mask(__mmask64 m, __m512i a, __m512i b, int cmp);
VPCMPUB __mmask32 _mm256_cmp_epu8_mask(__m256i a, __m256i b, int cmp);
VPCMPUB __mmask32 _mm256_mask_cmp_epu8_mask(__mmask32 m, __m256i a, __m256i b, int cmp);
VPCMPUB __mmask16 _mm_cmp_epu8_mask(__m128i a, __m128i b, int cmp);
VPCMPUB __mmask16 _mm_mask_cmp_epu8_mask(__mmask16 m, __m128i a, __m128i b, int cmp);
VPCMPUB __mmask64 _mm512_cmp[eq|ge|gt|le|lt|neq]_epu8_mask(__m512i a, __m512i b, int cmp);
VPCMPUB __mmask64 _mm512_mask_cmp[eq|ge|gt|le|lt|neq]_epu8_mask(__mmask64 m, __m512i a, __m512i b, int cmp);
VPCMPUB __mmask32 _mm256_cmp[eq|ge|gt|le|lt|neq]_epu8_mask(__m256i a, __m256i b, int cmp);
VPCMPUB __mmask32 _mm256_mask_cmp[eq|ge|gt|le|lt|neq]_epu8_mask(__mmask32 m, __m256i a, __m256i b, int cmp);
VPCMPUB __mmask16 _mm_cmp[eq|ge|gt|le|lt|neq]_epu8_mask(__m128i a, __m128i b, int cmp);
VPCMPUB __mmask16 _mm_mask_cmp[eq|ge|gt|le|lt|neq]_epu8_mask(__mmask16 m, __m128i a, __m128i b, int cmp);

SIMD Floating-Point Exceptions

None

Other Exceptions

EVEX-encoded instruction, see Exceptions Type E4.nb.

VPCMPD/VPCMPUD—Compare Packed Integer Values into Mask

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.NDS.128.66.0F3A.W0 1F /r ib VPCMPD k1 {k2}, xmm2, xmm3/m128/m32bcst, imm8	A	V/V	AVX512VL AVX512F	Compare packed signed doubleword integer values in xmm3/m128/m32bcst and xmm2 using bits 2:0 of imm8 as a comparison predicate with writemask k2 and leave the result in mask register k1.
EVEX.NDS.256.66.0F3A.W0 1F /r ib VPCMPD k1 {k2}, ymm2, ymm3/m256/m32bcst, imm8	A	V/V	AVX512VL AVX512F	Compare packed signed doubleword integer values in ymm3/m256/m32bcst and ymm2 using bits 2:0 of imm8 as a comparison predicate with writemask k2 and leave the result in mask register k1.
EVEX.NDS.512.66.0F3A.W0 1F /r ib VPCMPD k1 {k2}, zmm2, zmm3/m512/m32bcst, imm8	A	V/V	AVX512F	Compare packed signed doubleword integer values in zmm2 and zmm3/m512/m32bcst using bits 2:0 of imm8 as a comparison predicate. The comparison results are written to the destination k1 under writemask k2.
EVEX.NDS.128.66.0F3A.W0 1E /r ib VPCMPUD k1 {k2}, xmm2, xmm3/m128/m32bcst, imm8	A	V/V	AVX512VL AVX512F	Compare packed unsigned doubleword integer values in xmm3/m128/m32bcst and xmm2 using bits 2:0 of imm8 as a comparison predicate with writemask k2 and leave the result in mask register k1.
EVEX.NDS.256.66.0F3A.W0 1E /r ib VPCMPUD k1 {k2}, ymm2, ymm3/m256/m32bcst, imm8	A	V/V	AVX512VL AVX512F	Compare packed unsigned doubleword integer values in ymm3/m256/m32bcst and ymm2 using bits 2:0 of imm8 as a comparison predicate with writemask k2 and leave the result in mask register k1.
EVEX.NDS.512.66.0F3A.W0 1E /r ib VPCMPUD k1 {k2}, zmm2, zmm3/m512/m32bcst, imm8	A	V/V	AVX512F	Compare packed unsigned doubleword integer values in zmm2 and zmm3/m512/m32bcst using bits 2:0 of imm8 as a comparison predicate. The comparison results are written to the destination k1 under writemask k2.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Full Vector	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	Imm8

Description

Performs a SIMD compare of the packed integer values in the second source operand and the first source operand and returns the results of the comparison to the mask destination operand. The comparison predicate operand (immediate byte) specifies the type of comparison performed on each pair of packed values in the two source operands. The result of each comparison is a single mask bit result of 1 (comparison true) or 0 (comparison false).

VPCMPD/VPCMPUD performs a comparison between pairs of signed/unsigned doubleword integer values.

The first source operand (second operand) is a ZMM/YMM/XMM register. The second source operand can be a ZMM/YMM/XMM register or a 512/256/128-bit memory location or a 512-bit vector broadcasted from a 32-bit memory location. The destination operand (first operand) is a mask register k1. Up to 16/8/4 comparisons are performed with results written to the destination operand under the writemask k2.

The comparison predicate operand is an 8-bit immediate: bits 2:0 define the type of comparison to be performed. Bits 3 through 7 of the immediate are reserved. Compiler can implement the pseudo-op mnemonic listed in Table 5-10.

Operation

```
CASE (COMPARISON PREDICATE) OF
    0: OP ← EQ;
    1: OP ← LT;
    2: OP ← LE;
    3: OP ← FALSE;
    4: OP ← NEQ;
    5: OP ← NLT;
    6: OP ← NLE;
    7: OP ← TRUE;
ESAC;
```

VPCMPD (EVEX encoded versions)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j ← 0 TO KL-1
    i ← j * 32
    IF k2[j] OR *no writemask*
        THEN
            IF (EVEX.b = 1) AND (SRC2 *is memory*)
                THEN CMP ← SRC1[i+31:i] OP SRC2[31:0];
                ELSE CMP ← SRC1[i+31:i] OP SRC2[i+31:i];
            FI;
            IF CMP = TRUE
                THEN DEST[j] ← 1;
                ELSE DEST[j] ← 0; FI;
        ELSE    DEST[j] ← 0                     ; zeroing-masking onlyFI;
    FI;
ENDFOR
DEST[MAX_KL-1:KL] ← 0
```

VPCMPUD (EVEX encoded versions)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j ← 0 TO KL-1
    i ← j * 32
    IF k2[j] OR *no writemask*
        THEN
            IF (EVEX.b = 1) AND (SRC2 *is memory*)
                THEN CMP ← SRC1[i+31:i] OP SRC2[31:0];
                ELSE CMP ← SRC1[i+31:i] OP SRC2[i+31:i];
            FI;
            IF CMP = TRUE
                THEN DEST[j] ← 1;
                ELSE DEST[j] ← 0; FI;
        ELSE    DEST[j] ← 0                     ; zeroing-masking onlyFI;
    FI;
ENDFOR
DEST[MAX_KL-1:KL] ← 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VPCMPD __mmask16 _mm512_cmp_epi32_mask(__m512i a, __m512i b, int imm);
VPCMPD __mmask16 _mm512_mask_cmp_epi32_mask(__mmask16 k, __m512i a, __m512i b, int imm);
VPCMPD __mmask16 _mm512_cmp[eq|ge|gt|le|lt|neq]_epi32_mask(__m512i a, __m512i b);
VPCMPD __mmask16 _mm512_mask_cmp[eq|ge|gt|le|lt|neq]_epi32_mask(__mmask16 k, __m512i a, __m512i b);
VPCMPUD __mmask16 _mm512_cmp_epu32_mask(__m512i a, __m512i b, int imm);
VPCMPUD __mmask16 _mm512_mask_cmp_epu32_mask(__mmask16 k, __m512i a, __m512i b, int imm);
VPCMPUD __mmask16 _mm512_cmp[eq|ge|gt|le|lt|neq]_epu32_mask(__m512i a, __m512i b);
VPCMPUD __mmask16 _mm512_mask_cmp[eq|ge|gt|le|lt|neq]_epu32_mask(__mmask16 k, __m512i a, __m512i b);
VPCMPD __mmask8 _mm256_cmp_epi32_mask(__m256i a, __m256i b, int imm);
VPCMPD __mmask8 _mm256_mask_cmp_epi32_mask(__mmask8 k, __m256i a, __m256i b, int imm);
VPCMPD __mmask8 _mm256_cmp[eq|ge|gt|le|lt|neq]_epi32_mask(__m256i a, __m256i b);
VPCMPD __mmask8 _mm256_mask_cmp[eq|ge|gt|le|lt|neq]_epi32_mask(__mmask8 k, __m256i a, __m256i b);
VPCMPUD __mmask8 _mm256_cmp_epu32_mask(__m256i a, __m256i b, int imm);
VPCMPUD __mmask8 _mm256_mask_cmp_epu32_mask(__mmask8 k, __m256i a, __m256i b, int imm);
VPCMPUD __mmask8 _mm256_cmp[eq|ge|gt|le|lt|neq]_epu32_mask(__m256i a, __m256i b);
VPCMPUD __mmask8 _mm256_mask_cmp[eq|ge|gt|le|lt|neq]_epu32_mask(__mmask8 k, __m256i a, __m256i b);
VPCMPD __mmask8 _mm_cmp_epi32_mask(__m128i a, __m128i b, int imm);
VPCMPD __mmask8 _mm_mask_cmp_epi32_mask(__mmask8 k, __m128i a, __m128i b, int imm);
VPCMPD __mmask8 _mm_cmp[eq|ge|gt|le|lt|neq]_epi32_mask(__m128i a, __m128i b);
VPCMPD __mmask8 _mm_mask_cmp[eq|ge|gt|le|lt|neq]_epi32_mask(__mmask8 k, __m128i a, __m128i b);
VPCMPUD __mmask8 _mm_cmp_epu32_mask(__m128i a, __m128i b, int imm);
VPCMPUD __mmask8 _mm_mask_cmp_epu32_mask(__mmask8 k, __m128i a, __m128i b, int imm);
VPCMPUD __mmask8 _mm_cmp[eq|ge|gt|le|lt|neq]_epu32_mask(__m128i a, __m128i b);
VPCMPUD __mmask8 _mm_mask_cmp[eq|ge|gt|le|lt|neq]_epu32_mask(__mmask8 k, __m128i a, __m128i b);

SIMD Floating-Point Exceptions

None

Other Exceptions

EVEX-encoded instruction, see Exceptions Type E4.

VPCMPQ/VPCMPUQ—Compare Packed Integer Values into Mask

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.NDS.128.66.0F3A.W1 1F /r ib VPCMPQ k1 {k2}, xmm2, xmm3/m128/m64bcst, imm8	A	V/V	AVX512VL AVX512F	Compare packed signed quadword integer values in xmm3/m128/m64bcst and xmm2 using bits 2:0 of imm8 as a comparison predicate with writemask k2 and leave the result in mask register k1.
EVEX.NDS.256.66.0F3A.W1 1F /r ib VPCMPQ k1 {k2}, ymm2, ymm3/m256/m64bcst, imm8	A	V/V	AVX512VL AVX512F	Compare packed signed quadword integer values in ymm3/m256/m64bcst and ymm2 using bits 2:0 of imm8 as a comparison predicate with writemask k2 and leave the result in mask register k1.
EVEX.NDS.512.66.0F3A.W1 1F /r ib VPCMPQ k1 {k2}, zmm2, zmm3/m512/m64bcst, imm8	A	V/V	AVX512F	Compare packed signed quadword integer values in zmm3/m512/m64bcst and zmm2 using bits 2:0 of imm8 as a comparison predicate with writemask k2 and leave the result in mask register k1.
EVEX.NDS.128.66.0F3A.W1 1E /r ib VPCMPUQ k1 {k2}, xmm2, xmm3/m128/m64bcst, imm8	A	V/V	AVX512VL AVX512F	Compare packed unsigned quadword integer values in xmm3/m128/m64bcst and xmm2 using bits 2:0 of imm8 as a comparison predicate with writemask k2 and leave the result in mask register k1.
EVEX.NDS.256.66.0F3A.W1 1E /r ib VPCMPUQ k1 {k2}, ymm2, ymm3/m256/m64bcst, imm8	A	V/V	AVX512VL AVX512F	Compare packed unsigned quadword integer values in ymm3/m256/m64bcst and ymm2 using bits 2:0 of imm8 as a comparison predicate with writemask k2 and leave the result in mask register k1.
EVEX.NDS.512.66.0F3A.W1 1E /r ib VPCMPUQ k1 {k2}, zmm2, zmm3/m512/m64bcst, imm8	A	V/V	AVX512F	Compare packed unsigned quadword integer values in zmm3/m512/m64bcst and zmm2 using bits 2:0 of imm8 as a comparison predicate with writemask k2 and leave the result in mask register k1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Full Vector	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	Imm8

Description

Performs a SIMD compare of the packed integer values in the second source operand and the first source operand and returns the results of the comparison to the mask destination operand. The comparison predicate operand (immediate byte) specifies the type of comparison performed on each pair of packed values in the two source operands. The result of each comparison is a single mask bit result of 1 (comparison true) or 0 (comparison false).

VPCMPQ/VPCMPUQ performs a comparison between pairs of signed/unsigned quadword integer values.

The first source operand (second operand) is a ZMM/YMM/XMM register. The second source operand can be a ZMM/YMM/XMM register or a 512/256/128-bit memory location or a 512-bit vector broadcasted from a 64-bit memory location. The destination operand (first operand) is a mask register k1. Up to 8/4/2 comparisons are performed with results written to the destination operand under the writemask k2.

The comparison predicate operand is an 8-bit immediate: bits 2:0 define the type of comparison to be performed. Bits 3 through 7 of the immediate are reserved. Compiler can implement the pseudo-op mnemonic listed in Table 5-10.

Operation

```
CASE (COMPARISON PREDICATE) OF
    0: OP ← EQ;
    1: OP ← LT;
    2: OP ← LE;
    3: OP ← FALSE;
    4: OP ← NEQ;
    5: OP ← NLT;
    6: OP ← NLE;
    7: OP ← TRUE;
ESAC;
```

VPCMPQ (EVEX encoded versions)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j ← 0 TO KL-1
    i ← j * 64
    IF k2[j] OR *no writemask*
        THEN
            IF (EVEX.b = 1) AND (SRC2 *is memory*)
                THEN CMP ← SRC1[i+63:i] OP SRC2[63:0];
                ELSE CMP ← SRC1[i+63:i] OP SRC2[i+63:i];
            FI;
            IF CMP = TRUE
                THEN DEST[j] ← 1;
                ELSE DEST[j] ← 0; FI;
        ELSE    DEST[j] ← 0             ; zeroing-masking only
    FI;
ENDFOR
DEST[MAX_KL-1:KL] ← 0
```

VPCMPUQ (EVEX encoded versions)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j ← 0 TO KL-1
    i ← j * 64
    IF k2[j] OR *no writemask*
        THEN
            IF (EVEX.b = 1) AND (SRC2 *is memory*)
                THEN CMP ← SRC1[i+63:i] OP SRC2[63:0];
                ELSE CMP ← SRC1[i+63:i] OP SRC2[i+63:i];
            FI;
            IF CMP = TRUE
                THEN DEST[j] ← 1;
                ELSE DEST[j] ← 0; FI;
        ELSE    DEST[j] ← 0             ; zeroing-masking only
    FI;
ENDFOR
DEST[MAX_KL-1:KL] ← 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VPCMPQ __mmask8 _mm512_cmp_epi64_mask(__m512i a, __m512i b, int imm);
VPCMPQ __mmask8 _mm512_mask_cmp_epi64_mask(__mmask8 k, __m512i a, __m512i b, int imm);
VPCMPQ __mmask8 _mm512_cmp[eq|ge|gt|le|lt|neq]_epi64_mask(__m512i a, __m512i b);
VPCMPQ __mmask8 _mm512_mask_cmp[eq|ge|gt|le|lt|neq]_epi64_mask(__mmask8 k, __m512i a, __m512i b);
VPCMPUQ __mmask8 _mm512_cmp_epu64_mask(__m512i a, __m512i b, int imm);
VPCMPUQ __mmask8 _mm512_mask_cmp_epu64_mask(__mmask8 k, __m512i a, __m512i b, int imm);
VPCMPUQ __mmask8 _mm512_cmp[eq|ge|gt|le|lt|neq]_epu64_mask(__m512i a, __m512i b);
VPCMPUQ __mmask8 _mm512_mask_cmp[eq|ge|gt|le|lt|neq]_epu64_mask(__mmask8 k, __m512i a, __m512i b);
VPCMPQ __mmask8 _mm256_cmp_epi64_mask(__m256i a, __m256i b, int imm);
VPCMPQ __mmask8 _mm256_mask_cmp_epi64_mask(__mmask8 k, __m256i a, __m256i b, int imm);
VPCMPQ __mmask8 _mm256_cmp[eq|ge|gt|le|lt|neq]_epi64_mask(__m256i a, __m256i b);
VPCMPQ __mmask8 _mm256_mask_cmp[eq|ge|gt|le|lt|neq]_epi64_mask(__mmask8 k, __m256i a, __m256i b);
VPCMPUQ __mmask8 _mm256_cmp_epu64_mask(__m256i a, __m256i b, int imm);
VPCMPUQ __mmask8 _mm256_mask_cmp_epu64_mask(__mmask8 k, __m256i a, __m256i b, int imm);
VPCMPUQ __mmask8 _mm256_cmp[eq|ge|gt|le|lt|neq]_epu64_mask(__m256i a, __m256i b);
VPCMPUQ __mmask8 _mm256_mask_cmp[eq|ge|gt|le|lt|neq]_epu64_mask(__mmask8 k, __m256i a, __m256i b);
VPCMPQ __mmask8 _mm_cmp_epi64_mask(__m128i a, __m128i b, int imm);
VPCMPQ __mmask8 _mm_mask_cmp_epi64_mask(__mmask8 k, __m128i a, __m128i b, int imm);
VPCMPQ __mmask8 _mm_cmp[eq|ge|gt|le|lt|neq]_epi64_mask(__m128i a, __m128i b);
VPCMPQ __mmask8 _mm_mask_cmp[eq|ge|gt|le|lt|neq]_epi64_mask(__mmask8 k, __m128i a, __m128i b);
VPCMPUQ __mmask8 _mm_cmp_epu64_mask(__m128i a, __m128i b, int imm);
VPCMPUQ __mmask8 _mm_mask_cmp_epu64_mask(__mmask8 k, __m128i a, __m128i b, int imm);
VPCMPUQ __mmask8 _mm_cmp[eq|ge|gt|le|lt|neq]_epu64_mask(__m128i a, __m128i b);
VPCMPUQ __mmask8 _mm_mask_cmp[eq|ge|gt|le|lt|neq]_epu64_mask(__mmask8 k, __m128i a, __m128i b);

SIMD Floating-Point Exceptions

None

Other Exceptions

EVEX-encoded instruction, see Exceptions Type E4.

VPCMPW/VPCMPUW—Compare Packed Word Values Into Mask

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.NDS.128.66.0F3A.W1 3F /r ib VPCMPW k1 {k2}, xmm2, xmm3/m128, imm8	A	V/V	AVX512VL AVX512BW	Compare packed signed word integers in xmm3/m128 and xmm2 using bits 2:0 of imm8 as a comparison predicate with writemask k2 and leave the result in mask register k1.
EVEX.NDS.256.66.0F3A.W1 3F /r ib VPCMPW k1 {k2}, ymm2, ymm3/m256, imm8	A	V/V	AVX512VL AVX512BW	Compare packed signed word integers in ymm3/m256 and ymm2 using bits 2:0 of imm8 as a comparison predicate with writemask k2 and leave the result in mask register k1.
EVEX.NDS.512.66.0F3A.W1 3F /r ib VPCMPW k1 {k2}, zmm2, zmm3/m512, imm8	A	V/V	AVX512BW	Compare packed signed word integers in zmm3/m512 and zmm2 using bits 2:0 of imm8 as a comparison predicate with writemask k2 and leave the result in mask register k1.
EVEX.NDS.128.66.0F3A.W1 3E /r ib VPCMPUW k1 {k2}, xmm2, xmm3/m128, imm8	A	V/V	AVX512VL AVX512BW	Compare packed unsigned word integers in xmm3/m128 and xmm2 using bits 2:0 of imm8 as a comparison predicate with writemask k2 and leave the result in mask register k1.
EVEX.NDS.256.66.0F3A.W1 3E /r ib VPCMPUW k1 {k2}, ymm2, ymm3/m256, imm8	A	V/V	AVX512VL AVX512BW	Compare packed unsigned word integers in ymm3/m256 and ymm2 using bits 2:0 of imm8 as a comparison predicate with writemask k2 and leave the result in mask register k1.
VPCMPUW k1 {k2}, zmm2, zmm3/m512, imm8	A	V/V	AVX512BW	Compare packed unsigned word integers in zmm3/m512 and zmm2 using bits 2:0 of imm8 as a comparison predicate with writemask k2 and leave the result in mask register k1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Full Vector Mem	ModRM:reg (w)	vvvv (r)	ModRM:r/m (r)	NA

Description

Performs a SIMD compare of the packed integer word in the second source operand and the first source operand and returns the results of the comparison to the mask destination operand. The comparison predicate operand (immediate byte) specifies the type of comparison performed on each pair of packed values in the two source operands. The result of each comparison is a single mask bit result of 1 (comparison true) or 0 (comparison false).

VPCMPW performs a comparison between pairs of signed word values.

VPCMPUW performs a comparison between pairs of unsigned word values.

The first source operand (second operand) is a ZMM/YMM/XMM register. The second source operand can be a ZMM/YMM/XMM register or a 512/256/128-bit memory location. The destination operand (first operand) is a mask register k1. Up to 32/16/8 comparisons are performed with results written to the destination operand under the writemask k2.

The comparison predicate operand is an 8-bit immediate: bits 2:0 define the type of comparison to be performed. Bits 3 through 7 of the immediate are reserved. Compiler can implement the pseudo-op mnemonic listed in Table 5-10.

Operation

```
CASE (COMPARISON PREDICATE) OF
    0: OP ← EQ;
    1: OP ← LT;
    2: OP ← LE;
    3: OP ← FALSE;
    4: OP ← NEQ;
    5: OP ← NLT;
    6: OP ← NLE;
    7: OP ← TRUE;
ESAC;
```

VPCMPW (EVEX encoded versions)

```
(KL, VL) = (8, 128), (16, 256), (32, 512)
FOR j ← 0 TO KL-1
    i ← j * 16
    IF k2[j] OR *no writemask*
        THEN
            ICMP ← SRC1[i+15:i] OP SRC2[i+15:i];
            IF CMP = TRUE
                THEN DEST[j] ← 1;
                ELSE DEST[j] ← 0; FI;
        ELSE    DEST[j] = 0             ; zeroing-masking only
    FI;
ENDFOR
DEST[MAX_KL-1:KL] ← 0
```

VPCMPUW (EVEX encoded versions)

```
(KL, VL) = (8, 128), (16, 256), (32, 512)
FOR j ← 0 TO KL-1
    i ← j * 16
    IF k2[j] OR *no writemask*
        THEN
            CMP ← SRC1[i+15:i] OP SRC2[i+15:i];
            IF CMP = TRUE
                THEN DEST[j] ← 1;
                ELSE DEST[j] ← 0; FI;
        ELSE    DEST[j] = 0             ; zeroing-masking only
    FI;
ENDFOR
DEST[MAX_KL-1:KL] ← 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VPCMPW __mmask32 _mm512_cmp_epi16_mask(__m512i a, __m512i b, int cmp);
VPCMPW __mmask32 _mm512_mask_cmp_epi16_mask(__mmask32 m, __m512i a, __m512i b, int cmp);
VPCMPW __mmask16 _mm256_cmp_epi16_mask(__m256i a, __m256i b, int cmp);
VPCMPW __mmask16 _mm256_mask_cmp_epi16_mask(__mmask16 m, __m256i a, __m256i b, int cmp);
VPCMPW __mmask8 _mm_cmp_epi16_mask(__m128i a, __m128i b, int cmp);
VPCMPW __mmask8 _mm_mask_cmp_epi16_mask(__mmask8 m, __m128i a, __m128i b, int cmp);
VPCMPW __mmask32 _mm512_cmp[eq|ge|gt|le|lt|neq]_epi16_mask(__m512i a, __m512i b);
VPCMPW __mmask32 _mm512_mask_cmp[eq|ge|gt|le|lt|neq]_epi16_mask(__mmask32 m, __m512i a, __m512i b);
VPCMPW __mmask16 _mm256_cmp[eq|ge|gt|le|lt|neq]_epi16_mask(__m256i a, __m256i b);
VPCMPW __mmask16 _mm256_mask_cmp[eq|ge|gt|le|lt|neq]_epi16_mask(__mmask16 m, __m256i a, __m256i b);
VPCMPW __mmask8 _mm_cmp[eq|ge|gt|le|lt|neq]_epi16_mask(__m128i a, __m128i b);
VPCMPW __mmask8 _mm_mask_cmp[eq|ge|gt|le|lt|neq]_epi16_mask(__mmask8 m, __m128i a, __m128i b);
VPCMPUW __mmask32 _mm512_cmp_epu16_mask(__m512i a, __m512i b, int cmp);
VPCMPUW __mmask32 _mm512_mask_cmp_epu16_mask(__mmask32 m, __m512i a, __m512i b, int cmp);
VPCMPUW __mmask16 _mm256_cmp_epu16_mask(__m256i a, __m256i b, int cmp);
VPCMPUW __mmask16 _mm256_mask_cmp_epu16_mask(__mmask16 m, __m256i a, __m256i b, int cmp);
VPCMPUW __mmask8 _mm_cmp_epu16_mask(__m128i a, __m128i b, int cmp);
VPCMPUW __mmask8 _mm_mask_cmp_epu16_mask(__mmask8 m, __m128i a, __m128i b, int cmp);
VPCMPUW __mmask32 _mm512_cmp[eq|ge|gt|le|lt|neq]_epu16_mask(__m512i a, __m512i b, int cmp);
VPCMPUW __mmask32 _mm512_mask_cmp[eq|ge|gt|le|lt|neq]_epu16_mask(__mmask32 m, __m512i a, __m512i b, int cmp);
VPCMPUW __mmask16 _mm256_cmp[eq|ge|gt|le|lt|neq]_epu16_mask(__m256i a, __m256i b, int cmp);
VPCMPUW __mmask16 _mm256_mask_cmp[eq|ge|gt|le|lt|neq]_epu16_mask(__mmask16 m, __m256i a, __m256i b, int cmp);
VPCMPUW __mmask8 _mm_cmp[eq|ge|gt|le|lt|neq]_epu16_mask(__m128i a, __m128i b, int cmp);
VPCMPUW __mmask8 _mm_mask_cmp[eq|ge|gt|le|lt|neq]_epu16_mask(__mmask8 m, __m128i a, __m128i b, int cmp);

SIMD Floating-Point Exceptions

None

Other Exceptions

EVEX-encoded instruction, see Exceptions Type E4.nb.

VPCOMPRESSD—Store Sparse Packed Doubleword Integer Values into Dense Memory/Register

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.128.66.0F38.W0 8B /r VPCOMPRESSD xmm1/m128 {k1}{z}, xmm2	A	V/V	AVX512VL AVX512F	Compress packed doubleword integer values from xmm2 to xmm1/m128 using controlmask k1.
EVEX.256.66.0F38.W0 8B /r VPCOMPRESSD ymm1/m256 {k1}{z}, ymm2	A	V/V	AVX512VL AVX512F	Compress packed doubleword integer values from ymm2 to ymm1/m256 using controlmask k1.
EVEX.512.66.0F38.W0 8B /r VPCOMPRESSD zmm1/m512 {k1}{z}, zmm2	A	V/V	AVX512F	Compress packed doubleword integer values from zmm2 to zmm1/m512 using controlmask k1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Tuple1 Scalar	ModRM:r/m (w)	ModRM:reg (r)	NA	NA

Description

Compress (store) up to 16/8/4 doubleword integer values from the source operand (second operand) to the destination operand (first operand). The source operand is a ZMM/YMM/XMM register, the destination operand can be a ZMM/YMM/XMM register or a 512/256/128-bit memory location.

The opmask register k1 selects the active elements (partial vector or possibly non-contiguous if less than 16 active elements) from the source operand to compress into a contiguous vector. The contiguous vector is written to the destination starting from the low element of the destination operand.

Memory destination version: Only the contiguous vector is written to the destination memory location. EVEX.z must be zero.

Register destination version: If the vector length of the contiguous vector is less than that of the input vector in the source operand, the upper bits of the destination register are unmodified if EVEX.z is not set, otherwise the upper bits are zeroed.

Note: EVEX.vvvv is reserved and must be 1111b otherwise instructions will #UD.

Note that the compressed displacement assumes a pre-scaling (N) corresponding to the size of one single element instead of the size of the full vector.

Operation

VPCOMPRESSD (EVEX encoded versions) store form

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
SIZE ← 32
k ← 0
FOR j ← 0 TO KL-1
    i ← j * 32
    IF k1[j] OR *no controlmask*
        THEN
            DEST[k+SIZE-1:k]← SRC[i+31:i]
            k ← k + SIZE
    FI;
ENDFOR;
```

VPCOMPRESSD (EVEX encoded versions) reg-reg form

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
SIZE ← 32
k ← 0
FOR j ← 0 TO KL-1
    i ← j * 32
    IF k1[j] OR *no controlmask*
        THEN
            DEST[k+SIZE-1:k]← SRC[i+31:i]
            k ← k + SIZE
    FI;
ENDFOR
IF *merging-masking*
        THEN *DEST[VL-1:k] remains unchanged*
        ELSE DEST[VL-1:k] ← 0
FI
DEST[MAXVL-1:VL] ← 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VPCOMPRESSD __m512i _mm512_mask_compress_epi32(__m512i s, __mmask16 c, __m512i a);
VPCOMPRESSD __m512i _mm512_maskz_compress_epi32(__mmask16 c, __m512i a);
VPCOMPRESSD void _mm512_mask_compressstoreu_epi32(void * a, __mmask16 c, __m512i s);
VPCOMPRESSD __m256i _mm256_mask_compress_epi32(__m256i s, __mmask8 c, __m256i a);
VPCOMPRESSD __m256i _mm256_maskz_compress_epi32(__mmask8 c, __m256i a);
VPCOMPRESSD void _mm256_mask_compressstoreu_epi32(void * a, __mmask8 c, __m256i s);
VPCOMPRESSD __m128i _mm_mask_compress_epi32(__m128i s, __mmask8 c, __m128i a);
VPCOMPRESSD __m128i _mm_maskz_compress_epi32(__mmask8 c, __m128i a);
VPCOMPRESSD void _mm_mask_compressstoreu_epi32(void * a, __mmask8 c, __m128i s);

SIMD Floating-Point Exceptions

None

Other Exceptions

EVEX-encoded instruction, see Exceptions Type E4.nb.

VPCOMPRESSQ—Store Sparse Packed Quadword Integer Values into Dense Memory/Register

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.128.66.0F38.W1 8B /r VPCOMPRESSQ xmm1/m128 {k1}{z}, xmm2	A	V/V	AVX512VL AVX512F	Compress packed quadword integer values from xmm2 to xmm1/m128 using controlmask k1.
EVEX.256.66.0F38.W1 8B /r VPCOMPRESSQ ymm1/m256 {k1}{z}, ymm2	A	V/V	AVX512VL AVX512F	Compress packed quadword integer values from ymm2 to ymm1/m256 using controlmask k1.
EVEX.512.66.0F38.W1 8B /r VPCOMPRESSQ zmm1/m512 {k1}{z}, zmm2	A	V/V	AVX512F	Compress packed quadword integer values from zmm2 to zmm1/m512 using controlmask k1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Tuple1 Scalar	ModRM:r/m (w)	ModRM:reg (r)	NA	NA

Description

Compress (stores) up to 8/4/2 quadword integer values from the source operand (second operand) to the destination operand (first operand). The source operand is a ZMM/YMM/XMM register, the destination operand can be a ZMM/YMM/XMM register or a 512/256/128-bit memory location.

The opmask register k1 selects the active elements (partial vector or possibly non-contiguous if less than 8 active elements) from the source operand to compress into a contiguous vector. The contiguous vector is written to the destination starting from the low element of the destination operand.

Memory destination version: Only the contiguous vector is written to the destination memory location. EVEX.z must be zero.

Register destination version: If the vector length of the contiguous vector is less than that of the input vector in the source operand, the upper bits of the destination register are unmodified if EVEX.z is not set, otherwise the upper bits are zeroed.

Note: EVEX.vvvv is reserved and must be 1111b otherwise instructions will #UD.

Note that the compressed displacement assumes a pre-scaling (N) corresponding to the size of one single element instead of the size of the full vector.

Operation

VPCOMPRESSQ (EVEX encoded versions) store form

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
SIZE ← 64
k ← 0
FOR j ← 0 TO KL-1
    i ← j * 64
    IF k1[j] OR *no controlmask*
        THEN
            DEST[k+SIZE-1:k]← SRC[i+63:i]
            k ← k + SIZE
    FI;
ENFOR
```

VPCOMPRESSQ (EVEX encoded versions) reg-reg form

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
SIZE ← 64
k ← 0
FOR j ← 0 TO KL-1
    i ← j * 64
    IF k1[j] OR *no controlmask*
        THEN
            DEST[k+SIZE-1:k]← SRC[i+63:i]
            k ← k + SIZE
    FI;
ENDFOR
IF *merging-masking*
        THEN *DEST[VL-1:k] remains unchanged*
        ELSE DEST[VL-1:k] ← 0
FI
DEST[MAXVL-1:VL] ← 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VPCOMPRESSQ __m512i _mm512_mask_compress_epi64(__m512i s, __mmask8 c, __m512i a);
VPCOMPRESSQ __m512i _mm512_maskz_compress_epi64(__mmask8 c, __m512i a);
VPCOMPRESSQ void _mm512_mask_compressstoreu_epi64(void * a, __mmask8 c, __m512i s);
VPCOMPRESSQ __m256i _mm256_mask_compress_epi64(__m256i s, __mmask8 c, __m256i a);
VPCOMPRESSQ __m256i _mm256_maskz_compress_epi64(__mmask8 c, __m256i a);
VPCOMPRESSQ void _mm256_mask_compressstoreu_epi64(void * a, __mmask8 c, __m256i s);
VPCOMPRESSQ __m128i _mm_mask_compress_epi64(__m128i s, __mmask8 c, __m128i a);
VPCOMPRESSQ __m128i _mm_maskz_compress_epi64(__mmask8 c, __m128i a);
VPCOMPRESSQ void _mm_mask_compressstoreu_epi64(void * a, __mmask8 c, __m128i s);

SIMD Floating-Point Exceptions

None

Other Exceptions

EVEX-encoded instruction, see Exceptions Type E4.nb.

VPCONFLICTD/Q—Detect Conflicts Within a Vector of Packed Dword/Qword Values into Dense Memory/ Register

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.128.66.0F38.W0 C4 /r VPCONFLICTD xmm1 {k1}{z}, xmm2/m128/m32bcst	A	V/V	AVX512VL AVX512CD	Detect duplicate double-word values in xmm2/m128/m32bcst using writemask k1.
EVEX.256.66.0F38.W0 C4 /r VPCONFLICTD ymm1 {k1}{z}, ymm2/m256/m32bcst	A	V/V	AVX512VL AVX512CD	Detect duplicate double-word values in ymm2/m256/m32bcst using writemask k1.
EVEX.512.66.0F38.W0 C4 /r VPCONFLICTD zmm1 {k1}{z}, zmm2/m512/m32bcst	A	V/V	AVX512CD	Detect duplicate double-word values in zmm2/m512/m32bcst using writemask k1.
EVEX.128.66.0F38.W1 C4 /r VPCONFLICTQ xmm1 {k1}{z}, xmm2/m128/m64bcst	A	V/V	AVX512VL AVX512CD	Detect duplicate quad-word values in xmm2/m128/m64bcst using writemask k1.
EVEX.256.66.0F38.W1 C4 /r VPCONFLICTQ ymm1 {k1}{z}, ymm2/m256/m64bcst	A	V/V	AVX512VL AVX512CD	Detect duplicate quad-word values in ymm2/m256/m64bcst using writemask k1.
EVEX.512.66.0F38.W1 C4 /r VPCONFLICTQ zmm1 {k1}{z}, zmm2/m512/m64bcst	A	V/V	AVX512CD	Detect duplicate quad-word values in zmm2/m512/m64bcst using writemask k1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Full Vector	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

Description

Test each dword/qword element of the source operand (the second operand) for equality with all other elements in the source operand closer to the least significant element. Each element's comparison results form a bit vector, which is then zero extended and written to the destination according to the writemask.

EVEX.512 encoded version: The source operand is a ZMM register, a 512-bit memory location, or a 512-bit vector broadcasted from a 32/64-bit memory location. The destination operand is a ZMM register, conditionally updated using writemask k1.

EVEX.256 encoded version: The source operand is a YMM register, a 256-bit memory location, or a 256-bit vector broadcasted from a 32/64-bit memory location. The destination operand is a YMM register, conditionally updated using writemask k1.

EVEX.128 encoded version: The source operand is a XMM register, a 128-bit memory location, or a 128-bit vector broadcasted from a 32/64-bit memory location. The destination operand is a XMM register, conditionally updated using writemask k1.

EVEX.vvvv is reserved and must be 1111b otherwise instructions will #UD.

Operation

VPCONFLICTD

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j ← 0 TO KL-1
    i ← j*32
    IF MaskBit(j) OR *no writemask*THEN
        FOR k ← 0 TO j-1
            m ← k*32
            IF ((SRC[i+31:i] = SRC[m+31:m])) THEN
                DEST[i+k] ← 1
            ELSE
                DEST[i+k] ← 0
            FI
        ENDFOR
        DEST[i+31:i+j] ← 0
    ELSE
        IF *merging-masking* THEN
            *DEST[i+31:i] remains unchanged*
        ELSE
            DEST[i+31:i] ← 0
        FI
    FI
ENDFOR
DEST[MAXVL-1:VL] ← 0
```

VPCONFLICTQ

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j ← 0 TO KL-1
     i ← j*64
     IF MaskBit(j) OR *no writemask*THEN
        FOR k ← 0 TO j-1
            m ← k*64
            IF ((SRC[i+63:i] = SRC[m+63:m])) THEN
                DEST[i+k] ← 1
            ELSE
                DEST[i+k] ← 0
            FI
        ENDFOR
        DEST[i+63:i+j] ← 0
    ELSE
        IF *merging-masking* THEN
            *DEST[i+63:i] remains unchanged*
        ELSE
            DEST[i+63:i] ← 0
        FI
    FI
ENDFOR
DEST[MAXVL-1:VL] ← 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VPCONFLICTD __m512i _mm512_conflict_epi32(__m512i a);
VPCONFLICTD __m512i _mm512_mask_conflict_epi32(__m512i s, __mmask16 m, __m512i a);
VPCONFLICTD __m512i _mm512_maskz_conflict_epi32(__mmask16 m, __m512i a);
VPCONFLICTQ __m512i _mm512_conflict_epi64(__m512i a);
VPCONFLICTQ __m512i _mm512_mask_conflict_epi64(__m512i s, __mmask8 m, __m512i a);
VPCONFLICTQ __m512i _mm512_maskz_conflict_epi64(__mmask8 m, __m512i a);
VPCONFLICTD __m256i _mm256_conflict_epi32(__m256i a);
VPCONFLICTD __m256i _mm256_mask_conflict_epi32(__m256i s, __mmask8 m, __m256i a);
VPCONFLICTD __m256i _mm256_maskz_conflict_epi32(__mmask8 m, __m256i a);
VPCONFLICTQ __m256i _mm256_conflict_epi64(__m256i a);
VPCONFLICTQ __m256i _mm256_mask_conflict_epi64(__m256i s, __mmask8 m, __m256i a);
VPCONFLICTQ __m256i _mm256_maskz_conflict_epi64(__mmask8 m, __m256i a);
VPCONFLICTD __m128i _mm_conflict_epi32(__m128i a);
VPCONFLICTD __m128i _mm_mask_conflict_epi32(__m128i s, __mmask8 m, __m128i a);
VPCONFLICTD __m128i _mm_maskz_conflict_epi32(__mmask8 m, __m128i a);
VPCONFLICTQ __m128i _mm_conflict_epi64(__m128i a);
VPCONFLICTQ __m128i _mm_mask_conflict_epi64(__m128i s, __mmask8 m, __m128i a);
VPCONFLICTQ __m128i _mm_maskz_conflict_epi64(__mmask8 m, __m128i a);

SIMD Floating-Point Exceptions

None

Other Exceptions

EVEX-encoded instruction, see Exceptions Type E4.

VPERM2F128 — Permute Floating-Point Values

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
VEX.NDS.256.66.0F3A.W0 06 /r ib VPERM2F128 *ymm1, ymm2, ymm3/m256, imm8*	RVMI	V/V	AVX	Permute 128-bit floating-point fields in *ymm2* and *ymm3/mem* using controls from *imm8* and store result in *ymm1*.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RVMI	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	imm8

Description

Permute 128 bit floating-point-containing fields from the first source operand (second operand) and second source operand (third operand) using bits in the 8-bit immediate and store results in the destination operand (first operand). The first source operand is a YMM register, the second source operand is a YMM register or a 256-bit memory location, and the destination operand is a YMM register.

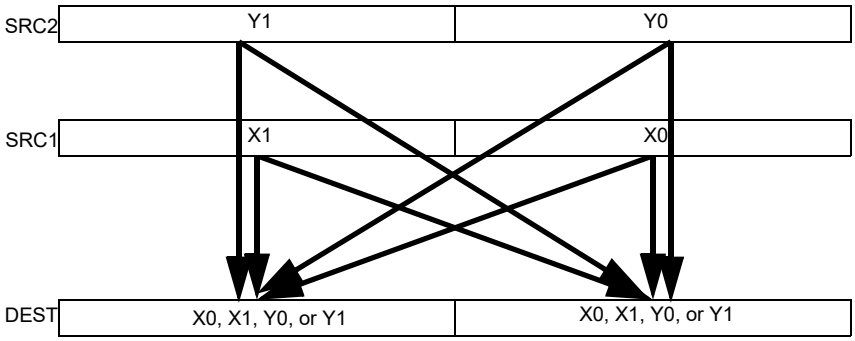


Figure 5-21. VPERM2F128 Operation

Imm8[1:0] select the source for the first destination 128-bit field, imm8[5:4] select the source for the second destination field. If imm8[3] is set, the low 128-bit field is zeroed. If imm8[7] is set, the high 128-bit field is zeroed.

VEX.L must be 1, otherwise the instruction will #UD.

Operation

VPERM2F128

```
CASE IMM8[1:0] of
0: DEST[127:0] ← SRC1[127:0]
1: DEST[127:0] ← SRC1[255:128]
2: DEST[127:0] ← SRC2[127:0]
3: DEST[127:0] ← SRC2[255:128]
ESAC

CASE IMM8[5:4] of
0: DEST[255:128] ← SRC1[127:0]
1: DEST[255:128] ← SRC1[255:128]
2: DEST[255:128] ← SRC2[127:0]
3: DEST[255:128] ← SRC2[255:128]
ESAC
IF (imm8[3])
DEST[127:0] ← 0
FI

IF (imm8[7])
DEST[MAXVL-1:128] ← 0
FI
```

Intel C/C++ Compiler Intrinsic Equivalent

VPERM2F128: __m256 _mm256_permute2f128_ps (__m256 a, __m256 b, int control)

VPERM2F128: __m256d _mm256_permute2f128_pd (__m256d a, __m256d b, int control)

VPERM2F128: __m256i _mm256_permute2f128_si256 (__m256i a, __m256i b, int control)

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Exceptions Type 6; additionally

#UD	If VEX.L = 0
	If VEX.W = 1.

VPERM2I128 — Permute Integer Values

Opcode/ Instruction	Op/ En	64/32 -bit Mode	CPUID Feature Flag	Description
VEX.NDS.256.66.0F3A.W0 46 /r ib VPERM2I128 *ymm1, ymm2, ymm3/m256, imm8*	RVMI	V/V	AVX2	Permute 128-bit integer data in *ymm2* and *ymm3/mem* using controls from *imm8* and store result in *ymm1*.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RVMI	ModRM:reg (w)	VEX.vvvv	ModRM:r/m (r)	Imm8

Description

Permute 128 bit integer data from the first source operand (second operand) and second source operand (third operand) using bits in the 8-bit immediate and store results in the destination operand (first operand). The first source operand is a YMM register, the second source operand is a YMM register or a 256-bit memory location, and the destination operand is a YMM register.

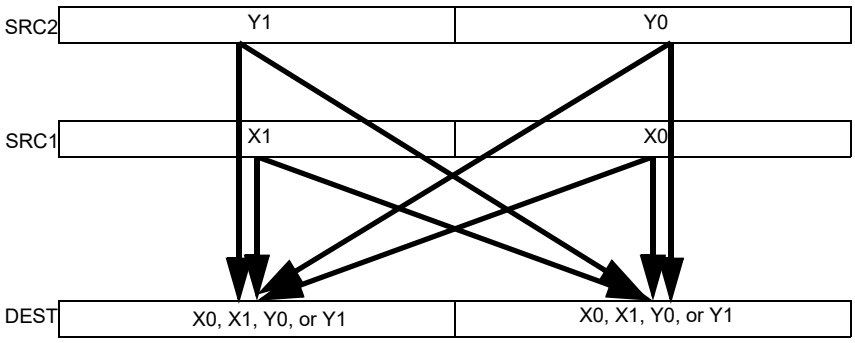


Figure 5-22. VPERM2I128 Operation

Imm8[1:0] select the source for the first destination 128-bit field, imm8[5:4] select the source for the second destination field. If imm8[3] is set, the low 128-bit field is zeroed. If imm8[7] is set, the high 128-bit field is zeroed.

VEX.L must be 1, otherwise the instruction will #UD.

Operation

VPERM2I128

```
CASE IMM8[1:0] of
0: DEST[127:0] ← SRC1[127:0]
1: DEST[127:0] ← SRC1[255:128]
2: DEST[127:0] ← SRC2[127:0]
3: DEST[127:0] ← SRC2[255:128]
ESAC
CASE IMM8[5:4] of
0: DEST[255:128] ← SRC1[127:0]
1: DEST[255:128] ← SRC1[255:128]
2: DEST[255:128] ← SRC2[127:0]
3: DEST[255:128] ← SRC2[255:128]
ESAC
IF (imm8[3])
DEST[127:0] ← 0
FI

IF (imm8[7])
DEST[255:128] ← 0
FI
```

Intel C/C++ Compiler Intrinsic Equivalent

VPERM2I128: __m256i _mm256_permute2x128_si256 (__m256i a, __m256i b, int control)

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type 6; additionally

#UD	If VEX.L = 0,
	If VEX.W = 1.

VPERMD/VPERMW—Permute Packed Doublewords/Words Elements

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
VEX.NDS.256.66.0F38.W0 36 /r VPERMD ymm1, ymm2, ymm3/m256	A	V/V	AVX2	Permute doublewords in ymm3/m256 using indices in ymm2 and store the result in ymm1.
EVEX.NDS.256.66.0F38.W0 36 /r VPERMD ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst	B	V/V	AVX512VL AVX512F	Permute doublewords in ymm3/m256/m32bcst using indexes in ymm2 and store the result in ymm1 using writemask k1.
EVEX.NDS.512.66.0F38.W0 36 /r VPERMD zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst	B	V/V	AVX512F	Permute doublewords in zmm3/m512/m32bcst using indices in zmm2 and store the result in zmm1 using writemask k1.
EVEX.NDS.128.66.0F38.W1 8D /r VPERMW xmm1 {k1}{z}, xmm2, xmm3/m128	C	V/V	AVX512VL AVX512BW	Permute word integers in xmm3/m128 using indexes in xmm2 and store the result in xmm1 using writemask k1.
EVEX.NDS.256.66.0F38.W1 8D /r VPERMW ymm1 {k1}{z}, ymm2, ymm3/m256	C	V/V	AVX512VL AVX512BW	Permute word integers in ymm3/m256 using indexes in ymm2 and store the result in ymm1 using writemask k1.
EVEX.NDS.512.66.0F38.W1 8D /r VPERMW zmm1 {k1}{z}, zmm2, zmm3/m512	C	V/V	AVX512BW	Permute word integers in zmm3/m512 using indexes in zmm2 and store the result in zmm1 using writemask k1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	NA	ModRM:reg (w)	VEX.vvvv	ModRM:r/m (r)	NA
B	Full Vector	ModRM:reg (w)	EVEX.vvvv	ModRM:r/m (r)	NA
C	Full Vector Mem	ModRM:reg (w)	VEX.vvvv	ModRM:r/m (r)	NA

Description

Copies doublewords (or words) from the second source operand (the third operand) to the destination operand (the first operand) according to the indices in the first source operand (the second operand). Note that this instruction permits a doubleword (word) in the source operand to be copied to more than one location in the destination operand.

VEX.256 encoded VPERMD: The first and second operands are YMM registers, the third operand can be a YMM register or memory location. Bits (MAXVL-1:256) of the corresponding destination register are zeroed.

EVEX encoded VPERMD: The first and second operands are ZMM/YMM registers, the third operand can be a ZMM/YMM register, a 512/256-bit memory location or a 512/256-bit vector broadcasted from a 32-bit memory location. The elements in the destination are updated using the writemask k1.

VPERMW: first and second operands are ZMM/YMM/XMM registers, the third operand can be a ZMM/YMM/XMM register, or a 512/256/128-bit memory location. The destination is updated using the writemask k1.

EVEX.128 encoded versions: Bits (MAXVL-1:128) of the corresponding ZMM register are zeroed.

Operation

VPERMD (EVEX encoded versions)

```
(KL, VL) = (8, 256), (16, 512)
IF VL = 256 THEN n ← 2; FI;
IF VL = 512 THEN n ← 3; FI;
FOR j ← 0 TO KL-1
    i ← j * 32
    id ← 32*SRC1[i+n:i]
    IF k1[j] OR *no writemask*
        THEN
            IF (EVEX.b = 1) AND (SRC2 *is memory*)
                THEN DEST[i+31:i] ← SRC2[31:0];
                ELSE DEST[i+31:i] ← SRC2[id+31:id];
            FI;
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE                                ; zeroing-masking
                    DEST[i+31:i] ← 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0
```

VPERMD (VEX.256 encoded version)

```
DEST[31:0] ← (SRC2[255:0] >> (SRC1[2:0] * 32))[31:0];
DEST[63:32] ← (SRC2[255:0] >> (SRC1[34:32] * 32))[31:0];
DEST[95:64] ← (SRC2[255:0] >> (SRC1[66:64] * 32))[31:0];
DEST[127:96] ← (SRC2[255:0] >> (SRC1[98:96] * 32))[31:0];
DEST[159:128] ← (SRC2[255:0] >> (SRC1[130:128] * 32))[31:0];
DEST[191:160] ← (SRC2[255:0] >> (SRC1[162:160] * 32))[31:0];
DEST[223:192] ← (SRC2[255:0] >> (SRC1[194:192] * 32))[31:0];
DEST[255:224] ← (SRC2[255:0] >> (SRC1[226:224] * 32))[31:0];
DEST[MAXVL-1:256] ← 0
```

VPERMW (EVEX encoded versions)

```
(KL, VL) = (8, 128), (16, 256), (32, 512)
IF VL = 128 THEN n ← 2; FI;
IF VL = 256 THEN n ← 3; FI;
IF VL = 512 THEN n ← 4; FI;
FOR j ← 0 TO KL-1
    i ← j * 16
    id ← 16*SRC1[i+n:i]
    IF k1[j] OR *no writemask*
        THEN DEST[i+15:i] ← SRC2[id+15:id]
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+15:i] remains unchanged*
                ELSE                                ; zeroing-masking
                    DEST[i+15:i] ← 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VPERMD __m512i _mm512_permutexvar_epi32(__m512i idx, __m512i a);
VPERMD __m512i _mm512_mask_permutexvar_epi32(__m512i s, __mmask16 k, __m512i idx, __m512i a);
VPERMD __m512i _mm512_maskz_permutexvar_epi32(__mmask16 k, __m512i idx, __m512i a);
VPERMD __m256i _mm256_permutexvar_epi32(__m256i idx, __m256i a);
VPERMD __m256i _mm256_mask_permutexvar_epi32(__m256i s, __mmask8 k, __m256i idx, __m256i a);
VPERMD __m256i _mm256_maskz_permutexvar_epi32(__mmask8 k, __m256i idx, __m256i a);
VPERMW __m512i _mm512_permutexvar_epi16(__m512i idx, __m512i a);
VPERMW __m512i _mm512_mask_permutexvar_epi16(__m512i s, __mmask32 k, __m512i idx, __m512i a);
VPERMW __m512i _mm512_maskz_permutexvar_epi16(__mmask32 k, __m512i idx, __m512i a);
VPERMW __m256i _mm256_permutexvar_epi16(__m256i idx, __m256i a);
VPERMW __m256i _mm256_mask_permutexvar_epi16(__m256i s, __mmask16 k, __m256i idx, __m256i a);
VPERMW __m256i _mm256_maskz_permutexvar_epi16(__mmask16 k, __m256i idx, __m256i a);
VPERMW __m128i _mm_permutexvar_epi16(__m128i idx, __m128i a);
VPERMW __m128i _mm_mask_permutexvar_epi16(__m128i s, __mmask8 k, __m128i idx, __m128i a);
VPERMW __m128i _mm_maskz_permutexvar_epi16(__mmask8 k, __m128i idx, __m128i a);

SIMD Floating-Point Exceptions

None

Other Exceptions

Non-EVEX-encoded instruction, see Exceptions Type 4.
EVEX-encoded VPERMD, see Exceptions Type E4NF.
EVEX-encoded VPERMW, see Exceptions Type E4NF.nb.

#UD	If VEX.L = 0.
	If EVEX.L'L = 0 for VPERMD.

VPERMI2W/D/Q/PS/PD—Full Permute From Two Tables Overwriting the Index

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.DDS.128.66.0F38.W1 75 /r VPERMI2W xmm1 {k1}{z}, xmm2, xmm3/m128	A	V/V	AVX512VL AVX512BW	Permute word integers from two tables in xmm3/m128 and xmm2 using indexes in xmm1 and store the result in xmm1 using writemask k1.
EVEX.DDS.256.66.0F38.W1 75 /r VPERMI2W ymm1 {k1}{z}, ymm2, ymm3/m256	A	V/V	AVX512VL AVX512BW	Permute word integers from two tables in ymm3/m256 and ymm2 using indexes in ymm1 and store the result in ymm1 using writemask k1.
EVEX.DDS.512.66.0F38.W1 75 /r VPERMI2W zmm1 {k1}{z}, zmm2, zmm3/m512	A	V/V	AVX512BW	Permute word integers from two tables in zmm3/m512 and zmm2 using indexes in zmm1 and store the result in zmm1 using writemask k1.
EVEX.DDS.128.66.0F38.W0 76 /r VPERMI2D xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst	B	V/V	AVX512VL AVX512F	Permute double-words from two tables in xmm3/m128/m32bcst and xmm2 using indexes in xmm1 and store the result in xmm1 using writemask k1.
EVEX.DDS.256.66.0F38.W0 76 /r VPERMI2D ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst	B	V/V	AVX512VL AVX512F	Permute double-words from two tables in ymm3/m256/m32bcst and ymm2 using indexes in ymm1 and store the result in ymm1 using writemask k1.
EVEX.DDS.512.66.0F38.W0 76 /r VPERMI2D zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst	B	V/V	AVX512F	Permute double-words from two tables in zmm3/m512/m32bcst and zmm2 using indices in zmm1 and store the result in zmm1 using writemask k1.
EVEX.DDS.128.66.0F38.W1 76 /r VPERMI2Q xmm1 {k1}{z}, xmm2, xmm3/m128/m64bcst	B	V/V	AVX512VL AVX512F	Permute quad-words from two tables in xmm3/m128/m64bcst and xmm2 using indexes in xmm1 and store the result in xmm1 using writemask k1.
EVEX.DDS.256.66.0F38.W1 76 /r VPERMI2Q ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst	B	V/V	AVX512VL AVX512F	Permute quad-words from two tables in ymm3/m256/m64bcst and ymm2 using indexes in ymm1 and store the result in ymm1 using writemask k1.
EVEX.DDS.512.66.0F38.W1 76 /r VPERMI2Q zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst	B	V/V	AVX512F	Permute quad-words from two tables in zmm3/m512/m64bcst and zmm2 using indices in zmm1 and store the result in zmm1 using writemask k1.
EVEX.DDS.128.66.0F38.W0 77 /r VPERMI2PS xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst	B	V/V	AVX512VL AVX512F	Permute single-precision FP values from two tables in xmm3/m128/m32bcst and xmm2 using indexes in xmm1 and store the result in xmm1 using writemask k1.
EVEX.DDS.256.66.0F38.W0 77 /r VPERMI2PS ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst	B	V/V	AVX512VL AVX512F	Permute single-precision FP values from two tables in ymm3/m256/m32bcst and ymm2 using indexes in ymm1 and store the result in ymm1 using writemask k1.
EVEX.DDS.512.66.0F38.W0 77 /r VPERMI2PS zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst	B	V/V	AVX512F	Permute single-precision FP values from two tables in zmm3/m512/m32bcst and zmm2 using indices in zmm1 and store the result in zmm1 using writemask k1.

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.DDS.128.66.0F38.W1 77 /r VPERMI2PD xmm1 {k1}{z}, xmm2, xmm3/m128/m64bcst	B	V/V	AVX512VL AVX512F	Permute double-precision FP values from two tables in xmm3/m128/m64bcst and xmm2 using indexes in xmm1 and store the result in xmm1 using writemask k1.
EVEX.DDS.256.66.0F38.W1 77 /r VPERMI2PD ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst	B	V/V	AVX512VL AVX512F	Permute double-precision FP values from two tables in ymm3/m256/m64bcst and ymm2 using indexes in ymm1 and store the result in ymm1 using writemask k1.
EVEX.DDS.512.66.0F38.W1 77 /r VPERMI2PD zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst	B	V/V	AVX512F	Permute double-precision FP values from two tables in zmm3/m512/m64bcst and zmm2 using indices in zmm1 and store the result in zmm1 using writemask k1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Full Vector Mem	ModRM:reg (r,w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA
B	Full Vector	ModRM:reg (r, w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA

Description

Permutes 16-bit/32-bit/64-bit values in the second operand (the first source operand) and the third operand (the second source operand) using indices in the first operand to select elements from the second and third operands. The selected elements are written to the destination operand (the first operand) according to the writemask k1.

The first and second operands are ZMM/YMM/XMM registers. The first operand contains input indices to select elements from the two input tables in the 2nd and 3rd operands. The first operand is also the destination of the result.

D/Q/PS/PD element versions: The second source operand can be a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 32/64-bit memory location. Broadcast from the low 32/64-bit memory location is performed if EVEX.b and the id bit for table selection are set (selecting table_2).

Dword/PS versions: The id bit for table selection is bit 4/3/2, depending on VL=512, 256, 128. Bits [3:0]/[2:0]/[1:0] of each element in the input index vector select an element within the two source operands, If the id bit is 0, table_1 (the first source) is selected; otherwise the second source operand is selected.

Qword/PD versions: The id bit for table selection is bit 3/2/1, and bits [2:0]/[1:0] /bit 0 selects element within each input table.

Word element versions: The second source operand can be a ZMM/YMM/XMM register, or a 512/256/128-bit memory location. The id bit for table selection is bit 5/4/3, and bits [4:0]/[3:0]/[2:0] selects element within each input table.

Note that these instructions permit a 16-bit/32-bit/64-bit value in the source operands to be copied to more than one location in the destination operand. Note also that in this case, the same table can be reused for example for a second iteration, while the index elements are overwritten.

Bits (MAXVL-1:256/128) of the destination are zeroed for VL=256,128.

Operation

VPERMI2W (EVEX encoded versions)

```
(KL, VL) = (8, 128), (16, 256), (32, 512)
IF VL = 128
    id ← 2
FI;
IF VL = 256
    id ← 3
FI;
IF VL = 512
    id ← 4
FI;
TMP_DEST← DEST
FOR j ← 0 TO KL-1
    i ← j * 16
    off ← 16*TMP_DEST[i+id:i]
    IF k1[j] OR *no writemask*
        THEN
            DEST[i+15:i]=TMP_DEST[i+id+1] ? SRC2[off+15:off]
                    : SRC1[off+15:off]
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+15:i] remains unchanged*
                ELSE                                ; zeroing-masking
                    DEST[i+15:i] ← 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0
```

VPERMI2D/VPERMI2PS (EVEX encoded versions)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
IF VL = 128
    id ← 1
FI;
IF VL = 256
    id ← 2
FI;
IF VL = 512
    id ← 3
FI;
TMP_DEST← DEST
FOR j ← 0 TO KL-1
    i ← j * 32
    off ← 32*TMP_DEST[i+id:i]
    IF k1[j] OR *no writemask*
        THEN
            IF (EVEX.b = 1) AND (SRC2 *is memory*)
                THEN
                    DEST[i+31:i]  ← TMP_DEST[i+id+1] ? SRC2[31:0]
                    : SRC1[off+31:off]
            ELSE
                DEST[i+31:i] ← TMP_DEST[i+id+1] ? SRC2[off+31:off]
                    : SRC1[off+31:off]
```

```
            FI
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE                                ; zeroing-masking
                    DEST[i+31:i] ← 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0
```

VPERMI2Q/VPERMI2PD (EVEX encoded versions)

```
(KL, VL) = (2, 128), (4, 256), (8 512)
IF VL = 128
    id ← 0
FI;
IF VL = 256
    id ← 1
FI;
IF VL = 512
    id ← 2
FI;
TMP_DEST← DEST
FOR j ← 0 TO KL-1
    i ← j * 64
    off ← 64*TMP_DEST[i+id:i]
    IF k1[j] OR *no writemask*
        THEN
            IF (EVEX.b = 1) AND (SRC2 *is memory*)
                THEN
                    DEST[i+63:i] ← TMP_DEST[i+id+1] ? SRC2[63:0]
                    : SRC1[off+63:off]
            ELSE
                DEST[i+63:i] ← TMP_DEST[i+id+1] ? SRC2[off+63:off]
                    : SRC1[off+63:off]
            FI
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE                                ; zeroing-masking
                    DEST[i+63:i] ← 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VPERMI2D __m512i _mm512_permutex2var_epi32(__m512i a, __m512i idx, __m512i b);
VPERMI2D __m512i _mm512_mask_permutex2var_epi32(__m512i a, __mmask16 k, __m512i idx, __m512i b);
VPERMI2D __m512i _mm512_mask2_permutex2var_epi32(__m512i a, __m512i idx, __mmask16 k, __m512i b);
VPERMI2D __m512i _mm512_maskz_permutex2var_epi32(__mmask16 k, __m512i a, __m512i idx, __m512i b);
VPERMI __m256i _mm256_permutex2var_epi32(__m256i a, __m256i idx, __m256i b);
VPERMI2D __m256i _mm256_mask_permutex2var_epi32(__m256i a, __mmask8 k, __m256i idx, __m256i b);
VPERMI2D __m256i _mm256_mask2_permutex2var_epi32(__m256i a, __m256i idx, __mmask8 k, __m256i b);
VPERMI2D __m256i _mm256_maskz_permutex2var_epi32(__mmask8 k, __m256i a, __m256i idx, __m256i b);
VPERMI2D __m128i _mm_permutex2var_epi32(__m128i a, __m128i idx, __m128i b);
VPERMI2D __m128i _mm_mask_permutex2var_epi32(__m128i a, __mmask8 k, __m128i idx, __m128i b);
VPERMI2D __m128i _mm_mask2_permutex2var_epi32(__m128i a, __m128i idx, __mmask8 k, __m128i b);
VPERMI2D __m128i _mm_maskz_permutex2var_epi32(__mmask8 k, __m128i a, __m128i idx, __m128i b);
VPERMI2PD __m512d _mm512_permutex2var_pd(__m512d a, __m512i idx, __m512d b);
VPERMI2PD __m512d _mm512_mask_permutex2var_pd(__m512d a, __mmask8 k, __m512i idx, __m512d b);
VPERMI2PD __m512d _mm512_mask2_permutex2var_pd(__m512d a, __m512i idx, __mmask8 k, __m512d b);
VPERMI2PD __m512d _mm512_maskz_permutex2var_pd(__mmask8 k, __m512d a, __m512i idx, __m512d b);
VPERMI2PD __m256d _mm256_permutex2var_pd(__m256d a, __m256i idx, __m256d b);
VPERMI2PD __m256d _mm256_mask_permutex2var_pd(__m256d a, __mmask8 k, __m256i idx, __m256d b);
VPERMI2PD __m256d _mm256_mask2_permutex2var_pd(__m256d a, __m256i idx, __mmask8 k, __m256d b);
VPERMI2PD __m256d _mm256_maskz_permutex2var_pd(__mmask8 k, __m256d a, __m256i idx, __m256d b);
VPERMI2PD __m128d _mm_permutex2var_pd(__m128d a, __m128i idx, __m128d b);
VPERMI2PD __m128d _mm_mask_permutex2var_pd(__m128d a, __mmask8 k, __m128i idx, __m128d b);
VPERMI2PD __m128d _mm_mask2_permutex2var_pd(__m128d a, __m128i idx, __mmask8 k, __m128d b);
VPERMI2PD __m128d _mm_maskz_permutex2var_pd(__mmask8 k, __m128d a, __m128i idx, __m128d b);
VPERMI2PS __m512 _mm512_permutex2var_ps(__m512 a, __m512i idx, __m512 b);
VPERMI2PS __m512 _mm512_mask_permutex2var_ps(__m512 a, __mmask16 k, __m512i idx, __m512 b);
VPERMI2PS __m512 _mm512_mask2_permutex2var_ps(__m512 a, __m512i idx, __mmask16 k, __m512 b);
VPERMI2PS __m512 _mm512_maskz_permutex2var_ps(__mmask16 k, __m512 a, __m512i idx, __m512 b);
VPERMI2PS __m256 _mm256_permutex2var_ps(__m256 a, __m256i idx, __m256 b);
VPERMI2PS __m256 _mm256_mask_permutex2var_ps(__m256 a, __mmask8 k, __m256i idx, __m256 b);
VPERMI2PS __m256 _mm256_mask2_permutex2var_ps(__m256 a, __m256i idx, __mmask8 k, __m256 b);
VPERMI2PS __m256 _mm256_maskz_permutex2var_ps(__mmask8 k, __m256 a, __m256i idx, __m256 b);
VPERMI2PS __m128 _mm_permutex2var_ps(__m128 a, __m128i idx, __m128 b);
VPERMI2PS __m128 _mm_mask_permutex2var_ps(__m128 a, __mmask8 k, __m128i idx, __m128 b);
VPERMI2PS __m128 _mm_mask2_permutex2var_ps(__m128 a, __m128i idx, __mmask8 k, __m128 b);
VPERMI2PS __m128 _mm_maskz_permutex2var_ps(__mmask8 k, __m128 a, __m128i idx, __m128 b);
VPERMI2Q __m512i _mm512_permutex2var_epi64(__m512i a, __m512i idx, __m512i b);
VPERMI2Q __m512i _mm512_mask_permutex2var_epi64(__m512i a, __mmask8 k, __m512i idx, __m512i b);
VPERMI2Q __m512i _mm512_mask2_permutex2var_epi64(__m512i a, __m512i idx, __mmask8 k, __m512i b);
VPERMI2Q __m512i _mm512_maskz_permutex2var_epi64(__mmask8 k, __m512i a, __m512i idx, __m512i b);
VPERMI2Q __m256i _mm256_permutex2var_epi64(__m256i a, __m256i idx, __m256i b);
VPERMI2Q __m256i _mm256_mask_permutex2var_epi64(__m256i a, __mmask8 k, __m256i idx, __m256i b);
VPERMI2Q __m256i _mm256_mask2_permutex2var_epi64(__m256i a, __m256i idx, __mmask8 k, __m256i b);
VPERMI2Q __m256i _mm256_maskz_permutex2var_epi64(__mmask8 k, __m256i a, __m256i idx, __m256i b);
VPERMI2Q __m128i _mm_permutex2var_epi64(__m128i a, __m128i idx, __m128i b);
VPERMI2Q __m128i _mm_mask_permutex2var_epi64(__m128i a, __mmask8 k, __m128i idx, __m128i b);
VPERMI2Q __m128i _mm_mask2_permutex2var_epi64(__m128i a, __m128i idx, __mmask8 k, __m128i b);
VPERMI2Q __m128i _mm_maskz_permutex2var_epi64(__mmask8 k, __m128i a, __m128i idx, __m128i b);

VPERMI2W __m512i _mm512_permutex2var_epi16(__m512i a, __m512i idx, __m512i b);
VPERMI2W __m512i _mm512_mask_permutex2var_epi16(__m512i a, __mmask32 k, __m512i idx, __m512i b);
VPERMI2W __m512i _mm512_mask2_permutex2var_epi16(__m512i a, __m512i idx, __mmask32 k, __m512i b);
VPERMI2W __m512i _mm512_maskz_permutex2var_epi16(__mmask32 k, __m512i a, __m512i idx, __m512i b);
VPERMI2W __m256i _mm256_permutex2var_epi16(__m256i a, __m256i idx, __m256i b);
VPERMI2W __m256i _mm256_mask_permutex2var_epi16(__m256i a, __mmask16 k, __m256i idx, __m256i b);
VPERMI2W __m256i _mm256_mask2_permutex2var_epi16(__m256i a, __m256i idx, __mmask16 k, __m256i b);
VPERMI2W __m256i _mm256_maskz_permutex2var_epi16(__mmask16 k, __m256i a, __m256i idx, __m256i b);
VPERMI2W __m128i _mm_permutex2var_epi16(__m128i a, __m128i idx, __m128i b);
VPERMI2W __m128i _mm_mask_permutex2var_epi16(__m128i a, __mmask8 k, __m128i idx, __m128i b);
VPERMI2W __m128i _mm_mask2_permutex2var_epi16(__m128i a, __m128i idx, __mmask8 k, __m128i b);
VPERMI2W __m128i _mm_maskz_permutex2var_epi16(__mmask8 k, __m128i a, __m128i idx, __m128i b);

SIMD Floating-Point Exceptions

None

Other Exceptions

VPERMI2D/Q/PS/PD: See Exceptions Type E4NF.
VPERMI2W: See Exceptions Type E4NF.nb.

VPERMILPD—Permute In-Lane of Pairs of Double-Precision Floating-Point Values

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
VEX.NDS.128.66.0F38.W0 0D /r VPERMILPD xmm1, xmm2, xmm3/m128	A	V/V	AVX	Permute double-precision floating-point values in xmm2 using controls from xmm3/m128 and store result in xmm1.
VEX.NDS.256.66.0F38.W0 0D /r VPERMILPD ymm1, ymm2, ymm3/m256	A	V/V	AVX	Permute double-precision floating-point values in ymm2 using controls from ymm3/m256 and store result in ymm1.
EVEX.NDS.128.66.0F38.W1 0D /r VPERMILPD xmm1 {k1}{z}, xmm2, xmm3/m128/m64bcst	C	V/V	AVX512VL AVX512F	Permute double-precision floating-point values in xmm2 using control from xmm3/m128/m64bcst and store the result in xmm1 using writemask k1.
EVEX.NDS.256.66.0F38.W1 0D /r VPERMILPD ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst	C	V/V	AVX512VL AVX512F	Permute double-precision floating-point values in ymm2 using control from ymm3/m256/m64bcst and store the result in ymm1 using writemask k1.
EVEX.NDS.512.66.0F38.W1 0D /r VPERMILPD zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst	C	V/V	AVX512F	Permute double-precision floating-point values in zmm2 using control from zmm3/m512/m64bcst and store the result in zmm1 using writemask k1.
VEX.128.66.0F3A.W0 05 /r ib VPERMILPD xmm1, xmm2/m128, imm8	B	V/V	AVX	Permute double-precision floating-point values in xmm2/m128 using controls from imm8.
VEX.256.66.0F3A.W0 05 /r ib VPERMILPD ymm1, ymm2/m256, imm8	B	V/V	AVX	Permute double-precision floating-point values in ymm2/m256 using controls from imm8.
EVEX.128.66.0F3A.W1 05 /r ib VPERMILPD xmm1 {k1}{z}, xmm2/m128/m64bcst, imm8	D	V/V	AVX512VL AVX512F	Permute double-precision floating-point values in xmm2/m128/m64bcst using controls from imm8 and store the result in xmm1 using writemask k1.
EVEX.256.66.0F3A.W1 05 /r ib VPERMILPD ymm1 {k1}{z}, ymm2/m256/m64bcst, imm8	D	V/V	AVX512VL AVX512F	Permute double-precision floating-point values in ymm2/m256/m64bcst using controls from imm8 and store the result in ymm1 using writemask k1.
EVEX.512.66.0F3A.W1 05 /r ib VPERMILPD zmm1 {k1}{z}, zmm2/m512/m64bcst, imm8	D	V/V	AVX512F	Permute double-precision floating-point values in zmm2/m512/m64bcst using controls from imm8 and store the result in zmm1 using writemask k1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	NA	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA
B	NA	ModRM:reg (w)	ModRM:r/m (r)	NA	NA
C	Full Vector	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA
D	Full Vector	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

Description

(variable control version)

Permute pairs of double-precision floating-point values in the first source operand (second operand), each using a 1-bit control field residing in the corresponding quadword element of the second source operand (third operand). Permuted results are stored in the destination operand (first operand).

The control bits are located at bit 0 of each quadword element (see Figure 5-24). Each control determines which of the source element in an input pair is selected for the destination element. Each pair of source elements must lie in the same 128-bit region as the destination.

EVEX version: The second source operand (third operand) is a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 64-bit memory location. Permuted results are written to the destination under the writemask.

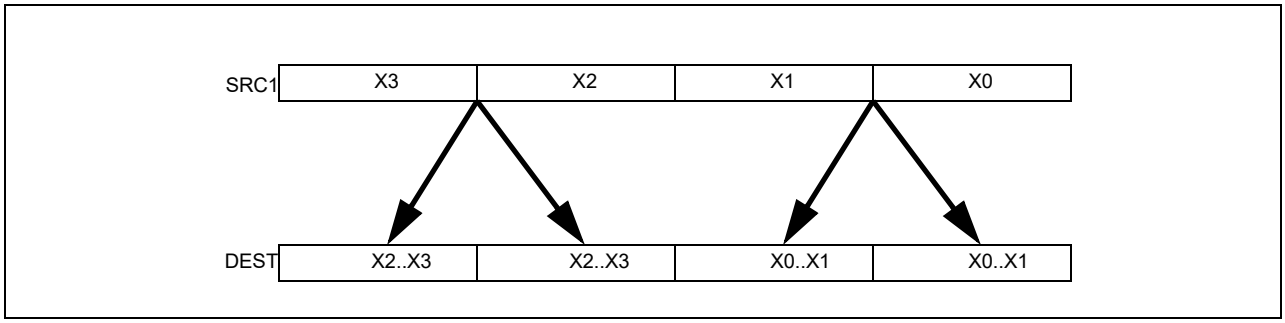


Figure 5-23. VPERMILPD Operation

VEX.256 encoded version: Bits (MAXVL-1:256) of the corresponding ZMM register are zeroed.

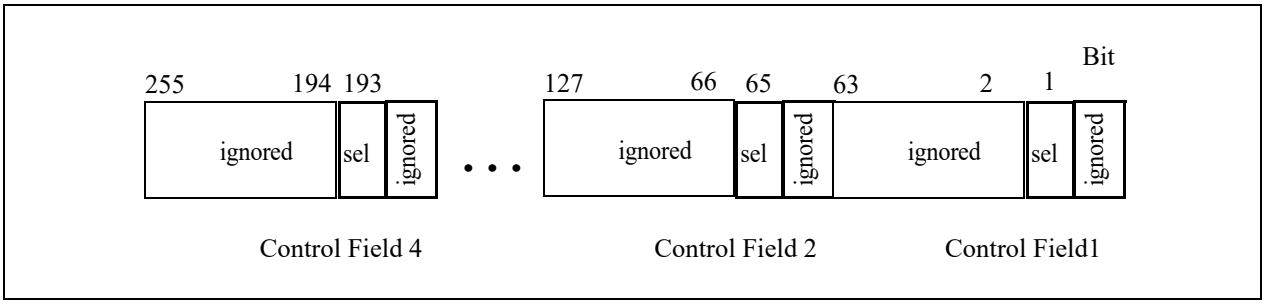


Figure 5-24. VPERMILPD Shuffle Control

(immediate control version)

Permute pairs of double-precision floating-point values in the first source operand (second operand), each pair using a 1-bit control field in the imm8 byte. Each element in the destination operand (first operand) use a separate control bit of the imm8 byte.

VEX version: The source operand is a YMM/XMM register or a 256/128-bit memory location and the destination operand is a YMM/XMM register. Imm8 byte provides the lower 4/2 bit as permute control fields.

EVEX version: The source operand (second operand) is a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 64-bit memory location. Permuted results are written to the destination under the writemask. Imm8 byte provides the lower 8/4/2 bit as permute control fields.

Note: For the imm8 versions, VEX.vvvv and EVEX.vvvv are reserved and must be 1111b otherwise instruction will #UD.

Operation

VPERMILPD (EVEX immediate versions)

```
(KL, VL) = (8, 512)
FOR j ← 0 TO KL-1
    i ← j * 64
    IF (EVEX.b = 1) AND (SRC1 *is memory*)
        THEN TMP_SRC1[i+63:i] ← SRC1[63:0];
        ELSE TMP_SRC1[i+63:i] ← SRC1[i+63:i];
    FI;
ENDFOR;
IF (imm8[0] = 0) THEN TMP_DEST[63:0] ← SRC1[63:0]; FI;
IF (imm8[0] = 1) THEN TMP_DEST[63:0] ← TMP_SRC1[127:64]; FI;
IF (imm8[1] = 0) THEN TMP_DEST[127:64] ← TMP_SRC1[63:0]; FI;
IF (imm8[1] = 1) THEN TMP_DEST[127:64] ← TMP_SRC1[127:64]; FI;
IF VL >= 256
    IF (imm8[2] = 0) THEN TMP_DEST[191:128] ← TMP_SRC1[191:128]; FI;
    IF (imm8[2] = 1) THEN TMP_DEST[191:128] ← TMP_SRC1[255:192]; FI;
    IF (imm8[3] = 0) THEN TMP_DEST[255:192] ← TMP_SRC1[191:128]; FI;
    IF (imm8[3] = 1) THEN TMP_DEST[255:192] ← TMP_SRC1[255:192]; FI;
FI;
IF VL >= 512
    IF (imm8[4] = 0) THEN TMP_DEST[319:256] ← TMP_SRC1[319:256]; FI;
    IF (imm8[4] = 1) THEN TMP_DEST[319:256] ← TMP_SRC1[383:320]; FI;
    IF (imm8[5] = 0) THEN TMP_DEST[383:320] ← TMP_SRC1[319:256]; FI;
    IF (imm8[5] = 1) THEN TMP_DEST[383:320] ← TMP_SRC1[383:320]; FI;
    IF (imm8[6] = 0) THEN TMP_DEST[447:384] ← TMP_SRC1[447:384]; FI;
    IF (imm8[6] = 1) THEN TMP_DEST[447:384] ← TMP_SRC1[511:448]; FI;
    IF (imm8[7] = 0) THEN TMP_DEST[511:448] ← TMP_SRC1[447:384]; FI;
    IF (imm8[7] = 1) THEN TMP_DEST[511:448] ← TMP_SRC1[511:448]; FI;
FI;
FOR j ← 0 TO KL-1
    i ← j * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+63:i] ← TMP_DEST[i+63:i]
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE                                ; zeroing-masking
                    DEST[i+63:i] ← 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] ←    0
```

VPERMILPD (256-bit immediate version)

```
IF (imm8[0] = 0) THEN DEST[63:0]←SRC1[63:0]
IF (imm8[0] = 1) THEN DEST[63:0]←SRC1[127:64]
IF (imm8[1] = 0) THEN DEST[127:64]←SRC1[63:0]
IF (imm8[1] = 1) THEN DEST[127:64]←SRC1[127:64]
IF (imm8[2] = 0) THEN DEST[191:128]←SRC1[191:128]
IF (imm8[2] = 1) THEN DEST[191:128]←SRC1[255:192]
IF (imm8[3] = 0) THEN DEST[255:192]←SRC1[191:128]
IF (imm8[3] = 1) THEN DEST[255:192]←SRC1[255:192]
DEST[MAXVL-1:256]←0
```

VPERMILPD (128-bit immediate version)

```
IF (imm8[0] = 0) THEN DEST[63:0]←SRC1[63:0]
IF (imm8[0] = 1) THEN DEST[63:0]←SRC1[127:64]
IF (imm8[1] = 0) THEN DEST[127:64]←SRC1[63:0]
IF (imm8[1] = 1) THEN DEST[127:64]←SRC1[127:64]
DEST[MAXVL-1:128]←0
```

VPERMILPD (EVEX variable versions)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j ← 0 TO KL-1
    i ← j * 64
    IF (EVEX.b = 1) AND (SRC2 *is memory*)
        THEN TMP_SRC2[i+63:i] ← SRC2[63:0];
        ELSE TMP_SRC2[i+63:i] ← SRC2[i+63:i];
    FI;
ENDFOR;

IF (TMP_SRC2[1] = 0) THEN TMP_DEST[63:0] ← SRC1[63:0]; FI;
IF (TMP_SRC2[1] = 1) THEN TMP_DEST[63:0] ← SRC1[127:64]; FI;
IF (TMP_SRC2[65] = 0) THEN TMP_DEST[127:64] ← SRC1[63:0]; FI;
IF (TMP_SRC2[65] = 1) THEN TMP_DEST[127:64] ← SRC1[127:64]; FI;
IF VL >= 256
    IF (TMP_SRC2[129] = 0) THEN TMP_DEST[191:128] ← SRC1[191:128]; FI;
    IF (TMP_SRC2[129] = 1) THEN TMP_DEST[191:128] ← SRC1[255:192]; FI;
    IF (TMP_SRC2[193] = 0) THEN TMP_DEST[255:192] ← SRC1[191:128]; FI;
    IF (TMP_SRC2[193] = 1) THEN TMP_DEST[255:192] ← SRC1[255:192]; FI;
FI;
IF VL >= 512
    IF (TMP_SRC2[257] = 0) THEN TMP_DEST[319:256] ← SRC1[319:256]; FI;
    IF (TMP_SRC2[257] = 1) THEN TMP_DEST[319:256] ← SRC1[383:320]; FI;
    IF (TMP_SRC2[321] = 0) THEN TMP_DEST[383:320] ← SRC1[319:256]; FI;
    IF (TMP_SRC2[321] = 1) THEN TMP_DEST[383:320] ← SRC1[383:320]; FI;
    IF (TMP_SRC2[385] = 0) THEN TMP_DEST[447:384] ← SRC1[447:384]; FI;
    IF (TMP_SRC2[385] = 1) THEN TMP_DEST[447:384] ← SRC1[511:448]; FI;
    IF (TMP_SRC2[449] = 0) THEN TMP_DEST[511:448] ← SRC1[447:384]; FI;
    IF (TMP_SRC2[449] = 1) THEN TMP_DEST[511:448] ← SRC1[511:448]; FI;
FI;

FOR j ← 0 TO KL-1
    i ← j * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+63:i] ← TMP_DEST[i+63:i]
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE                                ; zeroing-masking
                    DEST[i+63:i] ← 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] ←    0
```

VPERMILPD (256-bit variable version)

```
IF (SRC2[1] = 0) THEN DEST[63:0]←SRC1[63:0]
IF (SRC2[1] = 1) THEN DEST[63:0]←SRC1[127:64]
IF (SRC2[65] = 0) THEN DEST[127:64]←SRC1[63:0]
IF (SRC2[65] = 1) THEN DEST[127:64]←SRC1[127:64]
IF (SRC2[129] = 0) THEN DEST[191:128]←SRC1[191:128]
IF (SRC2[129] = 1) THEN DEST[191:128]←SRC1[255:192]
IF (SRC2[193] = 0) THEN DEST[255:192]←SRC1[191:128]
IF (SRC2[193] = 1) THEN DEST[255:192]←SRC1[255:192]
DEST[MAXVL-1:256]←0
```

VPERMILPD (128-bit variable version)

```
IF (SRC2[1] = 0) THEN DEST[63:0]←SRC1[63:0]
IF (SRC2[1] = 1) THEN DEST[63:0]←SRC1[127:64]
IF (SRC2[65] = 0) THEN DEST[127:64]←SRC1[63:0]
IF (SRC2[65] = 1) THEN DEST[127:64]←SRC1[127:64]
DEST[MAXVL-1:128]←0
```

Intel C/C++ Compiler Intrinsic Equivalent

VPERMILPD __m512d _mm512_permute_pd(__m512d a, int imm);
VPERMILPD __m512d _mm512_mask_permute_pd(__m512d s, __mmask8 k, __m512d a, int imm);
VPERMILPD __m512d _mm512_maskz_permute_pd(__mmask8 k, __m512d a, int imm);
VPERMILPD __m256d _mm256_mask_permute_pd(__m256d s, __mmask8 k, __m256d a, int imm);
VPERMILPD __m256d _mm256_maskz_permute_pd(__mmask8 k, __m256d a, int imm);
VPERMILPD __m128d _mm_mask_permute_pd(__m128d s, __mmask8 k, __m128d a, int imm);
VPERMILPD __m128d _mm_maskz_permute_pd(__mmask8 k, __m128d a, int imm);
VPERMILPD __m512d _mm512_permutevar_pd(__m512i i, __m512d a);
VPERMILPD __m512d _mm512_mask_permutevar_pd(__m512d s, __mmask8 k, __m512i i, __m512d a);
VPERMILPD __m512d _mm512_maskz_permutevar_pd(__mmask8 k, __m512i i, __m512d a);
VPERMILPD __m256d _mm256_mask_permutevar_pd(__m256d s, __mmask8 k, __m256d i, __m256d a);
VPERMILPD __m256d _mm256_maskz_permutevar_pd(__mmask8 k, __m256d i, __m256d a);
VPERMILPD __m128d _mm_mask_permutevar_pd(__m128d s, __mmask8 k, __m128d i, __m128d a);
VPERMILPD __m128d _mm_maskz_permutevar_pd(__mmask8 k, __m128d i, __m128d a);
VPERMILPD __m128d _mm_permute_pd (__m128d a, int control)
VPERMILPD __m256d _mm256_permute_pd (__m256d a, int control)
VPERMILPD __m128d _mm_permutevar_pd (__m128d a, __m128i control);
VPERMILPD __m256d _mm256_permutevar_pd (__m256d a, __m256i control);

SIMD Floating-Point Exceptions

None

Other Exceptions

Non-EVEX-encoded instruction, see Exceptions Type 4; additionally

#UD	If VEX.W = 1.

EVEX-encoded instruction, see Exceptions Type E4NF.

#UD	If either (E)VEX.vvvv != 1111B and with imm8.

VPERMILPS—Permute In-Lane of Quadruples of Single-Precision Floating-Point Values

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
VEX.NDS.128.66.0F38.W0 0C /r VPERMILPS xmm1, xmm2, xmm3/m128	A	V/V	AVX	Permute single-precision floating-point values in xmm2 using controls from xmm3/m128 and store result in xmm1.
VEX.128.66.0F3A.W0 04 /r ib VPERMILPS xmm1, xmm2/m128, imm8	B	V/V	AVX	Permute single-precision floating-point values in xmm2/m128 using controls from imm8 and store result in xmm1.
VEX.NDS.256.66.0F38.W0 0C /r VPERMILPS ymm1, ymm2, ymm3/m256	A	V/V	AVX	Permute single-precision floating-point values in ymm2 using controls from ymm3/m256 and store result in ymm1.
VEX.256.66.0F3A.W0 04 /r ib VPERMILPS ymm1, ymm2/m256, imm8	B	V/V	AVX	Permute single-precision floating-point values in ymm2/m256 using controls from imm8 and store result in ymm1.
EVEX.NDS.128.66.0F38.W0 0C /r VPERMILPS xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst	C	V/V	AVX512VL AVX512F	Permute single-precision floating-point values xmm2 using control from xmm3/m128/m32bcst and store the result in xmm1 using writemask k1.
EVEX.NDS.256.66.0F38.W0 0C /r VPERMILPS ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst	C	V/V	AVX512VL AVX512F	Permute single-precision floating-point values ymm2 using control from ymm3/m256/m32bcst and store the result in ymm1 using writemask k1.
EVEX.NDS.512.66.0F38.W0 0C /r VPERMILPS zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst	C	V/V	AVX512F	Permute single-precision floating-point values zmm2 using control from zmm3/m512/m32bcst and store the result in zmm1 using writemask k1.
EVEX.128.66.0F3A.W0 04 /r ib VPERMILPS xmm1 {k1}{z}, xmm2/m128/m32bcst, imm8	D	V/V	AVX512VL AVX512F	Permute single-precision floating-point values xmm2/m128/m32bcst using controls from imm8 and store the result in xmm1 using writemask k1.
EVEX.256.66.0F3A.W0 04 /r ib VPERMILPS ymm1 {k1}{z}, ymm2/m256/m32bcst, imm8	D	V/V	AVX512VL AVX512F	Permute single-precision floating-point values ymm2/m256/m32bcst using controls from imm8 and store the result in ymm1 using writemask k1.
EVEX.512.66.0F3A.W0 04 /r ibVPERMILPS zmm1 {k1}{z}, zmm2/m512/m32bcst, imm8	D	V/V	AVX512F	Permute single-precision floating-point values zmm2/m512/m32bcst using controls from imm8 and store the result in zmm1 using writemask k1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	NA	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA
B	NA	ModRM:reg (w)	ModRM:r/m (r)	NA	NA
C	Full Vector	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA
D	Full Vector	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

Description

(variable control version)

Permute quadruples of single-precision floating-point values in the first source operand (second operand), each quadruplet using a 2-bit control field in the corresponding dword element of the second source operand. Permuted results are stored in the destination operand (first operand).

The 2-bit control fields are located at the low two bits of each dword element (see Figure 5-26). Each control determines which of the source element in an input quadruple is selected for the destination element. Each quadruple of source elements must lie in the same 128-bit region as the destination.

EVEX version: The second source operand (third operand) is a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 32-bit memory location. Permuted results are written to the destination under the writemask.

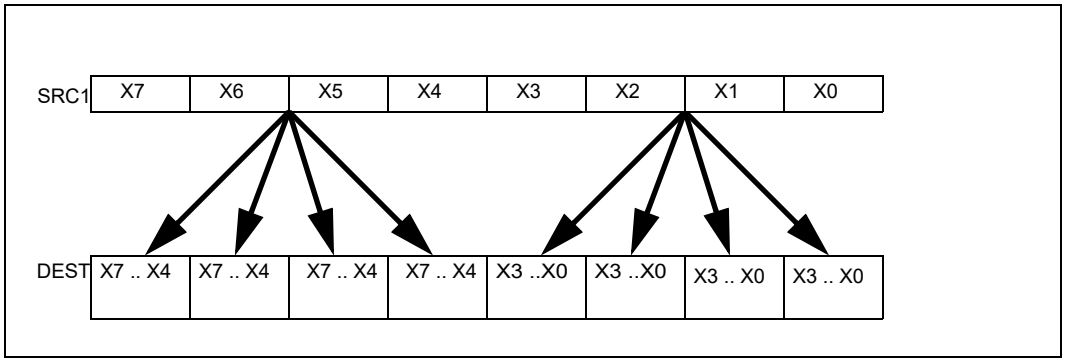


Figure 5-25. VPERMILPS Operation

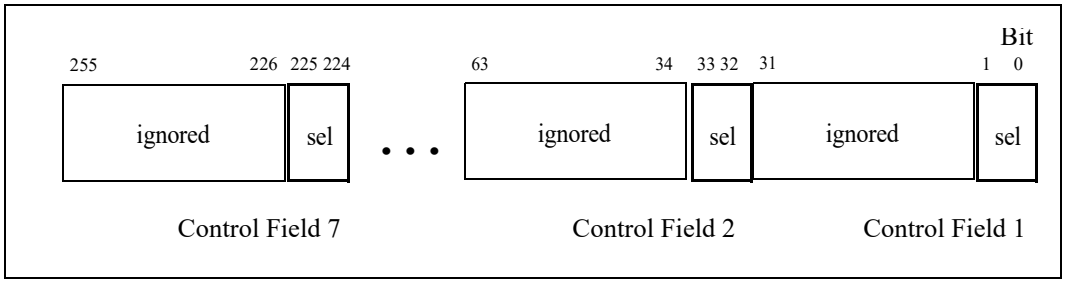


Figure 5-26. VPERMILPS Shuffle Control

(immediate control version)

Permute quadruples of single-precision floating-point values in the first source operand (second operand), each quadruplet using a 2-bit control field in the imm8 byte. Each 128-bit lane in the destination operand (first operand) use the four control fields of the same imm8 byte.

VEX version: The source operand is a YMM/XMM register or a 256/128-bit memory location and the destination operand is a YMM/XMM register.

EVEX version: The source operand (second operand) is a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 32-bit memory location. Permuted results are written to the destination under the writemask.

Note: For the imm8 version, VEX.vvvv and EVEX.vvvv are reserved and must be 1111b otherwise instruction will #UD.

Operation

```
Select4(SRC, control) {
CASE (control[1:0]) OF
    0:    TMP ←SRC[31:0];
    1:    TMP ←SRC[63:32];
    2:    TMP ←SRC[95:64];
    3:    TMP ←SRC[127:96];
ESAC;
RETURN TMP
}
```

VPERMILPS (EVEX immediate versions)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j ← 0 TO KL-1
    i ← j * 32
    IF (EVEX.b = 1) AND (SRC1 *is memory*)
        THEN TMP_SRC1[i+31:i] ← SRC1[31:0];
        ELSE TMP_SRC1[i+31:i] ← SRC1[i+31:i];
    FI;
ENDFOR;

TMP_DEST[31:0] ← Select4(TMP_SRC1[127:0], imm8[1:0]);
TMP_DEST[63:32] ← Select4(TMP_SRC1[127:0], imm8[3:2]);
TMP_DEST[95:64] ← Select4(TMP_SRC1[127:0], imm8[5:4]);
TMP_DEST[127:96] ← Select4(TMP_SRC1[127:0], imm8[7:6]); FI;
IF VL >= 256
    TMP_DEST[159:128] ← Select4(TMP_SRC1[255:128], imm8[1:0]); FI;
    TMP_DEST[191:160] ← Select4(TMP_SRC1[255:128], imm8[3:2]); FI;
    TMP_DEST[223:192] ← Select4(TMP_SRC1[255:128], imm8[5:4]); FI;
    TMP_DEST[255:224] ← Select4(TMP_SRC1[255:128], imm8[7:6]); FI;
FI;
IF VL >= 512
    TMP_DEST[287:256] ← Select4(TMP_SRC1[383:256], imm8[1:0]); FI;
    TMP_DEST[319:288] ← Select4(TMP_SRC1[383:256], imm8[3:2]); FI;
    TMP_DEST[351:320] ← Select4(TMP_SRC1[383:256], imm8[5:4]); FI;
    TMP_DEST[383:352] ← Select4(TMP_SRC1[383:256], imm8[7:6]); FI;
    TMP_DEST[415:384] ← Select4(TMP_SRC1[511:384], imm8[1:0]); FI;
    TMP_DEST[447:416] ← Select4(TMP_SRC1[511:384], imm8[3:2]); FI;
    TMP_DEST[479:448] ← Select4(TMP_SRC1[511:384], imm8[5:4]); FI;
    TMP_DEST[511:480] ← Select4(TMP_SRC1[511:384], imm8[7:6]); FI;
FI;
FOR j ← 0 TO KL-1
    i ← j * 32
    IF k1[j] OR *no writemask*
        THEN DEST[i+31:i] ← TMP_DEST[i+31:i]
        ELSE
            IF *merging-masking*
                THEN *DEST[i+31:i] remains unchanged*
                ELSE DEST[i+31:i] ← 0                    ;zeroing-masking
            FI;
    FI;
ENDFOR
DEST[MAXVL-1:VL] ←    0
```

VPERMILPS (256-bit immediate version)

```
DEST[31:0] ←Select4(SRC1[127:0], imm8[1:0]);
DEST[63:32] ←Select4(SRC1[127:0], imm8[3:2]);
DEST[95:64] ←Select4(SRC1[127:0], imm8[5:4]);
DEST[127:96] ←Select4(SRC1[127:0], imm8[7:6]);
DEST[159:128] ←Select4(SRC1[255:128], imm8[1:0]);
DEST[191:160] ←Select4(SRC1[255:128], imm8[3:2]);
DEST[223:192] ←Select4(SRC1[255:128], imm8[5:4]);
DEST[255:224] ←Select4(SRC1[255:128], imm8[7:6]);
```

VPERMILPS (128-bit immediate version)

```
DEST[31:0] ←Select4(SRC1[127:0], imm8[1:0]);
DEST[63:32] ←Select4(SRC1[127:0], imm8[3:2]);
DEST[95:64] ←Select4(SRC1[127:0], imm8[5:4]);
DEST[127:96] ←Select4(SRC1[127:0], imm8[7:6]);
DEST[MAXVL-1:128]←0
```

VPERMILPS (EVEX variable versions)

```
(KL, VL) = (16, 512)
FOR j ← 0 TO KL-1
    i ← j * 32
    IF (EVEX.b = 1) AND (SRC2 *is memory*)
        THEN TMP_SRC2[i+31:i] ← SRC2[31:0];
        ELSE TMP_SRC2[i+31:i] ← SRC2[i+31:i];
    FI;
ENDFOR;
TMP_DEST[31:0] ← Select4(SRC1[127:0], TMP_SRC2[1:0]);
TMP_DEST[63:32] ← Select4(SRC1[127:0], TMP_SRC2[33:32]);
TMP_DEST[95:64] ← Select4(SRC1[127:0], TMP_SRC2[65:64]);
TMP_DEST[127:96] ← Select4(SRC1[127:0], TMP_SRC2[97:96]);
IF VL >= 256
    TMP_DEST[159:128] ← Select4(SRC1[255:128], TMP_SRC2[129:128]);
    TMP_DEST[191:160] ← Select4(SRC1[255:128], TMP_SRC2[161:160]);
    TMP_DEST[223:192] ← Select4(SRC1[255:128], TMP_SRC2[193:192]);
    TMP_DEST[255:224] ← Select4(SRC1[255:128], TMP_SRC2[225:224]);
FI;
IF VL >= 512
    TMP_DEST[287:256] ← Select4(SRC1[383:256], TMP_SRC2[257:256]);
    TMP_DEST[319:288] ← Select4(SRC1[383:256], TMP_SRC2[289:288]);
    TMP_DEST[351:320] ← Select4(SRC1[383:256], TMP_SRC2[321:320]);
    TMP_DEST[383:352] ← Select4(SRC1[383:256], TMP_SRC2[353:352]);
    TMP_DEST[415:384] ← Select4(SRC1[511:384], TMP_SRC2[385:384]);
    TMP_DEST[447:416] ← Select4(SRC1[511:384], TMP_SRC2[417:416]);
    TMP_DEST[479:448] ← Select4(SRC1[511:384], TMP_SRC2[449:448]);
    TMP_DEST[511:480] ← Select4(SRC1[511:384], TMP_SRC2[481:480]);
FI;
FOR j ← 0 TO KL-1
    i ← j * 32
    IF k1[j] OR *no writemask*
        THEN DEST[i+31:i] ← TMP_DEST[i+31:i]
        ELSE
            IF *merging-masking*
                THEN *DEST[i+31:i] remains unchanged*
                ELSE DEST[i+31:i] ← 0                    ;zeroing-masking
```

```
            FI;
    FI;
ENDFOR
DEST[MAXVL-1:VL] ←  0
```

VPERMILPS (256-bit variable version)

```
DEST[31:0] ←Select4(SRC1[127:0], SRC2[1:0]);
DEST[63:32] ←Select4(SRC1[127:0], SRC2[33:32]);
DEST[95:64] ←Select4(SRC1[127:0], SRC2[65:64]);
DEST[127:96] ←Select4(SRC1[127:0], SRC2[97:96]);
DEST[159:128] ←Select4(SRC1[255:128], SRC2[129:128]);
DEST[191:160] ←Select4(SRC1[255:128], SRC2[161:160]);
DEST[223:192] ←Select4(SRC1[255:128], SRC2[193:192]);
DEST[255:224] ←Select4(SRC1[255:128], SRC2[225:224]);
DEST[MAXVL-1:256]←0
```

VPERMILPS (128-bit variable version)

```
DEST[31:0] ←Select4(SRC1[127:0], SRC2[1:0]);
DEST[63:32] ←Select4(SRC1[127:0], SRC2[33:32]);
DEST[95:64] ←Select4(SRC1[127:0], SRC2[65:64]);
DEST[127:96] ←Select4(SRC1[127:0], SRC2[97:96]);
DEST[MAXVL-1:128]←0
```

Intel C/C++ Compiler Intrinsic Equivalent

VPERMILPS __m512 _mm512_permute_ps(__m512 a, int imm);
VPERMILPS __m512 _mm512_mask_permute_ps(__m512 s, __mmask16 k, __m512 a, int imm);
VPERMILPS __m512 _mm512_maskz_permute_ps(__mmask16 k, __m512 a, int imm);
VPERMILPS __m256 _mm256_mask_permute_ps(__m256 s, __mmask8 k, __m256 a, int imm);
VPERMILPS __m256 _mm256_maskz_permute_ps(__mmask8 k, __m256 a, int imm);
VPERMILPS __m128 _mm_mask_permute_ps(__m128 s, __mmask8 k, __m128 a, int imm);
VPERMILPS __m128 _mm_maskz_permute_ps(__mmask8 k, __m128 a, int imm);
VPERMILPS __m512 _mm512_permutevar_ps(__m512i i, __m512 a);
VPERMILPS __m512 _mm512_mask_permutevar_ps(__m512 s, __mmask16 k, __m512i i, __m512 a);
VPERMILPS __m512 _mm512_maskz_permutevar_ps(__mmask16 k, __m512i i, __m512 a);
VPERMILPS __m256 _mm256_mask_permutevar_ps(__m256 s, __mmask8 k, __m256 i, __m256 a);
VPERMILPS __m256 _mm256_maskz_permutevar_ps(__mmask8 k, __m256 i, __m256 a);
VPERMILPS __m128 _mm_mask_permutevar_ps(__m128 s, __mmask8 k, __m128 i, __m128 a);
VPERMILPS __m128 _mm_maskz_permutevar_ps(__mmask8 k, __m128 i, __m128 a);
VPERMILPS __m128 _mm_permute_ps (__m128 a, int control);
VPERMILPS __m256 _mm256_permute_ps (__m256 a, int control);
VPERMILPS __m128 _mm_permutevar_ps (__m128 a, __m128i control);
VPERMILPS __m256 _mm256_permutevar_ps (__m256 a, __m256i control);

SIMD Floating-Point Exceptions

None

Other Exceptions

Non-EVEX-encoded instruction, see Exceptions Type 4;

#UD	If VEX.W = 1.

EVEX-encoded instruction, see Exceptions Type E4NF.

#UD	If either (E)VEX.vvvv != 1111B and with imm8.

VPERMPD—Permute Double-Precision Floating-Point Elements

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
VEX.256.66.0F3A.W1 01 /r ib VPERMPD ymm1, ymm2/m256, imm8	A	V/V	AVX2	Permute double-precision floating-point elements in ymm2/m256 using indices in imm8 and store the result in ymm1.
EVEX.256.66.0F3A.W1 01 /r ib VPERMPD ymm1 {k1}{z}, ymm2/m256/m64bcst, imm8	B	V/V	AVX512VL AVX512F	Permute double-precision floating-point elements in ymm2/m256/m64bcst using indexes in imm8 and store the result in ymm1 subject to writemask k1.
EVEX.512.66.0F3A.W1 01 /r ib VPERMPD zmm1 {k1}{z}, zmm2/m512/m64bcst, imm8	B	V/V	AVX512F	Permute double-precision floating-point elements in zmm2/m512/m64bcst using indices in imm8 and store the result in zmm1 subject to writemask k1.
EVEX.NDS.256.66.0F38.W1 16 /r VPERMPD ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst	C	V/V	AVX512VL AVX512F	Permute double-precision floating-point elements in ymm3/m256/m64bcst using indexes in ymm2 and store the result in ymm1 subject to writemask k1.
EVEX.NDS.512.66.0F38.W1 16 /r VPERMPD zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst	C	V/V	AVX512F	Permute double-precision floating-point elements in zmm3/m512/m64bcst using indices in zmm2 and store the result in zmm1 subject to writemask k1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	NA	ModRM:reg (w)	ModRM:r/m (r)	Imm8	NA
B	Full Vector	ModRM:reg (w)	ModRM:r/m (r)	Imm8	NA
C	Full Vector	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA

Description

The imm8 version: Copies quadword elements of double-precision floating-point values from the source operand (the second operand) to the destination operand (the first operand) according to the indices specified by the immediate operand (the third operand). Each two-bit value in the immediate byte selects a qword element in the source operand.

VEX version: The source operand can be a YMM register or a memory location. Bits (MAXVL-1:256) of the corresponding destination register are zeroed.

In EVEX.512 encoded version, The elements in the destination are updated using the writemask k1 and the imm8 bits are reused as control bits for the upper 256-bit half when the control bits are coming from immediate. The source operand can be a ZMM register, a 512-bit memory location or a 512-bit vector broadcasted from a 64-bit memory location.

The imm8 versions: VEX.vvvv and EVEX.vvvv are reserved and must be 1111b otherwise instructions will #UD.

The vector control version: Copies quadword elements of double-precision floating-point values from the second source operand (the third operand) to the destination operand (the first operand) according to the indices in the first source operand (the second operand). The first 3 bits of each 64 bit element in the index operand selects which quadword in the second source operand to copy. The first and second operands are ZMM registers, the third operand can be a ZMM register, a 512-bit memory location or a 512-bit vector broadcasted from a 64-bit memory location. The elements in the destination are updated using the writemask k1.

Note that this instruction permits a qword in the source operand to be copied to multiple locations in the destination operand.

If VPERMPD is encoded with VEX.L= 0, an attempt to execute the instruction encoded with VEX.L= 0 will cause an #UD exception.

Operation

VPERMPD (EVEX - imm8 control forms)

```
(KL, VL) = (4, 256), (8, 512)
FOR j ← 0 TO KL-1
    i ← j * 64
    IF (EVEX.b = 1) AND (SRC *is memory*)
        THEN TMP_SRC[i+63:i] ← SRC[63:0];
        ELSE TMP_SRC[i+63:i] ← SRC[i+63:i];
    FI;
ENDFOR;

TMP_DEST[63:0] ← (TMP_SRC[256:0] >> (IMM8[1:0] * 64))[63:0];
TMP_DEST[127:64] ← (TMP_SRC[256:0] >> (IMM8[3:2] * 64))[63:0];
TMP_DEST[191:128] ← (TMP_SRC[256:0] >> (IMM8[5:4] * 64))[63:0];
TMP_DEST[255:192] ← (TMP_SRC[256:0] >> (IMM8[7:6] * 64))[63:0];
IF VL >= 512
    TMP_DEST[319:256] ← (TMP_SRC[511:256] >> (IMM8[1:0] * 64))[63:0];
    TMP_DEST[383:320] ← (TMP_SRC[511:256] >> (IMM8[3:2] * 64))[63:0];
    TMP_DEST[447:384] ← (TMP_SRC[511:256] >> (IMM8[5:4] * 64))[63:0];
    TMP_DEST[511:448] ← (TMP_SRC[511:256] >> (IMM8[7:6] * 64))[63:0];
FI;
FOR j ← 0 TO KL-1
    i ← j * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+63:i] ← TMP_DEST[i+63:i]
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE                                ; zeroing-masking
                    DEST[i+63:i] ← 0                        ;zeroing-masking
            FI;
    FI;
ENDFOR
DEST[MAXVL-1:VL] ←    0
```

VPERMPD (EVEX - vector control forms)

```
(KL, VL) = (4, 256), (8, 512)
FOR j ← 0 TO KL-1
    i ← j * 64
    IF (EVEX.b = 1) AND (SRC2 *is memory*)
        THEN TMP_SRC2[i+63:i] ← SRC2[63:0];
        ELSE TMP_SRC2[i+63:i] ← SRC2[i+63:i];
    FI;
ENDFOR;

IF VL = 256
    TMP_DEST[63:0] ← (TMP_SRC2[255:0] >> (SRC1[1:0] * 64))[63:0];
    TMP_DEST[127:64] ← (TMP_SRC2[255:0] >> (SRC1[65:64] * 64))[63:0];
    TMP_DEST[191:128] ← (TMP_SRC2[255:0] >> (SRC1[129:128] * 64))[63:0];
    TMP_DEST[255:192] ← (TMP_SRC2[255:0] >> (SRC1[193:192] * 64))[63:0];
FI;
IF VL = 512
    TMP_DEST[63:0] ← (TMP_SRC2[511:0] >> (SRC1[2:0] * 64))[63:0];
```

```
    TMP_DEST[127:64] ← (TMP_SRC2[511:0] >> (SRC1[66:64] * 64))[63:0];
    TMP_DEST[191:128] ← (TMP_SRC2[511:0] >> (SRC1[130:128] * 64))[63:0];
    TMP_DEST[255:192] ← (TMP_SRC2[511:0] >> (SRC1[194:192] * 64))[63:0];
    TMP_DEST[319:256] ← (TMP_SRC2[511:0] >> (SRC1[258:256] * 64))[63:0];
    TMP_DEST[383:320] ← (TMP_SRC2[511:0] >> (SRC1[322:320] * 64))[63:0];
    TMP_DEST[447:384] ← (TMP_SRC2[511:0] >> (SRC1[386:384] * 64))[63:0];
    TMP_DEST[511:448] ← (TMP_SRC2[511:0] >> (SRC1[450:448] * 64))[63:0];
FI;
FOR j ← 0 TO KL-1
    i ← j * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+63:i] ← TMP_DEST[i+63:i]
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE                            ; zeroing-masking
                    DEST[i+63:i] ← 0                    ;zeroing-masking
            FI;
    FI;
ENDFOR
DEST[MAXVL-1:VL] ←  0
```

VPERMPD (VEX.256 encoded version)

```
DEST[63:0] ←(SRC[255:0] >> (IMM8[1:0] * 64))[63:0];
DEST[127:64] ←(SRC[255:0] >> (IMM8[3:2] * 64))[63:0];
DEST[191:128] ←(SRC[255:0] >> (IMM8[5:4] * 64))[63:0];
DEST[255:192] ←(SRC[255:0] >> (IMM8[7:6] * 64))[63:0];
DEST[MAXVL-1:256] ←  0
```

Intel C/C++ Compiler Intrinsic Equivalent

VPERMPD __m512d _mm512_permutex_pd(__m512d a, int imm);
VPERMPD __m512d _mm512_mask_permutex_pd(__m512d s, __mmask16 k, __m512d a, int imm);
VPERMPD __m512d _mm512_maskz_permutex_pd(__mmask16 k, __m512d a, int imm);
VPERMPD __m512d _mm512_permutexvar_pd(__m512i i, __m512d a);
VPERMPD __m512d _mm512_mask_permutexvar_pd(__m512d s, __mmask16 k, __m512i i, __m512d a);
VPERMPD __m512d _mm512_maskz_permutexvar_pd(__mmask16 k, __m512i i, __m512d a);
VPERMPD __m256d _mm256_permutex_epi64(__m256d a, int imm);
VPERMPD __m256d _mm256_mask_permutex_epi64(__m256i s, __mmask8 k, __m256d a, int imm);
VPERMPD __m256d _mm256_maskz_permutex_epi64(__mmask8 k, __m256d a, int imm);
VPERMPD __m256d _mm256_permutexvar_epi64(__m256i i, __m256d a);
VPERMPD __m256d _mm256_mask_permutexvar_epi64(__m256i s, __mmask8 k, __m256i i, __m256d a);
VPERMPD __m256d _mm256_maskz_permutexvar_epi64(__mmask8 k, __m256i i, __m256d a);

SIMD Floating-Point Exceptions

None

Other Exceptions

Non-EVEX-encoded instruction, see Exceptions Type 4; additionally

#UD	If VEX.L = 0.
	If VEX.vvvv != 1111B.

EVEX-encoded instruction, see Exceptions Type E4NF.

#UD	If encoded with EVEX.128.
	If EVEX.vvvv != 1111B and with imm8.

VPERMPS—Permute Single-Precision Floating-Point Elements

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
VEX.256.66.0F38.W0 16 /r VPERMPS ymm1, ymm2, ymm3/m256	A	V/V	AVX2	Permute single-precision floating-point elements in ymm3/m256 using indices in ymm2 and store the result in ymm1.
EVEX.NDS.256.66.0F38.W0 16 /r VPERMPS ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst	B	V/V	AVX512VL AVX512F	Permute single-precision floating-point elements in ymm3/m256/m32bcst using indexes in ymm2 and store the result in ymm1 subject to write mask k1.
EVEX.NDS.512.66.0F38.W0 16 /r VPERMPS zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst	B	V/V	AVX512F	Permute single-precision floating-point values in zmm3/m512/m32bcst using indices in zmm2 and store the result in zmm1 subject to write mask k1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	NA	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA
B	Full Vector	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA

Description

Copies doubleword elements of single-precision floating-point values from the second source operand (the third operand) to the destination operand (the first operand) according to the indices in the first source operand (the second operand). Note that this instruction permits a doubleword in the source operand to be copied to more than one location in the destination operand.

VEX.256 versions: The first and second operands are YMM registers, the third operand can be a YMM register or memory location. Bits (MAXVL-1:256) of the corresponding destination register are zeroed.

EVEX encoded version: The first and second operands are ZMM registers, the third operand can be a ZMM register, a 512-bit memory location or a 512-bit vector broadcasted from a 32-bit memory location. The elements in the destination are updated using the writemask k1.

If VPERMPS is encoded with VEX.L= 0, an attempt to execute the instruction encoded with VEX.L= 0 will cause an #UD exception.

Operation

VPERMPS (EVEX forms)

```
(KL, VL) (8, 256),= (16, 512)
FOR j ← 0 TO KL-1
    i ← j * 64
    IF (EVEX.b = 1) AND (SRC2 *is memory*)
        THEN TMP_SRC2[i+31:i] ← SRC2[31:0];
        ELSE TMP_SRC2[i+31:i] ← SRC2[i+31:i];
    FI;
ENDFOR;

IF VL = 256
    TMP_DEST[31:0] ← (TMP_SRC2[255:0] >> (SRC1[2:0] * 32))[31:0];
    TMP_DEST[63:32] ← (TMP_SRC2[255:0] >> (SRC1[34:32] * 32))[31:0];
    TMP_DEST[95:64] ← (TMP_SRC2[255:0] >> (SRC1[66:64] * 32))[31:0];
    TMP_DEST[127:96] ← (TMP_SRC2[255:0] >> (SRC1[98:96] * 32))[31:0];
    TMP_DEST[159:128] ← (TMP_SRC2[255:0] >> (SRC1[130:128] * 32))[31:0];
    TMP_DEST[191:160] ← (TMP_SRC2[255:0] >> (SRC1[162:160] * 32))[31:0];
    TMP_DEST[223:192] ← (TMP_SRC2[255:0] >> (SRC1[193:192] * 32))[31:0];
    TMP_DEST[255:224] ← (TMP_SRC2[255:0] >> (SRC1[226:224] * 32))[31:0];
```

```
FI;
IF VL = 512
    TMP_DEST[31:0] ← (TMP_SRC2[511:0] >> (SRC1[3:0] * 32))[31:0];
    TMP_DEST[63:32] ← (TMP_SRC2[511:0] >> (SRC1[35:32] * 32))[31:0];
    TMP_DEST[95:64] ← (TMP_SRC2[511:0] >> (SRC1[67:64] * 32))[31:0];
    TMP_DEST[127:96] ← (TMP_SRC2[511:0] >> (SRC1[99:96] * 32))[31:0];
    TMP_DEST[159:128] ← (TMP_SRC2[511:0] >> (SRC1[131:128] * 32))[31:0];
    TMP_DEST[191:160] ← (TMP_SRC2[511:0] >> (SRC1[163:160] * 32))[31:0];
    TMP_DEST[223:192] ← (TMP_SRC2[511:0] >> (SRC1[195:192] * 32))[31:0];
    TMP_DEST[255:224] ← (TMP_SRC2[511:0] >> (SRC1[227:224] * 32))[31:0];
    TMP_DEST[287:256] ← (TMP_SRC2[511:0] >> (SRC1[259:256] * 32))[31:0];
    TMP_DEST[319:288] ← (TMP_SRC2[511:0] >> (SRC1[291:288] * 32))[31:0];
    TMP_DEST[351:320] ← (TMP_SRC2[511:0] >> (SRC1[323:320] * 32))[31:0];
    TMP_DEST[383:352] ← (TMP_SRC2[511:0] >> (SRC1[355:352] * 32))[31:0];
    TMP_DEST[415:384] ← (TMP_SRC2[511:0] >> (SRC1[387:384] * 32))[31:0];
    TMP_DEST[447:416] ← (TMP_SRC2[511:0] >> (SRC1[419:416] * 32))[31:0];
    TMP_DEST[479:448] ←(TMP_SRC2[511:0] >> (SRC1[451:448] * 32))[31:0];
    TMP_DEST[511:480] ← (TMP_SRC2[511:0] >> (SRC1[483:480] * 32))[31:0];
FI;
FOR j ← 0 TO KL-1
    i ← j * 32
    IF k1[j] OR *no writemask*
        THEN DEST[i+31:i] ← TMP_DEST[i+31:i]
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE                                ; zeroing-masking
                    DEST[i+31:i] ← 0                        ;zeroing-masking
            FI;
    FI;
ENDFOR
DEST[MAXVL-1:VL] ←    0
```

VPERMPS (VEX.256 encoded version)

```
DEST[31:0] ←(SRC2[255:0] >> (SRC1[2:0] * 32))[31:0];
DEST[63:32] ←(SRC2[255:0] >> (SRC1[34:32] * 32))[31:0];
DEST[95:64] ←(SRC2[255:0] >> (SRC1[66:64] * 32))[31:0];
DEST[127:96] ←(SRC2[255:0] >> (SRC1[98:96] * 32))[31:0];
DEST[159:128] ←(SRC2[255:0] >> (SRC1[130:128] * 32))[31:0];
DEST[191:160] ←(SRC2[255:0] >> (SRC1[162:160] * 32))[31:0];
DEST[223:192] ←(SRC2[255:0] >> (SRC1[194:192] * 32))[31:0];
DEST[255:224] ←(SRC2[255:0] >> (SRC1[226:224] * 32))[31:0];
DEST[MAXVL-1:256] ←    0
```

Intel C/C++ Compiler Intrinsic Equivalent

VPERMPS __m512 _mm512_permutexvar_ps(__m512i i, __m512 a);
VPERMPS __m512 _mm512_mask_permutexvar_ps(__m512 s, __mmask16 k, __m512i i, __m512 a);
VPERMPS __m512 _mm512_maskz_permutexvar_ps(__mmask16 k, __m512i i, __m512 a);
VPERMPS __m256 _mm256_permutexvar_ps(__m256 i, __m256 a);
VPERMPS __m256 _mm256_mask_permutexvar_ps(__m256 s, __mmask8 k, __m256 i, __m256 a);
VPERMPS __m256 _mm256_maskz_permutexvar_ps(__mmask8 k, __m256 i, __m256 a);

SIMD Floating-Point Exceptions

None

Other Exceptions

Non-EVEX-encoded instruction, see Exceptions Type 4; additionally

#UD	If VEX.L = 0.

EVEX-encoded instruction, see Exceptions Type E4NF.

VPERMQ—Qwords Element Permutation

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
VEX.256.66.0F3A.W1 00 /r ib VPERMQ ymm1, ymm2/m256, imm8	A	V/V	AVX2	Permute qwords in ymm2/m256 using indices in imm8 and store the result in ymm1.
EVEX.256.66.0F3A.W1 00 /r ib VPERMQ ymm1 {k1}{z}, ymm2/m256/m64bcst, imm8	B	V/V	AVX512VL AVX512F	Permute qwords in ymm2/m256/m64bcst using indexes in imm8 and store the result in ymm1.
EVEX.512.66.0F3A.W1 00 /r ib VPERMQ zmm1 {k1}{z}, zmm2/m512/m64bcst, imm8	B	V/V	AVX512F	Permute qwords in zmm2/m512/m64bcst using indices in imm8 and store the result in zmm1.
EVEX.NDS.256.66.0F38.W1 36 /r VPERMQ ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst	C	V/V	AVX512VL AVX512F	Permute qwords in ymm3/m256/m64bcst using indexes in ymm2 and store the result in ymm1.
EVEX.NDS.512.66.0F38.W1 36 /r VPERMQ zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst	C	V/V	AVX512F	Permute qwords in zmm3/m512/m64bcst using indices in zmm2 and store the result in zmm1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	NA	ModRM:reg (w)	ModRM:r/m (r)	Imm8	NA
B	Full Vector	ModRM:reg (w)	ModRM:r/m (r)	Imm8	NA
C	Full Vector	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA

Description

The imm8 version: Copies quadwords from the source operand (the second operand) to the destination operand (the first operand) according to the indices specified by the immediate operand (the third operand). Each two-bit value in the immediate byte selects a qword element in the source operand.

VEX version: The source operand can be a YMM register or a memory location. Bits (MAXVL-1:256) of the corresponding destination register are zeroed.

In EVEX.512 encoded version, The elements in the destination are updated using the writemask k1 and the imm8 bits are reused as control bits for the upper 256-bit half when the control bits are coming from immediate. The source operand can be a ZMM register, a 512-bit memory location or a 512-bit vector broadcasted from a 64-bit memory location.

Immediate control versions: VEX.vvvv and EVEX.vvvv are reserved and must be 1111b otherwise instructions will #UD.

The vector control version: Copies quadwords from the second source operand (the third operand) to the destination operand (the first operand) according to the indices in the first source operand (the second operand). The first 3 bits of each 64 bit element in the index operand selects which quadword in the second source operand to copy. The first and second operands are ZMM registers, the third operand can be a ZMM register, a 512-bit memory location or a 512-bit vector broadcasted from a 64-bit memory location. The elements in the destination are updated using the writemask k1.

Note that this instruction permits a qword in the source operand to be copied to multiple locations in the destination operand.

If VPERMPQ is encoded with VEX.L= 0 or EVEX.128, an attempt to execute the instruction will cause an #UD exception.

Operation

VPERMQ (EVEX - imm8 control forms)

```
(KL, VL) = (4, 256), (8, 512)
FOR j ← 0 TO KL-1
    i ← j * 64
    IF (EVEX.b = 1) AND (SRC *is memory*)
        THEN TMP_SRC[i+63:i] ← SRC[63:0];
        ELSE TMP_SRC[i+63:i] ← SRC[i+63:i];
    FI;
ENDFOR;
    TMP_DEST[63:0] ← (TMP_SRC[255:0] >> (IMM8[1:0] * 64))[63:0];
    TMP_DEST[127:64] ← (TMP_SRC[255:0] >> (IMM8[3:2] * 64))[63:0];
    TMP_DEST[191:128] ← (TMP_SRC[255:0] >> (IMM8[5:4] * 64))[63:0];
    TMP_DEST[255:192] ← (TMP_SRC[255:0] >> (IMM8[7:6] * 64))[63:0];
IF VL >= 512
    TMP_DEST[319:256] ← (TMP_SRC[511:256] >> (IMM8[1:0] * 64))[63:0];
    TMP_DEST[383:320] ← (TMP_SRC[511:256] >> (IMM8[3:2] * 64))[63:0];
    TMP_DEST[447:384] ← (TMP_SRC[511:256] >> (IMM8[5:4] * 64))[63:0];
    TMP_DEST[511:448] ← (TMP_SRC[511:256] >> (IMM8[7:6] * 64))[63:0];
FI;
FOR j ← 0 TO KL-1
    i ← j * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+63:i] ← TMP_DEST[i+63:i]
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE                                ; zeroing-masking
                    DEST[i+63:i] ← 0                    ;zeroing-masking
            FI;
    FI;
ENDFOR
DEST[MAXVL-1:VL] ←   0
```

VPERMQ (EVEX - vector control forms)

```
(KL, VL) = (4, 256), (8, 512)
FOR j ← 0 TO KL-1
    i ← j * 64
    IF (EVEX.b = 1) AND (SRC2 *is memory*)
        THEN TMP_SRC2[i+63:i] ← SRC2[63:0];
        ELSE TMP_SRC2[i+63:i] ← SRC2[i+63:i];
    FI;
ENDFOR;
IF VL = 256
    TMP_DEST[63:0] ← (TMP_SRC2[255:0] >> (SRC1[1:0] * 64))[63:0];
    TMP_DEST[127:64] ← (TMP_SRC2[255:0] >> (SRC1[65:64] * 64))[63:0];
    TMP_DEST[191:128] ← (TMP_SRC2[255:0] >> (SRC1[129:128] * 64))[63:0];
    TMP_DEST[255:192] ← (TMP_SRC2[255:0] >> (SRC1[193:192] * 64))[63:0];
FI;
IF VL = 512
    TMP_DEST[63:0] ← (TMP_SRC2[511:0] >> (SRC1[2:0] * 64))[63:0];
    TMP_DEST[127:64] ← (TMP_SRC2[511:0] >> (SRC1[66:64] * 64))[63:0];
    TMP_DEST[191:128] ← (TMP_SRC2[511:0] >> (SRC1[130:128] * 64))[63:0];
    TMP_DEST[255:192] ← (TMP_SRC2[511:0] >> (SRC1[194:192] * 64))[63:0];
```

```
    TMP_DEST[319:256] ← (TMP_SRC2[511:0] >> (SRC1[258:256] * 64))[63:0];
    TMP_DEST[383:320] ← (TMP_SRC2[511:0] >> (SRC1[322:320] * 64))[63:0];
    TMP_DEST[447:384] ← (TMP_SRC2[511:0] >> (SRC1[386:384] * 64))[63:0];
    TMP_DEST[511:448] ← (TMP_SRC2[511:0] >> (SRC1[450:448] * 64))[63:0];
FI;
FOR j ← 0 TO KL-1
    i ← j * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+63:i] ← TMP_DEST[i+63:i]
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE                            ; zeroing-masking
                    DEST[i+63:i] ← 0                    ;zeroing-masking
            FI;
    FI;
ENDFOR
DEST[MAXVL-1:VL] ←  0
```

VPERMQ (VEX.256 encoded version)

```
DEST[63:0] ←(SRC[255:0] >> (IMM8[1:0] * 64))[63:0];
DEST[127:64] ←(SRC[255:0] >> (IMM8[3:2] * 64))[63:0];
DEST[191:128] ←(SRC[255:0] >> (IMM8[5:4] * 64))[63:0];
DEST[255:192] ←(SRC[255:0] >> (IMM8[7:6] * 64))[63:0];
DEST[MAXVL-1:256] ←  0
```

Intel C/C++ Compiler Intrinsic Equivalent

VPERMQ __m512i _mm512_permutex_epi64(__m512i a, int imm);
VPERMQ __m512i _mm512_mask_permutex_epi64(__m512i s, __mmask8 k, __m512i a, int imm);
VPERMQ __m512i _mm512_maskz_permutex_epi64(__mmask8 k, __m512i a, int imm);
VPERMQ __m512i _mm512_permutexvar_epi64(__m512i a, __m512i b);
VPERMQ __m512i _mm512_mask_permutexvar_epi64(__m512i s, __mmask8 k, __m512i a, __m512i b);
VPERMQ __m512i _mm512_maskz_permutexvar_epi64(__mmask8 k, __m512i a, __m512i b);
VPERMQ __m256i _mm256_permutex_epi64(__m256i a, int imm);
VPERMQ __m256i _mm256_mask_permutex_epi64(__m256i s, __mmask8 k, __m256i a, int imm);
VPERMQ __m256i _mm256_maskz_permutex_epi64(__mmask8 k, __m256i a, int imm);
VPERMQ __m256i _mm256_permutexvar_epi64(__m256i a, __m256i b);
VPERMQ __m256i _mm256_mask_permutexvar_epi64(__m256i s, __mmask8 k, __m256i a, __m256i b);
VPERMQ __m256i _mm256_maskz_permutexvar_epi64(__mmask8 k, __m256i a, __m256i b);

SIMD Floating-Point Exceptions

None

Other Exceptions

Non-EVEX-encoded instruction, see Exceptions Type 4; additionally

#UD	If VEX.L = 0.
	If VEX.vvvv != 1111B.

EVEX-encoded instruction, see Exceptions Type E4NF.

#UD	If encoded with EVEX.128.
	If EVEX.vvvv != 1111B and with imm8.

VPERMT2W/D/Q/PS/PD—Full Permute from Two Tables Overwriting one Table

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.DDS.128.66.0F38.W1 7D /r VPERMT2W xmm1 {k1}{z}, xmm2, xmm3/m128	A	V/V	AVX512VL AVX512BW	Permute word integers from two tables in xmm3/m128 and xmm1 using indexes in xmm2 and store the result in xmm1 using writemask k1.
EVEX.DDS.256.66.0F38.W1 7D /r VPERMT2W ymm1 {k1}{z}, ymm2, ymm3/m256	A	V/V	AVX512VL AVX512BW	Permute word integers from two tables in ymm3/m256 and ymm1 using indexes in ymm2 and store the result in ymm1 using writemask k1.
EVEX.DDS.512.66.0F38.W1 7D /r VPERMT2W zmm1 {k1}{z}, zmm2, zmm3/m512	A	V/V	AVX512BW	Permute word integers from two tables in zmm3/m512 and zmm1 using indexes in zmm2 and store the result in zmm1 using writemask k1.
EVEX.DDS.128.66.0F38.W0 7E /r VPERMT2D xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst	B	V/V	AVX512VL AVX512F	Permute double-words from two tables in xmm3/m128/m32bcst and xmm1 using indexes in xmm2 and store the result in xmm1 using writemask k1.
EVEX.DDS.256.66.0F38.W0 7E /r VPERMT2D ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst	B	V/V	AVX512VL AVX512F	Permute double-words from two tables in ymm3/m256/m32bcst and ymm1 using indexes in ymm2 and store the result in ymm1 using writemask k1.
EVEX.DDS.512.66.0F38.W0 7E /r VPERMT2D zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst	B	V/V	AVX512F	Permute double-words from two tables in zmm3/m512/m32bcst and zmm1 using indices in zmm2 and store the result in zmm1 using writemask k1.
EVEX.DDS.128.66.0F38.W1 7E /r VPERMT2Q xmm1 {k1}{z}, xmm2, xmm3/m128/m64bcst	B	V/V	AVX512VL AVX512F	Permute quad-words from two tables in xmm3/m128/m64bcst and xmm1 using indexes in xmm2 and store the result in xmm1 using writemask k1.
EVEX.DDS.256.66.0F38.W1 7E /r VPERMT2Q ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst	B	V/V	AVX512VL AVX512F	Permute quad-words from two tables in ymm3/m256/m64bcst and ymm1 using indexes in ymm2 and store the result in ymm1 using writemask k1.
EVEX.DDS.512.66.0F38.W1 7E /r VPERMT2Q zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst	B	V/V	AVX512F	Permute quad-words from two tables in zmm3/m512/m64bcst and zmm1 using indices in zmm2 and store the result in zmm1 using writemask k1.
EVEX.DDS.128.66.0F38.W0 7F /r VPERMT2PS xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst	B	V/V	AVX512VL AVX512F	Permute single-precision FP values from two tables in xmm3/m128/m32bcst and xmm1 using indexes in xmm2 and store the result in xmm1 using writemask k1.
EVEX.DDS.256.66.0F38.W0 7F /r VPERMT2PS ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst	B	V/V	AVX512VL AVX512F	Permute single-precision FP values from two tables in ymm3/m256/m32bcst and ymm1 using indexes in ymm2 and store the result in ymm1 using writemask k1.
EVEX.DDS.512.66.0F38.W0 7F /r VPERMT2PS zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst	B	V/V	AVX512F	Permute single-precision FP values from two tables in zmm3/m512/m32bcst and zmm1 using indices in zmm2 and store the result in zmm1 using writemask k1.
EVEX.DDS.128.66.0F38.W1 7F /r VPERMT2PD xmm1 {k1}{z}, xmm2, xmm3/m128/m64bcst	B	V/V	AVX512VL AVX512F	Permute double-precision FP values from two tables in xmm3/m128/m64bcst and xmm1 using indexes in xmm2 and store the result in xmm1 using writemask k1.
EVEX.DDS.256.66.0F38.W1 7F /r VPERMT2PD ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst	B	V/V	AVX512VL AVX512F	Permute double-precision FP values from two tables in ymm3/m256/m64bcst and ymm1 using indexes in ymm2 and store the result in ymm1 using writemask k1.
EVEX.DDS.512.66.0F38.W1 7F /r VPERMT2PD zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst	B	V/V	AVX512F	Permute double-precision FP values from two tables in zmm3/m512/m64bcst and zmm1 using indices in zmm2 and store the result in zmm1 using writemask k1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Full Mem	ModRM:reg (r,w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA
B	Full	ModRM:reg (r, w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA

Description

Permutes 16-bit/32-bit/64-bit values in the first operand and the third operand (the second source operand) using indices in the second operand (the first source operand) to select elements from the first and third operands. The selected elements are written to the destination operand (the first operand) according to the writemask k1.

The first and second operands are ZMM/YMM/XMM registers. The second operand contains input indices to select elements from the two input tables in the 1st and 3rd operands. The first operand is also the destination of the result.

D/Q/PS/PD element versions: The second source operand can be a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 32/64-bit memory location. Broadcast from the low 32/64-bit memory location is performed if EVEX.b and the id bit for table selection are set (selecting table_2).

Dword/PS versions: The id bit for table selection is bit 4/3/2, depending on VL=512, 256, 128. Bits [3:0]/[2:0]/[1:0] of each element in the input index vector select an element within the two source operands, If the id bit is 0, table_1 (the first source) is selected; otherwise the second source operand is selected.

Qword/PD versions: The id bit for table selection is bit 3/2/1, and bits [2:0]/[1:0] /bit 0 selects element within each input table.

Word element versions: The second source operand can be a ZMM/YMM/XMM register, or a 512/256/128-bit memory location. The id bit for table selection is bit 5/4/3, and bits [4:0]/[3:0]/[2:0] selects element within each input table.

Note that these instructions permit a 16-bit/32-bit/64-bit value in the source operands to be copied to more than one location in the destination operand. Note also that in this case, the same index can be reused for example for a second iteration, while the table elements being permuted are overwritten.

Bits (MAXVL-1:256/128) of the destination are zeroed for VL=256,128.

Operation

VPERMT2W (EVEX encoded versions)

```
(KL, VL) = (8, 128), (16, 256), (32, 512)
IF VL = 128
    id ← 2
FI;
IF VL = 256
    id ← 3
FI;
IF VL = 512
    id ← 4
FI;
TMP_DEST← DEST
FOR j ← 0 TO KL-1
    i ← j * 16
    off ← 16*SRC1[i+id:i]
    IF k1[j] OR *no writemask*
        THEN
            DEST[i+15:i]=SRC1[i+id+1] ? SRC2[off+15:off]
                    : TMP_DEST[off+15:off]
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+15:i] remains unchanged*
                ELSE                                ; zeroing-masking
```

```
                        DEST[i+15:i] ← 0
                FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0
```

VPERMT2D/VPERMT2PS (EVEX encoded versions)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
IF VL = 128
    id ← 1
FI;
IF VL = 256
    id ← 2
FI;
IF VL = 512
    id ← 3
FI;
TMP_DEST← DEST
FOR j ← 0 TO KL-1
    i ← j * 32
    off ← 32*SRC1[i+id:i]
    IF k1[j] OR *no writemask*
        THEN
            IF (EVEX.b = 1) AND (SRC2 *is memory*)
                THEN
                    DEST[i+31:i] ← SRC1[i+id+1] ? SRC2[31:0]
                    : TMP_DEST[off+31:off]
            ELSE
                DEST[i+31:i] ← SRC1[i+id+1] ? SRC2[off+31:off]
                    : TMP_DEST[off+31:off]
            FI
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE                                ; zeroing-masking
                    DEST[i+31:i] ← 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0
```

VPERMT2Q/VPERMT2PD (EVEX encoded versions)

```
(KL, VL) = (2, 128), (4, 256), (8 512)
IF VL = 128
    id ← 0
FI;
IF VL = 256
    id ← 1
FI;
IF VL = 512
    id ← 2
FI;
TMP_DEST← DEST
FOR j ← 0 TO KL-1
```

```
    i ← j * 64
    off ← 64*SRC1[i+id:i]
    IF k1[j] OR *no writemask*
        THEN
            IF (EVEX.b = 1) AND (SRC2 *is memory*)
                THEN
                    DEST[i+63:i] ← SRC1[i+id+1] ? SRC2[63:0]
                    : TMP_DEST[off+63:off]
            ELSE
                DEST[i+63:i] ← SRC1[i+id+1] ? SRC2[off+63:off]
                    : TMP_DEST[off+63:off]
            FI
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE                                ; zeroing-masking
                    DEST[i+63:i] ← 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VPERMT2D __m512i _mm512_permutex2var_epi32(__m512i a, __m512i idx, __m512i b);
VPERMT2D __m512i _mm512_mask_permutex2var_epi32(__m512i a, __mmask16 k, __m512i idx, __m512i b);
VPERMT2D __m512i _mm512_mask2_permutex2var_epi32(__m512i a, __m512i idx, __mmask16 k, __m512i b);
VPERMT2D __m512i _mm512_maskz_permutex2var_epi32(__mmask16 k, __m512i a, __m512i idx, __m512i b);
VPERMT2D __m256i _mm256_permutex2var_epi32(__m256i a, __m256i idx, __m256i b);
VPERMT2D __m256i _mm256_mask_permutex2var_epi32(__m256i a, __mmask8 k, __m256i idx, __m256i b);
VPERMT2D __m256i _mm256_mask2_permutex2var_epi32(__m256i a, __m256i idx, __mmask8 k, __m256i b);
VPERMT2D __m256i _mm256_maskz_permutex2var_epi32(__mmask8 k, __m256i a, __m256i idx, __m256i b);
VPERMT2D __m128i _mm_permutex2var_epi32(__m128i a, __m128i idx, __m128i b);
VPERMT2D __m128i _mm_mask_permutex2var_epi32(__m128i a, __mmask8 k, __m128i idx, __m128i b);
VPERMT2D __m128i _mm_mask2_permutex2var_epi32(__m128i a, __m128i idx, __mmask8 k, __m128i b);
VPERMT2D __m128i _mm_maskz_permutex2var_epi32(__mmask8 k, __m128i a, __m128i idx, __m128i b);
VPERMT2PD __m512d _mm512_permutex2var_pd(__m512d a, __m512i idx, __m512d b);
VPERMT2PD __m512d _mm512_mask_permutex2var_pd(__m512d a, __mmask8 k, __m512i idx, __m512d b);
VPERMT2PD __m512d _mm512_mask2_permutex2var_pd(__m512d a, __m512i idx, __mmask8 k, __m512d b);
VPERMT2PD __m512d _mm512_maskz_permutex2var_pd(__mmask8 k, __m512d a, __m512i idx, __m512d b);
VPERMT2PD __m256d _mm256_permutex2var_pd(__m256d a, __m256i idx, __m256d b);
VPERMT2PD __m256d _mm256_mask_permutex2var_pd(__m256d a, __mmask8 k, __m256i idx, __m256d b);
VPERMT2PD __m256d _mm256_mask2_permutex2var_pd(__m256d a, __m256i idx, __mmask8 k, __m256d b);
VPERMT2PD __m256d _mm256_maskz_permutex2var_pd(__mmask8 k, __m256d a, __m256i idx, __m256d b);
VPERMT2PD __m128d _mm_permutex2var_pd(__m128d a, __m128i idx, __m128d b);
VPERMT2PD __m128d _mm_mask_permutex2var_pd(__m128d a, __mmask8 k, __m128i idx, __m128d b);
VPERMT2PD __m128d _mm_mask2_permutex2var_pd(__m128d a, __m128i idx, __mmask8 k, __m128d b);
VPERMT2PD __m128d _mm_maskz_permutex2var_pd(__mmask8 k, __m128d a, __m128i idx, __m128d b);
VPERMT2PS __m512 _mm512_permutex2var_ps(__m512 a, __m512i idx, __m512 b);
VPERMT2PS __m512 _mm512_mask_permutex2var_ps(__m512 a, __mmask16 k, __m512i idx, __m512 b);
VPERMT2PS __m512 _mm512_mask2_permutex2var_ps(__m512 a, __m512i idx, __mmask16 k, __m512 b);
VPERMT2PS __m512 _mm512_maskz_permutex2var_ps(__mmask16 k, __m512 a, __m512i idx, __m512 b);

VPERMT2PS __m256 _mm256_permutex2var_ps(__m256 a, __m256i idx, __m256 b);
VPERMT2PS __m256 _mm256_mask_permutex2var_ps(__m256 a, __mmask8 k, __m256i idx, __m256 b);
VPERMT2PS __m256 _mm256_mask2_permutex2var_ps(__m256 a, __m256i idx, __mmask8 k, __m256 b);
VPERMT2PS __m256 _mm256_maskz_permutex2var_ps(__mmask8 k, __m256 a, __m256i idx, __m256 b);
VPERMT2PS __m128 _mm_permutex2var_ps(__m128 a, __m128i idx, __m128 b);
VPERMT2PS __m128 _mm_mask_permutex2var_ps(__m128 a, __mmask8 k, __m128i idx, __m128 b);
VPERMT2PS __m128 _mm_mask2_permutex2var_ps(__m128 a, __m128i idx, __mmask8 k, __m128 b);
VPERMT2PS __m128 _mm_maskz_permutex2var_ps(__mmask8 k, __m128 a, __m128i idx, __m128 b);
VPERMT2Q __m512i _mm512_permutex2var_epi64(__m512i a, __m512i idx, __m512i b);
VPERMT2Q __m512i _mm512_mask_permutex2var_epi64(__m512i a, __mmask8 k, __m512i idx, __m512i b);
VPERMT2Q __m512i _mm512_mask2_permutex2var_epi64(__m512i a, __m512i idx, __mmask8 k, __m512i b);
VPERMT2Q __m512i _mm512_maskz_permutex2var_epi64(__mmask8 k, __m512i a, __m512i idx, __m512i b);
VPERMT2Q __m256i _mm256_permutex2var_epi64(__m256i a, __m256i idx, __m256i b);
VPERMT2Q __m256i _mm256_mask_permutex2var_epi64(__m256i a, __mmask8 k, __m256i idx, __m256i b);
VPERMT2Q __m256i _mm256_mask2_permutex2var_epi64(__m256i a, __m256i idx, __mmask8 k, __m256i b);
VPERMT2Q __m256i _mm256_maskz_permutex2var_epi64(__mmask8 k, __m256i a, __m256i idx, __m256i b);
VPERMT2Q __m128i _mm_permutex2var_epi64(__m128i a, __m128i idx, __m128i b);
VPERMT2Q __m128i _mm_mask_permutex2var_epi64(__m128i a, __mmask8 k, __m128i idx, __m128i b);
VPERMT2Q __m128i _mm_mask2_permutex2var_epi64(__m128i a, __m128i idx, __mmask8 k, __m128i b);
VPERMT2Q __m128i _mm_maskz_permutex2var_epi64(__mmask8 k, __m128i a, __m128i idx, __m128i b);
VPERMT2W __m512i _mm512_permutex2var_epi16(__m512i a, __m512i idx, __m512i b);
VPERMT2W __m512i _mm512_mask_permutex2var_epi16(__m512i a, __mmask32 k, __m512i idx, __m512i b);
VPERMT2W __m512i _mm512_mask2_permutex2var_epi16(__m512i a, __m512i idx, __mmask32 k, __m512i b);
VPERMT2W __m512i _mm512_maskz_permutex2var_epi16(__mmask32 k, __m512i a, __m512i idx, __m512i b);
VPERMT2W __m256i _mm256_permutex2var_epi16(__m256i a, __m256i idx, __m256i b);
VPERMT2W __m256i _mm256_mask_permutex2var_epi16(__m256i a, __mmask16 k, __m256i idx, __m256i b);
VPERMT2W __m256i _mm256_mask2_permutex2var_epi16(__m256i a, __m256i idx, __mmask16 k, __m256i b);
VPERMT2W __m256i _mm256_maskz_permutex2var_epi16(__mmask16 k, __m256i a, __m256i idx, __m256i b);
VPERMT2W __m128i _mm_permutex2var_epi16(__m128i a, __m128i idx, __m128i b);
VPERMT2W __m128i _mm_mask_permutex2var_epi16(__m128i a, __mmask8 k, __m128i idx, __m128i b);
VPERMT2W __m128i _mm_mask2_permutex2var_epi16(__m128i a, __m128i idx, __mmask8 k, __m128i b);
VPERMT2W __m128i _mm_maskz_permutex2var_epi16(__mmask8 k, __m128i a, __m128i idx, __m128i b);

SIMD Floating-Point Exceptions

None.

Other Exceptions

VPERMT2D/Q/PS/PD: See Exceptions Type E4NF.
VPERMT2W: See Exceptions Type E4NF.nb.

VPEXPANDD—Load Sparse Packed Doubleword Integer Values from Dense Memory / Register

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.128.66.0F38.W0 89 /r VPEXPANDD xmm1 {k1}{z}, xmm2/m128	A	V/V	AVX512VL AVX512F	Expand packed double-word integer values from xmm2/m128 to xmm1 using writemask k1.
EVEX.256.66.0F38.W0 89 /r VPEXPANDD ymm1 {k1}{z}, ymm2/m256	A	V/V	AVX512VL AVX512F	Expand packed double-word integer values from ymm2/m256 to ymm1 using writemask k1.
EVEX.512.66.0F38.W0 89 /r VPEXPANDD zmm1 {k1}{z}, zmm2/m512	A	V/V	AVX512F	Expand packed double-word integer values from zmm2/m512 to zmm1 using writemask k1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Tuple1 Scalar	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

Description

Expand (load) up to 16 contiguous doubleword integer values of the input vector in the source operand (the second operand) to sparse elements in the destination operand (the first operand), selected by the writemask k1. The destination operand is a ZMM register, the source operand can be a ZMM register or memory location.

The input vector starts from the lowest element in the source operand. The opmask register k1 selects the destination elements (a partial vector or sparse elements if less than 8 elements) to be replaced by the ascending elements in the input vector. Destination elements not selected by the writemask k1 are either unmodified or zeroed, depending on EVEX.z.

Note: EVEX.vvvv is reserved and must be 1111b otherwise instructions will #UD.

Note that the compressed displacement assumes a pre-scaling (N) corresponding to the size of one single element instead of the size of the full vector.

Operation

VPEXPANDD (EVEX encoded versions)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
k ← 0
FOR j ← 0 TO KL-1
    i ← j * 32
    IF k1[j] OR *no writemask*
        THEN
            DEST[i+31:i] ← SRC[k+31:k];
            k ← k + 32
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE                                ; zeroing-masking
                    DEST[i+31:i] ← 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] ←  0
```

Intel C/C++ Compiler Intrinsic Equivalent

VPEXPANDD __m512i _mm512_mask_expandloadu_epi32(__m512i s, __mmask16 k, void * a);
VPEXPANDD __m512i _mm512_maskz_expandloadu_epi32(__mmask16 k, void * a);
VPEXPANDD __m512i _mm512_mask_expand_epi32(__m512i s, __mmask16 k, __m512i a);
VPEXPANDD __m512i _mm512_maskz_expand_epi32(__mmask16 k, __m512i a);
VPEXPANDD __m256i _mm256_mask_expandloadu_epi32(__m256i s, __mmask8 k, void * a);
VPEXPANDD __m256i _mm256_maskz_expandloadu_epi32(__mmask8 k, void * a);
VPEXPANDD __m256i _mm256_mask_expand_epi32(__m256i s, __mmask8 k, __m256i a);
VPEXPANDD __m256i _mm256_maskz_expand_epi32(__mmask8 k, __m256i a);
VPEXPANDD __m128i _mm_mask_expandloadu_epi32(__m128i s, __mmask8 k, void * a);
VPEXPANDD __m128i _mm_maskz_expandloadu_epi32(__mmask8 k, void * a);
VPEXPANDD __m128i _mm_mask_expand_epi32(__m128i s, __mmask8 k, __m128i a);
VPEXPANDD __m128i _mm_maskz_expand_epi32(__mmask8 k, __m128i a);

SIMD Floating-Point Exceptions

None

Other Exceptions

EVEX-encoded instruction, see Exceptions Type E4.nb.

#UD	If EVEX.vvvv != 1111B.

VPEXPANDQ—Load Sparse Packed Quadword Integer Values from Dense Memory / Register

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.128.66.0F38.W1 89 /r VPEXPANDQ xmm1 {k1}{z}, xmm2/m128	A	V/V	AVX512VL AVX512F	Expand packed quad-word integer values from xmm2/m128 to xmm1 using writemask k1.
EVEX.256.66.0F38.W1 89 /r VPEXPANDQ ymm1 {k1}{z}, ymm2/m256	A	V/V	AVX512VL AVX512F	Expand packed quad-word integer values from ymm2/m256 to ymm1 using writemask k1.
EVEX.512.66.0F38.W1 89 /r VPEXPANDQ zmm1 {k1}{z}, zmm2/m512	A	V/V	AVX512F	Expand packed quad-word integer values from zmm2/m512 to zmm1 using writemask k1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Tuple1 Scalar	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

Description

Expand (load) up to 8 quadword integer values from the source operand (the second operand) to sparse elements in the destination operand (the first operand), selected by the writemask k1. The destination operand is a ZMM register, the source operand can be a ZMM register or memory location.

The input vector starts from the lowest element in the source operand. The opmask register k1 selects the destination elements (a partial vector or sparse elements if less than 8 elements) to be replaced by the ascending elements in the input vector. Destination elements not selected by the writemask k1 are either unmodified or zeroed, depending on EVEX.z.

Note: EVEX.vvvv is reserved and must be 1111b otherwise instructions will #UD.

Note that the compressed displacement assumes a pre-scaling (N) corresponding to the size of one single element instead of the size of the full vector.

Operation

VPEXPANDQ (EVEX encoded versions)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
k ← 0
FOR j ← 0 TO KL-1
    i ← j * 64
    IF k1[j] OR *no writemask*
        THEN
            DEST[i+63:i] ← SRC[k+63:k];
            k ← k + 64
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE                                ; zeroing-masking
                    THEN DEST[i+63:i] ← 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] ←   0
```

Intel C/C++ Compiler Intrinsic Equivalent

VPEXPANDQ __m512i _mm512_mask_expandloadu_epi64(__m512i s, __mmask8 k, void * a);
VPEXPANDQ __m512i _mm512_maskz_expandloadu_epi64(__mmask8 k, void * a);
VPEXPANDQ __m512i _mm512_mask_expand_epi64(__m512i s, __mmask8 k, __m512i a);
VPEXPANDQ __m512i _mm512_maskz_expand_epi64(__mmask8 k, __m512i a);
VPEXPANDQ __m256i _mm256_mask_expandloadu_epi64(__m256i s, __mmask8 k, void * a);
VPEXPANDQ __m256i _mm256_maskz_expandloadu_epi64(__mmask8 k, void * a);
VPEXPANDQ __m256i _mm256_mask_expand_epi64(__m256i s, __mmask8 k, __m256i a);
VPEXPANDQ __m256i _mm256_maskz_expand_epi64(__mmask8 k, __m256i a);
VPEXPANDQ __m128i _mm_mask_expandloadu_epi64(__m128i s, __mmask8 k, void * a);
VPEXPANDQ __m128i _mm_maskz_expandloadu_epi64(__mmask8 k, void * a);
VPEXPANDQ __m128i _mm_mask_expand_epi64(__m128i s, __mmask8 k, __m128i a);
VPEXPANDQ __m128i _mm_maskz_expand_epi64(__mmask8 k, __m128i a);

SIMD Floating-Point Exceptions

None

Other Exceptions

EVEX-encoded instruction, see Exceptions Type E4.nb.

#UD	If EVEX.vvvv != 1111B.

VPLZCNTD/Q—Count the Number of Leading Zero Bits for Packed Dword, Packed Qword Values

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.128.66.0F38.W0 44 /r VPLZCNTD xmm1 {k1}{z}, xmm2/m128/m32bcst	A	V/V	AVX512VL AVX512CD	Count the number of leading zero bits in each dword element of xmm2/m128/m32bcst using writemask k1.
EVEX.256.66.0F38.W0 44 /r VPLZCNTD ymm1 {k1}{z}, ymm2/m256/m32bcst	A	V/V	AVX512VL AVX512CD	Count the number of leading zero bits in each dword element of ymm2/m256/m32bcst using writemask k1.
EVEX.512.66.0F38.W0 44 /r VPLZCNTD zmm1 {k1}{z}, zmm2/m512/m32bcst	A	V/V	AVX512CD	Count the number of leading zero bits in each dword element of zmm2/m512/m32bcst using writemask k1.
EVEX.128.66.0F38.W1 44 /r VPLZCNTQ xmm1 {k1}{z}, xmm2/m128/m64bcst	A	V/V	AVX512VL AVX512CD	Count the number of leading zero bits in each qword element of xmm2/m128/m64bcst using writemask k1.
EVEX.256.66.0F38.W1 44 /r VPLZCNTQ ymm1 {k1}{z}, ymm2/m256/m64bcst	A	V/V	AVX512VL AVX512CD	Count the number of leading zero bits in each qword element of ymm2/m256/m64bcst using writemask k1.
EVEX.512.66.0F38.W1 44 /r VPLZCNTQ zmm1 {k1}{z}, zmm2/m512/m64bcst	A	V/V	AVX512CD	Count the number of leading zero bits in each qword element of zmm2/m512/m64bcst using writemask k1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Full Vector	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

Description

Counts the number of leading most significant zero bits in each dword or qword element of the source operand (the second operand) and stores the results in the destination register (the first operand) according to the writemask. If an element is zero, the result for that element is the operand size of the element.

EVEX.512 encoded version: The source operand is a ZMM register, a 512-bit memory location, or a 512-bit vector broadcasted from a 32/64-bit memory location. The destination operand is a ZMM register, conditionally updated using writemask k1.

EVEX.256 encoded version: The source operand is a YMM register, a 256-bit memory location, or a 256-bit vector broadcasted from a 32/64-bit memory location. The destination operand is a YMM register, conditionally updated using writemask k1.

EVEX.128 encoded version: The source operand is a XMM register, a 128-bit memory location, or a 128-bit vector broadcasted from a 32/64-bit memory location. The destination operand is a XMM register, conditionally updated using writemask k1.

EVEX.vvvv is reserved and must be 1111b otherwise instructions will #UD.

Operation

VPLZCNTD

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j ← 0 TO KL-1
    i ← j*32
    IF MaskBit(j) OR *no writemask*
        THEN
            temp ← 32
            DEST[i+31:i] ← 0
            WHILE (temp > 0) AND (SRC[i+temp-1] = 0)
            DO
                temp ← temp – 1
                DEST[i+31:i] ← DEST[i+31:i] + 1
            OD
        ELSE
          IF *merging-masking*
            THEN *DEST[i+31:i] remains unchanged*
            ELSE DEST[i+31:i] ← 0
          FI
    FI
ENDFOR
DEST[MAXVL-1:VL] ← 0
```

VPLZCNTQ

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j ← 0 TO KL-1
    i ← j*64
    IF MaskBit(j) OR *no writemask*
        THEN
            temp ← 64
            DEST[i+63:i] ← 0
            WHILE (temp > 0) AND (SRC[i+temp-1] = 0)
            DO
                temp ← temp – 1
                DEST[i+63:i] ← DEST[i+63:i] + 1
            OD
        ELSE
          IF *merging-masking*
            THEN *DEST[i+63:i] remains unchanged*
            ELSE DEST[i+63:i] ← 0
          FI
    FI
ENDFOR
DEST[MAXVL-1:VL] ← 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VPLZCNTD __m512i _mm512_lzcnt_epi32(__m512i a);
VPLZCNTD __m512i _mm512_mask_lzcnt_epi32(__m512i s, __mmask16 m, __m512i a);
VPLZCNTD __m512i _mm512_maskz_lzcnt_epi32(__mmask16 m, __m512i a);
VPLZCNTQ __m512i _mm512_lzcnt_epi64(__m512i a);
VPLZCNTQ __m512i _mm512_mask_lzcnt_epi64(__m512i s, __mmask8 m, __m512i a);
VPLZCNTQ __m512i _mm512_maskz_lzcnt_epi64(__mmask8 m, __m512i a);
VPLZCNTD __m256i _mm256_lzcnt_epi32(__m256i a);
VPLZCNTD __m256i _mm256_mask_lzcnt_epi32(__m256i s, __mmask8 m, __m256i a);
VPLZCNTD __m256i _mm256_maskz_lzcnt_epi32(__mmask8 m, __m256i a);
VPLZCNTQ __m256i _mm256_lzcnt_epi64(__m256i a);
VPLZCNTQ __m256i _mm256_mask_lzcnt_epi64(__m256i s, __mmask8 m, __m256i a);
VPLZCNTQ __m256i _mm256_maskz_lzcnt_epi64(__mmask8 m, __m256i a);
VPLZCNTD __m128i _mm_lzcnt_epi32(__m128i a);
VPLZCNTD __m128i _mm_mask_lzcnt_epi32(__m128i s, __mmask8 m, __m128i a);
VPLZCNTD __m128i _mm_maskz_lzcnt_epi32(__mmask8 m, __m128i a);
VPLZCNTQ __m128i _mm_lzcnt_epi64(__m128i a);
VPLZCNTQ __m128i _mm_mask_lzcnt_epi64(__m128i s, __mmask8 m, __m128i a);
VPLZCNTQ __m128i _mm_maskz_lzcnt_epi64(__mmask8 m, __m128i a);

SIMD Floating-Point Exceptions

None

Other Exceptions

EVEX-encoded instruction, see Exceptions Type E4.

VPMASKMOV — Conditional SIMD Integer Packed Loads and Stores

Opcode/ Instruction	Op/ En	64/32 -bit Mode	CPUID Feature Flag	Description
VEX.NDS.128.66.0F38.W0 8C /r VPMASKMOVD *xmm1, xmm2, m128*	RVM	V/V	AVX2	Conditionally load dword values from *m128* using mask in *xmm2* and store in *xmm1*.
VEX.NDS.256.66.0F38.W0 8C /r VPMASKMOVD *ymm1, ymm2, m256*	RVM	V/V	AVX2	Conditionally load dword values from *m256* using mask in *ymm2* and store in *ymm1*.
VEX.NDS.128.66.0F38.W1 8C /r VPMASKMOVQ *xmm1, xmm2, m128*	RVM	V/V	AVX2	Conditionally load qword values from *m128* using mask in *xmm2* and store in *xmm1*.
VEX.NDS.256.66.0F38.W1 8C /r VPMASKMOVQ *ymm1, ymm2, m256*	RVM	V/V	AVX2	Conditionally load qword values from *m256* using mask in *ymm2* and store in *ymm1*.
VEX.NDS.128.66.0F38.W0 8E /r VPMASKMOVD *m128, xmm1, xmm2*	MVR	V/V	AVX2	Conditionally store dword values from *xmm2* using mask in *xmm1*.
VEX.NDS.256.66.0F38.W0 8E /r VPMASKMOVD *m256, ymm1, ymm2*	MVR	V/V	AVX2	Conditionally store dword values from *ymm2* using mask in *ymm1*.
VEX.NDS.128.66.0F38.W1 8E /r VPMASKMOVQ *m128, xmm1, xmm2*	MVR	V/V	AVX2	Conditionally store qword values from *xmm2* using mask in *xmm1*.
VEX.NDS.256.66.0F38.W1 8E /r VPMASKMOVQ *m256, ymm1, ymm2*	MVR	V/V	AVX2	Conditionally store qword values from *ymm2* using mask in *ymm1*.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RVM	ModRM:reg (w)	VEX.vvvv	ModRM:r/m (r)	NA
MVR	ModRM:r/m (w)	VEX.vvvv	ModRM:reg (r)	NA

Description

Conditionally moves packed data elements from the second source operand into the corresponding data element of the destination operand, depending on the mask bits associated with each data element. The mask bits are specified in the first source operand.

The mask bit for each data element is the most significant bit of that element in the first source operand. If a mask is 1, the corresponding data element is copied from the second source operand to the destination operand. If the mask is 0, the corresponding data element is set to zero in the load form of these instructions, and unmodified in the store form.

The second source operand is a memory address for the load form of these instructions. The destination operand is a memory address for the store form of these instructions. The other operands are either XMM registers (for VEX.128 version) or YMM registers (for VEX.256 version).

Faults occur only due to mask-bit required memory accesses that caused the faults. Faults will not occur due to referencing any memory location if the corresponding mask bit for that memory location is 0. For example, no faults will be detected if the mask bits are all zero.

Unlike previous MASKMOV instructions (MASKMOVQ and MASKMOVDQU), a nontemporal hint is not applied to these instructions.

Instruction behavior on alignment check reporting with mask bits of less than all 1s are the same as with mask bits of all 1s.

VMASKMOV should not be used to access memory mapped I/O as the ordering of the individual loads or stores it does is implementation specific.

In cases where mask bits indicate data should not be loaded or stored paging A and D bits will be set in an implementation dependent way. However, A and D bits are always set for pages where data is actually loaded/stored.

Note: for load forms, the first source (the mask) is encoded in VEX.vvvv; the second source is encoded in rm_field, and the destination register is encoded in reg_field.

Note: for store forms, the first source (the mask) is encoded in VEX.vvvv; the second source register is encoded in reg_field, and the destination memory location is encoded in rm_field.

Operation

VPMASKMOVD - 256-bit load

```
DEST[31:0] ← IF (SRC1[31]) Load_32(mem) ELSE 0
DEST[63:32] ← IF (SRC1[63]) Load_32(mem + 4) ELSE 0
DEST[95:64] ← IF (SRC1[95]) Load_32(mem + 8) ELSE 0
DEST[127:96] ← IF (SRC1[127]) Load_32(mem + 12) ELSE 0
DEST[159:128] ← IF (SRC1[159]) Load_32(mem + 16) ELSE 0
DEST[191:160] ← IF (SRC1[191]) Load_32(mem + 20) ELSE 0
DEST[223:192] ← IF (SRC1[223]) Load_32(mem + 24) ELSE 0
DEST[255:224] ← IF (SRC1[255]) Load_32(mem + 28) ELSE 0
```

VPMASKMOVD -128-bit load

```
DEST[31:0] ← IF (SRC1[31]) Load_32(mem) ELSE 0
DEST[63:32] ← IF (SRC1[63]) Load_32(mem + 4) ELSE 0
DEST[95:64] ← IF (SRC1[95]) Load_32(mem + 8) ELSE 0
DEST[127:97] ← IF (SRC1[127]) Load_32(mem + 12) ELSE 0
DEST[MAXVL-1:128] ← 0
```

VPMASKMOVQ - 256-bit load

```
DEST[63:0] ← IF (SRC1[63]) Load_64(mem) ELSE 0
DEST[127:64] ← IF (SRC1[127]) Load_64(mem + 8) ELSE 0
DEST[195:128] ← IF (SRC1[191]) Load_64(mem + 16) ELSE 0
DEST[255:196] ← IF (SRC1[255]) Load_64(mem + 24) ELSE 0
```

VPMASKMOVQ - 128-bit load

```
DEST[63:0] ← IF (SRC1[63]) Load_64(mem) ELSE 0
DEST[127:64] ← IF (SRC1[127]) Load_64(mem + 16) ELSE 0
DEST[MAXVL-1:128] ← 0
```

VPMASKMOVD - 256-bit store

```
IF (SRC1[31]) DEST[31:0] ← SRC2[31:0]
IF (SRC1[63]) DEST[63:32] ← SRC2[63:32]
IF (SRC1[95]) DEST[95:64] ← SRC2[95:64]
IF (SRC1[127]) DEST[127:96] ← SRC2[127:96]
IF (SRC1[159]) DEST[159:128] ←SRC2[159:128]
IF (SRC1[191]) DEST[191:160] ← SRC2[191:160]
IF (SRC1[223]) DEST[223:192] ← SRC2[223:192]
IF (SRC1[255]) DEST[255:224] ← SRC2[255:224]
```

VPMASKMOVD - 128-bit store
IF (SRC1[31]) DEST[31:0] ← SRC2[31:0]
IF (SRC1[63]) DEST[63:32] ← SRC2[63:32]
IF (SRC1[95]) DEST[95:64] ← SRC2[95:64]
IF (SRC1[127]) DEST[127:96] ← SRC2[127:96]

VPMASKMOVQ - 256-bit store
IF (SRC1[63]) DEST[63:0] ← SRC2[63:0]
IF (SRC1[127]) DEST[127:64] ←SRC2[127:64]
IF (SRC1[191]) DEST[191:128] ← SRC2[191:128]
IF (SRC1[255]) DEST[255:192] ← SRC2[255:192]

VPMASKMOVQ - 128-bit store
IF (SRC1[63]) DEST[63:0] ← SRC2[63:0]
IF (SRC1[127]) DEST[127:64] ←SRC2[127:64]

Intel C/C++ Compiler Intrinsic Equivalent

VPMASKMOVD: __m256i _mm256_maskload_epi32(int const *a, __m256i mask)

VPMASKMOVD: void _mm256_maskstore_epi32(int *a, __m256i mask, __m256i b)

VPMASKMOVQ: __m256i _mm256_maskload_epi64(__int64 const *a, __m256i mask);

VPMASKMOVQ: void _mm256_maskstore_epi64(__int64 *a, __m256i mask, __m256d b);

VPMASKMOVD: __m128i _mm_maskload_epi32(int const *a, __m128i mask)

VPMASKMOVD: void _mm_maskstore_epi32(int *a, __m128i mask, __m128 b)

VPMASKMOVQ: __m128i _mm_maskload_epi64(__int cont *a, __m128i mask);

VPMASKMOVQ: void _mm_maskstore_epi64(__int64 *a, __m128i mask, __m128i b);

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type 6 (No AC# reported for any mask bit combinations).

VPMOVB2M/VPMOVW2M/VPMOVD2M/VPMOVQ2M—Convert a Vector Register to a Mask

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.128.F3.0F38.W0 29 /r VPMOVB2M k1, xmm1	RM	V/V	AVX512VL AVX512BW	Sets each bit in k1 to 1 or 0 based on the value of the most significant bit of the corresponding byte in XMM1.
EVEX.256.F3.0F38.W0 29 /r VPMOVB2M k1, ymm1	RM	V/V	AVX512VL AVX512BW	Sets each bit in k1 to 1 or 0 based on the value of the most significant bit of the corresponding byte in YMM1.
EVEX.512.F3.0F38.W0 29 /r VPMOVB2M k1, zmm1	RM	V/V	AVX512BW	Sets each bit in k1 to 1 or 0 based on the value of the most significant bit of the corresponding byte in ZMM1.
EVEX.128.F3.0F38.W1 29 /r VPMOVW2M k1, xmm1	RM	V/V	AVX512VL AVX512BW	Sets each bit in k1 to 1 or 0 based on the value of the most significant bit of the corresponding word in XMM1.
EVEX.256.F3.0F38.W1 29 /r VPMOVW2M k1, ymm1	RM	V/V	AVX512VL AVX512BW	Sets each bit in k1 to 1 or 0 based on the value of the most significant bit of the corresponding word in YMM1.
EVEX.512.F3.0F38.W1 29 /r VPMOVW2M k1, zmm1	RM	V/V	AVX512BW	Sets each bit in k1 to 1 or 0 based on the value of the most significant bit of the corresponding word in ZMM1.
EVEX.128.F3.0F38.W0 39 /r VPMOVD2M k1, xmm1	RM	V/V	AVX512VL AVX512DQ	Sets each bit in k1 to 1 or 0 based on the value of the most significant bit of the corresponding doubleword in XMM1.
EVEX.256.F3.0F38.W0 39 /r VPMOVD2M k1, ymm1	RM	V/V	AVX512VL AVX512DQ	Sets each bit in k1 to 1 or 0 based on the value of the most significant bit of the corresponding doubleword in YMM1.
EVEX.512.F3.0F38.W0 39 /r VPMOVD2M k1, zmm1	RM	V/V	AVX512DQ	Sets each bit in k1 to 1 or 0 based on the value of the most significant bit of the corresponding doubleword in ZMM1.
EVEX.128.F3.0F38.W1 39 /r VPMOVQ2M k1, xmm1	RM	V/V	AVX512VL AVX512DQ	Sets each bit in k1 to 1 or 0 based on the value of the most significant bit of the corresponding quadword in XMM1.
EVEX.256.F3.0F38.W1 39 /r VPMOVQ2M k1, ymm1	RM	V/V	AVX512VL AVX512DQ	Sets each bit in k1 to 1 or 0 based on the value of the most significant bit of the corresponding quadword in YMM1.
EVEX.512.F3.0F38.W1 39 /r VPMOVQ2M k1, zmm1	RM	V/V	AVX512DQ	Sets each bit in k1 to 1 or 0 based on the value of the most significant bit of the corresponding quadword in ZMM1.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

Description

Converts a vector register to a mask register. Each element in the destination register is set to 1 or 0 depending on the value of most significant bit of the corresponding element in the source register.

The source operand is a ZMM/YMM/XMM register. The destination operand is a mask register.

EVEX.vvvv is reserved and must be 1111b otherwise instructions will #UD.

Operation

VPMOVB2M (EVEX encoded versions)

```
(KL, VL) = (16, 128), (32, 256), (64, 512)
FOR j ← 0 TO KL-1
    i ← j * 8
    IF SRC[i+7]
        THEN    DEST[j] ← 1
        ELSE    DEST[j] ← 0
    FI;
ENDFOR
DEST[MAX_KL-1:KL] ← 0
```

VPMOVW2M (EVEX encoded versions)

```
(KL, VL) = (8, 128), (16, 256), (32, 512)
FOR j ← 0 TO KL-1
    i ← j * 16
    IF SRC[i+15]
        THEN    DEST[j] ← 1
        ELSE    DEST[j] ← 0
    FI;
ENDFOR
DEST[MAX_KL-1:KL] ← 0
```

VPMOVD2M (EVEX encoded versions)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j ← 0 TO KL-1
    i ← j * 32
    IF SRC[i+31]
        THEN    DEST[j] ← 1
        ELSE    DEST[j] ← 0
    FI;
ENDFOR
DEST[MAX_KL-1:KL] ← 0
```

VPMOVQ2M (EVEX encoded versions)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j ← 0 TO KL-1
    i ← j * 64
    IF SRC[i+63]
        THEN    DEST[j] ← 1
        ELSE    DEST[j] ← 0
    FI;
ENDFOR
DEST[MAX_KL-1:KL] ← 0
```

Intel C/C++ Compiler Intrinsic Equivalents

VPMPOVB2M __mmask64 _mm512_movepi8_mask(__m512i);
VPMPOVD2M __mmask16 _mm512_movepi32_mask(__m512i);
VPMPOVQ2M __mmask8 _mm512_movepi64_mask(__m512i);
VPMPOVW2M __mmask32 _mm512_movepi16_mask(__m512i);
VPMPOVB2M __mmask32 _mm256_movepi8_mask(__m256i);
VPMPOVD2M __mmask8 _mm256_movepi32_mask(__m256i);
VPMPOVQ2M __mmask8 _mm256_movepi64_mask(__m256i);
VPMPOVW2M __mmask16 _mm256_movepi16_mask(__m256i);
VPMPOVB2M __mmask16 _mm_movepi8_mask(__m128i);
VPMPOVD2M __mmask8 _mm_movepi32_mask(__m128i);
VPMPOVQ2M __mmask8 _mm_movepi64_mask(__m128i);
VPMPOVW2M __mmask8 _mm_movepi16_mask(__m128i);

SIMD Floating-Point Exceptions

None

Other Exceptions

EVEX-encoded instruction, see Exceptions Type E7NM

#UD	If EVEX.vvvv != 1111B.

VPMOVDB/VPMOVSDB/VPMOVUSDB—Down Convert DWord to Byte

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.128.F3.0F38.W0 31 /r VPMOVDB *xmm1/m32 {k1}{z}, xmm2*	A	V/V	AVX512VL AVX512F	Converts 4 packed double-word integers from *xmm2* into 4 packed byte integers in *xmm1/m32* with truncation under writemask k1.
EVEX.128.F3.0F38.W0 21 /r VPMOVSDB *xmm1/m32 {k1}{z}, xmm2*	A	V/V	AVX512VL AVX512F	Converts 4 packed signed double-word integers from *xmm2* into 4 packed signed byte integers in *xmm1/m32* using signed saturation under writemask k1.
EVEX.128.F3.0F38.W0 11 /r VPMOVUSDB *xmm1/m32 {k1}{z}, xmm2*	A	V/V	AVX512VL AVX512F	Converts 4 packed unsigned double-word integers from *xmm2* into 4 packed unsigned byte integers in *xmm1/m32* using unsigned saturation under writemask k1.
EVEX.256.F3.0F38.W0 31 /r VPMOVDB *xmm1/m64 {k1}{z}, ymm2*	A	V/V	AVX512VL AVX512F	Converts 8 packed double-word integers from *ymm2* into 8 packed byte integers in *xmm1/m64* with truncation under writemask k1.
EVEX.256.F3.0F38.W0 21 /r VPMOVSDB *xmm1/m64 {k1}{z}, ymm2*	A	V/V	AVX512VL AVX512F	Converts 8 packed signed double-word integers from *ymm2* into 8 packed signed byte integers in *xmm1/m64* using signed saturation under writemask k1.
EVEX.256.F3.0F38.W0 11 /r VPMOVUSDB *xmm1/m64 {k1}{z}, ymm2*	A	V/V	AVX512VL AVX512F	Converts 8 packed unsigned double-word integers from *ymm2* into 8 packed unsigned byte integers in *xmm1/m64* using unsigned saturation under writemask k1.
EVEX.512.F3.0F38.W0 31 /r VPMOVDB *xmm1/m128 {k1}{z}, zmm2*	A	V/V	AVX512F	Converts 16 packed double-word integers from *zmm2* into 16 packed byte integers in *xmm1/m128* with truncation under writemask k1.
EVEX.512.F3.0F38.W0 21 /r VPMOVSDB *xmm1/m128 {k1}{z}, zmm2*	A	V/V	AVX512F	Converts 16 packed signed double-word integers from *zmm2* into 16 packed signed byte integers in *xmm1/m128* using signed saturation under writemask k1.
EVEX.512.F3.0F38.W0 11 /r VPMOVUSDB *xmm1/m128 {k1}{z}, zmm2*	A	V/V	AVX512F	Converts 16 packed unsigned double-word integers from *zmm2* into 16 packed unsigned byte integers in *xmm1/m128* using unsigned saturation under writemask k1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Quarter Vector Mem	ModRM:r/m (w)	ModRM:reg (r)	NA	NA

Description

VPMOVDB down converts 32-bit integer elements in the source operand (the second operand) into packed bytes using truncation. VPMOVSDB converts signed 32-bit integers into packed signed bytes using signed saturation. VPMOVUSDB convert unsigned double-word values into unsigned byte values using unsigned saturation.

The source operand is a ZMM/YMM/XMM register. The destination operand is a XMM register or a 128/64/32-bit memory location.

Down-converted byte elements are written to the destination operand (the first operand) from the least-significant byte. Byte elements of the destination operand are updated according to the writemask. Bits (MAXVL-1:128/64/32) of the register destination are zeroed.

EVEX.vvvv is reserved and must be 1111b otherwise instructions will #UD.

Operation

VPMOVDB instruction (EVEX encoded versions) when dest is a register

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j ← 0 TO KL-1
    i ← j * 8
    m ← j * 32
    IF k1[j] OR *no writemask*
        THEN DEST[i+7:i] ← TruncateDoubleWordToByte (SRC[m+31:m])
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+7:i] remains unchanged*
                ELSE *zeroing-masking*              ; zeroing-masking
                    DEST[i+7:i] ← 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL/4] ← 0;
```

VPMOVDB instruction (EVEX encoded versions) when dest is memory

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j ← 0 TO KL-1
    i ← j * 8
    m ← j * 32
    IF k1[j] OR *no writemask*
        THEN DEST[i+7:i] ← TruncateDoubleWordToByte (SRC[m+31:m])
        ELSE *DEST[i+7:i] remains unchanged*        ; merging-masking
    FI;
ENDFOR
```

VPMOVSDB instruction (EVEX encoded versions) when dest is a register

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j ← 0 TO KL-1
    i ← j * 8
    m ← j * 32
    IF k1[j] OR *no writemask*
        THEN DEST[i+7:i] ← SaturateSignedDoubleWordToByte (SRC[m+31:m])
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+7:i] remains unchanged*
                ELSE *zeroing-masking*              ; zeroing-masking
                    DEST[i+7:i] ← 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL/4] ← 0;
```

VPMOVSDB instruction (EVEX encoded versions) when dest is memory

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j ← 0 TO KL-1
    i ← j * 8
    m ← j * 32
    IF k1[j] OR *no writemask*
        THEN DEST[i+7:i] ← SaturateSignedDoubleWordToByte (SRC[m+31:m])
        ELSE *DEST[i+7:i] remains unchanged*          ; merging-masking
    FI;
ENDFOR
```

VPMOVUSDB instruction (EVEX encoded versions) when dest is a register

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j ← 0 TO KL-1
    i ← j * 8
    m ← j * 32
    IF k1[j] OR *no writemask*
        THEN DEST[i+7:i] ← SaturateUnsignedDoubleWordToByte (SRC[m+31:m])
        ELSE
            IF *merging-masking*                  ; merging-masking
                THEN *DEST[i+7:i] remains unchanged*
                ELSE *zeroing-masking*            ; zeroing-masking
                    DEST[i+7:i] ← 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL/4] ← 0;
```

VPMOVUSDB instruction (EVEX encoded versions) when dest is memory

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j ← 0 TO KL-1
    i ← j * 8
    m ← j * 32
    IF k1[j] OR *no writemask*
        THEN DEST[i+7:i] ← SaturateUnsignedDoubleWordToByte (SRC[m+31:m])
        ELSE *DEST[i+7:i] remains unchanged*          ; merging-masking
    FI;
ENDFOR
```

Intel C/C++ Compiler Intrinsic Equivalents

VPMOVDB __m128i _mm512_cvtepi32_epi8(__m512i a);
VPMOVDB __m128i _mm512_mask_cvtepi32_epi8(__m128i s, __mmask16 k, __m512i a);
VPMOVDB __m128i _mm512_maskz_cvtepi32_epi8(__mmask16 k, __m512i a);
VPMOVDB void _mm512_mask_cvtepi32_storeu_epi8(void * d, __mmask16 k, __m512i a);
VPMOVSDB __m128i _mm512_cvtsepi32_epi8(__m512i a);
VPMOVSDB __m128i _mm512_mask_cvtsepi32_epi8(__m128i s, __mmask16 k, __m512i a);
VPMOVSDB __m128i _mm512_maskz_cvtsepi32_epi8(__mmask16 k, __m512i a);
VPMOVSDB void _mm512_mask_cvtsepi32_storeu_epi8(void * d, __mmask16 k, __m512i a);
VPMOVUSDB __m128i _mm512_cvtusepi32_epi8(__m512i a);
VPMOVUSDB __m128i _mm512_mask_cvtusepi32_epi8(__m128i s, __mmask16 k, __m512i a);
VPMOVUSDB __m128i _mm512_maskz_cvtusepi32_epi8(__mmask16 k, __m512i a);
VPMOVUSDB void _mm512_mask_cvtusepi32_storeu_epi8(void * d, __mmask16 k, __m512i a);
VPMOVUSDB __m128i _mm256_cvtusepi32_epi8(__m256i a);
VPMOVUSDB __m128i _mm256_mask_cvtusepi32_epi8(__m128i a, __mmask8 k, __m256i b);
VPMOVUSDB __m128i _mm256_maskz_cvtusepi32_epi8(__mmask8 k, __m256i b);
VPMOVUSDB void _mm256_mask_cvtusepi32_storeu_epi8(void * , __mmask8 k, __m256i b);
VPMOVUSDB __m128i _mm_cvtusepi32_epi8(__m128i a);
VPMOVUSDB __m128i _mm_mask_cvtusepi32_epi8(__m128i a, __mmask8 k, __m128i b);
VPMOVUSDB __m128i _mm_maskz_cvtusepi32_epi8(__mmask8 k, __m128i b);
VPMOVUSDB void _mm_mask_cvtusepi32_storeu_epi8(void * , __mmask8 k, __m128i b);
VPMOVSDB __m128i _mm256_cvtsepi32_epi8(__m256i a);
VPMOVSDB __m128i _mm256_mask_cvtsepi32_epi8(__m128i a, __mmask8 k, __m256i b);
VPMOVSDB __m128i _mm256_maskz_cvtsepi32_epi8(__mmask8 k, __m256i b);
VPMOVSDB void _mm256_mask_cvtsepi32_storeu_epi8(void * , __mmask8 k, __m256i b);
VPMOVSDB __m128i _mm_cvtsepi32_epi8(__m128i a);
VPMOVSDB __m128i _mm_mask_cvtsepi32_epi8(__m128i a, __mmask8 k, __m128i b);
VPMOVSDB __m128i _mm_maskz_cvtsepi32_epi8(__mmask8 k, __m128i b);
VPMOVSDB void _mm_mask_cvtsepi32_storeu_epi8(void * , __mmask8 k, __m128i b);
VPMOVDB __m128i _mm256_cvtepi32_epi8(__m256i a);
VPMOVDB __m128i _mm256_mask_cvtepi32_epi8(__m128i a, __mmask8 k, __m256i b);
VPMOVDB __m128i _mm256_maskz_cvtepi32_epi8(__mmask8 k, __m256i b);
VPMOVDB void _mm256_mask_cvtepi32_storeu_epi8(void * , __mmask8 k, __m256i b);
VPMOVDB __m128i _mm_cvtepi32_epi8(__m128i a);
VPMOVDB __m128i _mm_mask_cvtepi32_epi8(__m128i a, __mmask8 k, __m128i b);
VPMOVDB __m128i _mm_maskz_cvtepi32_epi8(__mmask8 k, __m128i b);
VPMOVDB void _mm_mask_cvtepi32_storeu_epi8(void * , __mmask8 k, __m128i b);

SIMD Floating-Point Exceptions

None

Other Exceptions

EVEX-encoded instruction, see Exceptions Type E6.

#UD	If EVEX.vvvv != 1111B.

VPMOVDW/VPMOVSDW/VPMOVUSDW—Down Convert DWord to Word

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.128.F3.0F38.W0 33 /r VPMOVDW *xmm1/m64 {k1}{z}, xmm2*	A	V/V	AVX512VL AVX512F	Converts 4 packed double-word integers from *xmm2* into 4 packed word integers in *xmm1/m64* with truncation under writemask k1.
EVEX.128.F3.0F38.W0 23 /r VPMOVSDW *xmm1/m64 {k1}{z}, xmm2*	A	V/V	AVX512VL AVX512F	Converts 4 packed signed double-word integers from *xmm2* into 4 packed signed word integers in *ymm1/m64* using signed saturation under writemask k1.
EVEX.128.F3.0F38.W0 13 /r VPMOVUSDW *xmm1/m64 {k1}{z}, xmm2*	A	V/V	AVX512VL AVX512F	Converts 4 packed unsigned double-word integers from *xmm2* into 4 packed unsigned word integers in *xmm1/m64* using unsigned saturation under writemask k1.
EVEX.256.F3.0F38.W0 33 /r VPMOVDW *xmm1/m128 {k1}{z}, ymm2*	A	V/V	AVX512VL AVX512F	Converts 8 packed double-word integers from *ymm2* into 8 packed word integers in *xmm1/m128* with truncation under writemask k1.
EVEX.256.F3.0F38.W0 23 /r VPMOVSDW *xmm1/m128 {k1}{z}, ymm2*	A	V/V	AVX512VL AVX512F	Converts 8 packed signed double-word integers from *ymm2* into 8 packed signed word integers in *xmm1/m128* using signed saturation under writemask k1.
EVEX.256.F3.0F38.W0 13 /r VPMOVUSDW *xmm1/m128 {k1}{z}, ymm2*	A	V/V	AVX512VL AVX512F	Converts 8 packed unsigned double-word integers from *ymm2* into 8 packed unsigned word integers in *xmm1/m128* using unsigned saturation under writemask k1.
EVEX.512.F3.0F38.W0 33 /r VPMOVDW *ymm1/m256 {k1}{z}, zmm2*	A	V/V	AVX512F	Converts 16 packed double-word integers from *zmm2* into 16 packed word integers in *ymm1/m256* with truncation under writemask k1.
EVEX.512.F3.0F38.W0 23 /r VPMOVSDW *ymm1/m256 {k1}{z}, zmm2*	A	V/V	AVX512F	Converts 16 packed signed double-word integers from *zmm2* into 16 packed signed word integers in *ymm1/m256* using signed saturation under writemask k1.
EVEX.512.F3.0F38.W0 13 /r VPMOVUSDW *ymm1/m256 {k1}{z}, zmm2*	A	V/V	AVX512F	Converts 16 packed unsigned double-word integers from *zmm2* into 16 packed unsigned word integers in *ymm1/m256* using unsigned saturation under writemask k1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Half Vector Mem	ModRM:r/m (w)	ModRM:reg (r)	NA	NA

Description

VPMOVDW down converts 32-bit integer elements in the source operand (the second operand) into packed words using truncation. VPMOVSDW converts signed 32-bit integers into packed signed words using signed saturation. VPMOVUSDW convert unsigned double-word values into unsigned word values using unsigned saturation.

The source operand is a ZMM/YMM/XMM register. The destination operand is a YMM/XMM/XMM register or a 256/128/64-bit memory location.

Down-converted word elements are written to the destination operand (the first operand) from the least-significant word. Word elements of the destination operand are updated according to the writemask. Bits (MAXVL-1:256/128/64) of the register destination are zeroed.

EVEX.vvvv is reserved and must be 1111b otherwise instructions will #UD.

Operation

VPMOVDW instruction (EVEX encoded versions) when dest is a register

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j ← 0 TO KL-1
    i ← j * 16
    m ← j * 32
    IF k1[j] OR *no writemask*
        THEN DEST[i+15:i] ← TruncateDoubleWordToWord (SRC[m+31:m])
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+15:i] remains unchanged*
                ELSE *zeroing-masking*              ; zeroing-masking
                    DEST[i+15:i] ← 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL/2] ← 0;
```

VPMOVDW instruction (EVEX encoded versions) when dest is memory

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j ← 0 TO KL-1
    i ← j * 16
    m ← j * 32
    IF k1[j] OR *no writemask*
        THEN DEST[i+15:i] ← TruncateDoubleWordToWord (SRC[m+31:m])
        ELSE
            *DEST[i+15:i] remains unchanged*    ; merging-masking
    FI;
ENDFOR
```

VPMOVSDW instruction (EVEX encoded versions) when dest is a register

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j ← 0 TO KL-1
    i ← j * 16
    m ← j * 32
    IF k1[j] OR *no writemask*
        THEN DEST[i+15:i] ← SaturateSignedDoubleWordToWord (SRC[m+31:m])
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+15:i] remains unchanged*
                ELSE *zeroing-masking*              ; zeroing-masking
                    DEST[i+15:i] ← 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL/2] ← 0;
```

VPMOVSDW instruction (EVEX encoded versions) when dest is memory

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j ← 0 TO KL-1
    i ← j * 16
    m ← j * 32
    IF k1[j] OR *no writemask*
        THEN DEST[i+15:i] ← SaturateSignedDoubleWordToWord (SRC[m+31:m])
        ELSE
            *DEST[i+15:i] remains unchanged*     ; merging-masking
    FI;
ENDFOR
```

VPMOVUSDW instruction (EVEX encoded versions) when dest is a register

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j ← 0 TO KL-1
    i ← j * 16
    m ← j * 32
    IF k1[j] OR *no writemask*
        THEN DEST[i+15:i] ← SaturateUnsignedDoubleWordToWord (SRC[m+31:m])
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+15:i] remains unchanged*
                ELSE *zeroing-masking*              ; zeroing-masking
                    DEST[i+15:i] ← 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL/2] ← 0;
```

VPMOVUSDW instruction (EVEX encoded versions) when dest is memory

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j ← 0 TO KL-1
    i ← j * 16
    m ← j * 32
    IF k1[j] OR *no writemask*
        THEN DEST[i+15:i] ← SaturateUnsignedDoubleWordToWord (SRC[m+31:m])
        ELSE
            *DEST[i+15:i] remains unchanged*     ; merging-masking
    FI;
ENDFOR
```

Intel C/C++ Compiler Intrinsic Equivalents

VPMOVDW __m256i _mm512_cvtepi32_epi16(__m512i a);
VPMOVDW __m256i _mm512_mask_cvtepi32_epi16(__m256i s, __mmask16 k, __m512i a);
VPMOVDW __m256i _mm512_maskz_cvtepi32_epi16(__mmask16 k, __m512i a);
VPMOVDW void _mm512_mask_cvtepi32_storeu_epi16(void * d, __mmask16 k, __m512i a);
VPMOVSDW __m256i _mm512_cvtsepi32_epi16(__m512i a);
VPMOVSDW __m256i _mm512_mask_cvtsepi32_epi16(__m256i s, __mmask16 k, __m512i a);
VPMOVSDW __m256i _mm512_maskz_cvtsepi32_epi16(__mmask16 k, __m512i a);
VPMOVSDW void _mm512_mask_cvtsepi32_storeu_epi16(void * d, __mmask16 k, __m512i a);
VPMOVUSDW __m256i _mm512_cvtusepi32_epi16 __m512i a);
VPMOVUSDW __m256i _mm512_mask_cvtusepi32_epi16(__m256i s, __mmask16 k, __m512i a);
VPMOVUSDW __m256i _mm512_maskz_cvtusepi32_epi16(__mmask16 k, __m512i a);
VPMOVUSDW void _mm512_mask_cvtusepi32_storeu_epi16(void * d, __mmask16 k, __m512i a);
VPMOVUSDW __m128i _mm256_cvtusepi32_epi16(__m256i a);
VPMOVUSDW __m128i _mm256_mask_cvtusepi32_epi16(__m128i a, __mmask8 k, __m256i b);
VPMOVUSDW __m128i _mm256_maskz_cvtusepi32_epi16(__mmask8 k, __m256i b);
VPMOVUSDW void _mm256_mask_cvtusepi32_storeu_epi16(void * , __mmask8 k, __m256i b);
VPMOVUSDW __m128i _mm_cvtusepi32_epi16(__m128i a);
VPMOVUSDW __m128i _mm_mask_cvtusepi32_epi16(__m128i a, __mmask8 k, __m128i b);
VPMOVUSDW __m128i _mm_maskz_cvtusepi32_epi16(__mmask8 k, __m128i b);
VPMOVUSDW void _mm_mask_cvtusepi32_storeu_epi16(void * , __mmask8 k, __m128i b);
VPMOVSDW __m128i _mm256_cvtsepi32_epi16(__m256i a);
VPMOVSDW __m128i _mm256_mask_cvtsepi32_epi16(__m128i a, __mmask8 k, __m256i b);
VPMOVSDW __m128i _mm256_maskz_cvtsepi32_epi16(__mmask8 k, __m256i b);
VPMOVSDW void _mm256_mask_cvtsepi32_storeu_epi16(void * , __mmask8 k, __m256i b);
VPMOVSDW __m128i _mm_cvtsepi32_epi16(__m128i a);
VPMOVSDW __m128i _mm_mask_cvtsepi32_epi16(__m128i a, __mmask8 k, __m128i b);
VPMOVSDW __m128i _mm_maskz_cvtsepi32_epi16(__mmask8 k, __m128i b);
VPMOVSDW void _mm_mask_cvtsepi32_storeu_epi16(void * , __mmask8 k, __m128i b);
VPMOVDW __m128i _mm256_cvtepi32_epi16(__m256i a);
VPMOVDW __m128i _mm256_mask_cvtepi32_epi16(__m128i a, __mmask8 k, __m256i b);
VPMOVDW __m128i _mm256_maskz_cvtepi32_epi16(__mmask8 k, __m256i b);
VPMOVDW void _mm256_mask_cvtepi32_storeu_epi16(void * , __mmask8 k, __m256i b);
VPMOVDW __m128i _mm_cvtepi32_epi16(__m128i a);
VPMOVDW __m128i _mm_mask_cvtepi32_epi16(__m128i a, __mmask8 k, __m128i b);
VPMOVDW __m128i _mm_maskz_cvtepi32_epi16(__mmask8 k, __m128i b);
VPMOVDW void _mm_mask_cvtepi32_storeu_epi16(void * , __mmask8 k, __m128i b);

SIMD Floating-Point Exceptions

None

Other Exceptions

EVEX-encoded instruction, see Exceptions Type E6.

#UD	If EVEX.vvvv != 1111B.

VPMOVM2B/VPMOVM2W/VPMOVM2D/VPMOVM2Q—Convert a Mask Register to a Vector Register

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.128.F3.0F38.W0 28 /r VPMOVM2B xmm1, k1	RM	V/V	AVX512VL AVX512BW	Sets each byte in XMM1 to all 1's or all 0's based on the value of the corresponding bit in k1.
EVEX.256.F3.0F38.W0 28 /r VPMOVM2B ymm1, k1	RM	V/V	AVX512VL AVX512BW	Sets each byte in YMM1 to all 1's or all 0's based on the value of the corresponding bit in k1.
EVEX.512.F3.0F38.W0 28 /r VPMOVM2B zmm1, k1	RM	V/V	AVX512BW	Sets each byte in ZMM1 to all 1's or all 0's based on the value of the corresponding bit in k1.
EVEX.128.F3.0F38.W1 28 /r VPMOVM2W xmm1, k1	RM	V/V	AVX512VL AVX512BW	Sets each word in XMM1 to all 1's or all 0's based on the value of the corresponding bit in k1.
EVEX.256.F3.0F38.W1 28 /r VPMOVM2W ymm1, k1	RM	V/V	AVX512VL AVX512BW	Sets each word in YMM1 to all 1's or all 0's based on the value of the corresponding bit in k1.
EVEX.512.F3.0F38.W1 28 /r VPMOVM2W zmm1, k1	RM	V/V	AVX512BW	Sets each word in ZMM1 to all 1's or all 0's based on the value of the corresponding bit in k1.
EVEX.128.F3.0F38.W0 38 /r VPMOVM2D xmm1, k1	RM	V/V	AVX512VL AVX512DQ	Sets each doubleword in XMM1 to all 1's or all 0's based on the value of the corresponding bit in k1.
EVEX.256.F3.0F38.W0 38 /r VPMOVM2D ymm1, k1	RM	V/V	AVX512VL AVX512DQ	Sets each doubleword in YMM1 to all 1's or all 0's based on the value of the corresponding bit in k1.
EVEX.512.F3.0F38.W0 38 /r VPMOVM2D zmm1, k1	RM	V/V	AVX512DQ	Sets each doubleword in ZMM1 to all 1's or all 0's based on the value of the corresponding bit in k1.
EVEX.128.F3.0F38.W1 38 /r VPMOVM2Q xmm1, k1	RM	V/V	AVX512VL AVX512DQ	Sets each quadword in XMM1 to all 1's or all 0's based on the value of the corresponding bit in k1.
EVEX.256.F3.0F38.W1 38 /r VPMOVM2Q ymm1, k1	RM	V/V	AVX512VL AVX512DQ	Sets each quadword in YMM1 to all 1's or all 0's based on the value of the corresponding bit in k1.
EVEX.512.F3.0F38.W1 38 /r VPMOVM2Q zmm1, k1	RM	V/V	AVX512DQ	Sets each quadword in ZMM1 to all 1's or all 0's based on the value of the corresponding bit in k1.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

Description

Converts a mask register to a vector register. Each element in the destination register is set to all 1's or all 0's depending on the value of the corresponding bit in the source mask register.

The source operand is a mask register. The destination operand is a ZMM/YMM/XMM register.

EVEX.vvvv is reserved and must be 1111b otherwise instructions will #UD.

Operation

VPMOVM2B (EVEX encoded versions)

```
(KL, VL) = (16, 128), (32, 256), (64, 512)
FOR j ← 0 TO KL-1
    i ← j * 8
    IF SRC[j]
        THEN    DEST[i+7:i] ← -1
        ELSE    DEST[i+7:i] ← 0
    FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0
```

VPMOVM2W (EVEX encoded versions)

```
(KL, VL) = (8, 128), (16, 256), (32, 512)
FOR j ← 0 TO KL-1
    i ← j * 16
    IF SRC[j]
        THEN    DEST[i+15:i] ← -1
        ELSE    DEST[i+15:i] ← 0
    FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0
```

VPMOVM2D (EVEX encoded versions)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j ← 0 TO KL-1
    i ← j * 32
    IF SRC[j]
        THEN    DEST[i+31:i] ← -1
        ELSE    DEST[i+31:i] ← 0
    FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0
```

VPMOVM2Q (EVEX encoded versions)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j ← 0 TO KL-1
    i ← j * 64
    IF SRC[j]
        THEN    DEST[i+63:i] ← -1
        ELSE    DEST[i+63:i] ← 0
    FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0
```

Intel C/C++ Compiler Intrinsic Equivalents

VPMOVM2B __m512i _mm512_movm_epi8(__mmask64);
VPMOVM2D __m512i _mm512_movm_epi32(__mmask8);
VPMOVM2Q __m512i _mm512_movm_epi64(__mmask16);
VPMOVM2W __m512i _mm512_movm_epi16(__mmask32);
VPMOVM2B __m256i _mm256_movm_epi8(__mmask32);
VPMOVM2D __m256i _mm256_movm_epi32(__mmask8);
VPMOVM2Q __m256i _mm256_movm_epi64(__mmask8);
VPMOVM2W __m256i _mm256_movm_epi16(__mmask16);
VPMOVM2B __m128i _mm_movm_epi8(__mmask16);
VPMOVM2D __m128i _mm_movm_epi32(__mmask8);
VPMOVM2Q __m128i _mm_movm_epi64(__mmask8);
VPMOVM2W __m128i _mm_movm_epi16(__mmask8);

SIMD Floating-Point Exceptions

None

Other Exceptions

EVEX-encoded instruction, see Exceptions Type E7NM

#UD	If EVEX.vvvv != 1111B.

VPMOVQB/VPMOVSQB/VPMOVUSQB—Down Convert QWord to Byte

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.128.F3.0F38.W0 32 /r VPMOVQB *xmm1/m16 {k1}{z}, xmm2*	A	V/V	AVX512VL AVX512F	Converts 2 packed quad-word integers from *xmm2* into 2 packed byte integers in *xmm1/m16* with truncation under writemask k1.
EVEX.128.F3.0F38.W0 22 /r VPMOVSQB *xmm1/m16 {k1}{z}, xmm2*	A	V/V	AVX512VL AVX512F	Converts 2 packed signed quad-word integers from *xmm2* into 2 packed signed byte integers in *xmm1/m16* using signed saturation under writemask k1.
EVEX.128.F3.0F38.W0 12 /r VPMOVUSQB *xmm1/m16 {k1}{z}, xmm2*	A	V/V	AVX512VL AVX512F	Converts 2 packed unsigned quad-word integers from *xmm2* into 2 packed unsigned byte integers in *xmm1/m16* using unsigned saturation under writemask k1.
EVEX.256.F3.0F38.W0 32 /r VPMOVQB *xmm1/m32 {k1}{z}, ymm2*	A	V/V	AVX512VL AVX512F	Converts 4 packed quad-word integers from *ymm2* into 4 packed byte integers in *xmm1/m32* with truncation under writemask k1.
EVEX.256.F3.0F38.W0 22 /r VPMOVSQB *xmm1/m32 {k1}{z}, ymm2*	A	V/V	AVX512VL AVX512F	Converts 4 packed signed quad-word integers from *ymm2* into 4 packed signed byte integers in *xmm1/m32* using signed saturation under writemask k1.
EVEX.256.F3.0F38.W0 12 /r VPMOVUSQB *xmm1/m32 {k1}{z}, ymm2*	A	V/V	AVX512VL AVX512F	Converts 4 packed unsigned quad-word integers from *ymm2* into 4 packed unsigned byte integers in *xmm1/m32* using unsigned saturation under writemask k1.
EVEX.512.F3.0F38.W0 32 /r VPMOVQB *xmm1/m64 {k1}{z}, zmm2*	A	V/V	AVX512F	Converts 8 packed quad-word integers from *zmm2* into 8 packed byte integers in *xmm1/m64* with truncation under writemask k1.
EVEX.512.F3.0F38.W0 22 /r VPMOVSQB *xmm1/m64 {k1}{z}, zmm2*	A	V/V	AVX512F	Converts 8 packed signed quad-word integers from *zmm2* into 8 packed signed byte integers in *xmm1/m64* using signed saturation under writemask k1.
EVEX.512.F3.0F38.W0 12 /r VPMOVUSQB *xmm1/m64 {k1}{z}, zmm2*	A	V/V	AVX512F	Converts 8 packed unsigned quad-word integers from *zmm2* into 8 packed unsigned byte integers in *xmm1/m64* using unsigned saturation under writemask k1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Oct Vector Mem	ModRM:r/m (w)	ModRM:reg (r)	NA	NA

Description

VPMOVQB down converts 64-bit integer elements in the source operand (the second operand) into packed byte elements using truncation. VPMOVSQB converts signed 64-bit integers into packed signed bytes using signed saturation. VPMOVUSQB convert unsigned quad-word values into unsigned byte values using unsigned saturation. The source operand is a vector register. The destination operand is an XMM register or a memory location.

Down-converted byte elements are written to the destination operand (the first operand) from the least-significant byte. Byte elements of the destination operand are updated according to the writemask. Bits (MAXVL-1:64) of the destination are zeroed.

EVEX.vvvv is reserved and must be 1111b otherwise instructions will #UD.

Operation

VPMOVQB instruction (EVEX encoded versions) when dest is a register

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j ← 0 TO KL-1
    i ← j * 8
    m ← j * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+7:i] ← TruncateQuadWordToByte (SRC[m+63:m])
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+7:i] remains unchanged*
                ELSE *zeroing-masking*          ; zeroing-masking
                    DEST[i+7:i] ← 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL/8] ← 0;
```

VPMOVQB instruction (EVEX encoded versions) when dest is memory

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j ← 0 TO KL-1
    i ← j * 8
    m ← j * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+7:i] ← TruncateQuadWordToByte (SRC[m+63:m])
        ELSE
            *DEST[i+7:i] remains unchanged*     ; merging-masking
    FI;
ENDFOR
```

VPMOVSQB instruction (EVEX encoded versions) when dest is a register

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j ← 0 TO KL-1
    i ← j * 8
    m ← j * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+7:i] ← SaturateSignedQuadWordToByte (SRC[m+63:m])
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+7:i] remains unchanged*
                ELSE *zeroing-masking*          ; zeroing-masking
                    DEST[i+7:i] ← 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL/8] ← 0;
```

VPMOVSQB instruction (EVEX encoded versions) when dest is memory

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j ← 0 TO KL-1
    i ← j * 8
    m ← j * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+7:i] ← SaturateSignedQuadWordToByte (SRC[m+63:m])
        ELSE
            *DEST[i+7:i] remains unchanged*          ; merging-masking
    FI;
ENDFOR
```

VPMOVUSQB instruction (EVEX encoded versions) when dest is a register

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j ← 0 TO KL-1
    i ← j * 8
    m ← j * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+7:i] ← SaturateUnsignedQuadWordToByte (SRC[m+63:m])
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+7:i] remains unchanged*
                ELSE *zeroing-masking*              ; zeroing-masking
                    DEST[i+7:i] ← 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL/8] ← 0;
```

VPMOVUSQB instruction (EVEX encoded versions) when dest is memory

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j ← 0 TO KL-1
    i ← j * 8
    m ← j * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+7:i] ← SaturateUnsignedQuadWordToByte (SRC[m+63:m])
        ELSE
            *DEST[i+7:i] remains unchanged*          ; merging-masking
    FI;
ENDFOR
```

Intel C/C++ Compiler Intrinsic Equivalents

VPMOVQB __m128i _mm512_cvtepi64_epi8(__m512i a);
VPMOVQB __m128i _mm512_mask_cvtepi64_epi8(__m128i s, __mmask8 k, __m512i a);
VPMOVQB __m128i _mm512_maskz_cvtepi64_epi8(__mmask8 k, __m512i a);
VPMOVQB void _mm512_mask_cvtepi64_storeu_epi8(void * d, __mmask8 k, __m512i a);
VPMOVSQB __m128i _mm512_cvtsepi64_epi8(__m512i a);
VPMOVSQB __m128i _mm512_mask_cvtsepi64_epi8(__m128i s, __mmask8 k, __m512i a);
VPMOVSQB __m128i _mm512_maskz_cvtsepi64_epi8(__mmask8 k, __m512i a);
VPMOVSQB void _mm512_mask_cvtsepi64_storeu_epi8(void * d, __mmask8 k, __m512i a);
VPMOVUSQB __m128i _mm512_cvtusepi64_epi8(__m512i a);
VPMOVUSQB __m128i _mm512_mask_cvtusepi64_epi8(__m128i s, __mmask8 k, __m512i a);
VPMOVUSQB __m128i _mm512_maskz_cvtusepi64_epi8(__mmask8 k, __m512i a);
VPMOVUSQB void _mm512_mask_cvtusepi64_storeu_epi8(void * d, __mmask8 k, __m512i a);
VPMOVUSQB __m128i _mm256_cvtusepi64_epi8(__m256i a);
VPMOVUSQB __m128i _mm256_mask_cvtusepi64_epi8(__m128i a, __mmask8 k, __m256i b);
VPMOVUSQB __m128i _mm256_maskz_cvtusepi64_epi8(__mmask8 k, __m256i b);
VPMOVUSQB void _mm256_mask_cvtusepi64_storeu_epi8(void * , __mmask8 k, __m256i b);
VPMOVUSQB __m128i _mm_cvtusepi64_epi8(__m128i a);
VPMOVUSQB __m128i _mm_mask_cvtusepi64_epi8(__m128i a, __mmask8 k, __m128i b);
VPMOVUSQB __m128i _mm_maskz_cvtusepi64_epi8(__mmask8 k, __m128i b);
VPMOVUSQB void _mm_mask_cvtusepi64_storeu_epi8(void * , __mmask8 k, __m128i b);
VPMOVSQB __m128i _mm256_cvtsepi64_epi8(__m256i a);
VPMOVSQB __m128i _mm256_mask_cvtsepi64_epi8(__m128i a, __mmask8 k, __m256i b);
VPMOVSQB __m128i _mm256_maskz_cvtsepi64_epi8(__mmask8 k, __m256i b);
VPMOVSQB void _mm256_mask_cvtsepi64_storeu_epi8(void * , __mmask8 k, __m256i b);
VPMOVSQB __m128i _mm_cvtsepi64_epi8(__m128i a);
VPMOVSQB __m128i _mm_mask_cvtsepi64_epi8(__m128i a, __mmask8 k, __m128i b);
VPMOVSQB __m128i _mm_maskz_cvtsepi64_epi8(__mmask8 k, __m128i b);
VPMOVSQB void _mm_mask_cvtsepi64_storeu_epi8(void * , __mmask8 k, __m128i b);
VPMOVQB __m128i _mm256_cvtepi64_epi8(__m256i a);
VPMOVQB __m128i _mm256_mask_cvtepi64_epi8(__m128i a, __mmask8 k, __m256i b);
VPMOVQB __m128i _mm256_maskz_cvtepi64_epi8(__mmask8 k, __m256i b);
VPMOVQB void _mm256_mask_cvtepi64_storeu_epi8(void * , __mmask8 k, __m256i b);
VPMOVQB __m128i _mm_cvtepi64_epi8(__m128i a);
VPMOVQB __m128i _mm_mask_cvtepi64_epi8(__m128i a, __mmask8 k, __m128i b);
VPMOVQB __m128i _mm_maskz_cvtepi64_epi8(__mmask8 k, __m128i b);
VPMOVQB void _mm_mask_cvtepi64_storeu_epi8(void * , __mmask8 k, __m128i b);

SIMD Floating-Point Exceptions

None

Other Exceptions

EVEX-encoded instruction, see Exceptions Type E6.

#UD	If EVEX.vvvv != 1111B.

VPMOVQD/VPMOVSQD/VPMOVUSQD—Down Convert QWord to DWord

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.128.F3.0F38.W0 35 /r VPMOVQD *xmm1/m128 {k1}{z}, xmm2*	A	V/V	AVX512VL AVX512F	Converts 2 packed quad-word integers from *xmm2* into 2 packed double-word integers in *xmm1/m128* with truncation subject to writemask k1.
EVEX.128.F3.0F38.W0 25 /r VPMOVSQD *xmm1/m64 {k1}{z}, xmm2*	A	V/V	AVX512VL AVX512F	Converts 2 packed signed quad-word integers from *xmm2* into 2 packed signed double-word integers in *xmm1/m64* using signed saturation subject to writemask k1.
EVEX.128.F3.0F38.W0 15 /r VPMOVUSQD *xmm1/m64 {k1}{z}, xmm2*	A	V/V	AVX512VL AVX512F	Converts 2 packed unsigned quad-word integers from *xmm2* into 2 packed unsigned double-word integers in *xmm1/m64* using unsigned saturation subject to writemask k1.
EVEX.256.F3.0F38.W0 35 /r VPMOVQD *xmm1/m128 {k1}{z}, ymm2*	A	V/V	AVX512VL AVX512F	Converts 4 packed quad-word integers from *ymm2* into 4 packed double-word integers in *xmm1/m128* with truncation subject to writemask k1.
EVEX.256.F3.0F38.W0 25 /r VPMOVSQD *xmm1/m128 {k1}{z}, ymm2*	A	V/V	AVX512VL AVX512F	Converts 4 packed signed quad-word integers from *ymm2* into 4 packed signed double-word integers in *xmm1/m128* using signed saturation subject to writemask k1.
EVEX.256.F3.0F38.W0 15 /r VPMOVUSQD *xmm1/m128 {k1}{z}, ymm2*	A	V/V	AVX512VL AVX512F	Converts 4 packed unsigned quad-word integers from y*mm2* into 4 packed unsigned double-word integers in *xmm1/m128* using unsigned saturation subject to writemask k1.
EVEX.512.F3.0F38.W0 35 /r VPMOVQD *ymm1/m256 {k1}{z}, zmm2*	A	V/V	AVX512F	Converts 8 packed quad-word integers from *zmm2* into 8 packed double-word integers in *ymm1/m256* with truncation subject to writemask k1.
EVEX.512.F3.0F38.W0 25 /r VPMOVSQD *ymm1/m256 {k1}{z}, zmm2*	A	V/V	AVX512F	Converts 8 packed signed quad-word integers from *zmm2* into 8 packed signed double-word integers in *ymm1/m256* using signed saturation subject to writemask k1.
EVEX.512.F3.0F38.W0 15 /r VPMOVUSQD *ymm1/m256 {k1}{z}, zmm2*	A	V/V	AVX512F	Converts 8 packed unsigned quad-word integers from *zmm2* into 8 packed unsigned double-word integers in *ymm1/m256* using unsigned saturation subject to writemask k1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Half Vector Mem	ModRM:r/m (w)	ModRM:reg (r)	NA	NA

Description

VPMOVQW down converts 64-bit integer elements in the source operand (the second operand) into packed double-words using truncation. VPMOVSQW converts signed 64-bit integers into packed signed doublewords using signed saturation. VPMOVUSQW convert unsigned quad-word values into unsigned double-word values using unsigned saturation.

The source operand is a ZMM/YMM/XMM register. The destination operand is a YMM/XMM/XMM register or a 256/128/64-bit memory location.

Down-converted doubleword elements are written to the destination operand (the first operand) from the least-significant doubleword. Doubleword elements of the destination operand are updated according to the writemask. Bits (MAXVL-1:256/128/64) of the register destination are zeroed.

EVEX.vvvv is reserved and must be 1111b otherwise instructions will #UD.

Operation

VPMOVQD instruction (EVEX encoded version) reg-reg form

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j ← 0 TO KL-1
    i ← j * 32
    m ← j * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+31:i] ← TruncateQuadWordToDWord (SRC[m+63:m])
        ELSE *zeroing-masking*                  ; zeroing-masking
                    DEST[i+31:i] ← 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL/2] ← 0;
```

VPMOVQD instruction (EVEX encoded version) memory form

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j ← 0 TO KL-1
    i ← j * 32
    m ← j * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+31:i] ← TruncateQuadWordToDWord (SRC[m+63:m])
        ELSE *DEST[i+31:i] remains unchanged*        ; merging-masking
    FI;
ENDFOR
```

VPMOVSQD instruction (EVEX encoded version) reg-reg form

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j ← 0 TO KL-1
    i ← j * 32
    m ← j * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+31:i] ← SaturateSignedQuadWordToDWord (SRC[m+63:m])
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE *zeroing-masking*          ; zeroing-masking
                    DEST[i+31:i] ← 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL/2] ← 0;
```

VPMOVSQD instruction (EVEX encoded version) memory form

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j ← 0 TO KL-1
    i ← j * 32
    m ← j * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+31:i] ← SaturateSignedQuadWordToDWord (SRC[m+63:m])
        ELSE *DEST[i+31:i] remains unchanged*          ; merging-masking
    FI;
ENDFOR
```

VPMOVUSQD instruction (EVEX encoded version) reg-reg form

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j ← 0 TO KL-1
    i ← j * 32
    m ← j * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+31:i] ← SaturateUnsignedQuadWordToDWord (SRC[m+63:m])
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE *zeroing-masking*              ; zeroing-masking
                    DEST[i+31:i] ← 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL/2] ← 0;
```

VPMOVUSQD instruction (EVEX encoded version) memory form

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j ← 0 TO KL-1
    i ← j * 32
    m ← j * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+31:i] ← SaturateUnsignedQuadWordToDWord (SRC[m+63:m])
        ELSE *DEST[i+31:i] remains unchanged*          ; merging-masking
    FI;
ENDFOR
```

Intel C/C++ Compiler Intrinsic Equivalents

VPMOVQD __m256i _mm512_cvtepi64_epi32(__m512i a);
VPMOVQD __m256i _mm512_mask_cvtepi64_epi32(__m256i s, __mmask8 k, __m512i a);
VPMOVQD __m256i _mm512_maskz_cvtepi64_epi32(__mmask8 k, __m512i a);
VPMOVQD void _mm512_mask_cvtepi64_storeu_epi32(void * d, __mmask8 k, __m512i a);
VPMOVSQD __m256i _mm512_cvtsepi64_epi32(__m512i a);
VPMOVSQD __m256i _mm512_mask_cvtsepi64_epi32(__m256i s, __mmask8 k, __m512i a);
VPMOVSQD __m256i _mm512_maskz_cvtsepi64_epi32(__mmask8 k, __m512i a);
VPMOVSQD void _mm512_mask_cvtsepi64_storeu_epi32(void * d, __mmask8 k, __m512i a);
VPMOVUSQD __m256i _mm512_cvtusepi64_epi32(__m512i a);
VPMOVUSQD __m256i _mm512_mask_cvtusepi64_epi32(__m256i s, __mmask8 k, __m512i a);
VPMOVUSQD __m256i _mm512_maskz_cvtusepi64_epi32(__mmask8 k, __m512i a);
VPMOVUSQD void _mm512_mask_cvtusepi64_storeu_epi32(void * d, __mmask8 k, __m512i a);
VPMOVUSQD __m128i _mm256_cvtusepi64_epi32(__m256i a);
VPMOVUSQD __m128i _mm256_mask_cvtusepi64_epi32(__m128i a, __mmask8 k, __m256i b);
VPMOVUSQD __m128i _mm256_maskz_cvtusepi64_epi32(__mmask8 k, __m256i b);
VPMOVUSQD void _mm256_mask_cvtusepi64_storeu_epi32(void * , __mmask8 k, __m256i b);
VPMOVUSQD __m128i _mm_cvtusepi64_epi32(__m128i a);
VPMOVUSQD __m128i _mm_mask_cvtusepi64_epi32(__m128i a, __mmask8 k, __m128i b);
VPMOVUSQD __m128i _mm_maskz_cvtusepi64_epi32(__mmask8 k, __m128i b);
VPMOVUSQD void _mm_mask_cvtusepi64_storeu_epi32(void * , __mmask8 k, __m128i b);
VPMOVSQD __m128i _mm256_cvtsepi64_epi32(__m256i a);
VPMOVSQD __m128i _mm256_mask_cvtsepi64_epi32(__m128i a, __mmask8 k, __m256i b);
VPMOVSQD __m128i _mm256_maskz_cvtsepi64_epi32(__mmask8 k, __m256i b);
VPMOVSQD void _mm256_mask_cvtsepi64_storeu_epi32(void * , __mmask8 k, __m256i b);
VPMOVSQD __m128i _mm_cvtsepi64_epi32(__m128i a);
VPMOVSQD __m128i _mm_mask_cvtsepi64_epi32(__m128i a, __mmask8 k, __m128i b);
VPMOVSQD __m128i _mm_maskz_cvtsepi64_epi32(__mmask8 k, __m128i b);
VPMOVSQD void _mm_mask_cvtsepi64_storeu_epi32(void * , __mmask8 k, __m128i b);
VPMOVQD __m128i _mm256_cvtepi64_epi32(__m256i a);
VPMOVQD __m128i _mm256_mask_cvtepi64_epi32(__m128i a, __mmask8 k, __m256i b);
VPMOVQD __m128i _mm256_maskz_cvtepi64_epi32(__mmask8 k, __m256i b);
VPMOVQD void _mm256_mask_cvtepi64_storeu_epi32(void * , __mmask8 k, __m256i b);
VPMOVQD __m128i _mm_cvtepi64_epi32(__m128i a);
VPMOVQD __m128i _mm_mask_cvtepi64_epi32(__m128i a, __mmask8 k, __m128i b);
VPMOVQD __m128i _mm_maskz_cvtepi64_epi32(__mmask8 k, __m128i b);
VPMOVQD void _mm_mask_cvtepi64_storeu_epi32(void * , __mmask8 k, __m128i b);

SIMD Floating-Point Exceptions

None

Other Exceptions

EVEX-encoded instruction, see Exceptions Type E6.

#UD	If EVEX.vvvv != 1111B.

VPMOVQW/VPMOVSQW/VPMOVUSQW—Down Convert QWord to Word

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.128.F3.0F38.W0 34 /r VPMOVQW *xmm1/m32 {k1}{z}, xmm2*	A	V/V	AVX512VL AVX512F	Converts 2 packed quad-word integers from *xmm2* into 2 packed word integers in *xmm1/m32* with truncation under writemask k1.
EVEX.128.F3.0F38.W0 24 /r VPMOVSQW *xmm1/m32 {k1}{z}, xmm2*	A	V/V	AVX512VL AVX512F	Converts 8 packed signed quad-word integers from *zmm2* into 8 packed signed word integers in *xmm1/m32* using signed saturation under writemask k1.
EVEX.128.F3.0F38.W0 14 /r VPMOVUSQW *xmm1/m32 {k1}{z}, xmm2*	A	V/V	AVX512VL AVX512F	Converts 2 packed unsigned quad-word integers from *xmm2* into 2 packed unsigned word integers in *xmm1/m32* using unsigned saturation under writemask k1.
EVEX.256.F3.0F38.W0 34 /r VPMOVQW *xmm1/m64 {k1}{z}, ymm2*	A	V/V	AVX512VL AVX512F	Converts 4 packed quad-word integers from *ymm2* into 4 packed word integers in *xmm1/m64* with truncation under writemask k1.
EVEX.256.F3.0F38.W0 24 /r VPMOVSQW *xmm1/m64 {k1}{z}, ymm2*	A	V/V	AVX512VL AVX512F	Converts 4 packed signed quad-word integers from *ymm2* into 4 packed signed word integers in *xmm1/m64* using signed saturation under writemask k1.
EVEX.256.F3.0F38.W0 14 /r VPMOVUSQW *xmm1/m64 {k1}{z}, ymm2*	A	V/V	AVX512VL AVX512F	Converts 4 packed unsigned quad-word integers from *ymm2* into 4 packed unsigned word integers in *xmm1/m64* using unsigned saturation under writemask k1.
EVEX.512.F3.0F38.W0 34 /r VPMOVQW *xmm1/m128 {k1}{z}, zmm2*	A	V/V	AVX512F	Converts 8 packed quad-word integers from *zmm2* into 8 packed word integers in *xmm1/m128* with truncation under writemask k1.
EVEX.512.F3.0F38.W0 24 /r VPMOVSQW *xmm1/m128 {k1}{z}, zmm2*	A	V/V	AVX512F	Converts 8 packed signed quad-word integers from *zmm2* into 8 packed signed word integers in *xmm1/m128* using signed saturation under writemask k1.
EVEX.512.F3.0F38.W0 14 /r VPMOVUSQW *xmm1/m128 {k1}{z}, zmm2*	A	V/V	AVX512F	Converts 8 packed unsigned quad-word integers from *zmm2* into 8 packed unsigned word integers in *xmm1/m128* using unsigned saturation under writemask k1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Quarter Vector Mem	ModRM:r/m (w)	ModRM:reg (r)	NA	NA

Description

VPMOVQW down converts 64-bit integer elements in the source operand (the second operand) into packed words using truncation. VPMOVSQW converts signed 64-bit integers into packed signed words using signed saturation. VPMOVUSQW convert unsigned quad-word values into unsigned word values using unsigned saturation.

The source operand is a ZMM/YMM/XMM register. The destination operand is a XMM register or a 128/64/32-bit memory location.

Down-converted word elements are written to the destination operand (the first operand) from the least-significant word. Word elements of the destination operand are updated according to the writemask. Bits (MAXVL-1:128/64/32) of the register destination are zeroed.

EVEX.vvvv is reserved and must be 1111b otherwise instructions will #UD.

Operation

VPMOVQW instruction (EVEX encoded versions) when dest is a register

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j ← 0 TO KL-1
    i ← j * 16
    m ← j * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+15:i] ← TruncateQuadWordToWord (SRC[m+63:m])
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+15:i] remains unchanged*
                ELSE *zeroing-masking*              ; zeroing-masking
                    DEST[i+15:i] ← 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL/4] ← 0;
```

VPMOVQW instruction (EVEX encoded versions) when dest is memory

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j ← 0 TO KL-1
    i ← j * 16
    m ← j * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+15:i] ← TruncateQuadWordToWord (SRC[m+63:m])
        ELSE
            *DEST[i+15:i] remains unchanged*    ; merging-masking
    FI;
ENDFOR
```

VPMOVSQW instruction (EVEX encoded versions) when dest is a register

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j ← 0 TO KL-1
    i ← j * 16
    m ← j * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+15:i] ← SaturateSignedQuadWordToWord (SRC[m+63:m])
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+15:i] remains unchanged*
                ELSE *zeroing-masking*              ; zeroing-masking
                    DEST[i+15:i] ← 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL/4] ← 0;
```

VPMOVSQW instruction (EVEX encoded versions) when dest is memory

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j ← 0 TO KL-1
    i ← j * 16
    m ← j * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+15:i] ← SaturateSignedQuadWordToWord (SRC[m+63:m])
        ELSE
            *DEST[i+15:i] remains unchanged*     ; merging-masking
    FI;
ENDFOR
```

VPMOVUSQW instruction (EVEX encoded versions) when dest is a register

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j ← 0 TO KL-1
    i ← j * 16
    m ← j * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+15:i] ← SaturateUnsignedQuadWordToWord (SRC[m+63:m])
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+15:i] remains unchanged*
                ELSE *zeroing-masking*              ; zeroing-masking
                    DEST[i+15:i] ← 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL/4] ← 0;
```

VPMOVUSQW instruction (EVEX encoded versions) when dest is memory

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j ← 0 TO KL-1
    i ← j * 16
    m ← j * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+15:i] ← SaturateUnsignedQuadWordToWord (SRC[m+63:m])
        ELSE
            *DEST[i+15:i] remains unchanged*     ; merging-masking
    FI;
ENDFOR
```

Intel C/C++ Compiler Intrinsic Equivalents

VPMOVQW __m128i _mm512_cvtepi64_epi16(__m512i a);
VPMOVQW __m128i _mm512_mask_cvtepi64_epi16(__m128i s, __mmask8 k, __m512i a);
VPMOVQW __m128i _mm512_maskz_cvtepi64_epi16(__mmask8 k, __m512i a);
VPMOVQW void _mm512_mask_cvtepi64_storeu_epi16(void * d, __mmask8 k, __m512i a);
VPMOVSQW __m128i _mm512_cvtsepi64_epi16(__m512i a);
VPMOVSQW __m128i _mm512_mask_cvtsepi64_epi16(__m128i s, __mmask8 k, __m512i a);
VPMOVSQW __m128i _mm512_maskz_cvtsepi64_epi16(__mmask8 k, __m512i a);
VPMOVSQW void _mm512_mask_cvtsepi64_storeu_epi16(void * d, __mmask8 k, __m512i a);
VPMOVUSQW __m128i _mm512_cvtusepi64_epi16(__m512i a);
VPMOVUSQW __m128i _mm512_mask_cvtusepi64_epi16(__m128i s, __mmask8 k, __m512i a);
VPMOVUSQW __m128i _mm512_maskz_cvtusepi64_epi16(__mmask8 k, __m512i a);
VPMOVUSQW void _mm512_mask_cvtusepi64_storeu_epi16(void * d, __mmask8 k, __m512i a);
VPMOVUSQD __m128i _mm256_cvtusepi64_epi32(__m256i a);
VPMOVUSQD __m128i _mm256_mask_cvtusepi64_epi32(__m128i a, __mmask8 k, __m256i b);
VPMOVUSQD __m128i _mm256_maskz_cvtusepi64_epi32(__mmask8 k, __m256i b);
VPMOVUSQD void _mm256_mask_cvtusepi64_storeu_epi32(void * , __mmask8 k, __m256i b);
VPMOVUSQD __m128i _mm_cvtusepi64_epi32(__m128i a);
VPMOVUSQD __m128i _mm_mask_cvtusepi64_epi32(__m128i a, __mmask8 k, __m128i b);
VPMOVUSQD __m128i _mm_maskz_cvtusepi64_epi32(__mmask8 k, __m128i b);
VPMOVUSQD void _mm_mask_cvtusepi64_storeu_epi32(void * , __mmask8 k, __m128i b);
VPMOVSQD __m128i _mm256_cvtsepi64_epi32(__m256i a);
VPMOVSQD __m128i _mm256_mask_cvtsepi64_epi32(__m128i a, __mmask8 k, __m256i b);
VPMOVSQD __m128i _mm256_maskz_cvtsepi64_epi32(__mmask8 k, __m256i b);
VPMOVSQD void _mm256_mask_cvtsepi64_storeu_epi32(void * , __mmask8 k, __m256i b);
VPMOVSQD __m128i _mm_cvtsepi64_epi32(__m128i a);
VPMOVSQD __m128i _mm_mask_cvtsepi64_epi32(__m128i a, __mmask8 k, __m128i b);
VPMOVSQD __m128i _mm_maskz_cvtsepi64_epi32(__mmask8 k, __m128i b);
VPMOVSQD void _mm_mask_cvtsepi64_storeu_epi32(void * , __mmask8 k, __m128i b);
VPMOVQD __m128i _mm256_cvtepi64_epi32(__m256i a);
VPMOVQD __m128i _mm256_mask_cvtepi64_epi32(__m128i a, __mmask8 k, __m256i b);
VPMOVQD __m128i _mm256_maskz_cvtepi64_epi32(__mmask8 k, __m256i b);
VPMOVQD void _mm256_mask_cvtepi64_storeu_epi32(void * , __mmask8 k, __m256i b);
VPMOVQD __m128i _mm_cvtepi64_epi32(__m128i a);
VPMOVQD __m128i _mm_mask_cvtepi64_epi32(__m128i a, __mmask8 k, __m128i b);
VPMOVQD __m128i _mm_maskz_cvtepi64_epi32(__mmask8 k, __m128i b);
VPMOVQD void _mm_mask_cvtepi64_storeu_epi32(void * , __mmask8 k, __m128i b);

SIMD Floating-Point Exceptions

None

Other Exceptions

EVEX-encoded instruction, see Exceptions Type E6.

#UD	If EVEX.vvvv != 1111B.

VPMOVWB/VPMOVSWB/VPMOVUSWB—Down Convert Word to Byte

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.128.F3.0F38.W0 30 /r VPMOVWB *xmm1/m64 {k1}{z}, xmm2*	A	V/V	AVX512VL AVX512BW	Converts 8 packed word integers from *xmm2* into 8 packed bytes in *xmm1/m64* with truncation under writemask k1.
EVEX.128.F3.0F38.W0 20 /r VPMOVSWB *xmm1/m64 {k1}{z}, xmm2*	A	V/V	AVX512VL AVX512BW	Converts 8 packed signed word integers from *xmm2* into 8 packed signed bytes in *xmm1/m64* using signed saturation under writemask k1.
EVEX.128.F3.0F38.W0 10 /r VPMOVUSWB *xmm1/m64 {k1}{z}, xmm2*	A	V/V	AVX512VL AVX512BW	Converts 8 packed unsigned word integers from *xmm2* into 8 packed unsigned bytes in *8mm1/m64* using unsigned saturation under writemask k1.
EVEX.256.F3.0F38.W0 30 /r VPMOVWB *xmm1/m128 {k1}{z}, ymm2*	A	V/V	AVX512VL AVX512BW	Converts 16 packed word integers from *ymm2* into 16 packed bytes in *xmm1/m128* with truncation under writemask k1.
EVEX.256.F3.0F38.W0 20 /r VPMOVSWB *xmm1/m128 {k1}{z}, ymm2*	A	V/V	AVX512VL AVX512BW	Converts 16 packed signed word integers from *ymm2* into 16 packed signed bytes in *xmm1/m128* using signed saturation under writemask k1.
EVEX.256.F3.0F38.W0 10 /r VPMOVUSWB *xmm1/m128 {k1}{z}, ymm2*	A	V/V	AVX512VL AVX512BW	Converts 16 packed unsigned word integers from *ymm2* into 16 packed unsigned bytes in *xmm1/m128* using unsigned saturation under writemask k1.
EVEX.512.F3.0F38.W0 30 /r VPMOVWB *ymm1/m256 {k1}{z}, zmm2*	A	V/V	AVX512BW	Converts 32 packed word integers from *zmm2* into 32 packed bytes in *ymm1/m256* with truncation under writemask k1.
EVEX.512.F3.0F38.W0 20 /r VPMOVSWB *ymm1/m256 {k1}{z}, zmm2*	A	V/V	AVX512BW	Converts 32 packed signed word integers from *zmm2* into 32 packed signed bytes in *ymm1/m256* using signed saturation under writemask k1.
EVEX.512.F3.0F38.W0 10 /r VPMOVUSWB *ymm1/m256 {k1}{z}, zmm2*	A	V/V	AVX512BW	Converts 32 packed unsigned word integers from *zmm2* into 32 packed unsigned bytes in *ymm1/m256* using unsigned saturation under writemask k1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Half Vector Mem	ModRM:r/m (w)	ModRM:reg (r)	NA	NA

Description

VPMOVWB down converts 16-bit integers into packed bytes using truncation. VPMOVSWB converts signed 16-bit integers into packed signed bytes using signed saturation. VPMOVUSWB convert unsigned word values into unsigned byte values using unsigned saturation.

The source operand is a ZMM/YMM/XMM register. The destination operand is a YMM/XMM/XMM register or a 256/128/64-bit memory location.

Down-converted byte elements are written to the destination operand (the first operand) from the least-significant byte. Byte elements of the destination operand are updated according to the writemask. Bits (MAXVL-1:256/128/64) of the register destination are zeroed.

Note: EVEX.vvvv is reserved and must be 1111b otherwise instructions will #UD.

Operation

VPMOVWB instruction (EVEX encoded versions) when dest is a register

```
(KL, VL) = (8, 128), (16, 256), (32, 512)
FOR j ← 0 TO KI-1
    i ← j * 8
    m ← j * 16
    IF k1[j] OR *no writemask*
        THEN DEST[i+7:i] ← TruncateWordToByte (SRC[m+15:m])
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+7:i] remains unchanged*
                ELSE *zeroing-masking*              ; zeroing-masking
                    DEST[i+7:i] = 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL/2] ← 0;
```

VPMOVWB instruction (EVEX encoded versions) when dest is memory

```
(KL, VL) = (8, 128), (16, 256), (32, 512)
FOR j ← 0 TO KI-1
    i ← j * 8
    m ← j * 16
    IF k1[j] OR *no writemask*
        THEN DEST[i+7:i] ← TruncateWordToByte (SRC[m+15:m])
        ELSE
            *DEST[i+7:i] remains unchanged*     ; merging-masking
    FI;
ENDFOR
```

VPMOVSWB instruction (EVEX encoded versions) when dest is a register

```
(KL, VL) = (8, 128), (16, 256), (32, 512)
FOR j ← 0 TO KI-1
    i ← j * 8
    m ← j * 16
    IF k1[j] OR *no writemask*
        THEN DEST[i+7:i] ← SaturateSignedWordToByte (SRC[m+15:m])
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+7:i] remains unchanged*
                ELSE *zeroing-masking*              ; zeroing-masking
                    DEST[i+7:i] = 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL/2] ← 0;
```

VPMOVSWB instruction (EVEX encoded versions) when dest is memory

```
(KL, VL) = (8, 128), (16, 256), (32, 512)
FOR j ← 0 TO Kl-1
    i ← j * 8
    m ← j * 16
    IF k1[j] OR *no writemask*
        THEN DEST[i+7:i] ← SaturateSignedWordToByte (SRC[m+15:m])
        ELSE
            *DEST[i+7:i] remains unchanged*      ; merging-masking
    FI;
ENDFOR
```

VPMOVUSWB instruction (EVEX encoded versions) when dest is a register

```
(KL, VL) = (8, 128), (16, 256), (32, 512)
FOR j ← 0 TO Kl-1
    i ← j * 8
    m ← j * 16
    IF k1[j] OR *no writemask*
        THEN DEST[i+7:i] ← SaturateUnsignedWordToByte (SRC[m+15:m])
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+7:i] remains unchanged*
                ELSE *zeroing-masking*          ; zeroing-masking
                    DEST[i+7:i] = 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL/2] ← 0;
```

VPMOVUSWB instruction (EVEX encoded versions) when dest is memory

```
(KL, VL) = (8, 128), (16, 256), (32, 512)
FOR j ← 0 TO Kl-1
    i ← j * 8
    m ← j * 16
    IF k1[j] OR *no writemask*
        THEN DEST[i+7:i] ← SaturateUnsignedWordToByte (SRC[m+15:m])
        ELSE
            *DEST[i+7:i] remains unchanged*      ; merging-masking
    FI;
ENDFOR
```

Intel C/C++ Compiler Intrinsic Equivalents

VPMOVUSWB __m256i _mm512_cvtusepi16_epi8(__m512i a);
VPMOVUSWB __m256i _mm512_mask_cvtusepi16_epi8(__m256i a, __mmask32 k, __m512i b);
VPMOVUSWB __m256i _mm512_maskz_cvtusepi16_epi8(__mmask32 k, __m512i b);
VPMOVUSWB void _mm512_mask_cvtusepi16_storeu_epi8(void * , __mmask32 k, __m512i b);
VPMOVSWB __m256i _mm512_cvtsepi16_epi8(__m512i a);
VPMOVSWB __m256i _mm512_mask_cvtsepi16_epi8(__m256i a, __mmask32 k, __m512i b);
VPMOVSWB __m256i _mm512_maskz_cvtsepi16_epi8(__mmask32 k, __m512i b);
VPMOVSWB void _mm512_mask_cvtsepi16_storeu_epi8(void * , __mmask32 k, __m512i b);
VPMOVWB __m256i _mm512_cvtepi16_epi8(__m512i a);
VPMOVWB __m256i _mm512_mask_cvtepi16_epi8(__m256i a, __mmask32 k, __m512i b);
VPMOVWB __m256i _mm512_maskz_cvtepi16_epi8(__mmask32 k, __m512i b);
VPMOVWB void _mm512_mask_cvtepi16_storeu_epi8(void * , __mmask32 k, __m512i b);
VPMOVUSWB __m128i _mm256_cvtusepi16_epi8(__m256i a);
VPMOVUSWB __m128i _mm256_mask_cvtusepi16_epi8(__m128i a, __mmask16 k, __m256i b);
VPMOVUSWB __m128i _mm256_maskz_cvtusepi16_epi8(__mmask16 k, __m256i b);
VPMOVUSWB void _mm256_mask_cvtusepi16_storeu_epi8(void * , __mmask16 k, __m256i b);
VPMOVUSWB __m128i _mm_cvtusepi16_epi8(__m128i a);
VPMOVUSWB __m128i _mm_mask_cvtusepi16_epi8(__m128i a, __mmask8 k, __m128i b);
VPMOVUSWB __m128i _mm_maskz_cvtusepi16_epi8(__mmask8 k, __m128i b);
VPMOVUSWB void _mm_mask_cvtusepi16_storeu_epi8(void * , __mmask8 k, __m128i b);
VPMOVSWB __m128i _mm256_cvtsepi16_epi8(__m256i a);
VPMOVSWB __m128i _mm256_mask_cvtsepi16_epi8(__m128i a, __mmask16 k, __m256i b);
VPMOVSWB __m128i _mm256_maskz_cvtsepi16_epi8(__mmask16 k, __m256i b);
VPMOVSWB void _mm256_mask_cvtsepi16_storeu_epi8(void * , __mmask16 k, __m256i b);
VPMOVSWB __m128i _mm_cvtsepi16_epi8(__m128i a);
VPMOVSWB __m128i _mm_mask_cvtsepi16_epi8(__m128i a, __mmask8 k, __m128i b);
VPMOVSWB __m128i _mm_maskz_cvtsepi16_epi8(__mmask8 k, __m128i b);
VPMOVSWB void _mm_mask_cvtsepi16_storeu_epi8(void * , __mmask8 k, __m128i b);
VPMOVWB __m128i _mm256_cvtepi16_epi8(__m256i a);
VPMOVWB __m128i _mm256_mask_cvtepi16_epi8(__m128i a, __mmask16 k, __m256i b);
VPMOVWB __m128i _mm256_maskz_cvtepi16_epi8(__mmask16 k, __m256i b);
VPMOVWB void _mm256_mask_cvtepi16_storeu_epi8(void * , __mmask16 k, __m256i b);
VPMOVWB __m128i _mm_cvtepi16_epi8(__m128i a);
VPMOVWB __m128i _mm_mask_cvtepi16_epi8(__m128i a, __mmask8 k, __m128i b);
VPMOVWB __m128i _mm_maskz_cvtepi16_epi8(__mmask8 k, __m128i b);
VPMOVWB void _mm_mask_cvtepi16_storeu_epi8(void * , __mmask8 k, __m128i b);

SIMD Floating-Point Exceptions

None

Other Exceptions

EVEX-encoded instruction, see Exceptions Type E6NF

#UD	If EVEX.vvvv != 1111B.

VPROLD/VPROLVD/VPROLQ/VPROLVQ—Bit Rotate Left

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.NDS.128.66.0F38.W0 15 /r VPROLVD xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst	B	V/V	AVX512VL AVX512F	Rotate doublewords in xmm2 left by count in the corresponding element of xmm3/m128/m32bcst. Result written to xmm1 under writemask k1.
EVEX.NDD.128.66.0F.W0 72 /1 ib VPROLD xmm1 {k1}{z}, xmm2/m128/m32bcst, imm8	A	V/V	AVX512VL AVX512F	Rotate doublewords in xmm2/m128/m32bcst left by imm8. Result written to xmm1 using writemask k1.
EVEX.NDS.128.66.0F38.W1 15 /r VPROLVQ xmm1 {k1}{z}, xmm2, xmm3/m128/m64bcst	B	V/V	AVX512VL AVX512F	Rotate quadwords in xmm2 left by count in the corresponding element of xmm3/m128/m64bcst. Result written to xmm1 under writemask k1.
EVEX.NDD.128.66.0F.W1 72 /1 ib VPROLQ xmm1 {k1}{z}, xmm2/m128/m64bcst, imm8	A	V/V	AVX512VL AVX512F	Rotate quadwords in xmm2/m128/m64bcst left by imm8. Result written to xmm1 using writemask k1.
EVEX.NDS.256.66.0F38.W0 15 /r VPROLVD ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst	B	V/V	AVX512VL AVX512F	Rotate doublewords in ymm2 left by count in the corresponding element of ymm3/m256/m32bcst. Result written to ymm1 under writemask k1.
EVEX.NDD.256.66.0F.W0 72 /1 ib VPROLD ymm1 {k1}{z}, ymm2/m256/m32bcst, imm8	A	V/V	AVX512VL AVX512F	Rotate doublewords in ymm2/m256/m32bcst left by imm8. Result written to ymm1 using writemask k1.
EVEX.NDS.256.66.0F38.W1 15 /r VPROLVQ ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst	B	V/V	AVX512VL AVX512F	Rotate quadwords in ymm2 left by count in the corresponding element of ymm3/m256/m64bcst. Result written to ymm1 under writemask k1.
EVEX.NDD.256.66.0F.W1 72 /1 ib VPROLQ ymm1 {k1}{z}, ymm2/m256/m64bcst, imm8	A	V/V	AVX512VL AVX512F	Rotate quadwords in ymm2/m256/m64bcst left by imm8. Result written to ymm1 using writemask k1.
EVEX.NDS.512.66.0F38.W0 15 /r VPROLVD zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst	B	V/V	AVX512F	Rotate left of doublewords in zmm2 by count in the corresponding element of zmm3/m512/m32bcst. Result written to zmm1 using writemask k1.
EVEX.NDD.512.66.0F.W0 72 /1 ib VPROLD zmm1 {k1}{z}, zmm2/m512/m32bcst, imm8	A	V/V	AVX512F	Rotate left of doublewords in zmm3/m512/m32bcst by imm8. Result written to zmm1 using writemask k1.
EVEX.NDS.512.66.0F38.W1 15 /r VPROLVQ zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst	B	V/V	AVX512F	Rotate quadwords in zmm2 left by count in the corresponding element of zmm3/m512/m64bcst. Result written to zmm1under writemask k1.
EVEX.NDD.512.66.0F.W1 72 /1 ib VPROLQ zmm1 {k1}{z}, zmm2/m512/m64bcst, imm8	A	V/V	AVX512F	Rotate quadwords in zmm2/m512/m64bcst left by imm8. Result written to zmm1 using writemask k1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Full Vector	VEX.vvvv (w)	ModRM:r/m (R)	Imm8	NA
B	Full Vector	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA

Description

Rotates the bits in the individual data elements (doublewords, or quadword) in the first source operand to the left by the number of bits specified in the count operand. If the value specified by the count operand is greater than 31 (for doublewords), or 63 (for a quadword), then the count operand modulo the data size (32 or 64) is used.

EVEX.128 encoded version: The destination operand is a XMM register. The source operand is a XMM register or a memory location (for immediate form). The count operand can come either from an XMM register or a memory location or an 8-bit immediate. Bits (MAXVL-1:128) of the corresponding ZMM register are zeroed.

EVEX.256 encoded version: The destination operand is a YMM register. The source operand is a YMM register or a memory location (for immediate form). The count operand can come either from an XMM register or a memory location or an 8-bit immediate. Bits (MAXVL-1:256) of the corresponding ZMM register are zeroed.

EVEX.512 encoded version: The destination operand is a ZMM register updated according to the writemask. For the count operand in immediate form, the source operand can be a ZMM register, a 512-bit memory location or a 512-bit vector broadcasted from a 32/64-bit memory location, the count operand is an 8-bit immediate. For the count operand in variable form, the first source operand (the second operand) is a ZMM register and the counter operand (the third operand) is a ZMM register, a 512-bit memory location or a 512-bit vector broadcasted from a 32/64-bit memory location.

Operation

```
LEFT_ROTATE_DWORDS(SRC, COUNT_SRC)
COUNT ← COUNT_SRC modulo 32;
DEST[31:0] ← (SRC << COUNT) | (SRC >> (32 - COUNT));

LEFT_ROTATE_QWORDS(SRC, COUNT_SRC)
COUNT ← COUNT_SRC modulo 64;
DEST[63:0] ← (SRC << COUNT) | (SRC >> (64 - COUNT));
```

VPROLD (EVEX encoded versions)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j ← 0 TO KL-1
    i ← j * 32
    IF k1[j] OR *no writemask* THEN
                IF (EVEX.b = 1) AND (SRC1 *is memory*)
                    THEN DEST[i+31:i] ← LEFT_ROTATE_DWORDS(SRC1[31:0], imm8)
                    ELSE DEST[i+31:i] ← LEFT_ROTATE_DWORDS(SRC1[i+31:i], imm8)
                FI;
          ELSE
                IF *merging-masking*                    ; merging-masking
                    THEN *DEST[i+31:i] remains unchanged*
                    ELSE *zeroing-masking*                  ; zeroing-masking
                        DEST[i+31:i] ← 0
                FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0
```

VPROLVD (EVEX encoded versions)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j ← 0 TO KL-1
    i ← j * 32
    IF k1[j] OR *no writemask* THEN
            IF (EVEX.b = 1) AND (SRC2 *is memory*)
                THEN DEST[i+31:i] ← LEFT_ROTATE_DWORDS(SRC1[i+31:i], SRC2[31:0])
                ELSE DEST[i+31:i] ← LEFT_ROTATE_DWORDS(SRC1[i+31:i], SRC2[i+31:i])
            FI;
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE *zeroing-masking*              ; zeroing-masking
                    DEST[i+31:i] ← 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0
```

VPROLQ (EVEX encoded versions)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j ← 0 TO KL-1
    i ← j * 64
    IF k1[j] OR *no writemask* THEN
            IF (EVEX.b = 1) AND (SRC1 *is memory*)
                THEN DEST[i+63:i] ← LEFT_ROTATE_QWORDS(SRC1[63:0], imm8)
                ELSE DEST[i+63:i] ← LEFT_ROTATE_QWORDS(SRC1[i+63:i], imm8)
            FI;
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE *zeroing-masking*              ; zeroing-masking
                    DEST[i+63:i] ← 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0
```

VPROLVQ (EVEX encoded versions)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j ← 0 TO KL-1
    i ← j * 64
    IF k1[j] OR *no writemask* THEN
                IF (EVEX.b = 1) AND (SRC2 *is memory*)
                    THEN DEST[i+63:i] ← LEFT_ROTATE_QWORDS(SRC1[i+63:i], SRC2[63:0])
                    ELSE DEST[i+63:i] ← LEFT_ROTATE_QWORDS(SRC1[i+63:i], SRC2[i+63:i])
                FI;
            ELSE
                IF *merging-masking*                    ; merging-masking
                    THEN *DEST[i+63:i] remains unchanged*
                    ELSE *zeroing-masking*                  ; zeroing-masking
                        DEST[i+63:i] ← 0
                FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VPROLD __m512i _mm512_rol_epi32(__m512i a, int imm);
VPROLD __m512i _mm512_mask_rol_epi32(__m512i a, __mmask16 k, __m512i b, int imm);
VPROLD __m512i _mm512_maskz_rol_epi32( __mmask16 k, __m512i a, int imm);
VPROLD __m256i _mm256_rol_epi32(__m256i a, int imm);
VPROLD __m256i _mm256_mask_rol_epi32(__m256i a, __mmask8 k, __m256i b, int imm);
VPROLD __m256i _mm256_maskz_rol_epi32( __mmask8 k, __m256i a, int imm);
VPROLD __m128i _mm_rol_epi32(__m128i a, int imm);
VPROLD __m128i _mm_mask_rol_epi32(__m128i a, __mmask8 k, __m128i b, int imm);
VPROLD __m128i _mm_maskz_rol_epi32( __mmask8 k, __m128i a, int imm);
VPROLQ __m512i _mm512_rol_epi64(__m512i a, int imm);
VPROLQ __m512i _mm512_mask_rol_epi64(__m512i a, __mmask8 k, __m512i b, int imm);
VPROLQ __m512i _mm512_maskz_rol_epi64(__mmask8 k, __m512i a, int imm);
VPROLQ __m256i _mm256_rol_epi64(__m256i a, int imm);
VPROLQ __m256i _mm256_mask_rol_epi64(__m256i a, __mmask8 k, __m256i b, int imm);
VPROLQ __m256i _mm256_maskz_rol_epi64( __mmask8 k, __m256i a, int imm);
VPROLQ __m128i _mm_rol_epi64(__m128i a, int imm);
VPROLQ __m128i _mm_mask_rol_epi64(__m128i a, __mmask8 k, __m128i b, int imm);
VPROLQ __m128i _mm_maskz_rol_epi64( __mmask8 k, __m128i a, int imm);
VPROLVD __m512i _mm512_rolv_epi32(__m512i a, __m512i cnt);
VPROLVD __m512i _mm512_mask_rolv_epi32(__m512i a, __mmask16 k, __m512i b, __m512i cnt);
VPROLVD __m512i _mm512_maskz_rolv_epi32(__mmask16 k, __m512i a, __m512i cnt);
VPROLVD __m256i _mm256_rolv_epi32(__m256i a, __m256i cnt);
VPROLVD __m256i _mm256_mask_rolv_epi32(__m256i a, __mmask8 k, __m256i b, __m256i cnt);
VPROLVD __m256i _mm256_maskz_rolv_epi32(__mmask8 k, __m256i a, __m256i cnt);
VPROLVD __m128i _mm_rolv_epi32(__m128i a, __m128i cnt);
VPROLVD __m128i _mm_mask_rolv_epi32(__m128i a, __mmask8 k, __m128i b, __m128i cnt);
VPROLVD __m128i _mm_maskz_rolv_epi32(__mmask8 k, __m128i a, __m128i cnt);
VPROLVQ __m512i _mm512_rolv_epi64(__m512i a, __m512i cnt);
VPROLVQ __m512i _mm512_mask_rolv_epi64(__m512i a, __mmask8 k, __m512i b, __m512i cnt);
VPROLVQ __m512i _mm512_maskz_rolv_epi64( __mmask8 k, __m512i a, __m512i cnt);
VPROLVQ __m256i _mm256_rolv_epi64(__m256i a, __m256i cnt);
VPROLVQ __m256i _mm256_mask_rolv_epi64(__m256i a, __mmask8 k, __m256i b, __m256i cnt);
VPROLVQ __m256i _mm256_maskz_rolv_epi64(__mmask8 k, __m256i a, __m256i cnt);
VPROLVQ __m128i _mm_rolv_epi64(__m128i a, __m128i cnt);
```

VPROLVQ __m128i _mm_mask_rolv_epi64(__m128i a, __mmask8 k, __m128i b, __m128i cnt);
VPROLVQ __m128i _mm_maskz_rolv_epi64(__mmask8 k, __m128i a, __m128i cnt);

SIMD Floating-Point Exceptions

None

Other Exceptions

EVEX-encoded instruction, see Exceptions Type E4.

VPRORD/VPRORVD/VPRORQ/VPRORVQ—Bit Rotate Right

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.NDS.128.66.0F38.W0 14 /r VPRORVD xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst	B	V/V	AVX512VL AVX512F	Rotate doublewords in xmm2 right by count in the corresponding element of xmm3/m128/m32bcst, store result using writemask k1.
EVEX.NDD.128.66.0F.W0 72 /0 ib VPRORD xmm1 {k1}{z}, xmm2/m128/m32bcst, imm8	A	V/V	AVX512VL AVX512F	Rotate doublewords in xmm2/m128/m32bcst right by imm8, store result using writemask k1.
EVEX.NDS.128.66.0F38.W1 14 /r VPRORVQ xmm1 {k1}{z}, xmm2, xmm3/m128/m64bcst	B	V/V	AVX512VL AVX512F	Rotate quadwords in xmm2 right by count in the corresponding element of xmm3/m128/m64bcst, store result using writemask k1.
EVEX.NDD.128.66.0F.W1 72 /0 ib VPRORQ xmm1 {k1}{z}, xmm2/m128/m64bcst, imm8	A	V/V	AVX512VL AVX512F	Rotate quadwords in xmm2/m128/m64bcst right by imm8, store result using writemask k1.
EVEX.NDS.256.66.0F38.W0 14 /r VPRORVD ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst	B	V/V	AVX512VL AVX512F	Rotate doublewords in ymm2 right by count in the corresponding element of ymm3/m256/m32bcst, store using result writemask k1.
EVEX.NDD.256.66.0F.W0 72 /0 ib VPRORD ymm1 {k1}{z}, ymm2/m256/m32bcst, imm8	A	V/V	AVX512VL AVX512F	Rotate doublewords in ymm2/m256/m32bcst right by imm8, store result using writemask k1.
EVEX.NDS.256.66.0F38.W1 14 /r VPRORVQ ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst	B	V/V	AVX512VL AVX512F	Rotate quadwords in ymm2 right by count in the corresponding element of ymm3/m256/m64bcst, store result using writemask k1.
EVEX.NDD.256.66.0F.W1 72 /0 ib VPRORQ ymm1 {k1}{z}, ymm2/m256/m64bcst, imm8	A	V/V	AVX512VL AVX512F	Rotate quadwords in ymm2/m256/m64bcst right by imm8, store result using writemask k1.
EVEX.NDS.512.66.0F38.W0 14 /r VPRORVD zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst	B	V/V	AVX512F	Rotate doublewords in zmm2 right by count in the corresponding element of zmm3/m512/m32bcst, store result using writemask k1.
EVEX.NDD.512.66.0F.W0 72 /0 ib VPRORD zmm1 {k1}{z}, zmm2/m512/m32bcst, imm8	A	V/V	AVX512F	Rotate doublewords in zmm2/m512/m32bcst right by imm8, store result using writemask k1.
EVEX.NDS.512.66.0F38.W1 14 /r VPRORVQ zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst	B	V/V	AVX512F	Rotate quadwords in zmm2 right by count in the corresponding element of zmm3/m512/m64bcst, store result using writemask k1.
EVEX.NDD.512.66.0F.W1 72 /0 ib VPRORQ zmm1 {k1}{z}, zmm2/m512/m64bcst, imm8	A	V/V	AVX512F	Rotate quadwords in zmm2/m512/m64bcst right by imm8, store result using writemask k1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Full Vector	VEX.vvvv (w)	ModRM:r/m (R)	Imm8	NA
B	Full Vector	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA

Description

Rotates the bits in the individual data elements (doublewords, or quadword) in the first source operand to the right by the number of bits specified in the count operand. If the value specified by the count operand is greater than 31 (for doublewords), or 63 (for a quadword), then the count operand modulo the data size (32 or 64) is used.

EVEX.128 encoded version: The destination operand is a XMM register. The source operand is a XMM register or a memory location (for immediate form). The count operand can come either from an XMM register or a memory location or an 8-bit immediate. Bits (MAXVL-1:128) of the corresponding ZMM register are zeroed.

EVEX.256 encoded version: The destination operand is a YMM register. The source operand is a YMM register or a memory location (for immediate form). The count operand can come either from an XMM register or a memory location or an 8-bit immediate. Bits (MAXVL-1:256) of the corresponding ZMM register are zeroed.

EVEX.512 encoded version: The destination operand is a ZMM register updated according to the writemask. For the count operand in immediate form, the source operand can be a ZMM register, a 512-bit memory location or a 512-bit vector broadcasted from a 32/64-bit memory location, the count operand is an 8-bit immediate. For the count operand in variable form, the first source operand (the second operand) is a ZMM register and the counter operand (the third operand) is a ZMM register, a 512-bit memory location or a 512-bit vector broadcasted from a 32/64-bit memory location.

Operation

```
RIGHT_ROTATE_DWORDS(SRC, COUNT_SRC)
COUNT ← COUNT_SRC modulo 32;
DEST[31:0] ← (SRC >> COUNT) | (SRC << (32 - COUNT));

RIGHT_ROTATE_QWORDS(SRC, COUNT_SRC)
COUNT ← COUNT_SRC modulo 64;
DEST[63:0] ← (SRC >> COUNT) | (SRC << (64 - COUNT));
```

VPRORD (EVEX encoded versions)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j ← 0 TO KL-1
    i ← j * 32
    IF k1[j] OR *no writemask* THEN
                IF (EVEX.b = 1) AND (SRC1 *is memory*)
                    THEN DEST[i+31:i] ← RIGHT_ROTATE_DWORDS( SRC1[31:0], imm8)
                    ELSE DEST[i+31:i] ← RIGHT_ROTATE_DWORDS(SRC1[i+31:i], imm8)
                FI;
          ELSE
                IF *merging-masking*                    ; merging-masking
                    THEN *DEST[i+31:i] remains unchanged*
                    ELSE *zeroing-masking*                  ; zeroing-masking
                        DEST[i+31:i] ← 0
                FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0
```

VPRORVD (EVEX encoded versions)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j ← 0 TO KL-1
    i ← j * 32
    IF k1[j] OR *no writemask* THEN
            IF (EVEX.b = 1) AND (SRC2 *is memory*)
                THEN DEST[i+31:i] ← RIGHT_ROTATE_DWORDS(SRC1[i+31:i], SRC2[31:0])
                ELSE DEST[i+31:i] ← RIGHT_ROTATE_DWORDS(SRC1[i+31:i], SRC2[i+31:i])
            FI;
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE *zeroing-masking*              ; zeroing-masking
                    DEST[i+31:i] ← 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0
```

VPRORQ (EVEX encoded versions)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j ← 0 TO KL-1
    i ← j * 64
    IF k1[j] OR *no writemask* THEN
            IF (EVEX.b = 1) AND (SRC1 *is memory*)
                THEN DEST[i+63:i] ← RIGHT_ROTATE_QWORDS(SRC1[63:0], imm8)
                ELSE DEST[i+63:i] ← RIGHT_ROTATE_QWORDS(SRC1[i+63:i], imm8])
            FI;
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE *zeroing-masking*              ; zeroing-masking
                    DEST[i+63:i] ← 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0
```

VPRORVQ (EVEX encoded versions)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j ← 0 TO KL-1
    i ← j * 64
    IF k1[j] OR *no writemask* THEN
            IF (EVEX.b = 1) AND (SRC2 *is memory*)
                THEN DEST[i+63:i] ← RIGHT_ROTATE_QWORDS(SRC1[i+63:i], SRC2[63:0])
                ELSE DEST[i+63:i] ← RIGHT_ROTATE_QWORDS(SRC1[i+63:i], SRC2[i+63:i])
            FI;
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE *zeroing-masking*                  ; zeroing-masking
                    DEST[i+63:i] ← 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VPRORD __m512i _mm512_ror_epi32(__m512i a, int imm);
VPRORD __m512i _mm512_mask_ror_epi32(__m512i a, __mmask16 k, __m512i b, int imm);
VPRORD __m512i _mm512_maskz_ror_epi32( __mmask16 k, __m512i a, int imm);
VPRORD __m256i _mm256_ror_epi32(__m256i a, int imm);
VPRORD __m256i _mm256_mask_ror_epi32(__m256i a, __mmask8 k, __m256i b, int imm);
VPRORD __m256i _mm256_maskz_ror_epi32( __mmask8 k, __m256i a, int imm);
VPRORD __m128i _mm_ror_epi32(__m128i a, int imm);
VPRORD __m128i _mm_mask_ror_epi32(__m128i a, __mmask8 k, __m128i b, int imm);
VPRORD __m128i _mm_maskz_ror_epi32( __mmask8 k, __m128i a, int imm);
VPRORQ __m512i _mm512_ror_epi64(__m512i a, int imm);
VPRORQ __m512i _mm512_mask_ror_epi64(__m512i a, __mmask8 k, __m512i b, int imm);
VPRORQ __m512i _mm512_maskz_ror_epi64(__mmask8 k, __m512i a, int imm);
VPRORQ __m256i _mm256_ror_epi64(__m256i a, int imm);
VPRORQ __m256i _mm256_mask_ror_epi64(__m256i a, __mmask8 k, __m256i b, int imm);
VPRORQ __m256i _mm256_maskz_ror_epi64( __mmask8 k, __m256i a, int imm);
VPRORQ __m128i _mm_ror_epi64(__m128i a, int imm);
VPRORQ __m128i _mm_mask_ror_epi64(__m128i a, __mmask8 k, __m128i b, int imm);
VPRORQ __m128i _mm_maskz_ror_epi64( __mmask8 k, __m128i a, int imm);
VPRORVD __m512i _mm512_rorv_epi32(__m512i a, __m512i cnt);
VPRORVD __m512i _mm512_mask_rorv_epi32(__m512i a, __mmask16 k, __m512i b, __m512i cnt);
VPRORVD __m512i _mm512_maskz_rorv_epi32(__mmask16 k, __m512i a, __m512i cnt);
VPRORVD __m256i _mm256_rorv_epi32(__m256i a, __m256i cnt);
VPRORVD __m256i _mm256_mask_rorv_epi32(__m256i a, __mmask8 k, __m256i b, __m256i cnt);
VPRORVD __m256i _mm256_maskz_rorv_epi32(__mmask8 k, __m256i a, __m256i cnt);
VPRORVD __m128i _mm_rorv_epi32(__m128i a, __m128i cnt);
VPRORVD __m128i _mm_mask_rorv_epi32(__m128i a, __mmask8 k, __m128i b, __m128i cnt);
VPRORVD __m128i _mm_maskz_rorv_epi32(__mmask8 k, __m128i a, __m128i cnt);
VPRORVQ __m512i _mm512_rorv_epi64(__m512i a, __m512i cnt);
VPRORVQ __m512i _mm512_mask_rorv_epi64(__m512i a, __mmask8 k, __m512i b, __m512i cnt);
VPRORVQ __m512i _mm512_maskz_rorv_epi64( __mmask8 k, __m512i a, __m512i cnt);
VPRORVQ __m256i _mm256_rorv_epi64(__m256i a, __m256i cnt);
VPRORVQ __m256i _mm256_mask_rorv_epi64(__m256i a, __mmask8 k, __m256i b, __m256i cnt);
VPRORVQ __m256i _mm256_maskz_rorv_epi64(__mmask8 k, __m256i a, __m256i cnt);
VPRORVQ __m128i _mm_rorv_epi64(__m128i a, __m128i cnt);
```

VPRORVQ __m128i _mm_mask_rorv_epi64(__m128i a, __mmask8 k, __m128i b, __m128i cnt);
VPRORVQ __m128i _mm_maskz_rorv_epi64(__mmask8 k, __m128i a, __m128i cnt);

SIMD Floating-Point Exceptions

None

Other Exceptions

EVEX-encoded instruction, see Exceptions Type E4.

VPSCATTERDD/VPSCATTERDQ/VPSCATTERQD/VPSCATTERQQ—Scatter Packed Dword, Packed Qword with Signed Dword, Signed Qword Indices

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.128.66.0F38.W0 A0 /vsib VPSCATTERDD vm32x {k1}, xmm1	A	V/V	AVX512VL AVX512F	Using signed dword indices, scatter dword values to memory using writemask k1.
EVEX.256.66.0F38.W0 A0 /vsib VPSCATTERDD vm32y {k1}, ymm1	A	V/V	AVX512VL AVX512F	Using signed dword indices, scatter dword values to memory using writemask k1.
EVEX.512.66.0F38.W0 A0 /vsib VPSCATTERDD vm32z {k1}, zmm1	A	V/V	AVX512F	Using signed dword indices, scatter dword values to memory using writemask k1.
EVEX.128.66.0F38.W1 A0 /vsib VPSCATTERDQ vm32x {k1}, xmm1	A	V/V	AVX512VL AVX512F	Using signed dword indices, scatter qword values to memory using writemask k1.
EVEX.256.66.0F38.W1 A0 /vsib VPSCATTERDQ vm32x {k1}, ymm1	A	V/V	AVX512VL AVX512F	Using signed dword indices, scatter qword values to memory using writemask k1.
EVEX.512.66.0F38.W1 A0 /vsib VPSCATTERDQ vm32y {k1}, zmm1	A	V/V	AVX512F	Using signed dword indices, scatter qword values to memory using writemask k1.
EVEX.128.66.0F38.W0 A1 /vsib VPSCATTERQD vm64x {k1}, xmm1	A	V/V	AVX512VL AVX512F	Using signed qword indices, scatter dword values to memory using writemask k1.
EVEX.256.66.0F38.W0 A1 /vsib VPSCATTERQD vm64y {k1}, xmm1	A	V/V	AVX512VL AVX512F	Using signed qword indices, scatter dword values to memory using writemask k1.
EVEX.512.66.0F38.W0 A1 /vsib VPSCATTERQD vm64z {k1}, ymm1	A	V/V	AVX512F	Using signed qword indices, scatter dword values to memory using writemask k1.
EVEX.128.66.0F38.W1 A1 /vsib VPSCATTERQQ vm64x {k1}, xmm1	A	V/V	AVX512VL AVX512F	Using signed qword indices, scatter qword values to memory using writemask k1.
EVEX.256.66.0F38.W1 A1 /vsib VPSCATTERQQ vm64y {k1}, ymm1	A	V/V	AVX512VL AVX512F	Using signed qword indices, scatter qword values to memory using writemask k1.
EVEX.512.66.0F38.W1 A1 /vsib VPSCATTERQQ vm64z {k1}, zmm1	A	V/V	AVX512F	Using signed qword indices, scatter qword values to memory using writemask k1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Tuple1 Scalar	BaseReg (R): VSIB:base, VectorReg(R): VSIB:index	ModRM:reg (r)	NA	NA

Description

Stores up to 16 elements (8 elements for qword indices) in doubleword vector or 8 elements in quadword vector to the memory locations pointed by base address BASE_ADDR and index vector VINDEX, with scale SCALE. The elements are specified via the VSIB (i.e., the index register is a vector register, holding packed indices). Elements will only be stored if their corresponding mask bit is one. The entire mask register will be set to zero by this instruction unless it triggers an exception.

This instruction can be suspended by an exception if at least one element is already scattered (i.e., if the exception is triggered by an element other than the rightmost one with its mask bit set). When this happens, the destination register and the mask register are partially updated. If any traps or interrupts are pending from already scattered elements, they will be delivered in lieu of the exception; in this case, EFLAG.RF is set to one so an instruction breakpoint is not re-triggered when the instruction is continued.

Note that:

- Only writes to overlapping vector indices are guaranteed to be ordered with respect to each other (from LSB to MSB of the source registers). Note that this also include partially overlapping vector indices. Writes that are not overlapped may happen in any order. Memory ordering with other instructions follows the Intel-64 memory ordering model. Note that this does not account for non-overlapping indices that map into the same physical address locations.

- If two or more destination indices completely overlap, the "earlier" write(s) may be skipped.
- Faults are delivered in a right-to-left manner. That is, if a fault is triggered by an element and delivered, all elements closer to the LSB of the destination ZMM will be completed (and non-faulting). Individual elements closer to the MSB may or may not be completed. If a given element triggers multiple faults, they are delivered in the conventional order.
- Elements may be scattered in any order, but faults must be delivered in a right-to left order; thus, elements to the left of a faulting one may be gathered before the fault is delivered. A given implementation of this instruction is repeatable - given the same input values and architectural state, the same set of elements to the left of the faulting one will be gathered.
- This instruction does not perform AC checks, and so will never deliver an AC fault.
- Not valid with 16-bit effective addresses. Will deliver a #UD fault.
- If this instruction overwrites itself and then takes a fault, only a subset of elements may be completed before the fault is delivered (as described above). If the fault handler completes and attempts to re-execute this instruction, the new instruction will be executed, and the scatter will not complete.

Note that the presence of VSIB byte is enforced in this instruction. Hence, the instruction will #UD fault if ModRM.rm is different than 100b.

This instruction has special disp8*N and alignment rules. N is considered to be the size of a single vector element.

The scaled index may require more bits to represent than the address bits used by the processor (e.g., in 32-bit mode, if the scale is greater than one). In this case, the most significant bits beyond the number of address bits are ignored.

The instruction will #UD fault if the k0 mask register is specified.

The instruction will #UD fault if EVEX.Z = 1.

Operation

BASE_ADDR stands for the memory operand base address (a GPR); may not exist
VINDEX stands for the memory operand vector of indices (a ZMM register)
SCALE stands for the memory operand scalar (1, 2, 4 or 8)
DISP is the optional 1, 2 or 4 byte displacement

VPSCATTERDD (EVEX encoded versions)

```
(KL, VL)= (4, 128), (8, 256), (16, 512)
FOR j ← 0 TO KL-1
    i ← j * 32
    IF k1[j] OR *no writemask*
        THEN MEM[BASE_ADDR +SignExtend(VINDEX[i+31:i]) * SCALE + DISP] ←    SRC[i+31:i]
            k1[j] ← 0

    FI;
ENDFOR
k1[MAX_KL-1:KL] ← 0
```

VPSCATTERDQ (EVEX encoded versions)

```
(KL, VL)= (2, 128), (4, 256), (8, 512)
FOR j ← 0 TO KL-1
    i ← j * 64
    k ← j * 32
    IF k1[j] OR *no writemask*
        THEN MEM[BASE_ADDR +SignExtend(VINDEX[k+31:k]) * SCALE + DISP] ←    SRC[i+63:i]
            k1[j] ← 0
    FI;
ENDFOR
k1[MAX_KL-1:KL] ← 0
```

VPSCATTERQD (EVEX encoded versions)

```
(KL, VL)= (2, 128), (4, 256), (8, 512)
FOR j ← 0 TO KL-1
    i ← j * 32
    k ← j * 64
    IF k1[j] OR *no writemask*
        THEN MEM[BASE_ADDR + (VINDEX[k+63:k]) * SCALE + DISP] ←    SRC[i+31:i]
            k1[j] ← 0

    FI;
ENDFOR
k1[MAX_KL-1:KL] ← 0
```

VPSCATTERQQ (EVEX encoded versions)

```
(KL, VL)= (2, 128), (4, 256), (8, 512)
FOR j ← 0 TO KL-1
    i ← j * 64
    IF k1[j] OR *no writemask*
        THEN MEM[BASE_ADDR + (VINDEX[j+63:j]) * SCALE + DISP] ←    SRC[i+63:i]
    FI;
ENDFOR
k1[MAX_KL-1:KL] ← 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VPSCATTERDD void _mm512_i32scatter_epi32(void * base, __m512i vdx, __m512i a, int scale);
VPSCATTERDD void _mm256_i32scatter_epi32(void * base, __m256i vdx, __m256i a, int scale);
VPSCATTERDD void _mm_i32scatter_epi32(void * base, __m128i vdx, __m128i a, int scale);
VPSCATTERDD void _mm512_mask_i32scatter_epi32(void * base, __mmask16 k, __m512i vdx, __m512i a, int scale);
VPSCATTERDD void _mm256_mask_i32scatter_epi32(void * base, __mmask8 k, __m256i vdx, __m256i a, int scale);
VPSCATTERDD void _mm_mask_i32scatter_epi32(void * base, __mmask8 k, __m128i vdx, __m128i a, int scale);
VPSCATTERDQ void _mm512_i32scatter_epi64(void * base, __m256i vdx, __m512i a, int scale);
VPSCATTERDQ void _mm256_i32scatter_epi64(void * base, __m128i vdx, __m256i a, int scale);
VPSCATTERDQ void _mm_i32scatter_epi64(void * base, __m128i vdx, __m128i a, int scale);
VPSCATTERDQ void _mm512_mask_i32scatter_epi64(void * base, __mmask8 k, __m256i vdx, __m512i a, int scale);
VPSCATTERDQ void _mm256_mask_i32scatter_epi64(void * base, __mmask8 k, __m128i vdx, __m256i a, int scale);
VPSCATTERDQ void _mm_mask_i32scatter_epi64(void * base, __mmask8 k, __m128i vdx, __m128i a, int scale);
VPSCATTERQD void _mm512_i64scatter_epi32(void * base, __m512i vdx, __m256i a, int scale);
VPSCATTERQD void _mm256_i64scatter_epi32(void * base, __m256i vdx, __m128i a, int scale);
VPSCATTERQD void _mm_i64scatter_epi32(void * base, __m128i vdx, __m128i a, int scale);
VPSCATTERQD void _mm512_mask_i64scatter_epi32(void * base, __mmask8 k, __m512i vdx, __m256i a, int scale);
VPSCATTERQD void _mm256_mask_i64scatter_epi32(void * base, __mmask8 k, __m256i vdx, __m128i a, int scale);
VPSCATTERQD void _mm_mask_i64scatter_epi32(void * base, __mmask8 k, __m128i vdx, __m128i a, int scale);
VPSCATTERQQ void _mm512_i64scatter_epi64(void * base, __m512i vdx, __m512i a, int scale);
VPSCATTERQQ void _mm256_i64scatter_epi64(void * base, __m256i vdx, __m256i a, int scale);
VPSCATTERQQ void _mm_i64scatter_epi64(void * base, __m128i vdx, __m128i a, int scale);
VPSCATTERQQ void _mm512_mask_i64scatter_epi64(void * base, __mmask8 k, __m512i vdx, __m512i a, int scale);
VPSCATTERQQ void _mm256_mask_i64scatter_epi64(void * base, __mmask8 k, __m256i vdx, __m256i a, int scale);
VPSCATTERQQ void _mm_mask_i64scatter_epi64(void * base, __mmask8 k, __m128i vdx, __m128i a, int scale);

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type E12.

VPSLLVW/VPSLLVD/VPSLLVQ—Variable Bit Shift Left Logical

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
VEX.NDS.128.66.0F38.W0 47 /r VPSLLVD xmm1, xmm2, xmm3/m128	A	V/V	AVX2	Shift doublewords in xmm2 left by amount specified in the corresponding element of xmm3/m128 while shifting in 0s.
VEX.NDS.128.66.0F38.W1 47 /r VPSLLVQ xmm1, xmm2, xmm3/m128	A	V/V	AVX2	Shift quadwords in xmm2 left by amount specified in the corresponding element of xmm3/m128 while shifting in 0s.
VEX.NDS.256.66.0F38.W0 47 /r VPSLLVD ymm1, ymm2, ymm3/m256	A	V/V	AVX2	Shift doublewords in ymm2 left by amount specified in the corresponding element of ymm3/m256 while shifting in 0s.
VEX.NDS.256.66.0F38.W1 47 /r VPSLLVQ ymm1, ymm2, ymm3/m256	A	V/V	AVX2	Shift quadwords in ymm2 left by amount specified in the corresponding element of ymm3/m256 while shifting in 0s.
EVEX.NDS.128.66.0F38.W1 12 /r VPSLLVW xmm1 {k1}{z}, xmm2, xmm3/m128	B	V/V	AVX512VL AVX512BW	Shift words in xmm2 left by amount specified in the corresponding element of xmm3/m128 while shifting in 0s using writemask k1.
EVEX.NDS.256.66.0F38.W1 12 /r VPSLLVW ymm1 {k1}{z}, ymm2, ymm3/m256	B	V/V	AVX512VL AVX512BW	Shift words in ymm2 left by amount specified in the corresponding element of ymm3/m256 while shifting in 0s using writemask k1.
EVEX.NDS.512.66.0F38.W1 12 /r VPSLLVW zmm1 {k1}{z}, zmm2, zmm3/m512	B	V/V	AVX512BW	Shift words in zmm2 left by amount specified in the corresponding element of zmm3/m512 while shifting in 0s using writemask k1.
EVEX.NDS.128.66.0F38.W0 47 /r VPSLLVD xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst	C	V/V	AVX512VL AVX512F	Shift doublewords in xmm2 left by amount specified in the corresponding element of xmm3/m128/m32bcst while shifting in 0s using writemask k1.
EVEX.NDS.256.66.0F38.W0 47 /r VPSLLVD ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst	C	V/V	AVX512VL AVX512F	Shift doublewords in ymm2 left by amount specified in the corresponding element of ymm3/m256/m32bcst while shifting in 0s using writemask k1.
EVEX.NDS.512.66.0F38.W0 47 /r VPSLLVD zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst	C	V/V	AVX512F	Shift doublewords in zmm2 left by amount specified in the corresponding element of zmm3/m512/m32bcst while shifting in 0s using writemask k1.
EVEX.NDS.128.66.0F38.W1 47 /r VPSLLVQ xmm1 {k1}{z}, xmm2, xmm3/m128/m64bcst	C	V/V	AVX512VL AVX512F	Shift quadwords in xmm2 left by amount specified in the corresponding element of xmm3/m128/m64bcst while shifting in 0s using writemask k1.
EVEX.NDS.256.66.0F38.W1 47 /r VPSLLVQ ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst	C	V/V	AVX512VL AVX512F	Shift quadwords in ymm2 left by amount specified in the corresponding element of ymm3/m256/m64bcst while shifting in 0s using writemask k1.
EVEX.NDS.512.66.0F38.W1 47 /r VPSLLVQ zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst	C	V/V	AVX512F	Shift quadwords in zmm2 left by amount specified in the corresponding element of zmm3/m512/m64bcst while shifting in 0s using writemask k1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	NA	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA
B	Full Vector Mem	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA
C	Full Vector	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA

Description

Shifts the bits in the individual data elements (words, doublewords or quadword) in the first source operand to the left by the count value of respective data elements in the second source operand. As the bits in the data elements are shifted left, the empty low-order bits are cleared (set to 0).

The count values are specified individually in each data element of the second source operand. If the unsigned integer value specified in the respective data element of the second source operand is greater than 15 (for word), 31 (for doublewords), or 63 (for a quadword), then the destination data element are written with 0.

VEX.128 encoded version: The destination and first source operands are XMM registers. The count operand can be either an XMM register or a 128-bit memory location. Bits (MAXVL-1:128) of the corresponding destination register are zeroed.

VEX.256 encoded version: The destination and first source operands are YMM registers. The count operand can be either an YMM register or a 256-bit memory. Bits (MAXVL-1:256) of the corresponding ZMM register are zeroed.

EVEX encoded VPSLLVD/Q: The destination and first source operands are ZMM/YMM/XMM registers. The count operand can be either a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512-bit vector broadcasted from a 32/64-bit memory location. The destination is conditionally updated with writemask k1.

EVEX encoded VPSLLVW: The destination and first source operands are ZMM/YMM/XMM registers. The count operand can be either a ZMM/YMM/XMM register, a 512/256/128-bit memory location. The destination is conditionally updated with writemask k1.

Operation

VPSLLVW (EVEX encoded version)

```
(KL, VL) = (8, 128), (16, 256), (32, 512)

FOR j ← 0 TO KL-1
    i ← j * 16
    IF k1[j] OR *no writemask*
        THEN DEST[i+15:i] ← ZeroExtend(SRC1[i+15:i] << SRC2[i+15:i])
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+15:i] remains unchanged*
                ELSE                            ; zeroing-masking
                    DEST[i+15:i] ← 0
            FI
    FI;
ENDFOR;
DEST[MAXVL-1:VL] ← 0;
```

VPSLLVD (VEX.128 version)

```
COUNT_0 ←SRC2[31 : 0]
    (* Repeat Each COUNT_i for the 2nd through 4th dwords of SRC2*)
COUNT_3 ←SRC2[100 : 96];
IF COUNT_0 < 32 THEN
DEST[31:0] ←ZeroExtend(SRC1[31:0] << COUNT_0);
ELSE
DEST[31:0] ←0;
    (* Repeat shift operation for 2nd through 4th dwords *)
IF COUNT_3 < 32 THEN
DEST[127:96] ←ZeroExtend(SRC1[127:96] << COUNT_3);
ELSE
DEST[127:96] ←0;
DEST[MAXVL-1:128] ←0;
```

VPSLLVD (VEX.256 version)

```
COUNT_0 ←SRC2[31 : 0];
    (* Repeat Each COUNT_i for the 2nd through 7th dwords of SRC2*)
COUNT_7 ←SRC2[228 : 224];
IF COUNT_0 < 32 THEN
DEST[31:0] ←ZeroExtend(SRC1[31:0] << COUNT_0);
ELSE
DEST[31:0] ←0;
    (* Repeat shift operation for 2nd through 7th dwords *)
IF COUNT_7 < 32 THEN
DEST[255:224] ←ZeroExtend(SRC1[255:224] << COUNT_7);
ELSE
DEST[255:224] ←0;
DEST[MAXVL-1:256] ← 0;
```

VPSLLVD (EVEX encoded version)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j ← 0 TO KL-1
    i ← j * 32
    IF k1[j] OR *no writemask* THEN
            IF (EVEX.b = 1) AND (SRC2 *is memory*)
                THEN DEST[i+31:i] ← ZeroExtend(SRC1[i+31:i] << SRC2[31:0])
                ELSE DEST[i+31:i] ← ZeroExtend(SRC1[i+31:i] << SRC2[i+31:i])
            FI;
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE                                ; zeroing-masking
                    DEST[i+31:i] ← 0
            FI
    FI;
ENDFOR;
DEST[MAXVL-1:VL] ← 0;
```

VPSLLVQ (VEX.128 version)

```
COUNT_0 ←SRC2[63 : 0];
COUNT_1 ←SRC2[127 : 64];
IF COUNT_0 < 64THEN
DEST[63:0] ←ZeroExtend(SRC1[63:0] << COUNT_0);
ELSE
DEST[63:0] ←0;
IF COUNT_1 < 64 THEN
DEST[127:64] ←ZeroExtend(SRC1[127:64] << COUNT_1);
ELSE
DEST[127:96] ←0;
DEST[MAXVL-1:128] ←0;
```

VPSLLVQ (VEX.256 version)

```
COUNT_0 ←SRC2[63 : 0];
    (* Repeat Each COUNT_i for the 2nd through 4th dwords of SRC2*)
COUNT_3 ←SRC2[197 : 192];
IF COUNT_0 < 64THEN
DEST[63:0] ←ZeroExtend(SRC1[63:0] << COUNT_0);
ELSE
DEST[63:0] ←0;
    (* Repeat shift operation for 2nd through 4th dwords *)
IF COUNT_3 < 64 THEN
DEST[255:192] ←ZeroExtend(SRC1[255:192] << COUNT_3);
ELSE
DEST[255:192] ←0;
DEST[MAXVL-1:256] ← 0;
```

VPSLLVQ (EVEX encoded version)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j ← 0 TO KL-1
    i ← j * 64
    IF k1[j] OR *no writemask* THEN
            IF (EVEX.b = 1) AND (SRC2 *is memory*)
                THEN DEST[i+63:i] ← ZeroExtend(SRC1[i+63:i] << SRC2[63:0])
                ELSE DEST[i+63:i] ← ZeroExtend(SRC1[i+63:i] << SRC2[i+63:i])
            FI;
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE                                ; zeroing-masking
                    DEST[i+63:i] ← 0
            FI
    FI;
ENDFOR;
DEST[MAXVL-1:VL] ← 0;
```

Intel C/C++ Compiler Intrinsic Equivalent

VPSLLVW __m512i _mm512_sllv_epi16(__m512i a, __m512i cnt);
VPSLLVW __m512i _mm512_mask_sllv_epi16(__m512i s, __mmask32 k, __m512i a, __m512i cnt);
VPSLLVW __m512i _mm512_maskz_sllv_epi16(__mmask32 k, __m512i a, __m512i cnt);
VPSLLVW __m256i _mm256_mask_sllv_epi16(__m256i s, __mmask16 k, __m256i a, __m256i cnt);
VPSLLVW __m256i _mm256_maskz_sllv_epi16(__mmask16 k, __m256i a, __m256i cnt);
VPSLLVW __m128i _mm_mask_sllv_epi16(__m128i s, __mmask8 k, __m128i a, __m128i cnt);
VPSLLVW __m128i _mm_maskz_sllv_epi16(__mmask8 k, __m128i a, __m128i cnt);
VPSLLVD __m512i _mm512_sllv_epi32(__m512i a, __m512i cnt);
VPSLLVD __m512i _mm512_mask_sllv_epi32(__m512i s, __mmask16 k, __m512i a, __m512i cnt);
VPSLLVD __m512i _mm512_maskz_sllv_epi32(__mmask16 k, __m512i a, __m512i cnt);
VPSLLVD __m256i _mm256_mask_sllv_epi32(__m256i s, __mmask8 k, __m256i a, __m256i cnt);
VPSLLVD __m256i _mm256_maskz_sllv_epi32(__mmask8 k, __m256i a, __m256i cnt);
VPSLLVD __m128i _mm_mask_sllv_epi32(__m128i s, __mmask8 k, __m128i a, __m128i cnt);
VPSLLVD __m128i _mm_maskz_sllv_epi32(__mmask8 k, __m128i a, __m128i cnt);
VPSLLVQ __m512i _mm512_sllv_epi64(__m512i a, __m512i cnt);
VPSLLVQ __m512i _mm512_mask_sllv_epi64(__m512i s, __mmask8 k, __m512i a, __m512i cnt);
VPSLLVQ __m512i _mm512_maskz_sllv_epi64(__mmask8 k, __m512i a, __m512i cnt);
VPSLLVD __m256i _mm256_mask_sllv_epi64(__m256i s, __mmask8 k, __m256i a, __m256i cnt);
VPSLLVD __m256i _mm256_maskz_sllv_epi64(__mmask8 k, __m256i a, __m256i cnt);
VPSLLVD __m128i _mm_mask_sllv_epi64(__m128i s, __mmask8 k, __m128i a, __m128i cnt);
VPSLLVD __m128i _mm_maskz_sllv_epi64(__mmask8 k, __m128i a, __m128i cnt);
VPSLLVD __m256i _mm256_sllv_epi32 (__m256i m, __m256i count)
VPSLLVQ __m256i _mm256_sllv_epi64 (__m256i m, __m256i count)

SIMD Floating-Point Exceptions

None

Other Exceptions

VEX-encoded instructions, see Exceptions Type 4.
EVEX-encoded VPSLLVD/VPSLLVQ, see Exceptions Type E4.
EVEX-encoded VPSLLVW, see Exceptions Type E4.nb.

VPSRAVW/VPSRAVD/VPSRAVQ—Variable Bit Shift Right Arithmetic

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
VEX.NDS.128.66.0F38.W0 46 /r VPSRAVD xmm1, xmm2, xmm3/m128	A	V/V	AVX2	Shift doublewords in xmm2 right by amount specified in the corresponding element of xmm3/m128 while shifting in sign bits.
VEX.NDS.256.66.0F38.W0 46 /r VPSRAVD ymm1, ymm2, ymm3/m256	A	V/V	AVX2	Shift doublewords in ymm2 right by amount specified in the corresponding element of ymm3/m256 while shifting in sign bits.
EVEX.NDS.128.66.0F38.W1 11 /r VPSRAVW xmm1 {k1}{z}, xmm2, xmm3/m128	B	V/V	AVX512VL AVX512BW	Shift words in xmm2 right by amount specified in the corresponding element of xmm3/m128 while shifting in sign bits using writemask k1.
EVEX.NDS.256.66.0F38.W1 11 /r VPSRAVW ymm1 {k1}{z}, ymm2, ymm3/m256	B	V/V	AVX512VL AVX512BW	Shift words in ymm2 right by amount specified in the corresponding element of ymm3/m256 while shifting in sign bits using writemask k1.
EVEX.NDS.512.66.0F38.W1 11 /r VPSRAVW zmm1 {k1}{z}, zmm2, zmm3/m512	B	V/V	AVX512BW	Shift words in zmm2 right by amount specified in the corresponding element of zmm3/m512 while shifting in sign bits using writemask k1.
EVEX.NDS.128.66.0F38.W0 46 /r VPSRAVD xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst	C	V/V	AVX512VL AVX512F	Shift doublewords in xmm2 right by amount specified in the corresponding element of xmm3/m128/m32bcst while shifting in sign bits using writemask k1.
EVEX.NDS.256.66.0F38.W0 46 /r VPSRAVD ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst	C	V/V	AVX512VL AVX512F	Shift doublewords in ymm2 right by amount specified in the corresponding element of ymm3/m256/m32bcst while shifting in sign bits using writemask k1.
EVEX.NDS.512.66.0F38.W0 46 /r VPSRAVD zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst	C	V/V	AVX512F	Shift doublewords in zmm2 right by amount specified in the corresponding element of zmm3/m512/m32bcst while shifting in sign bits using writemask k1.
EVEX.NDS.128.66.0F38.W1 46 /r VPSRAVQ xmm1 {k1}{z}, xmm2, xmm3/m128/m64bcst	C	V/V	AVX512VL AVX512F	Shift quadwords in xmm2 right by amount specified in the corresponding element of xmm3/m128/m64bcst while shifting in sign bits using writemask k1.
EVEX.NDS.256.66.0F38.W1 46 /r VPSRAVQ ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst	C	V/V	AVX512VL AVX512F	Shift quadwords in ymm2 right by amount specified in the corresponding element of ymm3/m256/m64bcst while shifting in sign bits using writemask k1.
EVEX.NDS.512.66.0F38.W1 46 /r VPSRAVQ zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst	C	V/V	AVX512F	Shift quadwords in zmm2 right by amount specified in the corresponding element of zmm3/m512/m64bcst while shifting in sign bits using writemask k1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	NA	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA
B	Full Vector Mem	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA
C	Full Vector	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA

Description

Shifts the bits in the individual data elements (word/doublewords/quadword) in the first source operand (the second operand) to the right by the number of bits specified in the count value of respective data elements in the second source operand (the third operand). As the bits in the data elements are shifted right, the empty high-order bits are set to the MSB (sign extension).

The count values are specified individually in each data element of the second source operand. If the unsigned integer value specified in the respective data element of the second source operand is greater than 15 (for words), 31 (for doublewords), or 63 (for a quadword), then the destination data element are filled with the corresponding sign bit of the source element.

The count values are specified individually in each data element of the second source operand. If the unsigned integer value specified in the respective data element of the second source operand is greater than 16 (for word), 31 (for doublewords), or 63 (for a quadword), then the destination data element are written with 0.

VEX.128 encoded version: The destination and first source operands are XMM registers. The count operand can be either an XMM register or a 128-bit memory location. Bits (MAXVL-1:128) of the corresponding destination register are zeroed.

VEX.256 encoded version: The destination and first source operands are YMM registers. The count operand can be either an YMM register or a 256-bit memory. Bits (MAXVL-1:256) of the corresponding destination register are zeroed.

EVEX.512/256/128 encoded VPSRAVD/W: The destination and first source operands are ZMM/YMM/XMM registers. The count operand can be either a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 32/64-bit memory location. The destination is conditionally updated with writemask k1.

EVEX.512/256/128 encoded VPSRAVQ: The destination and first source operands are ZMM/YMM/XMM registers. The count operand can be either a ZMM/YMM/XMM register, a 512/256/128-bit memory location. The destination is conditionally updated with writemask k1.

Operation

VPSRAVW (EVEX encoded version)

(KL, VL) = (8, 128), (16, 256), (32, 512)

```
FOR j ← 0 TO KL-1
    i ← j * 16
    IF k1[j] OR *no writemask*
        THEN
            COUNT ← SRC2[i+3:i]
            IF COUNT < 16
                THEN    DEST[i+15:i] ← SignExtend(SRC1[i+15:i] >> COUNT)
                ELSE
                    FOR k← 0 TO 15
                        DEST[i+k] ← SRC1[i+15]
                    ENDFOR;
            FI
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+15:i] remains unchanged*
                ELSE                                ; zeroing-masking
                    DEST[i+15:i] ← 0
            FI
    FI;
ENDFOR;
DEST[MAXVL-1:VL] ← 0;
```

VPSRAVD (VEX.128 version)

```
COUNT_0 ← SRC2[31 : 0]
    (* Repeat Each COUNT_i for the 2nd through 4th dwords of SRC2*)
COUNT_3 ← SRC2[100 : 96];
DEST[31:0] ← SignExtend(SRC1[31:0] >> COUNT_0);
    (* Repeat shift operation for 2nd through 4th dwords *)
DEST[127:96] ← SignExtend(SRC1[127:96] >> COUNT_3);
DEST[MAXVL-1:128] ← 0;
```

VPSRAVD (VEX.256 version)

```
COUNT_0 ← SRC2[31 : 0];
    (* Repeat Each COUNT_i for the 2nd through 8th dwords of SRC2*)
COUNT_7 ← SRC2[228 : 224];
DEST[31:0] ← SignExtend(SRC1[31:0] >> COUNT_0);
    (* Repeat shift operation for 2nd through 7th dwords *)
DEST[255:224] ← SignExtend(SRC1[255:224] >> COUNT_7);
DEST[MAXVL-1:256] ← 0;
```

VPSRAVD (EVEX encoded version)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j ← 0 TO KL-1
    i ← j * 32
    IF k1[j] OR *no writemask* THEN
            IF (EVEX.b = 1) AND (SRC2 *is memory*)
                THEN
                    COUNT ← SRC2[4:0]
                    IF COUNT < 32
                        THEN    DEST[i+31:i] ← SignExtend(SRC1[i+31:i] >> COUNT)
                        ELSE
                            FOR k← 0 TO 31
                                DEST[i+k] ← SRC1[i+31]
                            ENDFOR;
                    FI
                ELSE
                    COUNT ← SRC2[i+4:i]
                    IF COUNT < 32
                        THEN    DEST[i+31:i] ← SignExtend(SRC1[i+31:i] >> COUNT)
                        ELSE
                            FOR k← 0 TO 31
                                DEST[i+k] ← SRC1[i+31]
                            ENDFOR;
                    FI
            FI;
    ELSE
        IF *merging-masking*                    ; merging-masking
            THEN *DEST[31:0] remains unchanged*
            ELSE                                ; zeroing-masking
                DEST[31:0] ← 0
            FI
    FI;
ENDFOR;
DEST[MAXVL-1:VL] ← 0;
```

VPSRAVQ (EVEX encoded version)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j ← 0 TO KL-1
    i ← j * 64
    IF k1[j] OR *no writemask* THEN
            IF (EVEX.b = 1) AND (SRC2 *is memory*)
                THEN
                    COUNT ← SRC2[5:0]
                    IF COUNT < 64
                        THEN    DEST[i+63:i] ← SignExtend(SRC1[i+63:i] >> COUNT)
                        ELSE
                            FOR k← 0 TO 63
                                DEST[i+k] ← SRC1[i+63]
                            ENDFOR;
                    FI
                ELSE
                    COUNT ← SRC2[i+5:i]
                    IF COUNT < 64
                        THEN    DEST[i+63:i] ← SignExtend(SRC1[i+63:i] >> COUNT)
                        ELSE
                            FOR k← 0 TO 63
                                DEST[i+k] ← SRC1[i+63]
                            ENDFOR;
                    FI
            FI;
    ELSE
        IF *merging-masking*                    ; merging-masking
            THEN *DEST[63:0] remains unchanged*
            ELSE                                ; zeroing-masking
                DEST[63:0] ← 0
            FI
    FI;
ENDFOR;
DEST[MAXVL-1:VL] ← 0;
```

Intel C/C++ Compiler Intrinsic Equivalent

VPSRAVD __m512i _mm512_srav_epi32(__m512i a, __m512i cnt);
VPSRAVD __m512i _mm512_mask_srav_epi32(__m512i s, __mmask16 m, __m512i a, __m512i cnt);
VPSRAVD __m512i _mm512_maskz_srav_epi32(__mmask16 m, __m512i a, __m512i cnt);
VPSRAVD __m256i _mm256_srav_epi32(__m256i a, __m256i cnt);
VPSRAVD __m256i _mm256_mask_srav_epi32(__m256i s, __mmask8 m, __m256i a, __m256i cnt);
VPSRAVD __m256i _mm256_maskz_srav_epi32(__mmask8 m, __m256i a, __m256i cnt);
VPSRAVD __m128i _mm_srav_epi32(__m128i a, __m128i cnt);
VPSRAVD __m128i _mm_mask_srav_epi32(__m128i s, __mmask8 m, __m128i a, __m128i cnt);
VPSRAVD __m128i _mm_maskz_srav_epi32(__mmask8 m, __m128i a, __m128i cnt);
VPSRAVQ __m512i _mm512_srav_epi64(__m512i a, __m512i cnt);
VPSRAVQ __m512i _mm512_mask_srav_epi64(__m512i s, __mmask8 m, __m512i a, __m512i cnt);
VPSRAVQ __m512i _mm512_maskz_srav_epi64(__mmask8 m, __m512i a, __m512i cnt);
VPSRAVQ __m256i _mm256_srav_epi64(__m256i a, __m256i cnt);
VPSRAVQ __m256i _mm256_mask_srav_epi64(__m256i s, __mmask8 m, __m256i a, __m256i cnt);
VPSRAVQ __m256i _mm256_maskz_srav_epi64(__mmask8 m, __m256i a, __m256i cnt);
VPSRAVQ __m128i _mm_srav_epi64(__m128i a, __m128i cnt);
VPSRAVQ __m128i _mm_mask_srav_epi64(__m128i s, __mmask8 m, __m128i a, __m128i cnt);
VPSRAVQ __m128i _mm_maskz_srav_epi64(__mmask8 m, __m128i a, __m128i cnt);
VPSRAVW __m512i _mm512_srav_epi16(__m512i a, __m512i cnt);
VPSRAVW __m512i _mm512_mask_srav_epi16(__m512i s, __mmask32 m, __m512i a, __m512i cnt);
VPSRAVW __m512i _mm512_maskz_srav_epi16(__mmask32 m, __m512i a, __m512i cnt);
VPSRAVW __m256i _mm256_srav_epi16(__m256i a, __m256i cnt);
VPSRAVW __m256i _mm256_mask_srav_epi16(__m256i s, __mmask16 m, __m256i a, __m256i cnt);
VPSRAVW __m256i _mm256_maskz_srav_epi16(__mmask16 m, __m256i a, __m256i cnt);
VPSRAVW __m128i _mm_srav_epi16(__m128i a, __m128i cnt);
VPSRAVW __m128i _mm_mask_srav_epi16(__m128i s, __mmask8 m, __m128i a, __m128i cnt);
VPSRAVW __m128i _mm_maskz_srav_epi32(__mmask8 m, __m128i a, __m128i cnt);
VPSRAVD __m256i _mm256_srav_epi32 (__m256i m, __m256i count)

SIMD Floating-Point Exceptions

None

Other Exceptions

Non-EVEX-encoded instruction, see Exceptions Type 4.
EVEX-encoded instruction, see Exceptions Type E4.

VPSRLVW/VPSRLVD/VPSRLVQ—Variable Bit Shift Right Logical

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
VEX.NDS.128.66.0F38.W0 45 /r VPSRLVD xmm1, xmm2, xmm3/m128	A	V/V	AVX2	Shift doublewords in xmm2 right by amount specified in the corresponding element of xmm3/m128 while shifting in 0s.
VEX.NDS.128.66.0F38.W1 45 /r VPSRLVQ xmm1, xmm2, xmm3/m128	A	V/V	AVX2	Shift quadwords in xmm2 right by amount specified in the corresponding element of xmm3/m128 while shifting in 0s.
VEX.NDS.256.66.0F38.W0 45 /r VPSRLVD ymm1, ymm2, ymm3/m256	A	V/V	AVX2	Shift doublewords in ymm2 right by amount specified in the corresponding element of ymm3/m256 while shifting in 0s.
VEX.NDS.256.66.0F38.W1 45 /r VPSRLVQ ymm1, ymm2, ymm3/m256	A	V/V	AVX2	Shift quadwords in ymm2 right by amount specified in the corresponding element of ymm3/m256 while shifting in 0s.
EVEX.NDS.128.66.0F38.W1 10 /r VPSRLVW xmm1 {k1}{z}, xmm2, xmm3/m128	B	V/V	AVX512VL AVX512BW	Shift words in xmm2 right by amount specified in the corresponding element of xmm3/m128 while shifting in 0s using writemask k1.
EVEX.NDS.256.66.0F38.W1 10 /r VPSRLVW ymm1 {k1}{z}, ymm2, ymm3/m256	B	V/V	AVX512VL AVX512BW	Shift words in ymm2 right by amount specified in the corresponding element of ymm3/m256 while shifting in 0s using writemask k1.
EVEX.NDS.512.66.0F38.W1 10 /r VPSRLVW zmm1 {k1}{z}, zmm2, zmm3/m512	B	V/V	AVX512BW	Shift words in zmm2 right by amount specified in the corresponding element of zmm3/m512 while shifting in 0s using writemask k1.
EVEX.NDS.128.66.0F38.W0 45 /r VPSRLVD xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst	C	V/V	AVX512VL AVX512F	Shift doublewords in xmm2 right by amount specified in the corresponding element of xmm3/m128/m32bcst while shifting in 0s using writemask k1.
EVEX.NDS.256.66.0F38.W0 45 /r VPSRLVD ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst	C	V/V	AVX512VL AVX512F	Shift doublewords in ymm2 right by amount specified in the corresponding element of ymm3/m256/m32bcst while shifting in 0s using writemask k1.
EVEX.NDS.512.66.0F38.W0 45 /r VPSRLVD zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst	C	V/V	AVX512F	Shift doublewords in zmm2 right by amount specified in the corresponding element of zmm3/m512/m32bcst while shifting in 0s using writemask k1.
EVEX.NDS.128.66.0F38.W1 45 /r VPSRLVQ xmm1 {k1}{z}, xmm2, xmm3/m128/m64bcst	C	V/V	AVX512VL AVX512F	Shift quadwords in xmm2 right by amount specified in the corresponding element of xmm3/m128/m64bcst while shifting in 0s using writemask k1.
EVEX.NDS.256.66.0F38.W1 45 /r VPSRLVQ ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst	C	V/V	AVX512VL AVX512F	Shift quadwords in ymm2 right by amount specified in the corresponding element of ymm3/m256/m64bcst while shifting in 0s using writemask k1.
EVEX.NDS.512.66.0F38.W1 45 /r VPSRLVQ zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst	C	V/V	AVX512F	Shift quadwords in zmm2 right by amount specified in the corresponding element of zmm3/m512/m64bcst while shifting in 0s using writemask k1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	NA	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA
B	Full Vector Mem	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA
C	Full Vector	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA

Description

Shifts the bits in the individual data elements (words, doublewords or quadword) in the first source operand to the right by the count value of respective data elements in the second source operand. As the bits in the data elements are shifted right, the empty high-order bits are cleared (set to 0).

The count values are specified individually in each data element of the second source operand. If the unsigned integer value specified in the respective data element of the second source operand is greater than 15 (for word), 31 (for doublewords), or 63 (for a quadword), then the destination data element are written with 0.

VEX.128 encoded version: The destination and first source operands are XMM registers. The count operand can be either an XMM register or a 128-bit memory location. Bits (MAXVL-1:128) of the corresponding destination register are zeroed.

VEX.256 encoded version: The destination and first source operands are YMM registers. The count operand can be either an YMM register or a 256-bit memory. Bits (MAXVL-1:256) of the corresponding ZMM register are zeroed.

EVEX encoded VPSRLVD/Q: The destination and first source operands are ZMM/YMM/XMM registers. The count operand can be either a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512-bit vector broadcasted from a 32/64-bit memory location. The destination is conditionally updated with writemask k1.

EVEX encoded VPSRLVW: The destination and first source operands are ZMM/YMM/XMM registers. The count operand can be either a ZMM/YMM/XMM register, a 512/256/128-bit memory location. The destination is conditionally updated with writemask k1.

Operation

VPSRLVW (EVEX encoded version)

```
(KL, VL) = (8, 128), (16, 256), (32, 512)

FOR j ← 0 TO KL-1
    i ← j * 16
    IF k1[j] OR *no writemask*
        THEN DEST[i+15:i] ← ZeroExtend(SRC1[i+15:i] >> SRC2[i+15:i])
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+15:i] remains unchanged*
                ELSE                            ; zeroing-masking
                    DEST[i+15:i] ← 0
            FI
    FI;
ENDFOR;
DEST[MAXVL-1:VL] ← 0;
```

VPSRLVD (VEX.128 version)

```
COUNT_0 ←SRC2[31 : 0]
    (* Repeat Each COUNT_i for the 2nd through 4th dwords of SRC2*)
COUNT_3 ←SRC2[127 : 96];
IF COUNT_0 < 32 THEN
    DEST[31:0] ←ZeroExtend(SRC1[31:0] >> COUNT_0);
ELSE
    DEST[31:0] ←0;
    (* Repeat shift operation for 2nd through 4th dwords *)
IF COUNT_3 < 32 THEN
    DEST[127:96] ←ZeroExtend(SRC1[127:96] >> COUNT_3);
ELSE
    DEST[127:96] ←0;
DEST[MAXVL-1:128] ←0;
```

VPSRLVD (VEX.256 version)

```
COUNT_0 ←SRC2[31 : 0];
    (* Repeat Each COUNT_i for the 2nd through 7th dwords of SRC2*)
COUNT_7 ←SRC2[255 : 224];
IF COUNT_0 < 32 THEN
DEST[31:0] ←ZeroExtend(SRC1[31:0] >> COUNT_0);
ELSE
DEST[31:0] ←0;
    (* Repeat shift operation for 2nd through 7th dwords *)
IF COUNT_7 < 32 THEN
    DEST[255:224] ←ZeroExtend(SRC1[255:224] >> COUNT_7);
ELSE
    DEST[255:224] ←0;
DEST[MAXVL-1:256] ←0;
```

VPSRLVD (EVEX encoded version)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j ← 0 TO KL-1
    i ← j * 32
    IF k1[j] OR *no writemask* THEN
            IF (EVEX.b = 1) AND (SRC2 *is memory*)
                THEN DEST[i+31:i] ← ZeroExtend(SRC1[i+31:i] >> SRC2[31:0])
                ELSE DEST[i+31:i] ← ZeroExtend(SRC1[i+31:i] >> SRC2[i+31:i])
            FI;
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE                                ; zeroing-masking
                    DEST[i+31:i] ← 0
            FI
    FI;
ENDFOR;
DEST[MAXVL-1:VL] ← 0;
```

VPSRLVQ (VEX.128 version)

```
COUNT_0 ←SRC2[63 : 0];
COUNT_1 ←SRC2[127 : 64];
IF COUNT_0 < 64 THEN
    DEST[63:0] ←ZeroExtend(SRC1[63:0] >> COUNT_0);
ELSE
    DEST[63:0] ←0;
IF COUNT_1 < 64 THEN
    DEST[127:64] ←ZeroExtend(SRC1[127:64] >> COUNT_1);
ELSE
    DEST[127:64] ←0;
DEST[MAXVL-1:128] ←0;
```

VPSRLVQ (VEX.256 version)

```
COUNT_0 ←SRC2[63 : 0];
    (* Repeat Each COUNT_i for the 2nd through 4th dwords of SRC2*)
COUNT_3 ←SRC2[255 : 192];
IF COUNT_0 < 64 THEN
DEST[63:0] ←ZeroExtend(SRC1[63:0] >> COUNT_0);
ELSE
DEST[63:0] ←0;
    (* Repeat shift operation for 2nd through 4th dwords *)
IF COUNT_3 < 64 THEN
    DEST[255:192] ←ZeroExtend(SRC1[255:192] >> COUNT_3);
ELSE
    DEST[255:192] ←0;
DEST[MAXVL-1:256] ←0;
```

VPSRLVQ (EVEX encoded version)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j ← 0 TO KL-1
    i ← j * 64
    IF k1[j] OR *no writemask* THEN
            IF (EVEX.b = 1) AND (SRC2 *is memory*)
                THEN DEST[i+63:i] ← ZeroExtend(SRC1[i+63:i] >> SRC2[63:0])
                ELSE DEST[i+63:i] ← ZeroExtend(SRC1[i+63:i] >> SRC2[i+63:i])
            FI;
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE                                ; zeroing-masking
                    DEST[i+63:i] ← 0
            FI
    FI;
ENDFOR;
DEST[MAXVL-1:VL] ← 0;
```

Intel C/C++ Compiler Intrinsic Equivalent

VPSRLVW __m512i _mm512_srlv_epi16(__m512i a, __m512i cnt);
VPSRLVW __m512i _mm512_mask_srlv_epi16(__m512i s, __mmask32 k, __m512i a, __m512i cnt);
VPSRLVW __m512i _mm512_maskz_srlv_epi16(__mmask32 k, __m512i a, __m512i cnt);
VPSRLVW __m256i _mm256_mask_srlv_epi16(__m256i s, __mmask16 k, __m256i a, __m256i cnt);
VPSRLVW __m256i _mm256_maskz_srlv_epi16(__mmask16 k, __m256i a, __m256i cnt);
VPSRLVW __m128i _mm_mask_srlv_epi16(__m128i s, __mmask8 k, __m128i a, __m128i cnt);
VPSRLVW __m128i _mm_maskz_srlv_epi16(__mmask8 k, __m128i a, __m128i cnt);
VPSRLVW __m256i _mm256_srlv_epi32 (__m256i m, __m256i count)
VPSRLVD __m512i _mm512_srlv_epi32(__m512i a, __m512i cnt);
VPSRLVD __m512i _mm512_mask_srlv_epi32(__m512i s, __mmask16 k, __m512i a, __m512i cnt);
VPSRLVD __m512i _mm512_maskz_srlv_epi32(__mmask16 k, __m512i a, __m512i cnt);
VPSRLVD __m256i _mm256_mask_srlv_epi32(__m256i s, __mmask8 k, __m256i a, __m256i cnt);
VPSRLVD __m256i _mm256_maskz_srlv_epi32(__mmask8 k, __m256i a, __m256i cnt);
VPSRLVD __m128i _mm_mask_srlv_epi32(__m128i s, __mmask8 k, __m128i a, __m128i cnt);
VPSRLVD __m128i _mm_maskz_srlv_epi32(__mmask8 k, __m128i a, __m128i cnt);
VPSRLVQ __m512i _mm512_srlv_epi64(__m512i a, __m512i cnt);
VPSRLVQ __m512i _mm512_mask_srlv_epi64(__m512i s, __mmask8 k, __m512i a, __m512i cnt);
VPSRLVQ __m512i _mm512_maskz_srlv_epi64(__mmask8 k, __m512i a, __m512i cnt);
VPSRLVQ __m256i _mm256_mask_srlv_epi64(__m256i s, __mmask8 k, __m256i a, __m256i cnt);
VPSRLVQ __m256i _mm256_maskz_srlv_epi64(__mmask8 k, __m256i a, __m256i cnt);
VPSRLVQ __m128i _mm_mask_srlv_epi64(__m128i s, __mmask8 k, __m128i a, __m128i cnt);
VPSRLVQ __m128i _mm_maskz_srlv_epi64(__mmask8 k, __m128i a, __m128i cnt);
VPSRLVQ __m256i _mm256_srlv_epi64 (__m256i m, __m256i count)
VPSRLVD __m128i _mm_srlv_epi32(__m128i a, __m128i cnt);
VPSRLVQ __m128i _mm_srlv_epi64(__m128i a, __m128i cnt);

SIMD Floating-Point Exceptions

None

Other Exceptions

VEX-encoded instructions, see Exceptions Type 4.

EVEX-encoded VPSRLVD/Q, see Exceptions Type E4.

EVEX-encoded VPSRLVW, see Exceptions Type E4.nb.

VPTERNLOGD/VPTERNLOGQ—Bitwise Ternary Logic

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.DDS.128.66.0F3A.W0 25 /r ib VPTERNLOGD xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst, imm8	A	V/V	AVX512VL AVX512F	Bitwise ternary logic taking xmm1, xmm2 and xmm3/m128/m32bcst as source operands and writing the result to xmm1 under writemask k1 with dword granularity. The immediate value determines the specific binary function being implemented.
EVEX.DDS.256.66.0F3A.W0 25 /r ib VPTERNLOGD ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst, imm8	A	V/V	AVX512VL AVX512F	Bitwise ternary logic taking ymm1, ymm2 and ymm3/m256/m32bcst as source operands and writing the result to ymm1 under writemask k1 with dword granularity. The immediate value determines the specific binary function being implemented.
EVEX.DDS.512.66.0F3A.W0 25 /r ib VPTERNLOGD zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst, imm8	A	V/V	AVX512F	Bitwise ternary logic taking zmm1, zmm2 and zmm3/m512/m32bcst as source operands and writing the result to zmm1 under writemask k1 with dword granularity. The immediate value determines the specific binary function being implemented.
EVEX.DDS.128.66.0F3A.W1 25 /r ib VPTERNLOGQ xmm1 {k1}{z}, xmm2, xmm3/m128/m64bcst, imm8	A	V/V	AVX512VL AVX512F	Bitwise ternary logic taking xmm1, xmm2 and xmm3/m128/m64bcst as source operands and writing the result to xmm1 under writemask k1 with qword granularity. The immediate value determines the specific binary function being implemented.
EVEX.DDS.256.66.0F3A.W1 25 /r ib VPTERNLOGQ ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst, imm8	A	V/V	AVX512VL AVX512F	Bitwise ternary logic taking ymm1, ymm2 and ymm3/m256/m64bcst as source operands and writing the result to ymm1 under writemask k1 with qword granularity. The immediate value determines the specific binary function being implemented.
EVEX.DDS.512.66.0F3A.W1 25 /r ib VPTERNLOGQ zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst, imm8	A	V/V	AVX512F	Bitwise ternary logic taking zmm1, zmm2 and zmm3/m512/m64bcst as source operands and writing the result to zmm1 under writemask k1 with qword granularity. The immediate value determines the specific binary function being implemented.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Full Vector	ModRM:reg (r, w)	EVEX.vvvv (r)	ModRM:r/m (r)	Imm8

Description

VPTERNLOGD/Q takes three bit vectors of 512-bit length (in the first, second and third operand) as input data to form a set of 512 indices, each index is comprised of one bit from each input vector. The imm8 byte specifies a boolean logic table producing a binary value for each 3-bit index value. The final 512-bit boolean result is written to the destination operand (the first operand) using the writemask k1 with the granularity of doubleword element or quadword element into the destination.

The destination operand is a ZMM (EVEX.512)/YMM (EVEX.256)/XMM (EVEX.128) register. The first source operand is a ZMM/YMM/XMM register. The second source operand can be a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 32/64-bit memory location The destination operand is a ZMM register conditionally updated with writemask k1.

Table 5-11 shows two examples of Boolean functions specified by immediate values 0xE2 and 0xE4, with the look up result listed in the fourth column following the three columns containing all possible values of the 3-bit index.

Table 5-11. Examples of VPTERNLOGD/Q Imm8 Boolean Function and Input Index Values

VPTERNLOGD reg1, reg2, src3, 0xE2			Bit Result with Imm8=0xE2	VPTERNLOGD reg1, reg2, src3, 0xE4			Bit Result with Imm8=0xE4
Bit(reg1)	Bit(reg2)	Bit(src3)		Bit(reg1)	Bit(reg2)	Bit(src3)	
0	0	0	0	0	0	0	0
0	0	1	1	0	0	1	0
0	1	0	0	0	1	0	1
0	1	1	0	0	1	1	0
1	0	0	0	1	0	0	0
1	0	1	1	1	0	1	1
1	1	0	1	1	1	0	1
1	1	1	1	1	1	1	1

Specifying different values in imm8 will allow any arbitrary three-input Boolean functions to be implemented in software using VPTERNLOGD/Q. Table 5-1 and Table 5-2 provide a mapping of all 256 possible imm8 values to various Boolean expressions.

Operation

VPTERNLOGD (EVEX encoded versions)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j ← 0 TO KL-1
    i ← j * 32
    IF k1[j] OR *no writemask*
        THEN
            FOR k ← 0 TO 31
                IF (EVEX.b = 1) AND (SRC2 *is memory*)
                    THEN DEST[j][k] ← imm[(DEST[i+k] << 2) + (SRC1[ i+k ] << 1) + SRC2[ k ]]
                    ELSE DEST[j][k] ← imm[(DEST[i+k] << 2) + (SRC1[ i+k ] << 1) + SRC2[ i+k ]]
                FI;
                        ; table lookup of immediate bellow;
    ELSE
        IF *merging-masking*                    ; merging-masking
            THEN *DEST[31+i:i] remains unchanged*
            ELSE                                ; zeroing-masking
                DEST[31+i:i] ← 0
        FI;
    FI;
ENDFOR;
DEST[MAXVL-1:VL] ← 0
```

VPTERNLOGQ (EVEX encoded versions)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j ← 0 TO KL-1
    i ← j * 64
    IF k1[j] OR *no writemask*
        THEN
            FOR k ← 0 TO 63
                IF (EVEX.b = 1) AND (SRC2 *is memory*)
                    THEN DEST[j][k] ← imm[(DEST[i+k] << 2) + (SRC1[ i+k ] << 1) + SRC2[ k ]]
                    ELSE DEST[j][k] ← imm[(DEST[i+k] << 2) + (SRC1[ i+k ] << 1) + SRC2[ i+k ]]
                FI;                     ; table lookup of immediate bellow;
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[63+i:i] remains unchanged*
                ELSE                                ; zeroing-masking
                    DEST[63+i:i] ← 0
            FI;
    FI;
ENDFOR;
DEST[MAXVL-1:VL] ← 0
```

Intel C/C++ Compiler Intrinsic Equivalents

VPTERNLOGD __m512i _mm512_ternarylogic_epi32(__m512i a, __m512i b, int imm);
VPTERNLOGD __m512i _mm512_mask_ternarylogic_epi32(__m512i s, __mmask16 m, __m512i a, __m512i b, int imm);
VPTERNLOGD __m512i _mm512_maskz_ternarylogic_epi32(__mmask m, __m512i a, __m512i b, int imm);
VPTERNLOGD __m256i _mm256_ternarylogic_epi32(__m256i a, __m256i b, int imm);
VPTERNLOGD __m256i _mm256_mask_ternarylogic_epi32(__m256i s, __mmask8 m, __m256i a, __m256i b, int imm);
VPTERNLOGD __m256i _mm256_maskz_ternarylogic_epi32(__mmask8 m, __m256i a, __m256i b, int imm);
VPTERNLOGD __m128i _mm_ternarylogic_epi32(__m128i a, __m128i b, int imm);
VPTERNLOGD __m128i _mm_mask_ternarylogic_epi32(__m128i s, __mmask8 m, __m128i a, __m128i b, int imm);
VPTERNLOGD __m128i _mm_maskz_ternarylogic_epi32(__mmask8 m, __m128i a, __m128i b, int imm);
VPTERNLOGQ __m512i _mm512_ternarylogic_epi64(__m512i a, __m512i b, int imm);
VPTERNLOGQ __m512i _mm512_mask_ternarylogic_epi64(__m512i s, __mmask8 m, __m512i a, __m512i b, int imm);
VPTERNLOGQ __m512i _mm512_maskz_ternarylogic_epi64(__mmask8 m, __m512i a, __m512i b, int imm);
VPTERNLOGQ __m256i _mm256_ternarylogic_epi64(__m256i a, __m256i b, int imm);
VPTERNLOGQ __m256i _mm256_mask_ternarylogic_epi64(__m256i s, __mmask8 m, __m256i a, __m256i b, int imm);
VPTERNLOGQ __m256i _mm256_maskz_ternarylogic_epi64(__mmask8 m, __m256i a, __m256i b, int imm);
VPTERNLOGQ __m128i _mm_ternarylogic_epi64(__m128i a, __m128i b, int imm);
VPTERNLOGQ __m128i _mm_mask_ternarylogic_epi64(__m128i s, __mmask8 m, __m128i a, __m128i b, int imm);
VPTERNLOGQ __m128i _mm_maskz_ternarylogic_epi64(__mmask8 m, __m128i a, __m128i b, int imm);

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type E4.

VPTESTMB/VPTESTMW/VPTESTMD/VPTESTMQ—Logical AND and Set Mask

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.NDS.128.66.0F38.W0 26 /r VPTESTMB k2 {k1}, xmm2, xmm3/m128	A	V/V	AVX512VL AVX512BW	Bitwise AND of packed byte integers in xmm2 and xmm3/m128 and set mask k2 to reflect the zero/non-zero status of each element of the result, under writemask k1.
EVEX.NDS.256.66.0F38.W0 26 /r VPTESTMB k2 {k1}, ymm2, ymm3/m256	A	V/V	AVX512VL AVX512BW	Bitwise AND of packed byte integers in ymm2 and ymm3/m256 and set mask k2 to reflect the zero/non-zero status of each element of the result, under writemask k1.
EVEX.NDS.512.66.0F38.W0 26 /r VPTESTMB k2 {k1}, zmm2, zmm3/m512	A	V/V	AVX512BW	Bitwise AND of packed byte integers in zmm2 and zmm3/m512 and set mask k2 to reflect the zero/non-zero status of each element of the result, under writemask k1.
EVEX.NDS.128.66.0F38.W1 26 /r VPTESTMW k2 {k1}, xmm2, xmm3/m128	A	V/V	AVX512VL AVX512BW	Bitwise AND of packed word integers in xmm2 and xmm3/m128 and set mask k2 to reflect the zero/non-zero status of each element of the result, under writemask k1.
EVEX.NDS.256.66.0F38.W1 26 /r VPTESTMW k2 {k1}, ymm2, ymm3/m256	A	V/V	AVX512VL AVX512BW	Bitwise AND of packed word integers in ymm2 and ymm3/m256 and set mask k2 to reflect the zero/non-zero status of each element of the result, under writemask k1.
EVEX.NDS.512.66.0F38.W1 26 /r VPTESTMW k2 {k1}, zmm2, zmm3/m512	A	V/V	AVX512BW	Bitwise AND of packed word integers in zmm2 and zmm3/m512 and set mask k2 to reflect the zero/non-zero status of each element of the result, under writemask k1.
EVEX.NDS.128.66.0F38.W0 27 /r VPTESTMD k2 {k1}, xmm2, xmm3/m128/m32bcst	B	V/V	AVX512VL AVX512F	Bitwise AND of packed doubleword integers in xmm2 and xmm3/m128/m32bcst and set mask k2 to reflect the zero/non-zero status of each element of the result, under writemask k1.
EVEX.NDS.256.66.0F38.W0 27 /r VPTESTMD k2 {k1}, ymm2, ymm3/m256/m32bcst	B	V/V	AVX512VL AVX512F	Bitwise AND of packed doubleword integers in ymm2 and ymm3/m256/m32bcst and set mask k2 to reflect the zero/non-zero status of each element of the result, under writemask k1.
EVEX.NDS.512.66.0F38.W0 27 /r VPTESTMD k2 {k1}, zmm2, zmm3/m512/m32bcst	B	V/V	AVX512F	Bitwise AND of packed doubleword integers in zmm2 and zmm3/m512/m32bcst and set mask k2 to reflect the zero/non-zero status of each element of the result, under writemask k1.
EVEX.NDS.128.66.0F38.W1 27 /r VPTESTMQ k2 {k1}, xmm2, xmm3/m128/m64bcst	B	V/V	AVX512VL AVX512F	Bitwise AND of packed quadword integers in xmm2 and xmm3/m128/m64bcst and set mask k2 to reflect the zero/non-zero status of each element of the result, under writemask k1.
EVEX.NDS.256.66.0F38.W1 27 /r VPTESTMQ k2 {k1}, ymm2, ymm3/m256/m64bcst	B	V/V	AVX512VL AVX512F	Bitwise AND of packed quadword integers in ymm2 and ymm3/m256/m64bcst and set mask k2 to reflect the zero/non-zero status of each element of the result, under writemask k1.
EVEX.NDS.512.66.0F38.W1 27 /r VPTESTMQ k2 {k1}, zmm2, zmm3/m512/m64bcst	B	V/V	AVX512F	Bitwise AND of packed quadword integers in zmm2 and zmm3/m512/m64bcst and set mask k2 to reflect the zero/non-zero status of each element of the result, under writemask k1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Full Vector Mem	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA
B	Full Vector	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA

Description

Performs a bitwise logical AND operation on the first source operand (the second operand) and second source operand (the third operand) and stores the result in the destination operand (the first operand) under the writemask. Each bit of the result is set to 1 if the bitwise AND of the corresponding elements of the first and second src operands is non-zero; otherwise it is set to 0.

VPTESTMD/VPTESTMQ: The first source operand is a ZMM/YMM/XMM register. The second source operand can be a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 32/64-bit memory location. The destination operand is a mask register updated under the writemask.

VPTESTMB/VPTESTMW: The first source operand is a ZMM/YMM/XMM register. The second source operand can be a ZMM/YMM/XMM register or a 512/256/128-bit memory location. The destination operand is a mask register updated under the writemask.

Operation

VPTESTMB (EVEX encoded versions)

```
(KL, VL) = (16, 128), (32, 256), (64, 512)
FOR j ← 0 TO KL-1
    i ← j * 8
    IF k1[j] OR *no writemask*
        THEN    DEST[j] ← (SRC1[i+7:i] BITWISE AND SRC2[i+7:i] != 0)? 1 : 0;
        ELSE    DEST[j] = 0                     ; zeroing-masking only
    FI;
ENDFOR
DEST[MAX_KL-1:KL] ← 0
```

VPTESTMW (EVEX encoded versions)

```
(KL, VL) = (8, 128), (16, 256), (32, 512)
FOR j ← 0 TO KL-1
    i ← j * 16
    IF k1[j] OR *no writemask*
        THEN    DEST[j] ← (SRC1[i+15:i] BITWISE AND SRC2[i+15:i] != 0)? 1 : 0;
        ELSE    DEST[j] = 0                     ; zeroing-masking only
    FI;
ENDFOR
DEST[MAX_KL-1:KL] ← 0
```

VPTESTMD (EVEX encoded versions)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j ← 0 TO KL-1
    i ← j * 32
    IF k1[j] OR *no writemask*
        THEN
            IF (EVEX.b = 1) AND (SRC2 *is memory*)
                THEN DEST[j] ← (SRC1[i+31:i] BITWISE AND SRC2[31:0] != 0)? 1 : 0;
                ELSE DEST[j] ← (SRC1[i+31:i] BITWISE AND SRC2[i+31:i] != 0)? 1 : 0;
            FI;
        ELSE    DEST[j] ← 0                     ; zeroing-masking only
    FI;
ENDFOR
DEST[MAX_KL-1:KL] ← 0
```

VPTESTMQ (EVEX encoded versions)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j ← 0 TO KL-1
    i ← j * 64
    IF k1[j] OR *no writemask*
        THEN
            IF (EVEX.b = 1) AND (SRC2 *is memory*)
                THEN DEST[j] ← (SRC1[i+63:i] BITWISE AND SRC2[63:0] != 0)? 1 : 0;
                ELSE DEST[j] ← (SRC1[i+63:i] BITWISE AND SRC2[i+63:i] != 0)? 1 : 0;
            FI;
        ELSE    DEST[j] ← 0                     ; zeroing-masking only
    FI;
ENDFOR
DEST[MAX_KL-1:KL] ← 0
```

Intel C/C++ Compiler Intrinsic Equivalents

VPTESTMB __mmask64 _mm512_test_epi8_mask(__m512i a, __m512i b);
VPTESTMB __mmask64 _mm512_mask_test_epi8_mask(__mmask64, __m512i a, __m512i b);
VPTESTMW __mmask32 _mm512_test_epi16_mask(__m512i a, __m512i b);
VPTESTMW __mmask32 _mm512_mask_test_epi16_mask(__mmask32, __m512i a, __m512i b);
VPTESTMD __mmask16 _mm512_test_epi32_mask(__m512i a, __m512i b);
VPTESTMD __mmask16 _mm512_mask_test_epi32_mask(__mmask16, __m512i a, __m512i b);
VPTESTMQ __mmask8 _mm512_test_epi64_mask(__m512i a, __m512i b);
VPTESTMQ __mmask8 _mm512_mask_test_epi64_mask(__mmask8, __m512i a, __m512i b);

SIMD Floating-Point Exceptions

None

Other Exceptions

VPTESTMD/Q: See Exceptions Type E4.
VPTESTMB/W: See Exceptions Type E4.nb.

VPTESTNMB/W/D/Q—Logical NAND and Set

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID	Description
EVEX.NDS.128.F3.0F38.W0 26 /r VPTESTNMB k2 {k1}, xmm2, xmm3/m128	A	V/V	AVX512VL AVX512BW	Bitwise NAND of packed byte integers in xmm2 and xmm3/m128 and set mask k2 to reflect the zero/non-zero status of each element of the result, under writemask k1.
EVEX.NDS.256.F3.0F38.W0 26 /r VPTESTNMB k2 {k1}, ymm2, ymm3/m256	A	V/V	AVX512VL AVX512BW	Bitwise NAND of packed byte integers in ymm2 and ymm3/m256 and set mask k2 to reflect the zero/non-zero status of each element of the result, under writemask k1.
EVEX.NDS.512.F3.0F38.W0 26 /r VPTESTNMB k2 {k1}, zmm2, zmm3/m512	A	V/V	AVX512F AVX512BW	Bitwise NAND of packed byte integers in zmm2 and zmm3/m512 and set mask k2 to reflect the zero/non-zero status of each element of the result, under writemask k1.
EVEX.NDS.128.F3.0F38.W1 26 /r VPTESTNMW k2 {k1}, xmm2, xmm3/m128	A	V/V	AVX512VL AVX512BW	Bitwise NAND of packed word integers in xmm2 and xmm3/m128 and set mask k2 to reflect the zero/non-zero status of each element of the result, under writemask k1.
EVEX.NDS.256.F3.0F38.W1 26 /r VPTESTNMW k2 {k1}, ymm2, ymm3/m256	A	V/V	AVX512VL AVX512BW	Bitwise NAND of packed word integers in ymm2 and ymm3/m256 and set mask k2 to reflect the zero/non-zero status of each element of the result, under writemask k1.
EVEX.NDS.512.F3.0F38.W1 26 /r VPTESTNMW k2 {k1}, zmm2, zmm3/m512	A	V/V	AVX512F AVX512BW	Bitwise NAND of packed word integers in zmm2 and zmm3/m512 and set mask k2 to reflect the zero/non-zero status of each element of the result, under writemask k1.
EVEX.NDS.128.F3.0F38.W0 27 /r VPTESTNMD k2 {k1}, xmm2, xmm3/m128/m32bcst	B	V/V	AVX512VL AVX512F	Bitwise NAND of packed doubleword integers in xmm2 and xmm3/m128/m32bcst and set mask k2 to reflect the zero/non-zero status of each element of the result, under writemask k1.
EVEX.NDS.256.F3.0F38.W0 27 /r VPTESTNMD k2 {k1}, ymm2, ymm3/m256/m32bcst	B	V/V	AVX512VL AVX512F	Bitwise NAND of packed doubleword integers in ymm2 and ymm3/m256/m32bcst and set mask k2 to reflect the zero/non-zero status of each element of the result, under writemask k1.
EVEX.NDS.512.F3.0F38.W0 27 /r VPTESTNMD k2 {k1}, zmm2, zmm3/m512/m32bcst	B	V/V	AVX512F	Bitwise NAND of packed doubleword integers in zmm2 and zmm3/m512/m32bcst and set mask k2 to reflect the zero/non-zero status of each element of the result, under writemask k1.
EVEX.NDS.128.F3.0F38.W1 27 /r VPTESTNMQ k2 {k1}, xmm2, xmm3/m128/m64bcst	B	V/V	AVX512VL AVX512F	Bitwise NAND of packed quadword integers in xmm2 and xmm3/m128/m64bcst and set mask k2 to reflect the zero/non-zero status of each element of the result, under writemask k1.
EVEX.NDS.256.F3.0F38.W1 27 /r VPTESTNMQ k2 {k1}, ymm2, ymm3/m256/m64bcst	B	V/V	AVX512VL AVX512F	Bitwise NAND of packed quadword integers in ymm2 and ymm3/m256/m64bcst and set mask k2 to reflect the zero/non-zero status of each element of the result, under writemask k1.
EVEX.NDS.512.F3.0F38.W1 27 /r VPTESTNMQ k2 {k1}, zmm2, zmm3/m512/m64bcst	B	V/V	AVX512F	Bitwise NAND of packed quadword integers in zmm2 and zmm3/m512/m64bcst and set mask k2 to reflect the zero/non-zero status of each element of the result, under writemask k1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Full Vector Mem	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA
B	Full Vector	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA

Description

Performs a bitwise logical NAND operation on the byte/word/doubleword/quadword element of the first source operand (the second operand) with the corresponding element of the second source operand (the third operand) and stores the logical comparison result into each bit of the destination operand (the first operand) according to the writemask k1. Each bit of the result is set to 1 if the bitwise AND of the corresponding elements of the first and second src operands is zero; otherwise it is set to 0.

EVEX encoded VPTESTNMD/Q: The first source operand is a ZMM/YMM/XMM registers. The second source operand can be a ZMM/YMM/XMM register, a 512/256/128-bit memory location, or a 512/256/128-bit vector broadcasted from a 32/64-bit memory location. The destination is updated according to the writemask.

EVEX encoded VPTESTNMB/W: The first source operand is a ZMM/YMM/XMM registers. The second source operand can be a ZMM/YMM/XMM register, a 512/256/128-bit memory location. The destination is updated according to the writemask.

Operation

VPTESTNMB

```
(KL, VL) = (16, 128), (32, 256), (64, 512)
FOR j ← 0 TO KL-1
    i ← j*8
    IF MaskBit(j) OR *no writemask*
        THEN
            DEST[j] ← (SRC1[i+7:i] BITWISE AND SRC2[i+7:i] == 0)? 1 : 0
        ELSE DEST[j] ← 0; zeroing masking only
    FI
ENDFOR
DEST[MAX_KL-1:KL] ← 0
```

VPTESTNMW

```
(KL, VL) = (8, 128), (16, 256), (32, 512)
FOR j ← 0 TO KL-1
    i ← j*16
    IF MaskBit(j) OR *no writemask*
        THEN
            DEST[j] ← (SRC1[i+15:i] BITWISE AND SRC2[i+15:i] == 0)? 1 : 0
        ELSE DEST[j] ← 0; zeroing masking only
    FI
ENDFOR
DEST[MAX_KL-1:KL] ← 0
```

VPTESTNMD

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j ← 0 TO KL-1
    i ← j*32
    IF MaskBit(j) OR *no writemask*
        THEN
            IF (EVEX.b = 1) AND (SRC2 *is memory*)
                THEN DEST[i+31:i] ← (SRC1[i+31:i] BITWISE AND SRC2[31:0] == 0)? 1 : 0
                ELSE DEST[j] ← (SRC1[i+31:i] BITWISE AND SRC2[i+31:i] == 0)? 1 : 0
            FI
        ELSE DEST[j] ← 0; zeroing masking only
    FI
ENDFOR
DEST[MAX_KL-1:KL] ← 0
```

VPTESTNMQ

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j ← 0 TO KL-1
    i ← j*64
    IF MaskBit(j) OR *no writemask*
        THEN
            IF (EVEX.b = 1) AND (SRC2 *is memory*)
                THEN DEST[j] ← (SRC1[i+63:i] BITWISE AND SRC2[63:0] != 0)? 1 : 0;
                ELSE DEST[j] ← (SRC1[i+63:i] BITWISE AND SRC2[i+63:i] != 0)? 1 : 0;
            FI;
        ELSE DEST[j] ← 0; zeroing masking only
    FI
ENDFOR
DEST[MAX_KL-1:KL] ← 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VPTESTNMB __mmask64 _mm512_testn_epi8_mask(__m512i a, __m512i b);
VPTESTNMB __mmask64 _mm512_mask_testn_epi8_mask(__mmask64, __m512i a, __m512i b);
VPTESTNMB __mmask32 _mm256_testn_epi8_mask(__m256i a, __m256i b);
VPTESTNMB __mmask32 _mm256_mask_testn_epi8_mask(__mmask32, __m256i a, __m256i b);
VPTESTNMB __mmask16 _mm_testn_epi8_mask(__m128i a, __m128i b);
VPTESTNMB __mmask16 _mm_mask_testn_epi8_mask(__mmask16, __m128i a, __m128i b);
VPTESTNMW __mmask32 _mm512_testn_epi16_mask(__m512i a, __m512i b);
VPTESTNMW __mmask32 _mm512_mask_testn_epi16_mask(__mmask32, __m512i a, __m512i b);
VPTESTNMW __mmask16 _mm256_testn_epi16_mask(__m256i a, __m256i b);
VPTESTNMW __mmask16 _mm256_mask_testn_epi16_mask(__mmask16, __m256i a, __m256i b);
VPTESTNMW __mmask8 _mm_testn_epi16_mask(__m128i a, __m128i b);
VPTESTNMW __mmask8 _mm_mask_testn_epi16_mask(__mmask8, __m128i a, __m128i b);
VPTESTNMD __mmask16 _mm512_testn_epi32_mask(__m512i a, __m512i b);
VPTESTNMD __mmask16 _mm512_mask_testn_epi32_mask(__mmask16, __m512i a, __m512i b);
VPTESTNMD __mmask8 _mm256_testn_epi32_mask(__m256i a, __m256i b);
VPTESTNMD __mmask8 _mm256_mask_testn_epi32_mask(__mmask8, __m256i a, __m256i b);
VPTESTNMD __mmask8 _mm_testn_epi32_mask(__m128i a, __m128i b);
VPTESTNMD __mmask8 _mm_mask_testn_epi32_mask(__mmask8, __m128i a, __m128i b);
VPTESTNMQ __mmask8 _mm512_testn_epi64_mask(__m512i a, __m512i b);
VPTESTNMQ __mmask8 _mm512_mask_testn_epi64_mask(__mmask8, __m512i a, __m512i b);
VPTESTNMQ __mmask8 _mm256_testn_epi64_mask(__m256i a, __m256i b);
VPTESTNMQ __mmask8 _mm256_mask_testn_epi64_mask(__mmask8, __m256i a, __m256i b);
VPTESTNMQ __mmask8 _mm_testn_epi64_mask(__m128i a, __m128i b);

VPTESTNMQ __mmask8 _mm_mask_testn_epi64_mask(__mmask8, __m128i a, __m128i b);

SIMD Floating-Point Exceptions

None

Other Exceptions

VPTESTNMD/VPTESTNMQ: See Exceptions Type E4.

VPTESTNMB/VPTESTNMW: See Exceptions Type E4.nb.

VRANGEPD—Range Restriction Calculation For Packed Pairs of Float64 Values

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.NDS.128.66.0F3A.W1 50 /r ib VRANGEPD xmm1 {k1}{z}, xmm2, xmm3/m128/m64bcst, imm8	A	V/V	AVX512VL AVX512DQ	Calculate two RANGE operation output value from 2 pairs of double-precision floating-point values in xmm2 and xmm3/m128/m32bcst, store the results to xmm1 under the writemask k1. Imm8 specifies the comparison and sign of the range operation.
EVEX.NDS.256.66.0F3A.W1 50 /r ib VRANGEPD ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst, imm8	A	V/V	AVX512VL AVX512DQ	Calculate four RANGE operation output value from 4pairs of double-precision floating-point values in ymm2 and ymm3/m256/m32bcst, store the results to ymm1 under the writemask k1. Imm8 specifies the comparison and sign of the range operation.
EVEX.NDS.512.66.0F3A.W1 50 /r ib VRANGEPD zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst{sae}, imm8	A	V/V	AVX512DQ	Calculate eight RANGE operation output value from 8 pairs of double-precision floating-point values in zmm2 and zmm3/m512/m32bcst, store the results to zmm1 under the writemask k1. Imm8 specifies the comparison and sign of the range operation.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Full Vector	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	Imm8

Description

This instruction calculates 2/4/8 range operation outputs from two sets of packed input double-precision FP values in the first source operand (the second operand) and the second source operand (the third operand). The range outputs are written to the destination operand (the first operand) under the writemask k1.

Bits7:4 of imm8 byte must be zero. The range operation output is performed in two parts, each configured by a two-bit control field within imm8[3:0]:

- Imm8[1:0] specifies the initial comparison operation to be one of max, min, max absolute value or min absolute value of the input value pair. Each comparison of two input values produces an intermediate result that combines with the sign selection control (Imm8[3:2]) to determine the final range operation output.
- Imm8[3:2] specifies the sign of the range operation output to be one of the following: from the first input value, from the comparison result, set or clear.

The encodings of Imm8[1:0] and Imm8[3:2] are shown in Figure 5-27.

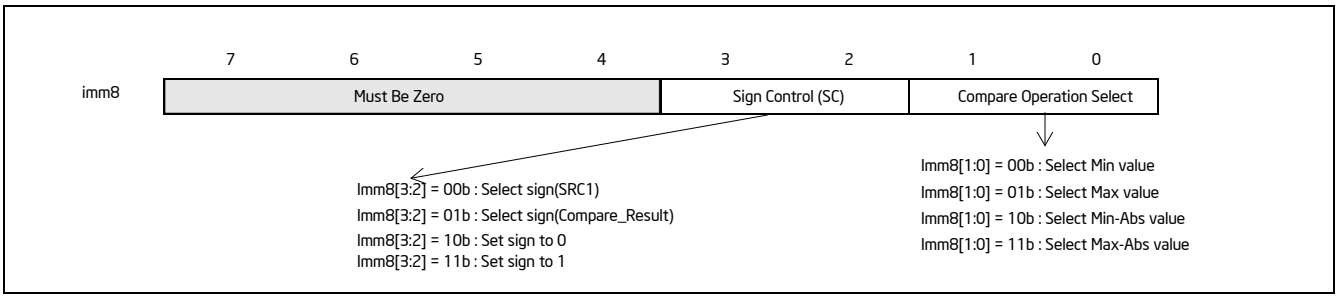


Figure 5-27. Imm8 Controls for VRANGEPD/SD/PS/SS

When one or more of the input value is a NAN, the comparison operation may signal invalid exception (IE). Details with one of more input value is NAN is listed in Table 5-12. If the comparison raises an IE, the sign select control (Imm8[3:2] has no effect to the range operation output, this is indicated also in Table 5-12.

When both input values are zeros of opposite signs, the comparison operation of MIN/MAX in the range compare operation is slightly different from the conceptually similar FP MIN/MAX operation that are found in the instructions VMAXPD/VMINPD. The details of MIN/MAX/MIN_ABS/MAX_ABS operation for VRANGEPD/PS/SD/SS for magnitude-0, opposite-signed input cases are listed in Table 5-13.

Additionally, non-zero, equal-magnitude with opposite-sign input values perform MIN_ABS or MAX_ABS comparison operation with result listed in Table 5-14.

Table 5-12. Signaling of Comparison Operation of One or More NaN Input Values and Effect of Imm8[3:2]

Src1	Src2	Result	IE Signaling Due to Comparison	Imm8[3:2] Effect to Range Output
sNaN1	sNaN2	Quiet(sNaN1)	Yes	Ignored
sNaN1	qNaN2	Quiet(sNaN1)	Yes	Ignored
sNaN1	Norm2	Quiet(sNaN1)	Yes	Ignored
qNaN1	sNaN2	Quiet(sNaN2)	Yes	Ignored
qNaN1	qNaN2	qNaN1	No	Applicable
qNaN1	Norm2	Norm2	No	Applicable
Norm1	sNaN2	Quiet(sNaN2)	Yes	Ignored
Norm1	qNaN2	Norm1	No	Applicable

Table 5-13. Comparison Result for Opposite-Signed Zero Cases for MIN, MIN_ABS and MAX, MAX_ABS

MIN and MIN_ABS			MAX and MAX_ABS		
Src1	Src2	Result	Src1	Src2	Result
+0	-0	-0	+0	-0	+0
-0	+0	-0	-0	+0	+0

Table 5-14. Comparison Result of Equal-Magnitude Input Cases for MIN_ABS and MAX_ABS, (|a| = |b|, a>0, b<0)

MIN_ABS (|a| = |b|, a>0, b<0)			MAX_ABS (|a| = |b|, a>0, b<0)		
Src1	Src2	Result	Src1	Src2	Result
a	b	b	a	b	a
b	a	b	b	a	a

Operation

```
RangeDP(SRC1[63:0], SRC2[63:0], CmpOpCtl[1:0], SignSelCtl[1:0])
{
    // Check if SNAN and report IE, see also Table 5-12
    IF (SRC1 = SNAN) THEN RETURN (QNAN(SRC1), set IE);
    IF (SRC2 = SNAN) THEN RETURN (QNAN(SRC2), set IE);

    Src1.exp ← SRC1[62:52];
    Src1.fraction ← SRC1[51:0];
    IF ((Src1.exp = 0 ) and (Src1.fraction != 0)) THEN// Src1 is a denormal number
        IF DAZ THEN Src1.fraction ← 0;
        ELSE IF (SRC2 <> QNAN) Set DE; FI;
    FI;

    Src2.exp ← SRC2[62:52];
    Src2.fraction ← SRC2[51:0];
    IF ((Src2.exp = 0) and (Src2.fraction !=0 )) THEN// Src2 is a denormal number
        IF DAZ THEN Src2.fraction ← 0;
        ELSE IF (SRC1 <> QNAN) Set DE; FI;
    FI;

    IF    (SRC2 = QNAN) THEN{TMP[63:0] ← SRC1[63:0]}
    ELSE IF(SRC1 = QNAN) THEN{TMP[63:0] ← SRC2[63:0]}
    ELSE IF (Both SRC1, SRC2 are magnitude-0 and opposite-signed) TMP[63:0] ← from Table 5-13
    ELSE IF (Both SRC1, SRC2 are magnitude-equal and opposite-signed and CmpOpCtl[1:0] > 01) TMP[63:0] ← from Table 5-14
    ELSE
        Case(CmpOpCtl[1:0])
        00: TMP[63:0] ← (SRC1[63:0] ≤ SRC2[63:0]) ? SRC1[63:0] : SRC2[63:0];
        01: TMP[63:0] ← (SRC1[63:0] ≤ SRC2[63:0]) ? SRC2[63:0] : SRC1[63:0];
        10: TMP[63:0] ← (ABS(SRC1[63:0]) ≤ ABS(SRC2[63:0])) ? SRC1[63:0] : SRC2[63:0];
        11: TMP[63:0] ← (ABS(SRC1[63:0]) ≤ ABS(SRC2[63:0])) ? SRC2[63:0] : SRC1[63:0];
        ESAC;
    FI;

    Case(SignSelCtl[1:0])
    00: dest ← (SRC1[63] << 63) OR (TMP[62:0]);// Preserve Src1 sign bit
    01: dest ← TMP[63:0];// Preserve sign of compare result
    10: dest ← (0 << 63) OR (TMP[62:0]);// Zero out sign bit
    11: dest ← (1 << 63) OR (TMP[62:0]);// Set the sign bit
    ESAC;
    RETURN dest[63:0];
}

CmpOpCtl[1:0]= imm8[1:0];
SignSelCtl[1:0]=imm8[3:2];
```

VRANGEPD (EVEX encoded versions)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j ← 0 TO KL-1
    i ← j * 64
    IF k1[j] OR *no writemask* THEN
            IF (EVEX.b == 1) AND (SRC2 *is memory*)
                THEN DEST[i+63:i] ← RangeDP (SRC1[i+63:i], SRC2[63:0], CmpOpCtl[1:0], SignSelCtl[1:0]);
                ELSE DEST[i+63:i] ← RangeDP (SRC1[i+63:i], SRC2[i+63:i], CmpOpCtl[1:0], SignSelCtl[1:0]);
            FI;
    ELSE
        IF *merging-masking*                    ; merging-masking
            THEN *DEST[i+63:i] remains unchanged*
            ELSE                                ; zeroing-masking
                DEST[i+63:i] = 0
        FI;
    FI;
ENDFOR;
DEST[MAXVL-1:VL] ← 0
```

The following example describes a common usage of this instruction for checking that the input operand is bounded between ±1023.

VRANGEPD zmm_dst, zmm_src, zmm_1023, 02h;

Where:

zmm_dst is the destination operand.
zmm_src is the input operand to compare against ±1023 (this is SRC1).
zmm_1023 is the reference operand, contains the value of 1023 (and this is SRC2).
IMM=02(imm8[1:0]='10) selects the Min Absolute value operation with selection of SRC1.sign.

In case |zmm_src| < 1023 (i.e. SRC1 is smaller than 1023 in magnitude), then its value will be written into zmm_dst. Otherwise, the value stored in zmm_dst will get the value of 1023 (received on zmm_1023, which is SRC2).

However, the sign control (imm8[3:2]='00) instructs to select the sign of SRC1 received from zmm_src. So, even in the case of |zmm_src| ≥ 1023, the selected sign of SRC1 is kept.

Thus, if zmm_src < -1023, the result of VRANGEPD will be the minimal value of -1023 while if zmm_src > +1023, the result of VRANGE will be the maximal value of +1023.

Intel C/C++ Compiler Intrinsic Equivalent

VRANGEPD __m512d _mm512_range_pd (__m512d a, __m512d b, int imm);
VRANGEPD __m512d _mm512_range_round_pd (__m512d a, __m512d b, int imm, int sae);
VRANGEPD __m512d _mm512_mask_range_pd (__m512 ds, __mmask8 k, __m512d a, __m512d b, int imm);
VRANGEPD __m512d _mm512_mask_range_round_pd (__m512d s, __mmask8 k, __m512d a, __m512d b, int imm, int sae);
VRANGEPD __m512d _mm512_maskz_range_pd (__mmask8 k, __m512d a, __m512d b, int imm);
VRANGEPD __m512d _mm512_maskz_range_round_pd (__mmask8 k, __m512d a, __m512d b, int imm, int sae);
VRANGEPD __m256d _mm256_range_pd (__m256d a, __m256d b, int imm);
VRANGEPD __m256d _mm256_mask_range_pd (__m256d s, __mmask8 k, __m256d a, __m256d b, int imm);
VRANGEPD __m256d _mm256_maskz_range_pd (__mmask8 k, __m256d a, __m256d b, int imm);
VRANGEPD __m128d _mm_range_pd (__m128 a, __m128d b, int imm);
VRANGEPD __m128d _mm_mask_range_pd (__m128 s, __mmask8 k, __m128d a, __m128d b, int imm);
VRANGEPD __m128d _mm_maskz_range_pd (__mmask8 k, __m128d a, __m128d b, int imm);

SIMD Floating-Point Exceptions

Invalid, Denormal

Other Exceptions

See Exceptions Type E2.

VRANGEPS—Range Restriction Calculation For Packed Pairs of Float32 Values

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.NDS.128.66.0F3A.W0 50 /r ib VRANGEPS xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst, imm8	A	V/V	AVX512VL AVX512DQ	Calculate four RANGE operation output value from 4 pairs of single-precision floating-point values in xmm2 and xmm3/m128/m32bcst, store the results to xmm1 under the writemask k1. Imm8 specifies the comparison and sign of the range operation.
EVEX.NDS.256.66.0F3A.W0 50 /r ib VRANGEPS ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst, imm8	A	V/V	AVX512VL AVX512DQ	Calculate eight RANGE operation output value from 8 pairs of single-precision floating-point values in ymm2 and ymm3/m256/m32bcst, store the results to ymm1 under the writemask k1. Imm8 specifies the comparison and sign of the range operation.
EVEX.NDS.512.66.0F3A.W0 50 /r ib VRANGEPS zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst{sae}, imm8	A	V/V	AVX512DQ	Calculate 16 RANGE operation output value from 16 pairs of single-precision floating-point values in zmm2 and zmm3/m512/m32bcst, store the results to zmm1 under the writemask k1. Imm8 specifies the comparison and sign of the range operation.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Full Vector	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	Imm8

Description

This instruction calculates 4/8/16 range operation outputs from two sets of packed input single-precision FP values in the first source operand (the second operand) and the second source operand (the third operand). The range outputs are written to the destination operand (the first operand) under the writemask k1.

Bits7:4 of imm8 byte must be zero. The range operation output is performed in two parts, each configured by a two-bit control field within imm8[3:0]:

- Imm8[1:0] specifies the initial comparison operation to be one of max, min, max absolute value or min absolute value of the input value pair. Each comparison of two input values produces an intermediate result that combines with the sign selection control (Imm8[3:2]) to determine the final range operation output.
- Imm8[3:2] specifies the sign of the range operation output to be one of the following: from the first input value, from the comparison result, set or clear.

The encodings of Imm8[1:0] and Imm8[3:2] are shown in Figure 5-27.

When one or more of the input value is a NAN, the comparison operation may signal invalid exception (IE). Details with one of more input value is NAN is listed in Table 5-12. If the comparison raises an IE, the sign select control (Imm8[3:2]) has no effect to the range operation output, this is indicated also in Table 5-12.

When both input values are zeros of opposite signs, the comparison operation of MIN/MAX in the range compare operation is slightly different from the conceptually similar FP MIN/MAX operation that are found in the instructions VMAXPD/VMINPD. The details of MIN/MAX/MIN_ABS/MAX_ABS operation for VRANGEPD/PS/SD/SS for magnitude-0, opposite-signed input cases are listed in Table 5-13.

Additionally, non-zero, equal-magnitude with opposite-sign input values perform MIN_ABS or MAX_ABS comparison operation with result listed in Table 5-14.

Operation

```
RangeSP(SRC1[31:0], SRC2[31:0], CmpOpCtl[1:0], SignSelCtl[1:0])
{
    // Check if SNAN and report IE, see also Table 5-12
    IF (SRC1=SNAN) THEN RETURN (QNAN(SRC1), set IE);
    IF (SRC2=SNAN) THEN RETURN (QNAN(SRC2), set IE);

    Src1.exp ← SRC1[30:23];
    Src1.fraction ← SRC1[22:0];
    IF ((Src1.exp = 0 ) and (Src1.fraction != 0 )) THEN// Src1 is a denormal number
        IF DAZ THEN Src1.fraction ← 0;
        ELSE IF (SRC2 <> QNAN) Set DE; FI;
    FI;
    Src2.exp ← SRC2[30:23];
    Src2.fraction ← SRC2[22:0];
    IF ((Src2.exp = 0 ) and (Src2.fraction != 0 )) THEN// Src2 is a denormal number
        IF DAZ THEN Src2.fraction ← 0;
        ELSE IF (SRC1 <> QNAN) Set DE; FI;
    FI;

    IF    (SRC2 = QNAN) THEN{TMP[31:0] ← SRC1[31:0]}
    ELSE IF(SRC1 = QNAN) THEN{TMP[31:0] ← SRC2[31:0]}
    ELSE IF (Both SRC1, SRC2 are magnitude-0 and opposite-signed) TMP[31:0] ← from Table 5-13
    ELSE IF (Both SRC1, SRC2 are magnitude-equal and opposite-signed and CmpOpCtl[1:0] > 01) TMP[31:0] ← from Table 5-14
    ELSE
        Case(CmpOpCtl[1:0])
        00: TMP[31:0] ← (SRC1[31:0] ≤ SRC2[31:0]) ? SRC1[31:0] : SRC2[31:0];
        01: TMP[31:0] ← (SRC1[31:0] ≤ SRC2[31:0]) ? SRC2[31:0] : SRC1[31:0];
        10: TMP[31:0] ← (ABS(SRC1[31:0]) ≤ ABS(SRC2[31:0])) ? SRC1[31:0] : SRC2[31:0];
        11: TMP[31:0] ← (ABS(SRC1[31:0]) ≤ ABS(SRC2[31:0])) ? SRC2[31:0] : SRC1[31:0];
        ESAC;
    FI;
    Case(SignSelCtl[1:0])
    00: dest ← (SRC1[31] << 31) OR (TMP[30:0]);// Preserve Src1 sign bit
    01: dest ← TMP[31:0];// Preserve sign of compare result
    10: dest ← (0 << 31) OR (TMP[30:0]);// Zero out sign bit
    11: dest ← (1 << 31) OR (TMP[30:0]);// Set the sign bit
    ESAC;
    RETURN dest[31:0];
}

CmpOpCtl[1:0]= imm8[1:0];
SignSelCtl[1:0]=imm8[3:2];
```

VRANGEPS

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j ← 0 TO KL-1
    i ← j * 32
    IF k1[j] OR *no writemask* THEN
            IF (EVEX.b == 1) AND (SRC2 *is memory*)
                THEN DEST[i+31:i] ← RangeSP (SRC1[i+31:i], SRC2[31:0], CmpOpCtl[1:0], SignSelCtl[1:0]);
                ELSE DEST[i+31:i] ← RangeSP (SRC1[i+31:i], SRC2[i+31:i], CmpOpCtl[1:0], SignSelCtl[1:0]);
            FI;
    ELSE
        IF *merging-masking*                    ; merging-masking
            THEN *DEST[i+31:i] remains unchanged*
            ELSE                                ; zeroing-masking
                DEST[i+31:i] = 0
        FI;
    FI;
ENDFOR;
DEST[MAXVL-1:VL] ← 0
```

The following example describes a common usage of this instruction for checking that the input operand is bounded between ±150.

VRANGEPS zmm_dst, zmm_src, zmm_150, 02h;

Where:
zmm_dst is the destination operand.
zmm_src is the input operand to compare against ±150.
zmm_150 is the reference operand, contains the value of 150.
IMM=02(imm8[1:0]=’10) selects the Min Absolute value operation with selection of src1.sign.

In case |zmm_src| < 150, then its value will be written into zmm_dst. Otherwise, the value stored in zmm_dst will get the value of 150 (received on zmm_150).
However, the sign control (imm8[3:2]=’00) instructs to select the sign of SRC1 received from zmm_src. So, even in the case of |zmm_src| ≥ 150, the selected sign of SRC1 is kept.
Thus, if zmm_src < -150, the result of VRANGEPS will be the minimal value of -150 while if zmm_src > +150, the result of VRANGE will be the maximal value of +150.

Intel C/C++ Compiler Intrinsic Equivalent

VRANGEPS __m512 _mm512_range_ps (__m512 a, __m512 b, int imm);
VRANGEPS __m512 _mm512_range_round_ps (__m512 a, __m512 b, int imm, int sae);
VRANGEPS __m512 _mm512_mask_range_ps (__m512 s, __mmask16 k, __m512 a, __m512 b, int imm);
VRANGEPS __m512 _mm512_mask_range_round_ps (__m512 s, __mmask16 k, __m512 a, __m512 b, int imm, int sae);
VRANGEPS __m512 _mm512_maskz_range_ps (__mmask16 k, __m512 a, __m512 b, int imm);
VRANGEPS __m512 _mm512_maskz_range_round_ps (__mmask16 k, __m512 a, __m512 b, int imm, int sae);
VRANGEPS __m256 _mm256_range_ps (__m256 a, __m256 b, int imm);
VRANGEPS __m256 _mm256_mask_range_ps (__m256 s, __mmask8 k, __m256 a, __m256 b, int imm);
VRANGEPS __m256 _mm256_maskz_range_ps (__mmask8 k, __m256 a, __m256 b, int imm);
VRANGEPS __m128 _mm_range_ps (__m128 a, __m128 b, int imm);
VRANGEPS __m128 _mm_mask_range_ps (__m128 s, __mmask8 k, __m128 a, __m128 b, int imm);
VRANGEPS __m128 _mm_maskz_range_ps (__mmask8 k, __m128 a, __m128 b, int imm);

SIMD Floating-Point Exceptions

Invalid, Denormal

Other Exceptions

See Exceptions Type E2.

VRANGESD—Range Restriction Calculation From a pair of Scalar Float64 Values

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.NDS.LIG.66.0F3A.W1 51 /r VRANGESD xmm1 {k1}{z}, xmm2, xmm3/m64{sae}, imm8	A	V/V	AVX512DQ	Calculate a RANGE operation output value from 2 double-precision floating-point values in xmm2 and xmm3/m64, store the output to xmm1 under writemask. Imm8 specifies the comparison and sign of the range operation.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Tuple1 Scalar	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	Imm8

Description

This instruction calculates a range operation output from two input double-precision FP values in the low qword element of the first source operand (the second operand) and second source operand (the third operand). The range output is written to the low qword element of the destination operand (the first operand) under the writemask k1.

Bits7:4 of imm8 byte must be zero. The range operation output is performed in two parts, each configured by a two-bit control field within imm8[3:0]:

- Imm8[1:0] specifies the initial comparison operation to be one of max, min, max absolute value or min absolute value of the input value pair. Each comparison of two input values produces an intermediate result that combines with the sign selection control (Imm8[3:2]) to determine the final range operation output.
- Imm8[3:2] specifies the sign of the range operation output to be one of the following: from the first input value, from the comparison result, set or clear.

The encodings of Imm8[1:0] and Imm8[3:2] are shown in Figure 5-27.

Bits 128:63 of the destination operand are copied from the respective element of the first source operand.

When one or more of the input value is a NAN, the comparison operation may signal invalid exception (IE). Details with one of more input value is NAN is listed in Table 5-12. If the comparison raises an IE, the sign select control (Imm8[3:2] has no effect to the range operation output, this is indicated also in Table 5-12.

When both input values are zeros of opposite signs, the comparison operation of MIN/MAX in the range compare operation is slightly different from the conceptually similar FP MIN/MAX operation that are found in the instructions VMAXPD/VMINPD. The details of MIN/MAX/MIN_ABS/MAX_ABS operation for VRANGEPD/PS/SD/SS for magnitude-0, opposite-signed input cases are listed in Table 5-13.

Additionally, non-zero, equal-magnitude with opposite-sign input values perform MIN_ABS or MAX_ABS comparison operation with result listed in Table 5-14.

Operation

```
RangeDP(SRC1[63:0], SRC2[63:0], CmpOpCtl[1:0], SignSelCtl[1:0])
{
    // Check if SNAN and report IE, see also Table 5-12
    IF (SRC1 = SNAN) THEN RETURN (QNAN(SRC1), set IE);
    IF (SRC2 = SNAN) THEN RETURN (QNAN(SRC2), set IE);

    Src1.exp ← SRC1[62:52];
    Src1.fraction ← SRC1[51:0];
    IF ((Src1.exp = 0 ) and (Src1.fraction != 0)) THEN// Src1 is a denormal number
        IF DAZ THEN Src1.fraction ← 0;
        ELSE IF (SRC2 <> QNAN) Set DE; FI;
    FI;

    Src2.exp ← SRC2[62:52];
    Src2.fraction ← SRC2[51:0];
    IF ((Src2.exp = 0) and (Src2.fraction !=0 )) THEN// Src2 is a denormal number
        IF DAZ THEN Src2.fraction ← 0;
        ELSE IF (SRC1 <> QNAN) Set DE; FI;
    FI;

    IF    (SRC2 = QNAN) THEN{TMP[63:0] ← SRC1[63:0]}
    ELSE IF(SRC1 = QNAN) THEN{TMP[63:0] ← SRC2[63:0]}
    ELSE IF (Both SRC1, SRC2 are magnitude-0 and opposite-signed) TMP[63:0] ← from Table 5-13
    ELSE IF (Both SRC1, SRC2 are magnitude-equal and opposite-signed and CmpOpCtl[1:0] > 01) TMP[63:0] ← from Table 5-14
    ELSE
        Case(CmpOpCtl[1:0])
        00: TMP[63:0] ← (SRC1[63:0] ≤ SRC2[63:0]) ? SRC1[63:0] : SRC2[63:0];
        01: TMP[63:0] ← (SRC1[63:0] ≤ SRC2[63:0]) ? SRC2[63:0] : SRC1[63:0];
        10: TMP[63:0] ← (ABS(SRC1[63:0]) ≤ ABS(SRC2[63:0])) ? SRC1[63:0] : SRC2[63:0];
        11: TMP[63:0] ← (ABS(SRC1[63:0]) ≤ ABS(SRC2[63:0])) ? SRC2[63:0] : SRC1[63:0];
        ESAC;
    FI;

    Case(SignSelCtl[1:0])
    00: dest ← (SRC1[63] << 63) OR (TMP[62:0]);// Preserve Src1 sign bit
    01: dest ← TMP[63:0];// Preserve sign of compare result
    10: dest ← (0 << 63) OR (TMP[62:0]);// Zero out sign bit
    11: dest ← (1 << 63) OR (TMP[62:0]);// Set the sign bit
    ESAC;
    RETURN dest[63:0];
}

CmpOpCtl[1:0]= imm8[1:0];
SignSelCtl[1:0]=imm8[3:2];
```

VRANGESD

```
IF k1[0] OR *no writemask*
        THEN DEST[63:0] ← RangeDP (SRC1[63:0], SRC2[63:0], CmpOpCtl[1:0], SignSelCtl[1:0]);
    ELSE
        IF *merging-masking*                    ; merging-masking
            THEN *DEST[63:0] remains unchanged*
            ELSE                                ; zeroing-masking
                DEST[63:0] = 0
        FI;
FI;
DEST[127:64] ← SRC1[127:64]
DEST[MAXVL-1:128] ← 0
```

The following example describes a common usage of this instruction for checking that the input operand is bounded between ±1023.

VRANGESD xmm_dst, xmm_src, xmm_1023, 02h;

Where:
xmm_dst is the destination operand.
xmm_src is the input operand to compare against ±1023.
xmm_1023 is the reference operand, contains the value of 1023.
IMM=02(imm8[1:0]=’10) selects the Min Absolute value operation with selection of src1.sign.

In case |xmm_src| < 1023, then its value will be written into xmm_dst. Otherwise, the value stored in xmm_dst will get the value of 1023 (received on xmm_1023).
However, the sign control (imm8[3:2]=’00) instructs to select the sign of SRC1 received from xmm_src. So, even in the case of |xmm_src| ≥ 1023, the selected sign of SRC1 is kept.
Thus, if xmm_src < -1023, the result of VRANGEPD will be the minimal value of -1023while if xmm_src > +1023, the result of VRANGE will be the maximal value of +1023.

Intel C/C++ Compiler Intrinsic Equivalent

VRANGESD __m128d _mm_range_sd (__m128d a, __m128d b, int imm);
VRANGESD __m128d _mm_range_round_sd (__m128d a, __m128d b, int imm, int sae);
VRANGESD __m128d _mm_mask_range_sd (__m128d s, __mmask8 k, __m128d a, __m128d b, int imm);
VRANGESD __m128d _mm_mask_range_round_sd (__m128d s, __mmask8 k, __m128d a, __m128d b, int imm, int sae);
VRANGESD __m128d _mm_maskz_range_sd (__mmask8 k, __m128d a, __m128d b, int imm);
VRANGESD __m128d _mm_maskz_range_round_sd (__mmask8 k, __m128d a, __m128d b, int imm, int sae);

SIMD Floating-Point Exceptions

Invalid, Denormal

Other Exceptions

See Exceptions Type E3.

VRANGESS—Range Restriction Calculation From a Pair of Scalar Float32 Values

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.NDS.LIG.66.0F3A.W0 51 /r VRANGESS xmm1 {k1}{z}, xmm2, xmm3/m32{sae}, imm8	A	V/V	AVX512DQ	Calculate a RANGE operation output value from 2 single-precision floating-point values in xmm2 and xmm3/m32, store the output to xmm1 under writemask. Imm8 specifies the comparison and sign of the range operation.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Tuple1 Scalar	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA

Description

This instruction calculates a range operation output from two input single-precision FP values in the low dword element of the first source operand (the second operand) and second source operand (the third operand). The range output is written to the low dword element of the destination operand (the first operand) under the writemask k1.

Bits7:4 of imm8 byte must be zero. The range operation output is performed in two parts, each configured by a two-bit control field within imm8[3:0]:

- Imm8[1:0] specifies the initial comparison operation to be one of max, min, max absolute value or min absolute value of the input value pair. Each comparison of two input values produces an intermediate result that combines with the sign selection control (Imm8[3:2]) to determine the final range operation output.
- Imm8[3:2] specifies the sign of the range operation output to be one of the following: from the first input value, from the comparison result, set or clear.

The encodings of Imm8[1:0] and Imm8[3:2] are shown in Figure 5-27.

Bits 128:31 of the destination operand are copied from the respective elements of the first source operand.

When one or more of the input value is a NAN, the comparison operation may signal invalid exception (IE). Details with one of more input value is NAN is listed in Table 5-12. If the comparison raises an IE, the sign select control (Imm8[3:2]) has no effect to the range operation output, this is indicated also in Table 5-12.

When both input values are zeros of opposite signs, the comparison operation of MIN/MAX in the range compare operation is slightly different from the conceptually similar FP MIN/MAX operation that are found in the instructions VMAXPD/VMINPD. The details of MIN/MAX/MIN_ABS/MAX_ABS operation for VRANGEPD/PS/SD/SS for magnitude-0, opposite-signed input cases are listed in Table 5-13.

Additionally, non-zero, equal-magnitude with opposite-sign input values perform MIN_ABS or MAX_ABS comparison operation with result listed in Table 5-14.

Operation

```
RangeSP(SRC1[31:0], SRC2[31:0], CmpOpCtl[1:0], SignSelCtl[1:0])
{
    // Check if SNAN and report IE, see also Table 5-12
    IF (SRC1=SNAN) THEN RETURN (QNAN(SRC1), set IE);
    IF (SRC2=SNAN) THEN RETURN (QNAN(SRC2), set IE);

    Src1.exp ← SRC1[30:23];
    Src1.fraction ← SRC1[22:0];
    IF ((Src1.exp = 0 ) and (Src1.fraction != 0 )) THEN// Src1 is a denormal number
        IF DAZ THEN Src1.fraction ← 0;
        ELSE IF (SRC2 <> QNAN) Set DE; FI;
    FI;
    Src2.exp ← SRC2[30:23];
    Src2.fraction ← SRC2[22:0];
    IF ((Src2.exp = 0 ) and (Src2.fraction != 0 )) THEN// Src2 is a denormal number
        IF DAZ THEN Src2.fraction ← 0;
        ELSE IF (SRC1 <> QNAN) Set DE; FI;
    FI;

    IF    (SRC2 = QNAN) THEN{TMP[31:0] ← SRC1[31:0]}
    ELSE IF(SRC1 = QNAN) THEN{TMP[31:0] ← SRC2[31:0]}
    ELSE IF (Both SRC1, SRC2 are magnitude-0 and opposite-signed) TMP[31:0] ← from Table 5-13
    ELSE IF (Both SRC1, SRC2 are magnitude-equal and opposite-signed and CmpOpCtl[1:0] > 01) TMP[31:0] ← from Table 5-14
    ELSE
        Case(CmpOpCtl[1:0])
        00: TMP[31:0] ← (SRC1[31:0] ≤ SRC2[31:0]) ? SRC1[31:0] : SRC2[31:0];
        01: TMP[31:0] ← (SRC1[31:0] ≤ SRC2[31:0]) ? SRC2[31:0] : SRC1[31:0];
        10: TMP[31:0] ← (ABS(SRC1[31:0]) ≤ ABS(SRC2[31:0])) ? SRC1[31:0] : SRC2[31:0];
        11: TMP[31:0] ← (ABS(SRC1[31:0]) ≤ ABS(SRC2[31:0])) ? SRC2[31:0] : SRC1[31:0];
        ESAC;
    FI;
    Case(SignSelCtl[1:0])
    00: dest ← (SRC1[31] << 31) OR (TMP[30:0]);// Preserve Src1 sign bit
    01: dest ← TMP[31:0];// Preserve sign of compare result
    10: dest ← (0 << 31) OR (TMP[30:0]);// Zero out sign bit
    11: dest ← (1 << 31) OR (TMP[30:0]);// Set the sign bit
    ESAC;
    RETURN dest[31:0];
}

CmpOpCtl[1:0]= imm8[1:0];
SignSelCtl[1:0]=imm8[3:2];
```

VRANGESS

```
IF k1[0] OR *no writemask*
        THEN DEST[31:0] ← RangeSP (SRC1[31:0], SRC2[31:0], CmpOpCtl[1:0], SignSelCtl[1:0]);
    ELSE
        IF *merging-masking*                    ; merging-masking
            THEN *DEST[31:0] remains unchanged*
            ELSE                                ; zeroing-masking
                DEST[31:0] = 0
        FI;
FI;
DEST[127:32] ← SRC1[127:32]
DEST[MAXVL-1:128] ← 0
```

The following example describes a common usage of this instruction for checking that the input operand is bounded between ±150.

VRANGESS zmm_dst, zmm_src, zmm_150, 02h;

Where:

xmm_dst is the destination operand.

xmm_src is the input operand to compare against ±150.

xmm_150 is the reference operand, contains the value of 150.

IMM=02(imm8[1:0]=’10) selects the Min Absolute value operation with selection of src1.sign.

In case |xmm_src| < 150, then its value will be written into zmm_dst. Otherwise, the value stored in xmm_dst will get the value of 150 (received on zmm_150).

However, the sign control (imm8[3:2]=’00) instructs to select the sign of SRC1 received from xmm_src. So, even in the case of |xmm_src| ≥ 150, the selected sign of SRC1 is kept.

Thus, if xmm_src < -150, the result of VRANGESS will be the minimal value of -150 while if xmm_src > +150, the result of VRANGE will be the maximal value of +150.

Intel C/C++ Compiler Intrinsic Equivalent

VRANGESS __m128 _mm_range_ss (__m128 a, __m128 b, int imm);
VRANGESS __m128 _mm_range_round_ss (__m128 a, __m128 b, int imm, int sae);
VRANGESS __m128 _mm_mask_range_ss (__m128 s, __mmask8 k, __m128 a, __m128 b, int imm);
VRANGESS __m128 _mm_mask_range_round_ss (__m128 s, __mmask8 k, __m128 a, __m128 b, int imm, int sae);
VRANGESS __m128 _mm_maskz_range_ss (__mmask8 k, __m128 a, __m128 b, int imm);
VRANGESS __m128 _mm_maskz_range_round_ss (__mmask8 k, __m128 a, __m128 b, int imm, int sae);

SIMD Floating-Point Exceptions

Invalid, Denormal

Other Exceptions

See Exceptions Type E3.

VRCP14PD—Compute Approximate Reciprocals of Packed Float64 Values

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.128.66.0F38.W1 4C /r VRCP14PD xmm1 {k1}{z}, xmm2/m128/m64bcst	A	V/V	AVX512VL AVX512F	Computes the approximate reciprocals of the packed double-precision floating-point values in xmm2/m128/m64bcst and stores the results in xmm1. Under writemask.
EVEX.256.66.0F38.W1 4C /r VRCP14PD ymm1 {k1}{z}, ymm2/m256/m64bcst	A	V/V	AVX512VL AVX512F	Computes the approximate reciprocals of the packed double-precision floating-point values in ymm2/m256/m64bcst and stores the results in ymm1. Under writemask.
EVEX.512.66.0F38.W1 4C /r VRCP14PD zmm1 {k1}{z}, zmm2/m512/m64bcst	A	V/V	AVX512F	Computes the approximate reciprocals of the packed double-precision floating-point values in zmm2/m512/m64bcst and stores the results in zmm1. Under writemask.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Full Vector	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

Description

This instruction performs a SIMD computation of the approximate reciprocals of eight/four/two packed double-precision floating-point values in the source operand (the second operand) and stores the packed double-precision floating-point results in the destination operand. The maximum relative error for this approximation is less than 2^{-14}.

The source operand can be a ZMM register, a 512-bit memory location, or a 512-bit vector broadcasted from a 64-bit memory location. The destination operand is a ZMM register conditionally updated according to the writemask.

The VRCP14PD instruction is not affected by the rounding control bits in the MXCSR register. When a source value is a 0.0, an ∞ with the sign of the source value is returned. A denormal source value will be treated as zero only in case of DAZ bit set in MXCSR. Otherwise it is treated correctly (i.e. not as a 0.0). Underflow results are flushed to zero only in case of FTZ bit set in MXCSR. Otherwise it will be treated correctly (i.e. correct underflow result is written) with the sign of the operand. When a source value is a SNaN or QNaN, the SNaN is converted to a QNaN or the source QNaN is returned.

EVEX.vvvv is reserved and must be 1111b otherwise instructions will #UD.

MXCSR exception flags are not affected by this instruction and floating-point exceptions are not reported.

Table 5-15. VRCP14PD/VRCP14SD Special Cases

Input value	Result value	Comments
$0 \le X \le 2^{-1024}$	INF	Very small denormal
$-2^{-1024} \le X \le -0$	-INF	Very small denormal
$X > 2^{1022}$	Underflow	Up to 18 bits of fractions are returned*
$X < -2^{1022}$	-Underflow	Up to 18 bits of fractions are returned*
$X = 2^{-n}$	2^{n}	
$X = -2^{-n}$	-2^{n}	

* in this case the mantissa is shifted right by one or two bits

A numerically exact implementation of VRCP14xx can be found at https://software.intel.com/en-us/articles/reference-implementations-for-IA-approximation-instructions-vrcp14-vrsqrt14-vrcp28-vrsqrt28-vexp2.

Operation

VRCP14PD ((EVEX encoded versions)
(KL, VL) = (2, 128), (4, 256), (8, 512)

```
FOR j ← 0 TO KL-1
    i ← j * 64
    IF k1[j] OR *no writemask* THEN
            IF (EVEX.b = 1) AND (SRC *is memory*)
                THEN DEST[i+63:i] ← APPROXIMATE(1.0/SRC[63:0]);
                ELSE DEST[i+63:i] ← APPROXIMATE(1.0/SRC[i+63:i]);
            FI;
    ELSE
        IF *merging-masking*                    ; merging-masking
            THEN *DEST[i+63:i] remains unchanged*
            ELSE                                ; zeroing-masking
                DEST[i+63:i] ← 0
        FI;
    FI;
ENDFOR;
DEST[MAXVL-1:VL] ← 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VRCP14PD __m512d _mm512_rcp14_pd(__m512d a);
VRCP14PD __m512d _mm512_mask_rcp14_pd(__m512d s, __mmask8 k, __m512d a);
VRCP14PD __m512d _mm512_maskz_rcp14_pd(__mmask8 k, __m512d a);

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type E4.

VRCP14SD—Compute Approximate Reciprocal of Scalar Float64 Value

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.NDS.LIG.66.0F38.W1 4D /r VRCP14SD xmm1 {k1}{z}, xmm2, xmm3/m64	A	V/V	AVX512F	Computes the approximate reciprocal of the scalar double-precision floating-point value in xmm3/m64 and stores the result in xmm1 using writemask k1. Also, upper double-precision floating-point value (bits[127:64]) from xmm2 is copied to xmm1[127:64].

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Tuple1 Scalar	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA

Description

This instruction performs a SIMD computation of the approximate reciprocal of the low double-precision floating-point value in the second source operand (the third operand) stores the result in the low quadword element of the destination operand (the first operand) according to the writemask k1. Bits (127:64) of the XMM register destination are copied from corresponding bits in the first source operand (the second operand). The maximum relative error for this approximation is less than 2^{-14}. The source operand can be an XMM register or a 64-bit memory location. The destination operand is an XMM register.

The VRCP14SD instruction is not affected by the rounding control bits in the MXCSR register. When a source value is a 0.0, an ∞ with the sign of the source value is returned. A denormal source value will be treated as zero only in case of DAZ bit set in MXCSR. Otherwise it is treated correctly (i.e. not as a 0.0). Underflow results are flushed to zero only in case of FTZ bit set in MXCSR. Otherwise it will be treated correctly (i.e. correct underflow result is written) with the sign of the operand. When a source value is a SNaN or QNaN, the SNaN is converted to a QNaN or the source QNaN is returned. See Table 5-15 for special-case input values.

MXCSR exception flags are not affected by this instruction and floating-point exceptions are not reported.

A numerically exact implementation of VRCP14xx can be found at https://software.intel.com/en-us/articles/reference-implementations-for-IA-approximation-instructions-vrcp14-vrsqrt14-vrcp28-vrsqrt28-vexp2.

Operation

VRCP14SD (EVEX version)

```
IF k1[0] OR *no writemask*
        THEN DEST[63:0] ← APPROXIMATE(1.0/SRC2[63:0]);
    ELSE
        IF *merging-masking*                    ; merging-masking
            THEN *DEST[63:0] remains unchanged*
            ELSE                                ; zeroing-masking
                DEST[63:0] ← 0
        FI;
FI;
DEST[127:64] ← SRC1[127:64]
DEST[MAXVL-1:128] ← 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VRCP14SD __m128d _mm_rcp14_sd(__m128d a, __m128d b);
VRCP14SD __m128d _mm_mask_rcp14_sd(__m128d s, __mmask8 k, __m128d a, __m128d b);
VRCP14SD __m128d _mm_maskz_rcp14_sd(__mmask8 k, __m128d a, __m128d b);

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type E5.

VRCP14PS—Compute Approximate Reciprocals of Packed Float32 Values

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.128.66.0F38.W0 4C /r VRCP14PS xmm1 {k1}{z}, xmm2/m128/m32bcst	A	V/V	AVX512VL AVX512F	Computes the approximate reciprocals of the packed single-precision floating-point values in xmm2/m128/m32bcst and stores the results in xmm1. Under writemask.
EVEX.256.66.0F38.W0 4C /r VRCP14PS ymm1 {k1}{z}, ymm2/m256/m32bcst	A	V/V	AVX512VL AVX512F	Computes the approximate reciprocals of the packed single-precision floating-point values in ymm2/m256/m32bcst and stores the results in ymm1. Under writemask.
EVEX.512.66.0F38.W0 4C /r VRCP14PS zmm1 {k1}{z}, zmm2/m512/m32bcst	A	V/V	AVX512F	Computes the approximate reciprocals of the packed single-precision floating-point values in zmm2/m512/m32bcst and stores the results in zmm1. Under writemask.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Full Vector	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

Description

This instruction performs a SIMD computation of the approximate reciprocals of the packed single-precision floating-point values in the source operand (the second operand) and stores the packed single-precision floating-point results in the destination operand (the first operand). The maximum relative error for this approximation is less than 2^{-14}.

The source operand can be a ZMM register, a 512-bit memory location or a 512-bit vector broadcasted from a 32-bit memory location. The destination operand is a ZMM register conditionally updated according to the writemask.

The VRCP14PS instruction is not affected by the rounding control bits in the MXCSR register. When a source value is a 0.0, an ∞ with the sign of the source value is returned. A denormal source value will be treated as zero only in case of DAZ bit set in MXCSR. Otherwise it is treated correctly (i.e. not as a 0.0). Underflow results are flushed to zero only in case of FTZ bit set in MXCSR. Otherwise it will be treated correctly (i.e. correct underflow result is written) with the sign of the operand. When a source value is a SNaN or QNaN, the SNaN is converted to a QNaN or the source QNaN is returned.

EVEX.vvvv is reserved and must be 1111b otherwise instructions will #UD.

MXCSR exception flags are not affected by this instruction and floating-point exceptions are not reported.

Table 5-16. VRCP14PS/VRCP14SS Special Cases

Input value	Result value	Comments
$0 \le X \le 2^{-128}$	INF	Very small denormal
$-2^{-128} \le X \le -0$	-INF	Very small denormal
$X > 2^{126}$	Underflow	Up to 18 bits of fractions are returned*
$X < -2^{126}$	-Underflow	Up to 18 bits of fractions are returned*
$X = 2^{-n}$	2^{n}	
$X = -2^{-n}$	-2^{n}	

* in this case the mantissa is shifted right by one or two bits

A numerically exact implementation of VRCP14xx can be found at https://software.intel.com/en-us/articles/reference-implementations-for-IA-approximation-instructions-vrcp14-vrsqrt14-vrcp28-vrsqrt28-vexp2.

Operation

VRCP14PS (EVEX encoded versions)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j ← 0 TO KL-1
    i ← j * 32
    IF k1[j] OR *no writemask* THEN
            IF (EVEX.b = 1) AND (SRC *is memory*)
                THEN DEST[i+31:i] ← APPROXIMATE(1.0/SRC[31:0]);
                ELSE DEST[i+31:i] ← APPROXIMATE(1.0/SRC[i+31:i]);
            FI;
    ELSE
        IF *merging-masking*                    ; merging-masking
            THEN *DEST[i+31:i] remains unchanged*
            ELSE                                ; zeroing-masking
                DEST[i+31:i] ← 0
        FI;
    FI;
ENDFOR;
DEST[MAXVL-1:VL] ← 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VRCP14PS __m512 _mm512_rcp14_ps(__m512 a);
VRCP14PS __m512 _mm512_mask_rcp14_ps(__m512 s, __mmask16 k, __m512 a);
VRCP14PS __m512 _mm512_maskz_rcp14_ps(__mmask16 k, __m512 a);
VRCP14PS __m256 _mm256_rcp14_ps(__m256 a);
VRCP14PS __m256 _mm512_mask_rcp14_ps(__m256 s, __mmask8 k, __m256 a);
VRCP14PS __m256 _mm512_maskz_rcp14_ps(__mmask8 k, __m256 a);
VRCP14PS __m128 _mm_rcp14_ps(__m128 a);
VRCP14PS __m128 _mm_mask_rcp14_ps(__m128 s, __mmask8 k, __m128 a);
VRCP14PS __m128 _mm_maskz_rcp14_ps(__mmask8 k, __m128 a);

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type E4.

VRCP14SS—Compute Approximate Reciprocal of Scalar Float32 Value

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.NDS.LIG.66.0F38.W0 4D /r VRCP14SS xmm1 {k1}{z}, xmm2, xmm3/m32	A	V/V	AVX512F	Computes the approximate reciprocal of the scalar single-precision floating-point value in xmm3/m32 and stores the results in xmm1 using writemask k1. Also, upper double-precision floating-point value (bits[127:32]) from xmm2 is copied to xmm1[127:32].

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Tuple1 Scalar	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA

Description

This instruction performs a SIMD computation of the approximate reciprocal of the low single-precision floating-point value in the second source operand (the third operand) and stores the result in the low quadword element of the destination operand (the first operand) according to the writemask k1. Bits (127:32) of the XMM register destination are copied from corresponding bits in the first source operand (the second operand). The maximum relative error for this approximation is less than 2^{-14}. The source operand can be an XMM register or a 32-bit memory location. The destination operand is an XMM register.

The VRCP14SS instruction is not affected by the rounding control bits in the MXCSR register. When a source value is a 0.0, an ∞ with the sign of the source value is returned. A denormal source value will be treated as zero only in case of DAZ bit set in MXCSR. Otherwise it is treated correctly (i.e. not as a 0.0). Underflow results are flushed to zero only in case of FTZ bit set in MXCSR. Otherwise it will be treated correctly (i.e. correct underflow result is written) with the sign of the operand. When a source value is a SNaN or QNaN, the SNaN is converted to a QNaN or the source QNaN is returned. See Table 5-16 for special-case input values.

MXCSR exception flags are not affected by this instruction and floating-point exceptions are not reported.

A numerically exact implementation of VRCP14xx can be found at https://software.intel.com/en-us/articles/reference-implementations-for-IA-approximation-instructions-vrcp14-vrsqrt14-vrcp28-vrsqrt28-vexp2.

Operation

VRCP14SS (EVEX version)

```
IF k1[0] OR *no writemask*
        THEN DEST[31:0] ← APPROXIMATE(1.0/SRC2[31:0]);
    ELSE
        IF *merging-masking*                    ; merging-masking
            THEN *DEST[31:0] remains unchanged*
            ELSE                                ; zeroing-masking
                DEST[31:0] ← 0
        FI;
FI;
DEST[127:32] ← SRC1[127:32]
DEST[MAXVL-1:128] ← 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VRCP14SS __m128 _mm_rcp14_ss(__m128 a, __m128 b);
VRCP14SS __m128 _mm_mask_rcp14_ss(__m128 s, __mmask8 k, __m128 a, __m128 b);
VRCP14SS __m128 _mm_maskz_rcp14_ss(__mmask8 k, __m128 a, __m128 b);

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type E5.

VRCP28PD—Approximation to the Reciprocal of Packed Double-Precision Floating-Point Values with Less Than 2^-28 Relative Error

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.512.66.0F38.W1 CA /r VRCP28PD zmm1 {k1}{z}, zmm2/m512/m64bcst {sae}	A	V/V	AVX512ER	Computes the approximate reciprocals (< 2^-28 relative error) of the packed double-precision floating-point values in zmm2/m512/m64bcst and stores the results in zmm1. Under writemask.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Full Vector	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

Description

Computes the reciprocal approximation of the float64 values in the source operand (the second operand) and store the results to the destination operand (the first operand). The approximate reciprocal is evaluated with less than 2^-28 of maximum relative error.

Denormal input values are treated as zeros and do not signal #DE, irrespective of MXCSR.DAZ. Denormal results are flushed to zeros and do not signal #UE, irrespective of MXCSR.FZ.

If any source element is NaN, the quietized NaN source value is returned for that element. If any source element is ±∞, ±0.0 is returned for that element. Also, if any source element is ±0.0, ±∞ is returned for that element.

The source operand is a ZMM register, a 512-bit memory location or a 512-bit vector broadcasted from a 64-bit memory location. The destination operand is a ZMM register, conditionally updated using writemask k1.

EVEX.vvvv is reserved and must be 1111b otherwise instructions will #UD.

A numerically exact implementation of VRCP28xx can be found at https://software.intel.com/en-us/articles/reference-implementations-for-IA-approximation-instructions-vrcp14-vrsqrt14-vrcp28-vrsqrt28-vexp2.

Operation

VRCP28PD (EVEX encoded versions)

```
(KL, VL) = (8, 512)

FOR j ← 0 TO KL-1
    i ← j * 64
    IF k1[j] OR *no writemask* THEN
            IF (EVEX.b = 1) AND (SRC *is memory*)
                THEN DEST[i+63:i] ← RCP_28_DP(1.0/SRC[63:0]);
                ELSE DEST[i+63:i] ← RCP_28_DP(1.0/SRC[i+63:i]);
            FI;
    ELSE
        IF *merging-masking*                ; merging-masking
            THEN *DEST[i+63:i] remains unchanged*
            ELSE                            ; zeroing-masking
                DEST[i+63:i] ← 0
        FI;
    FI;
ENDFOR;
```

Table 5-17. VRCP28PD Special Cases

Input value	Result value	Comments
NAN	QNAN(input)	If (SRC = SNaN) then #I
$0 \le X < 2^{-1022}$	INF	Positive input denormal or zero; #Z
$-2^{-1022} < X \le -0$	-INF	Negative input denormal or zero; #Z
$X > 2^{1022}$	+0.0f	
$X < -2^{1022}$	-0.0f	
$X = +\infty$	+0.0f	
$X = -\infty$	-0.0f	
$X = 2^{-n}$	2^{n}	Exact result (unless input/output is a denormal)
$X = -2^{-n}$	-2^{n}	Exact result (unless input/output is a denormal)

Intel C/C++ Compiler Intrinsic Equivalent

VRCP28PD __m512d _mm512_rcp28_round_pd (__m512d a, int sae);
VRCP28PD __m512d _mm512_mask_rcp28_round_pd(__m512d a, __mmask8 m, __m512d b, int sae);
VRCP28PD __m512d _mm512_maskz_rcp28_round_pd(__mmask8 m, __m512d b, int sae);

SIMD Floating-Point Exceptions

Invalid (if SNaN input), Divide-by-zero

Other Exceptions

See Exceptions Type E2.

VRCP28SD—Approximation to the Reciprocal of Scalar Double-Precision Floating-Point Value with Less Than 2^-28 Relative Error

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.NDS.LIG.66.0F38.W1 CB /r VRCP28SD xmm1 {k1}{z}, xmm2, xmm3/m64 {sae}	A	V/V	AVX512ER	Computes the approximate reciprocal (< 2^-28 relative error) of the scalar double-precision floating-point value in xmm3/m64 and stores the results in xmm1. Under writemask. Also, upper double-precision floating-point value (bits[127:64]) from xmm2 is copied to xmm1[127:64].

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Tuple1 Scalar	ModRM:reg (w)	EVEX.vvvv	ModRM:r/m (r)	NA

Description

Computes the reciprocal approximation of the low float64 value in the second source operand (the third operand) and store the result to the destination operand (the first operand). The approximate reciprocal is evaluated with less than 2^-28 of maximum relative error. The result is written into the low float64 element of the destination operand according to the writemask k1. Bits 127:64 of the destination is copied from the corresponding bits of the first source operand (the second operand).

A denormal input value is treated as zero and does not signal #DE, irrespective of MXCSR.DAZ. A denormal result is flushed to zero and does not signal #UE, irrespective of MXCSR.FZ.

If any source element is NaN, the quietized NaN source value is returned for that element. If any source element is ±∞, ±0.0 is returned for that element. Also, if any source element is ±0.0, ±∞ is returned for that element.

The first source operand is an XMM register. The second source operand is an XMM register or a 64-bit memory location. The destination operand is a XMM register, conditionally updated using writemask k1.

A numerically exact implementation of VRCP28xx can be found at https://software.intel.com/en-us/articles/reference-implementations-for-IA-approximation-instructions-vrcp14-vrsqrt14-vrcp28-vrsqrt28-vexp2.

Operation

VRCP28SD ((EVEX encoded versions)

```
IF k1[0] OR *no writemask* THEN
        DEST[63: 0] ← RCP_28_DP(1.0/SRC2[63: 0]);
ELSE
    IF *merging-masking*            ; merging-masking
        THEN *DEST[63: 0] remains unchanged*
        ELSE                        ; zeroing-masking
            DEST[63: 0] ← 0
    FI;
FI;
ENDFOR;
DEST[127:64] ← SRC1[127: 64]
DEST[MAXVL-1:128] ← 0
```

Table 5-18. VRCP28SD Special Cases

Input value	Result value	Comments
NAN	QNAN(input)	If (SRC = SNaN) then #I
$0 \le X < 2^{-1022}$	INF	Positive input denormal or zero; #Z
$-2^{-1022} < X \le -0$	-INF	Negative input denormal or zero; #Z
$X > 2^{1022}$	+0.0f	
$X < -2^{1022}$	-0.0f	
$X = +\infty$	+0.0f	
$X = -\infty$	-0.0f	
$X = 2^{-n}$	2^{n}	Exact result (unless input/output is a denormal)
$X = -2^{-n}$	-2^{n}	Exact result (unless input/output is a denormal)

Intel C/C++ Compiler Intrinsic Equivalent

VRCP28SD __m128d _mm_rcp28_round_sd (__m128d a, __m128d b, int sae);
VRCP28SD __m128d _mm_mask_rcp28_round_sd(__m128d s, __mmask8 m, __m128d a, __m128d b, int sae);
VRCP28SD __m128d _mm_maskz_rcp28_round_sd(__mmask8 m, __m128d a, __m128d b, int sae);

SIMD Floating-Point Exceptions

Invalid (if SNaN input), Divide-by-zero

Other Exceptions

See Exceptions Type E3.

VRCP28PS—Approximation to the Reciprocal of Packed Single-Precision Floating-Point Values with Less Than 2^-28 Relative Error

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.512.66.0F38.W0 CA /r VRCP28PS zmm1 {k1}{z}, zmm2/m512/m32bcst {sae}	A	V/V	AVX512ER	Computes the approximate reciprocals (< 2^-28 relative error) of the packed single-precision floating-point values in zmm2/m512/m32bcst and stores the results in zmm1. Under writemask.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Full Vector	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

Description

Computes the reciprocal approximation of the float32 values in the source operand (the second operand) and store the results to the destination operand (the first operand) using the writemask k1. The approximate reciprocal is evaluated with less than 2^-28 of maximum relative error prior to final rounding. The final results are rounded to < 2^-23 relative error before written to the destination.

Denormal input values are treated as zeros and do not signal #DE, irrespective of MXCSR.DAZ. Denormal results are flushed to zeros and do not signal #UE, irrespective of MXCSR.FZ.

If any source element is NaN, the quietized NaN source value is returned for that element. If any source element is ±∞, ±0.0 is returned for that element. Also, if any source element is ±0.0, ±∞ is returned for that element.

The source operand is a ZMM register, a 512-bit memory location, or a 512-bit vector broadcasted from a 32-bit memory location. The destination operand is a ZMM register, conditionally updated using writemask k1.

EVEX.vvvv is reserved and must be 1111b otherwise instructions will #UD.

A numerically exact implementation of VRCP28xx can be found at https://software.intel.com/en-us/articles/reference-implementations-for-IA-approximation-instructions-vrcp14-vrsqrt14-vrcp28-vrsqrt28-vexp2.

Operation

VRCP28PS (EVEX encoded versions)

```
(KL, VL) = (16, 512)

FOR j ← 0 TO KL-1
    i ← j * 32
    IF k1[j] OR *no writemask* THEN
            IF (EVEX.b = 1) AND (SRC *is memory*)
                THEN DEST[i+31:i] ← RCP_28_SP(1.0/SRC[31:0]);
                ELSE DEST[i+31:i] ← RCP_28_SP(1.0/SRC[i+31:i]);
            FI;
    ELSE
        IF *merging-masking*                    ; merging-masking
            THEN *DEST[i+31:i] remains unchanged*
            ELSE                                ; zeroing-masking
                DEST[i+31:i] ← 0
        FI;
    FI;
ENDFOR;
```

Table 5-19. VRCP28PS Special Cases

Input value	Result value	Comments
NAN	QNAN(input)	If (SRC = SNaN) then #I
$0 \leq X < 2^{-126}$	INF	Positive input denormal or zero; #Z
$-2^{-126} < X \leq -0$	-INF	Negative input denormal or zero; #Z
$X > 2^{126}$	+0.0f	
$X < -2^{126}$	-0.0f	
$X = +\infty$	+0.0f	
$X = -\infty$	-0.0f	
$X = 2^{-n}$	2^{n}	Exact result (unless input/output is a denormal)
$X = -2^{-n}$	-2^{n}	Exact result (unless input/output is a denormal)

Intel C/C++ Compiler Intrinsic Equivalent

VRCP28PS _mm512_rcp28_round_ps (__m512 a, int sae);
VRCP28PS __m512 _mm512_mask_rcp28_round_ps(__m512 s, __mmask16 m, __m512 a, int sae);
VRCP28PS __m512 _mm512_maskz_rcp28_round_ps(__mmask16 m, __m512 a, int sae);

SIMD Floating-Point Exceptions

Invalid (if SNaN input), Divide-by-zero

Other Exceptions

See Exceptions Type E2.

VRCP28SS—Approximation to the Reciprocal of Scalar Single-Precision Floating-Point Value with Less Than 2^-28 Relative Error

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.NDS.LIG.66.0F38.W0 CB /r VRCP28SS xmm1 {k1}{z}, xmm2, xmm3/m32 {sae}	A	V/V	AVX512ER	Computes the approximate reciprocal (< 2^-28 relative error) of the scalar single-precision floating-point value in xmm3/m32 and stores the results in xmm1. Under writemask. Also, upper 3 single-precision floating-point values (bits[127:32]) from xmm2 is copied to xmm1[127:32].

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Tuple1 Scalar	ModRM:reg (w)	EVEX.vvvv	ModRM:r/m (r)	NA

Description

Computes the reciprocal approximation of the low float32 value in the second source operand (the third operand) and store the result to the destination operand (the first operand). The approximate reciprocal is evaluated with less than 2^-28 of maximum relative error prior to final rounding. The final result is rounded to < 2^-23 relative error before written into the low float32 element of the destination according to writemask k1. Bits 127:32 of the destination is copied from the corresponding bits of the first source operand (the second operand).

A denormal input value is treated as zero and does not signal #DE, irrespective of MXCSR.DAZ. A denormal result is flushed to zero and does not signal #UE, irrespective of MXCSR.FZ.

If any source element is NaN, the quietized NaN source value is returned for that element. If any source element is ±∞, ±0.0 is returned for that element. Also, if any source element is ±0.0, ±∞ is returned for that element.

The first source operand is an XMM register. The second source operand is an XMM register or a 32-bit memory location. The destination operand is a XMM register, conditionally updated using writemask k1.

A numerically exact implementation of VRCP28xx can be found at https://software.intel.com/en-us/articles/reference-implementations-for-IA-approximation-instructions-vrcp14-vrsqrt14-vrcp28-vrsqrt28-vexp2.

Operation

VRCP28SS ((EVEX encoded versions)

```
IF k1[0] OR *no writemask* THEN
        DEST[31: 0] ← RCP_28_SP(1.0/SRC2[31: 0]);
ELSE
    IF *merging-masking*            ; merging-masking
        THEN *DEST[31: 0] remains unchanged*
        ELSE                        ; zeroing-masking
            DEST[31: 0] ← 0
    FI;
FI;
ENDFOR;
DEST[127:32] ← SRC1[127: 32]
DEST[MAXVL-1:128] ← 0
```

Table 5-20. VRCP28SS Special Cases

Input value	Result value	Comments
NAN	QNAN(input)	If (SRC = SNaN) then #I
$0 \le X < 2^{-126}$	INF	Positive input denormal or zero; #Z
$-2^{-126} < X \le -0$	-INF	Negative input denormal or zero; #Z
$X > 2^{126}$	+0.0f	
$X < -2^{126}$	-0.0f	
$X = +\infty$	+0.0f	
$X = -\infty$	-0.0f	
$X = 2^{-n}$	2^{n}	Exact result (unless input/output is a denormal)
$X = -2^{-n}$	-2^{n}	Exact result (unless input/output is a denormal)

Intel C/C++ Compiler Intrinsic Equivalent

VRCP28SS __m128 _mm_rcp28_round_ss (__m128 a, __m128 b, int sae);
VRCP28SS __m128 _mm_mask_rcp28_round_ss(__m128 s, __mmask8 m, __m128 a, __m128 b, int sae);
VRCP28SS __m128 _mm_maskz_rcp28_round_ss(__mmask8 m, __m128 a, __m128 b, int sae);

SIMD Floating-Point Exceptions

Invalid (if SNaN input), Divide-by-zero

Other Exceptions

See Exceptions Type E3.

VREDUCEPD—Perform Reduction Transformation on Packed Float64 Values

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.128.66.0F3A.W1 56 /r ib VREDUCEPD xmm1 {k1}{z}, xmm2/m128/m64bcst, imm8	A	V/V	AVX512VL AVX512DQ	Perform reduction transformation on packed double-precision floating point values in xmm2/m128/m32bcst by subtracting a number of fraction bits specified by the imm8 field. Stores the result in xmm1 register under writemask k1.
EVEX.256.66.0F3A.W1 56 /r ib VREDUCEPD ymm1 {k1}{z}, ymm2/m256/m64bcst, imm8	A	V/V	AVX512VL AVX512DQ	Perform reduction transformation on packed double-precision floating point values in ymm2/m256/m32bcst by subtracting a number of fraction bits specified by the imm8 field. Stores the result in ymm1 register under writemask k1.
EVEX.512.66.0F3A.W1 56 /r ib VREDUCEPD zmm1 {k1}{z}, zmm2/m512/m64bcst{sae}, imm8	A	V/V	AVX512DQ	Perform reduction transformation on double-precision floating point values in zmm2/m512/m32bcst by subtracting a number of fraction bits specified by the imm8 field. Stores the result in zmm1 register under writemask k1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Full Vector	ModRM:reg (w)	ModRM:r/m (r)	Imm8	NA

Description

Perform reduction transformation of the packed binary encoded double-precision FP values in the source operand (the second operand) and store the reduced results in binary FP format to the destination operand (the first operand) under the writemask k1.

The reduction transformation subtracts the integer part and the leading M fractional bits from the binary FP source value, where M is a unsigned integer specified by imm8[7:4], see Figure 5-28. Specifically, the reduction transformation can be expressed as:

dest = src – (ROUND(2^M*src))*2^{-M};

where "Round()" treats "src", "2^M", and their product as binary FP numbers with normalized significand and biased exponents.

The magnitude of the reduced result can be expressed by considering src= 2^p*man2,

where 'man2' is the normalized significand and 'p' is the unbiased exponent

Then if RC = RNE: 0<=|Reduced Result|<=2^{p-M-1}

Then if RC ≠ RNE: 0<=|Reduced Result|<2^{p-M}

This instruction might end up with a precision exception set. However, in case of SPE set (i.e. Suppress Precision Exception, which is imm8[3]=1), no precision exception is reported.

EVEX.vvvv is reserved and must be 1111b otherwise instructions will #UD.

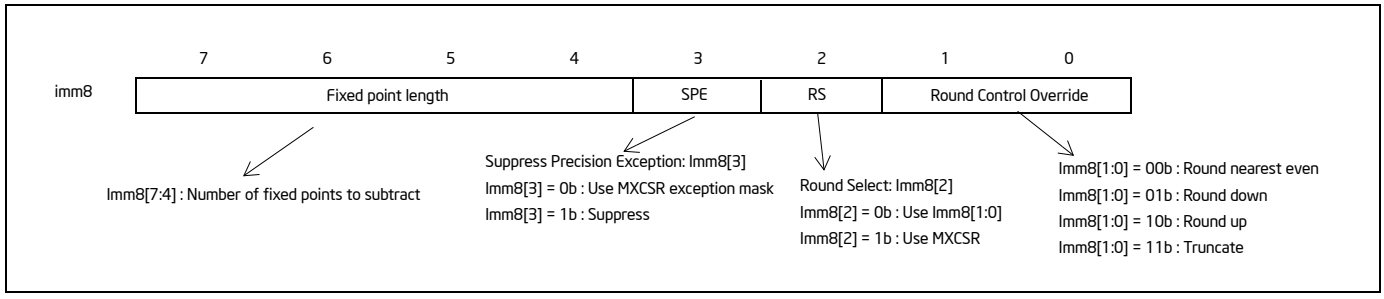


Figure 5-28. Imm8 Controls for VREDUCEPD/SD/PS/SS

Handling of special case of input values are listed in Table 5-21.

Table 5-21. VREDUCEPD/SD/PS/SS Special Cases

	Round Mode	Returned value
$\|Src1\| < 2^{-M-1}$	RNE	Src1
$\|Src1\| < 2^{-M}$	RPI, Src1 > 0	Round (Src1-2^{-M}) *
	RPI, Src1 ≤ 0	Src1
	RNI, Src1 ≥ 0	Src1
	RNI, Src1 < 0	Round (Src1+2^{-M}) *
Src1 = ±0, or Dest = ±0 (Src1!=INF)	NOT RNI	+0.0
	RNI	-0.0
Src1 = ±INF	any	+0.0
Src1= ±NAN	n/a	QNaN(Src1)

* Round control = (imm8.MS1)? MXCSR.RC: imm8.RC

Operation

```
ReduceArgumentDP(SRC[63:0], imm8[7:0])
{
    // Check for NaN
    IF (SRC [63:0] = NAN) THEN
        RETURN (Convert SRC[63:0] to QNaN); FI;
    M ← imm8[7:4]; // Number of fraction bits of the normalized significand to be subtracted
    RC ← imm8[1:0];// Round Control for ROUND() operation
    RC source ← imm[2];
    SPE ← 0;// Suppress Precision Exception
    TMP[63:0] ← 2^-M *{ROUND(2^M*SRC[63:0], SPE, RC_source, RC)}; // ROUND() treats SRC and 2^M as standard binary FP values
    TMP[63:0] ← SRC[63:0] – TMP[63:0]; // subtraction under the same RC,SPE controls
    RETURN TMP[63:0]; // binary encoded FP with biased exponent and normalized significand
}

VREDUCEPD
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j ← 0 TO KL-1
    i ← j * 64
    IF k1[j] OR *no writemask* THEN
            IF (EVEX.b == 1) AND (SRC *is memory*)
                THEN DEST[i+63:i] ← ReduceArgumentDP(SRC[63:0], imm8[7:0]);
                ELSE DEST[i+63:i] ← ReduceArgumentDP(SRC[i+63:i], imm8[7:0]);
            FI;
    ELSE
        IF *merging-masking*                ; merging-masking
            THEN *DEST[i+63:i] remains unchanged*
            ELSE                            ; zeroing-masking
                DEST[i+63:i] = 0
        FI;
    FI;
ENDFOR;
DEST[MAXVL-1:VL] ← 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VREDUCEPD __m512d _mm512_mask_reduce_pd(__m512d a, int imm, int sae)
VREDUCEPD __m512d _mm512_mask_reduce_pd(__m512d s, __mmask8 k, __m512d a, int imm, int sae)
VREDUCEPD __m512d _mm512_maskz_reduce_pd(__mmask8 k, __m512d a, int imm, int sae)
VREDUCEPD __m256d _mm256_mask_reduce_pd(__m256d a, int imm)
VREDUCEPD __m256d _mm256_mask_reduce_pd(__m256d s, __mmask8 k, __m256d a, int imm)
VREDUCEPD __m256d _mm256_maskz_reduce_pd(__mmask8 k, __m256d a, int imm)
VREDUCEPD __m128d _mm_mask_reduce_pd(__m128d a, int imm)
VREDUCEPD __m128d _mm_mask_reduce_pd(__m128d s, __mmask8 k, __m128d a, int imm)
VREDUCEPD __m128d _mm_maskz_reduce_pd(__mmask8 k, __m128d a, int imm)

SIMD Floating-Point Exceptions

Invalid, Precision

If SPE is enabled, precision exception is not reported (regardless of MXCSR exception mask).

Other Exceptions

See Exceptions Type E2, additionally

#UD	If EVEX.vvvv != 1111B.

VREDUCESD—Perform a Reduction Transformation on a Scalar Float64 Value

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.NDS.LIG.66.0F3A.W1 57 VREDUCESD xmm1 {k1}{z}, xmm2, xmm3/m64{sae}, imm8/r	A	V/V	AVX512DQ	Perform a reduction transformation on a scalar double-precision floating point value in xmm3/m64 by subtracting a number of fraction bits specified by the imm8 field. Also, upper double precision floating-point value (bits[127:64]) from xmm2 are copied to xmm1[127:64]. Stores the result in xmm1 register.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Tuple1 Scalar	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA

Description

Perform a reduction transformation of the binary encoded double-precision FP value in the low qword element of the second source operand (the third operand) and store the reduced result in binary FP format to the low qword element of the destination operand (the first operand) under the writemask k1. Bits 127:64 of the destination operand are copied from respective qword elements of the first source operand (the second operand).

The reduction transformation subtracts the integer part and the leading M fractional bits from the binary FP source value, where M is a unsigned integer specified by imm8[7:4], see Figure 5-28. Specifically, the reduction transformation can be expressed as:

dest = src – (ROUND(2^M*src))*2^{-M};

where "Round()" treats "src", "2^M", and their product as binary FP numbers with normalized significand and biased exponents.

The magnitude of the reduced result can be expressed by considering src= 2^p*man2,
where 'man2' is the normalized significand and 'p' is the unbiased exponent

Then if RC = RNE: 0<=|Reduced Result|<=2^{p-M-1}

Then if RC ≠ RNE: 0<=|Reduced Result|<2^{p-M}

This instruction might end up with a precision exception set. However, in case of SPE set (i.e. Suppress Precision Exception, which is imm8[3]=1), no precision exception is reported.

The operation is write masked.

Handling of special case of input values are listed in Table 5-21.

Operation

```
ReduceArgumentDP(SRC[63:0], imm8[7:0])
{
    // Check for NaN
    IF (SRC [63:0] = NAN) THEN
        RETURN (Convert SRC[63:0] to QNaN); FI;
    M ← imm8[7:4]; // Number of fraction bits of the normalized significand to be subtracted
    RC ← imm8[1:0];// Round Control for ROUND() operation
    RC source ← imm[2];
    SPE ← 0;// Suppress Precision Exception
    TMP[63:0] ← 2^-M *{ROUND(2^M*SRC[63:0], SPE, RC_source, RC)}; // ROUND() treats SRC and 2^M as standard binary FP values
    TMP[63:0] ← SRC[63:0] – TMP[63:0]; // subtraction under the same RC,SPE controls
    RETURN TMP[63:0]; // binary encoded FP with biased exponent and normalized significand
}
```

VREDUCESD

```
IF k1[0] or *no writemask*
    THEN     DEST[63:0] ← ReduceArgumentDP(SRC2[63:0], imm8[7:0])
    ELSE
        IF *merging-masking*                    ; merging-masking
            THEN *DEST[63:0] remains unchanged*
            ELSE                                ; zeroing-masking
                THEN DEST[63:0] = 0
        FI;
FI;
DEST[127:64] ← SRC1[127:64]
DEST[MAXVL-1:128] ← 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VREDUCESD __m128d _mm_mask_reduce_sd(__m128d a, __m128d b, int imm, int sae)
VREDUCESD __m128d _mm_mask_reduce_sd(__m128d s, __mmask16 k, __m128d a, __m128d b, int imm, int sae)
VREDUCESD __m128d _mm_maskz_reduce_sd(__mmask16 k, __m128d a, __m128d b, int imm, int sae)

SIMD Floating-Point Exceptions

Invalid, Precision

If SPE is enabled, precision exception is not reported (regardless of MXCSR exception mask).

Other Exceptions

See Exceptions Type E3.

VREDUCEPS—Perform Reduction Transformation on Packed Float32 Values

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.128.66.0F3A.W0 56 /r ib VREDUCEPS xmm1 {k1}{z}, xmm2/m128/m32bcst, imm8	A	V/V	AVX512VL AVX512DQ	Perform reduction transformation on packed single-precision floating point values in xmm2/m128/m32bcst by subtracting a number of fraction bits specified by the imm8 field. Stores the result in xmm1 register under writemask k1.
EVEX.256.66.0F3A.W0 56 /r ib VREDUCEPS ymm1 {k1}{z}, ymm2/m256/m32bcst, imm8	A	V/V	AVX512VL AVX512DQ	Perform reduction transformation on packed single-precision floating point values in ymm2/m256/m32bcst by subtracting a number of fraction bits specified by the imm8 field. Stores the result in ymm1 register under writemask k1.
EVEX.512.66.0F3A.W0 56 /r ib VREDUCEPS zmm1 {k1}{z}, zmm2/m512/m32bcst{sae}, imm8	A	V/V	AVX512DQ	Perform reduction transformation on packed single-precision floating point values in zmm2/m512/m32bcst by subtracting a number of fraction bits specified by the imm8 field. Stores the result in zmm1 register under writemask k1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Full Vector	ModRM:reg (w)	ModRM:r/m (r)	Imm8	NA

Description

Perform reduction transformation of the packed binary encoded single-precision FP values in the source operand (the second operand) and store the reduced results in binary FP format to the destination operand (the first operand) under the writemask k1.

The reduction transformation subtracts the integer part and the leading M fractional bits from the binary FP source value, where M is a unsigned integer specified by imm8[7:4], see Figure 5-28. Specifically, the reduction transformation can be expressed as:

dest = src – (ROUND(2^{M}*src))*2^{-M};

where "Round()" treats "src", "2^{M}", and their product as binary FP numbers with normalized significand and biased exponents.

The magnitude of the reduced result can be expressed by considering src= 2^{p}*man2,

where 'man2' is the normalized significand and 'p' is the unbiased exponent

Then if RC = RNE: 0<=|Reduced Result|<=2^{p-M-1}

Then if RC ≠ RNE: 0<=|Reduced Result|<2^{p-M}

This instruction might end up with a precision exception set. However, in case of SPE set (i.e. Suppress Precision Exception, which is imm8[3]=1), no precision exception is reported.

EVEX.vvvv is reserved and must be 1111b otherwise instructions will #UD.

Handling of special case of input values are listed in Table 5-21.

Operation

```
ReduceArgumentSP(SRC[31:0], imm8[7:0])
{
    // Check for NaN
    IF (SRC [31:0] = NAN) THEN
        RETURN (Convert SRC[31:0] to QNaN); FI
    M ← imm8[7:4]; // Number of fraction bits of the normalized significand to be subtracted
    RC ← imm8[1:0];// Round Control for ROUND() operation
    RC source ← imm[2];
    SPE ← 0;// Suppress Precision Exception
    TMP[31:0] ← 2^-M *{ROUND(2^M*SRC[31:0], SPE, RC_source, RC)}; // ROUND() treats SRC and 2^M as standard binary FP values
    TMP[31:0] ← SRC[31:0] – TMP[31:0]; // subtraction under the same RC,SPE controls
RETURN TMP[31:0]; // binary encoded FP with biased exponent and normalized significand
}

VREDUCEPS
(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j ← 0 TO KL-1
    i ← j * 32
    IF k1[j] OR *no writemask* THEN
            IF (EVEX.b == 1) AND (SRC *is memory*)
                THEN DEST[i+31:i] ← ReduceArgumentSP(SRC[31:0], imm8[7:0]);
                ELSE DEST[i+31:i] ← ReduceArgumentSP(SRC[i+31:i], imm8[7:0]);
            FI;
    ELSE
        IF *merging-masking*                ; merging-masking
            THEN *DEST[i+31:i] remains unchanged*
            ELSE                            ; zeroing-masking
                DEST[i+31:i] = 0
        FI;
    FI;
ENDFOR;
DEST[MAXVL-1:VL] ← 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VREDUCEPS __m512 _mm512_mask_reduce_ps(__m512 a, int imm, int sae)
VREDUCEPS __m512 _mm512_mask_reduce_ps(__m512 s, __mmask16 k, __m512 a, int imm, int sae)
VREDUCEPS __m512 _mm512_maskz_reduce_ps(__mmask16 k, __m512 a, int imm, int sae)
VREDUCEPS __m256 _mm256_mask_reduce_ps(__m256 a, int imm)
VREDUCEPS __m256 _mm256_mask_reduce_ps(__m256 s, __mmask8 k, __m256 a, int imm)
VREDUCEPS __m256 _mm256_maskz_reduce_ps(__mmask8 k, __m256 a, int imm)
VREDUCEPS __m128 _mm_mask_reduce_ps(__m128 a, int imm)
VREDUCEPS __m128 _mm_mask_reduce_ps(__m128 s, __mmask8 k, __m128 a, int imm)
VREDUCEPS __m128 _mm_maskz_reduce_ps(__mmask8 k, __m128 a, int imm)

SIMD Floating-Point Exceptions

Invalid, Precision

If SPE is enabled, precision exception is not reported (regardless of MXCSR exception mask).

Other Exceptions

See Exceptions Type E2, additionally

#UD	If EVEX.vvvv != 1111B.

VREDUCESS—Perform a Reduction Transformation on a Scalar Float32 Value

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.NDS.LIG.66.0F3A.W0 57 /r /ib VREDUCESS xmm1 {k1}{z}, xmm2, xmm3/m32{sae}, imm8	A	V/V	AVX512DQ	Perform a reduction transformation on a scalar single-precision floating point value in xmm3/m32 by subtracting a number of fraction bits specified by the imm8 field. Also, upper single precision floating-point values (bits[127:32]) from xmm2 are copied to xmm1[127:32]. Stores the result in xmm1 register.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Tuple1 Scalar	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA

Description

Perform a reduction transformation of the binary encoded single-precision FP value in the low dword element of the second source operand (the third operand) and store the reduced result in binary FP format to the low dword element of the destination operand (the first operand) under the writemask k1. Bits 127:32 of the destination operand are copied from respective dword elements of the first source operand (the second operand).

The reduction transformation subtracts the integer part and the leading M fractional bits from the binary FP source value, where M is a unsigned integer specified by imm8[7:4], see Figure 5-28. Specifically, the reduction transformation can be expressed as:

dest = src – (ROUND(2^M*src))*2^{-M};

where "Round()" treats "src", "2^M", and their product as binary FP numbers with normalized significand and biased exponents.

The magnitude of the reduced result can be expressed by considering src= 2^p*man2,
where 'man2' is the normalized significand and 'p' is the unbiased exponent

Then if RC = RNE: 0<=|Reduced Result|<=2^{p-M-1}

Then if RC ≠ RNE: 0<=|Reduced Result|<2^{p-M}

This instruction might end up with a precision exception set. However, in case of SPE set (i.e. Suppress Precision Exception, which is imm8[3]=1), no precision exception is reported.

Handling of special case of input values are listed in Table 5-21.

Operation

```
ReduceArgumentSP(SRC[31:0], imm8[7:0])
{
    // Check for NaN
    IF (SRC [31:0] = NAN) THEN
        RETURN (Convert SRC[31:0] to QNaN); FI
    M ← imm8[7:4]; // Number of fraction bits of the normalized significand to be subtracted
    RC ← imm8[1:0];// Round Control for ROUND() operation
    RC source ← imm[2];
    SPE ← 0;// Suppress Precision Exception
    TMP[31:0] ← 2^-M *{ROUND(2^M*SRC[31:0], SPE, RC_source, RC)}; // ROUND() treats SRC and 2^M as standard binary FP values
    TMP[31:0] ← SRC[31:0] – TMP[31:0]; // subtraction under the same RC,SPE controls
RETURN TMP[31:0]; // binary encoded FP with biased exponent and normalized significand
}
```

VREDUCESS

```
IF k1[0] or *no writemask*
    THEN     DEST[31:0] ← ReduceArgumentSP(SRC2[31:0], imm8[7:0])
    ELSE
        IF *merging-masking*                    ; merging-masking
            THEN *DEST[31:0] remains unchanged*
            ELSE                                ; zeroing-masking
                THEN DEST[31:0] = 0
        FI;
FI;
DEST[127:32] ← SRC1[127:32]
DEST[MAXVL-1:128] ← 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VREDUCESS __m128 _mm_mask_reduce_ss(__m128 a, __m128 b, int imm, int sae)
VREDUCESS __m128 _mm_mask_reduce_ss(__m128 s, __mmask16 k, __m128 a, __m128 b, int imm, int sae)
VREDUCESS __m128 _mm_maskz_reduce_ss(__mmask16 k, __m128 a, __m128 b, int imm, int sae)

SIMD Floating-Point Exceptions

Invalid, Precision

If SPE is enabled, precision exception is not reported (regardless of MXCSR exception mask).

Other Exceptions

See Exceptions Type E3.

VRNDSCALEPD—Round Packed Float64 Values To Include A Given Number Of Fraction Bits

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.128.66.0F3A.W1 09 /r ib VRNDSCALEPD xmm1 {k1}{z}, xmm2/m128/m64bcst, imm8	A	V/V	AVX512VL AVX512F	Rounds packed double-precision floating point values in xmm2/m128/m64bcst to a number of fraction bits specified by the imm8 field. Stores the result in xmm1 register. Under writemask.
EVEX.256.66.0F3A.W1 09 /r ib VRNDSCALEPD ymm1 {k1}{z}, ymm2/m256/m64bcst, imm8	A	V/V	AVX512VL AVX512F	Rounds packed double-precision floating point values in ymm2/m256/m64bcst to a number of fraction bits specified by the imm8 field. Stores the result in ymm1 register. Under writemask.
EVEX.512.66.0F3A.W1 09 /r ib VRNDSCALEPD zmm1 {k1}{z}, zmm2/m512/m64bcst{sae}, imm8	A	V/V	AVX512F	Rounds packed double-precision floating-point values in zmm2/m512/m64bcst to a number of fraction bits specified by the imm8 field. Stores the result in zmm1 register using writemask k1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Full Vector	ModRM:reg (w)	ModRM:r/m (r)	Imm8	NA

Description

Round the double-precision floating-point values in the source operand by the rounding mode specified in the immediate operand (see Figure 5-29) and places the result in the destination operand.

The destination operand (the first operand) is a ZMM/YMM/XMM register conditionally updated according to the writemask. The source operand (the second operand) can be a ZMM/YMM/XMM register, a 512/256/128-bit memory location, or a 512/256/128-bit vector broadcasted from a 64-bit memory location.

The rounding process rounds the input to an integral value, plus number bits of fraction that are specified by imm8[7:4] (to be included in the result) and returns the result as a double-precision floating-point value.

It should be noticed that no overflow is induced while executing this instruction (although the source is scaled by the imm8[7:4] value).

The immediate operand also specifies control fields for the rounding operation, three bit fields are defined and shown in the "Immediate Control Description" figure below. Bit 3 of the immediate byte controls the processor behavior for a precision exception, bit 2 selects the source of rounding mode control. Bits 1:0 specify a non-sticky rounding-mode value (Immediate control table below lists the encoded values for rounding-mode field).

The Precision Floating-Point Exception is signaled according to the immediate operand. If any source operand is an SNaN then it will be converted to a QNaN. If DAZ is set to '1 then denormals will be converted to zero before rounding.

The sign of the result of this instruction is preserved, including the sign of zero.

The formula of the operation on each data element for VRNDSCALEPD is

ROUND(x) = 2^{-M}*Round_to_INT(x*2^{M}, round_ctrl),

round_ctrl = imm[3:0];

M=imm[7:4];

The operation of x*2^{M} is computed as if the exponent range is unlimited (i.e. no overflow ever occurs).

VRNDSCALEPD is a more general form of the VEX-encoded VROUNDPD instruction. In VROUNDPD, the formula of the operation on each element is

ROUND(x) = Round_to_INT(x, round_ctrl),

round_ctrl = imm[3:0];

Note: EVEX.vvvv is reserved and must be 1111b, otherwise instructions will #UD.

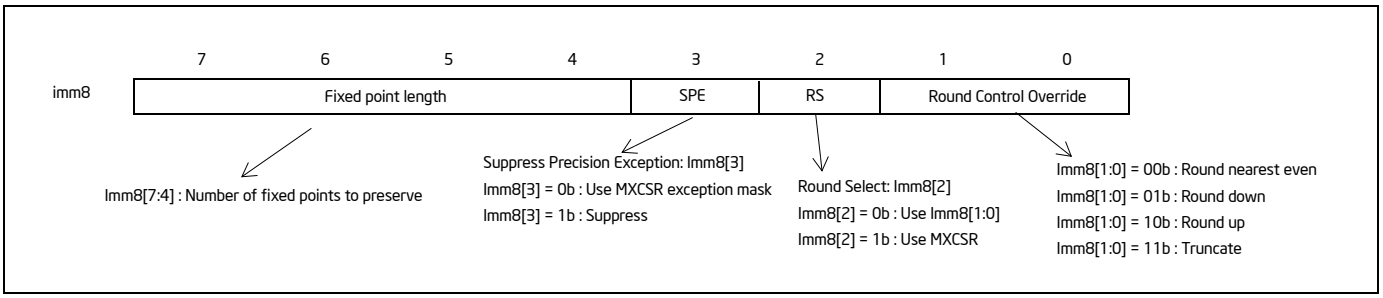


Figure 5-29. Imm8 Controls for VRNDSCALEPD/SD/PS/SS

Handling of special case of input values are listed in Table 5-22.

Table 5-22. VRNDSCALEPD/SD/PS/SS Special Cases

	Returned value
Src1=±inf	Src1
Src1=±NAN	Src1 converted to QNAN
Src1=±0	Src1

Operation

```
RoundToIntegerDP(SRC[63:0], imm8[7:0]) {
    if (imm8[2] = 1)
        rounding_direction ← MXCSR:RC       ; get round control from MXCSR
    else
        rounding_direction ← imm8[1:0]      ; get round control from imm8[1:0]
    FI
    M ← imm8[7:4]           ; get the scaling factor

    case (rounding_direction)
    00: TMP[63:0] ← round_to_nearest_even_integer(2^M*SRC[63:0])
    01: TMP[63:0] ← round_to_equal_or_smaller_integer(2^M*SRC[63:0])
    10: TMP[63:0] ← round_to_equal_or_larger_integer(2^M*SRC[63:0])
    11: TMP[63:0] ← round_to_nearest_smallest_magnitude_integer(2^M*SRC[63:0])
    ESAC

    Dest[63:0] ← 2^-M* TMP[63:0]            ; scale down back to 2^-M

    if (imm8[3] = 0) Then    ; check SPE
        if (SRC[63:0] != Dest[63:0]) Then       ; check precision lost
            set_precision()                     ; set #PE
        FI;
    FI;
    return(Dest[63:0])
}
```

VRNDSCALEPD (EVEX encoded versions)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
IF *src is a memory operand*
    THEN TMP_SRC ← BROADCAST64(SRC, VL, k1)
    ELSE TMP_SRC ← SRC
FI;

FOR j ← 0 TO KL-1
    i ← j * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+63:i] ← RoundToIntegerDP((TMP_SRC[i+63:i], imm8[7:0])
    ELSE
        IF *merging-masking*                    ; merging-masking
            THEN *DEST[i+63:i] remains unchanged*
            ELSE                                ; zeroing-masking
                DEST[i+63:i] ← 0
        FI;
    FI;
ENDFOR;
DEST[MAXVL-1:VL] ← 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VRNDSCALEPD __m512d _mm512_roundscale_pd(__m512d a, int imm);
VRNDSCALEPD __m512d _mm512_roundscale_round_pd(__m512d a, int imm, int sae);
VRNDSCALEPD __m512d _mm512_mask_roundscale_pd(__m512d s, __mmask8 k, __m512d a, int imm);
VRNDSCALEPD __m512d _mm512_mask_roundscale_round_pd(__m512d s, __mmask8 k, __m512d a, int imm, int sae);
VRNDSCALEPD __m512d _mm512_maskz_roundscale_pd(__mmask8 k, __m512d a, int imm);
VRNDSCALEPD __m512d _mm512_maskz_roundscale_round_pd(__mmask8 k, __m512d a, int imm, int sae);
VRNDSCALEPD __m256d _mm256_roundscale_pd(__m256d a, int imm);
VRNDSCALEPD __m256d _mm256_mask_roundscale_pd(__m256d s, __mmask8 k, __m256d a, int imm);
VRNDSCALEPD __m256d _mm256_maskz_roundscale_pd(__mmask8 k, __m256d a, int imm);
VRNDSCALEPD __m128d _mm_roundscale_pd(__m128d a, int imm);
VRNDSCALEPD __m128d _mm_mask_roundscale_pd(__m128d s, __mmask8 k, __m128d a, int imm);
VRNDSCALEPD __m128d _mm_maskz_roundscale_pd(__mmask8 k, __m128d a, int imm);

SIMD Floating-Point Exceptions

Invalid, Precision

If SPE is enabled, precision exception is not reported (regardless of MXCSR exception mask).

Other Exceptions

See Exceptions Type E2.

VRNDSCALESD—Round Scalar Float64 Value To Include A Given Number Of Fraction Bits

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.NDS.LIG.66.0F3A.W1 0B /r ib VRNDSCALESD xmm1 {k1}{z}, xmm2, xmm3/m64{sae}, imm8	A	V/V	AVX512F	Rounds scalar double-precision floating-point value in xmm3/m64 to a number of fraction bits specified by the imm8 field. Stores the result in xmm1 register.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Tuple1 Scalar	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	Imm8

Description

Rounds a double-precision floating-point value in the low quadword (see Figure 5-29) element the second source operand (the third operand) by the rounding mode specified in the immediate operand and places the result in the corresponding element of the destination operand (the third operand) according to the writemask. The quadword element at bits 127:64 of the destination is copied from the first source operand (the second operand).

The destination and first source operands are XMM registers, the 2nd source operand can be an XMM register or memory location. Bits MAXVL-1:128 of the destination register are cleared.

The rounding process rounds the input to an integral value, plus number bits of fraction that are specified by imm8[7:4] (to be included in the result) and returns the result as a double-precision floating-point value.

It should be noticed that no overflow is induced while executing this instruction (although the source is scaled by the imm8[7:4] value).

The immediate operand also specifies control fields for the rounding operation, three bit fields are defined and shown in the "Immediate Control Description" figure below. Bit 3 of the immediate byte controls the processor behavior for a precision exception, bit 2 selects the source of rounding mode control. Bits 1:0 specify a non-sticky rounding-mode value (Immediate control table below lists the encoded values for rounding-mode field).

The Precision Floating-Point Exception is signaled according to the immediate operand. If any source operand is an SNaN then it will be converted to a QNaN. If DAZ is set to `1 then denormals will be converted to zero before rounding.

The sign of the result of this instruction is preserved, including the sign of zero.

The formula of the operation for VRNDSCALESD is

ROUND(x) = 2^{-M}*Round_to_INT(x*2^{M}, round_ctrl),

round_ctrl = imm[3:0];

M=imm[7:4];

The operation of x*2^{M} is computed as if the exponent range is unlimited (i.e. no overflow ever occurs).

VRNDSCALESD is a more general form of the VEX-encoded VROUNDSD instruction. In VROUNDSD, the formula of the operation is

ROUND(x) = Round_to_INT(x, round_ctrl),

round_ctrl = imm[3:0];

EVEX encoded version: The source operand is a XMM register or a 64-bit memory location. The destination operand is a XMM register.

Handling of special case of input values are listed in Table 5-22.

Operation

```
RoundToIntegerDP(SRC[63:0], imm8[7:0]) {
    if (imm8[2] = 1)
        rounding_direction ← MXCSR:RC       ; get round control from MXCSR
    else
        rounding_direction ← imm8[1:0]      ; get round control from imm8[1:0]
    FI
    M ← imm8[7:4]           ; get the scaling factor

    case (rounding_direction)
    00: TMP[63:0] ← round_to_nearest_even_integer(2^M*SRC[63:0])
    01: TMP[63:0] ← round_to_equal_or_smaller_integer(2^M*SRC[63:0])
    10: TMP[63:0] ← round_to_equal_or_larger_integer(2^M*SRC[63:0])
    11: TMP[63:0] ← round_to_nearest_smallest_magnitude_integer(2^M*SRC[63:0])
    ESAC

    Dest[63:0] ← 2^-M* TMP[63:0]          ; scale down back to 2^-M

    if (imm8[3] = 0) Then    ; check SPE
        if (SRC[63:0] != Dest[63:0]) Then       ; check precision lost
            set_precision()                 ; set #PE
        FI;
    FI;
    return(Dest[63:0])
}

VRNDSCALESD (EVEX encoded version)
IF k1[0] or *no writemask*
    THEN    DEST[63:0] ← RoundToIntegerDP(SRC2[63:0], Zero_upper_imm[7:0])
    ELSE
        IF *merging-masking*                    ; merging-masking
            THEN *DEST[63:0] remains unchanged*
            ELSE                                ; zeroing-masking
                THEN DEST[63:0] ← 0
        FI;
FI;
DEST[127:64] ← SRC1[127:64]
DEST[MAXVL-1:128] ← 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VRNDSCALESD __m128d _mm_roundscale_sd (__m128d a, __m128d b, int imm);
VRNDSCALESD __m128d _mm_roundscale_round_sd (__m128d a, __m128d b, int imm, int sae);
VRNDSCALESD __m128d _mm_mask_roundscale_sd (__m128d s, __mmask8 k, __m128d a, __m128d b, int imm);
VRNDSCALESD __m128d _mm_mask_roundscale_round_sd (__m128d s, __mmask8 k, __m128d a, __m128d b, int imm, int sae);
VRNDSCALESD __m128d _mm_maskz_roundscale_sd (__mmask8 k, __m128d a, __m128d b, int imm);
VRNDSCALESD __m128d _mm_maskz_roundscale_round_sd (__mmask8 k, __m128d a, __m128d b, int imm, int sae);

SIMD Floating-Point Exceptions

Invalid, Precision

If SPE is enabled, precision exception is not reported (regardless of MXCSR exception mask).

Other Exceptions

See Exceptions Type E3.

VRNDSCALEPS—Round Packed Float32 Values To Include A Given Number Of Fraction Bits

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.128.66.0F3A.W0 08 /r ib VRNDSCALEPS xmm1 {k1}{z}, xmm2/m128/m32bcst, imm8	A	V/V	AVX512VL AVX512F	Rounds packed single-precision floating point values in xmm2/m128/m32bcst to a number of fraction bits specified by the imm8 field. Stores the result in xmm1 register. Under writemask.
EVEX.256.66.0F3A.W0 08 /r ib VRNDSCALEPS ymm1 {k1}{z}, ymm2/m256/m32bcst, imm8	A	V/V	AVX512VL AVX512F	Rounds packed single-precision floating point values in ymm2/m256/m32bcst to a number of fraction bits specified by the imm8 field. Stores the result in ymm1 register. Under writemask.
EVEX.512.66.0F3A.W0 08 /r ib VRNDSCALEPS zmm1 {k1}{z}, zmm2/m512/m32bcst{sae}, imm8	A	V/V	AVX512F	Rounds packed single-precision floating-point values in zmm2/m512/m32bcst to a number of fraction bits specified by the imm8 field. Stores the result in zmm1 register using writemask.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Full Vector	ModRM:reg (w)	ModRM:r/m (r)	Imm8	NA

Description

Round the single-precision floating-point values in the source operand by the rounding mode specified in the immediate operand (see Figure 5-29) and places the result in the destination operand.

The destination operand (the first operand) is a ZMM register conditionally updated according to the writemask. The source operand (the second operand) can be a ZMM register, a 512-bit memory location, or a 512-bit vector broadcasted from a 32-bit memory location.

The rounding process rounds the input to an integral value, plus number bits of fraction that are specified by imm8[7:4] (to be included in the result) and returns the result as a single-precision floating-point value.

It should be noticed that no overflow is induced while executing this instruction (although the source is scaled by the imm8[7:4] value).

The immediate operand also specifies control fields for the rounding operation, three bit fields are defined and shown in the "Immediate Control Description" figure below. Bit 3 of the immediate byte controls the processor behavior for a precision exception, bit 2 selects the source of rounding mode control. Bits 1:0 specify a non-sticky rounding-mode value (Immediate control table below lists the encoded values for rounding-mode field).

The Precision Floating-Point Exception is signaled according to the immediate operand. If any source operand is an SNaN then it will be converted to a QNaN. If DAZ is set to '1 then denormals will be converted to zero before rounding.

The sign of the result of this instruction is preserved, including the sign of zero.

The formula of the operation on each data element for VRNDSCALEPS is

ROUND(x) = 2^{-M}*Round_to_INT(x*2^{M}, round_ctrl),

round_ctrl = imm[3:0];

M=imm[7:4];

The operation of x*2^{M} is computed as if the exponent range is unlimited (i.e. no overflow ever occurs).

VRNDSCALEPS is a more general form of the VEX-encoded VROUNDPS instruction. In VROUNDPS, the formula of the operation on each element is

ROUND(x) = Round_to_INT(x, round_ctrl),

round_ctrl = imm[3:0];

Note: EVEX.vvvv is reserved and must be 1111b, otherwise instructions will #UD.
Handling of special case of input values are listed in Table 5-22.

Operation

```
RoundToIntegerSP(SRC[31:0], imm8[7:0]) {
    if (imm8[2] = 1)
        rounding_direction ← MXCSR:RC       ; get round control from MXCSR
    else
        rounding_direction ← imm8[1:0]      ; get round control from imm8[1:0]
    FI
    M ← imm8[7:4]           ; get the scaling factor

    case (rounding_direction)
    00: TMP[31:0] ← round_to_nearest_even_integer(2^M*SRC[31:0])
    01: TMP[31:0] ← round_to_equal_or_smaller_integer(2^M*SRC[31:0])
    10: TMP[31:0] ← round_to_equal_or_larger_integer(2^M*SRC[31:0])
    11: TMP[31:0] ← round_to_nearest_smallest_magnitude_integer(2^M*SRC[31:0])
    ESAC;

    Dest[31:0] ← 2^-M* TMP[31:0]          ; scale down back to 2^-M
    if (imm8[3] = 0) Then          ; check SPE
        if (SRC[31:0] != Dest[31:0]) Then       ; check precision lost
            set_precision()             ; set #PE
        FI;
    FI;
    return(Dest[31:0])
}

VRNDSCALEPS (EVEX encoded versions)
(KL, VL) = (4, 128), (8, 256), (16, 512)
IF *src is a memory operand*
    THEN TMP_SRC ← BROADCAST32(SRC, VL, k1)
    ELSE TMP_SRC ← SRC
FI;

FOR j ← 0 TO KL-1
    i ← j * 32
    IF k1[j] OR *no writemask*
        THEN DEST[i+31:i] ← RoundToIntegerSP(TMP_SRC[i+31:i]), imm8[7:0])
    ELSE
        IF *merging-masking*                    ; merging-masking
            THEN *DEST[i+31:i] remains unchanged*
            ELSE                                ; zeroing-masking
                DEST[i+31:i] ← 0
        FI;
    FI;
ENDFOR;
DEST[MAXVL-1:VL] ← 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VRNDSCALEPS __m512 _mm512_roundscale_ps(__m512 a, int imm);
VRNDSCALEPS __m512 _mm512_roundscale_round_ps(__m512 a, int imm, int sae);
VRNDSCALEPS __m512 _mm512_mask_roundscale_ps(__m512 s, __mmask16 k, __m512 a, int imm);
VRNDSCALEPS __m512 _mm512_mask_roundscale_round_ps(__m512 s, __mmask16 k, __m512 a, int imm, int sae);
VRNDSCALEPS __m512 _mm512_maskz_roundscale_ps(__mmask16 k, __m512 a, int imm);
VRNDSCALEPS __m512 _mm512_maskz_roundscale_round_ps(__mmask16 k, __m512 a, int imm, int sae);
VRNDSCALEPS __m256 _mm256_roundscale_ps(__m256 a, int imm);
VRNDSCALEPS __m256 _mm256_mask_roundscale_ps(__m256 s, __mmask8 k, __m256 a, int imm);
VRNDSCALEPS __m256 _mm256_maskz_roundscale_ps(__mmask8 k, __m256 a, int imm);
VRNDSCALEPS __m128 _mm_roundscale_ps(__m256 a, int imm);
VRNDSCALEPS __m128 _mm_mask_roundscale_ps(__m128 s, __mmask8 k, __m128 a, int imm);
VRNDSCALEPS __m128 _mm_maskz_roundscale_ps(__mmask8 k, __m128 a, int imm);

SIMD Floating-Point Exceptions

Invalid, Precision

If SPE is enabled, precision exception is not reported (regardless of MXCSR exception mask).

Other Exceptions

See Exceptions Type E2.

VRNDSCALESS—Round Scalar Float32 Value To Include A Given Number Of Fraction Bits

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.NDS.LIG.66.0F3A.W0 0A /r ib VRNDSCALESS xmm1 {k1}{z}, xmm2, xmm3/m32{sae}, imm8	A	V/V	AVX512F	Rounds scalar single-precision floating-point value in xmm3/m32 to a number of fraction bits specified by the imm8 field. Stores the result in xmm1 register under writemask.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Tuple1 Scalar	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA

Description

Rounds the single-precision floating-point value in the low doubleword element of the second source operand (the third operand) by the rounding mode specified in the immediate operand (see Figure 5-29) and places the result in the corresponding element of the destination operand (the first operand) according to the writemask. The double-word elements at bits 127:32 of the destination are copied from the first source operand (the second operand).

The destination and first source operands are XMM registers, the 2nd source operand can be an XMM register or memory location. Bits MAXVL-1:128 of the destination register are cleared.

The rounding process rounds the input to an integral value, plus number bits of fraction that are specified by imm8[7:4] (to be included in the result) and returns the result as a single-precision floating-point value.

It should be noticed that no overflow is induced while executing this instruction (although the source is scaled by the imm8[7:4] value).

The immediate operand also specifies control fields for the rounding operation, three bit fields are defined and shown in the "Immediate Control Description" figure below. Bit 3 of the immediate byte controls the processor behavior for a precision exception, bit 2 selects the source of rounding mode control. Bits 1:0 specify a non-sticky rounding-mode value (Immediate control tables below lists the encoded values for rounding-mode field).

The Precision Floating-Point Exception is signaled according to the immediate operand. If any source operand is an SNaN then it will be converted to a QNaN. If DAZ is set to `1 then denormals will be converted to zero before rounding.

The sign of the result of this instruction is preserved, including the sign of zero.

The formula of the operation for VRNDSCALESS is

ROUND(x) = 2^{-M}*Round_to_INT(x*2^{M}, round_ctrl),

round_ctrl = imm[3:0];

M=imm[7:4];

The operation of x*2^{M} is computed as if the exponent range is unlimited (i.e. no overflow ever occurs).

VRNDSCALESS is a more general form of the VEX-encoded VROUNDSS instruction. In VROUNDSS, the formula of the operation on each element is

ROUND(x) = Round_to_INT(x, round_ctrl),

round_ctrl = imm[3:0];

EVEX encoded version: The source operand is a XMM register or a 32-bit memory location. The destination operand is a XMM register.

Handling of special case of input values are listed in Table 5-22.

Operation

```
RoundToIntegerSP(SRC[31:0], imm8[7:0]) {
    if (imm8[2] = 1)
        rounding_direction ← MXCSR:RC       ; get round control from MXCSR
    else
        rounding_direction ← imm8[1:0]      ; get round control from imm8[1:0]
    FI
    M ← imm8[7:4]           ; get the scaling factor

    case (rounding_direction)
    00: TMP[31:0] ← round_to_nearest_even_integer(2^M*SRC[31:0])
    01: TMP[31:0] ← round_to_equal_or_smaller_integer(2^M*SRC[31:0])
    10: TMP[31:0] ← round_to_equal_or_larger_integer(2^M*SRC[31:0])
    11: TMP[31:0] ← round_to_nearest_smallest_magnitude_integer(2^M*SRC[31:0])
    ESAC;

    Dest[31:0] ← 2^-M* TMP[31:0]          ; scale down back to 2^-M
    if (imm8[3] = 0) Then          ; check SPE
        if (SRC[31:0] != Dest[31:0]) Then       ; check precision lost
            set_precision()             ; set #PE
        FI;
    FI;
    return(Dest[31:0])
}

VRNDSCALESS (EVEX encoded version)
IF k1[0] or *no writemask*
    THEN    DEST[31:0] ← RoundToIntegerSP(SRC2[31:0], Zero_upper_imm[7:0])
    ELSE
        IF *merging-masking*                    ; merging-masking
            THEN *DEST[31:0] remains unchanged*
            ELSE                                ; zeroing-masking
                THEN DEST[31:0] ← 0
        FI;
FI;
DEST[127:32] ← SRC1[127:32]
DEST[MAXVL-1:128] ← 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VRNDSCALESS __m128 _mm_roundscale_ss (__m128 a, __m128 b, int imm);
VRNDSCALESS __m128 _mm_roundscale_round_ss (__m128 a, __m128 b, int imm, int sae);
VRNDSCALESS __m128 _mm_mask_roundscale_ss (__m128 s, __mmask8 k, __m128 a, __m128 b, int imm);
VRNDSCALESS __m128 _mm_mask_roundscale_round_ss (__m128 s, __mmask8 k, __m128 a, __m128 b, int imm, int sae);
VRNDSCALESS __m128 _mm_maskz_roundscale_ss (__mmask8 k, __m128 a, __m128 b, int imm);
VRNDSCALESS __m128 _mm_maskz_roundscale_round_ss (__mmask8 k, __m128 a, __m128 b, int imm, int sae);

SIMD Floating-Point Exceptions

Invalid, Precision

If SPE is enabled, precision exception is not reported (regardless of MXCSR exception mask).

Other Exceptions

See Exceptions Type E3.

VRSQRT14PD—Compute Approximate Reciprocals of Square Roots of Packed Float64 Values

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.128.66.0F38.W1 4E /r VRSQRT14PD xmm1 {k1}{z}, xmm2/m128/m64bcst	A	V/V	AVX512VL AVX512F	Computes the approximate reciprocal square roots of the packed double-precision floating-point values in xmm2/m128/m64bcst and stores the results in xmm1. Under writemask.
EVEX.256.66.0F38.W1 4E /r VRSQRT14PD ymm1 {k1}{z}, ymm2/m256/m64bcst	A	V/V	AVX512VL AVX512F	Computes the approximate reciprocal square roots of the packed double-precision floating-point values in ymm2/m256/m64bcst and stores the results in ymm1. Under writemask.
EVEX.512.66.0F38.W1 4E /r VRSQRT14PD zmm1 {k1}{z}, zmm2/m512/m64bcst	A	V/V	AVX512F	Computes the approximate reciprocal square roots of the packed double-precision floating-point values in zmm2/m512/m64bcst and stores the results in zmm1 under writemask.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Full Vector	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

Description

This instruction performs a SIMD computation of the approximate reciprocals of the square roots of the eight packed double-precision floating-point values in the source operand (the second operand) and stores the packed double-precision floating-point results in the destination operand (the first operand) according to the writemask. The maximum relative error for this approximation is less than 2^{-14}.

EVEX.512 encoded version: The source operand can be a ZMM register, a 512-bit memory location, or a 512-bit vector broadcasted from a 64-bit memory location. The destination operand is a ZMM register, conditionally updated using writemask k1.

EVEX.256 encoded version: The source operand is a YMM register, a 256-bit memory location, or a 256-bit vector broadcasted from a 64-bit memory location. The destination operand is a YMM register, conditionally updated using writemask k1.

EVEX.128 encoded version: The source operand is a XMM register, a 128-bit memory location, or a 128-bit vector broadcasted from a 64-bit memory location. The destination operand is a XMM register, conditionally updated using writemask k1.

The VRSQRT14PD instruction is not affected by the rounding control bits in the MXCSR register. When a source value is a 0.0, an ∞ with the sign of the source value is returned. When the source operand is an +∞ then +ZERO value is returned. A denormal source value is treated as zero only if DAZ bit is set in MXCSR. Otherwise it is treated correctly and performs the approximation with the specified masked response. When a source value is a negative value (other than 0.0) a floating-point QNaN_indefinite is returned. When a source value is an SNaN or QNaN, the SNaN is converted to a QNaN or the source QNaN is returned.

MXCSR exception flags are not affected by this instruction and floating-point exceptions are not reported.

Note: EVEX.vvvv is reserved and must be 1111b, otherwise instructions will #UD.

A numerically exact implementation of VRSQRT14xx can be found at https://software.intel.com/en-us/articles/reference-implementations-for-IA-approximation-instructions-vrcp14-vrsqrt14-vrcp28-vrsqrt28-vexp2.

Operation

VRSQRT14PD (EVEX encoded versions)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j ← 0 TO KL-1
    i ← j * 64
    IF k1[j] OR *no writemask* THEN
            IF (EVEX.b = 1) AND (SRC *is memory*)
                THEN DEST[i+63:i] ← APPROXIMATE(1.0/ SQRT(SRC[63:0]));
                ELSE DEST[i+63:i] ← APPROXIMATE(1.0/ SQRT(SRC[i+63:i]));
            FI;
    ELSE
        IF *merging-masking*                    ; merging-masking
            THEN *DEST[i+63:i] remains unchanged*
            ELSE                                ; zeroing-masking
                DEST[i+63:i] ← 0
        FI;
    FI;
ENDFOR;
DEST[MAXVL-1:VL] ← 0
```

Table 5-23. VRSQRT14PD Special Cases

Input value	Result value	Comments
Any denormal	Normal	Cannot generate overflow
X = 2^{-2n}	2^n	
X < 0	QNaN_Indefinite	Including -INF
X = -0	-INF	
X = +0	+INF	
X = +INF	+0	

Intel C/C++ Compiler Intrinsic Equivalent

VRSQRT14PD __m512d _mm512_rsqrt14_pd(__m512d a);
VRSQRT14PD __m512d _mm512_mask_rsqrt14_pd(__m512d s, __mmask8 k, __m512d a);
VRSQRT14PD __m512d _mm512_maskz_rsqrt14_pd(__mmask8 k, __m512d a);
VRSQRT14PD __m256d _mm256_rsqrt14_pd(__m256d a);
VRSQRT14PD __m256d _mm512_mask_rsqrt14_pd(__m256d s, __mmask8 k, __m256d a);
VRSQRT14PD __m256d _mm512_maskz_rsqrt14_pd(__mmask8 k, __m256d a);
VRSQRT14PD __m128d _mm_rsqrt14_pd(__m128d a);
VRSQRT14PD __m128d _mm_mask_rsqrt14_pd(__m128d s, __mmask8 k, __m128d a);
VRSQRT14PD __m128d _mm_maskz_rsqrt14_pd(__mmask8 k, __m128d a);

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type E4.

VRSQRT14SD—Compute Approximate Reciprocal of Square Root of Scalar Float64 Value

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.NDS.LIG.66.0F38.W1 4F /r VRSQRT14SD xmm1 {k1}{z}, xmm2, xmm3/m64	A	V/V	AVX512F	Computes the approximate reciprocal square root of the scalar double-precision floating-point value in xmm3/m64 and stores the result in the low quadword element of xmm1 using writemask k1. Bits[127:64] of xmm2 is copied to xmm1[127:64].

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Tuple1 Scalar	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA

Description

Computes the approximate reciprocal of the square roots of the scalar double-precision floating-point value in the low quadword element of the source operand (the second operand) and stores the result in the low quadword element of the destination operand (the first operand) according to the writemask. The maximum relative error for this approximation is less than 2^{-14}. The source operand can be an XMM register or a 32-bit memory location. The destination operand is an XMM register.

Bits (127:64) of the XMM register destination are copied from corresponding bits in the first source operand. Bits (MAXVL-1:128) of the destination register are zeroed.

The VRSQRT14SD instruction is not affected by the rounding control bits in the MXCSR register. When a source value is a 0.0, an ∞ with the sign of the source value is returned. When the source operand is an +∞ then +ZERO value is returned. A denormal source value is treated as zero only if DAZ bit is set in MXCSR. Otherwise it is treated correctly and performs the approximation with the specified masked response. When a source value is a negative value (other than 0.0) a floating-point QNaN_indefinite is returned. When a source value is an SNaN or QNaN, the SNaN is converted to a QNaN or the source QNaN is returned.

MXCSR exception flags are not affected by this instruction and floating-point exceptions are not reported.

A numerically exact implementation of VRSQRT14xx can be found at https://software.intel.com/en-us/articles/reference-implementations-for-IA-approximation-instructions-vrcp14-vrsqrt14-vrcp28-vrsqrt28-vexp2.

Operation

VRSQRT14SD (EVEX version)

```
IF k1[0] or *no writemask*
    THEN     DEST[63:0] ← APPROXIMATE(1.0/ SQRT(SRC2[63:0]))
    ELSE
        IF *merging-masking*                    ; merging-masking
            THEN *DEST[63:0] remains unchanged*
            ELSE                                ; zeroing-masking
                THEN DEST[63:0] ← 0
        FI;
FI;
DEST[127:64] ← SRC1[127:64]
DEST[MAXVL-1:128] ← 0
```

Table 5-24. VRSQRT14SD Special Cases

Input value	Result value	Comments
Any denormal	Normal	Cannot generate overflow
X = 2^{-2n}	2^n	
X < 0	QNaN_Indefinite	Including -INF
X = -0	-INF	
X = +0	+INF	
X = +INF	+0	

Intel C/C++ Compiler Intrinsic Equivalent

VRSQRT14SD __m128d _mm_rsqrt14_sd(__m128d a, __m128d b);
VRSQRT14SD __m128d _mm_mask_rsqrt14_sd(__m128d s, __mmask8 k, __m128d a, __m128d b);
VRSQRT14SD __m128d _mm_maskz_rsqrt14_sd(__mmask8d m, __m128d a, __m128d b);

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type E5.

VRSQRT14PS—Compute Approximate Reciprocals of Square Roots of Packed Float32 Values

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.128.66.0F38.W0 4E /r VRSQRT14PS xmm1 {k1}{z}, xmm2/m128/m32bcst	A	V/V	AVX512VL AVX512F	Computes the approximate reciprocal square roots of the packed single-precision floating-point values in xmm2/m128/m32bcst and stores the results in xmm1. Under writemask.
EVEX.256.66.0F38.W0 4E /r VRSQRT14PS ymm1 {k1}{z}, ymm2/m256/m32bcst	A	V/V	AVX512VL AVX512F	Computes the approximate reciprocal square roots of the packed single-precision floating-point values in ymm2/m256/m32bcst and stores the results in ymm1. Under writemask.
EVEX.512.66.0F38.W0 4E /r VRSQRT14PS zmm1 {k1}{z}, zmm2/m512/m32bcst	A	V/V	AVX512F	Computes the approximate reciprocal square roots of the packed single-precision floating-point values in zmm2/m512/m32bcst and stores the results in zmm1. Under writemask.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Full Vector	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

Description

This instruction performs a SIMD computation of the approximate reciprocals of the square roots of 16 packed single-precision floating-point values in the source operand (the second operand) and stores the packed single-precision floating-point results in the destination operand (the first operand) according to the writemask. The maximum relative error for this approximation is less than 2^{-14}.

EVEX.512 encoded version: The source operand can be a ZMM register, a 512-bit memory location or a 512-bit vector broadcasted from a 32-bit memory location. The destination operand is a ZMM register, conditionally updated using writemask k1.

EVEX.256 encoded version: The source operand is a YMM register, a 256-bit memory location, or a 256-bit vector broadcasted from a 32-bit memory location. The destination operand is a YMM register, conditionally updated using writemask k1.

EVEX.128 encoded version: The source operand is a XMM register, a 128-bit memory location, or a 128-bit vector broadcasted from a 32-bit memory location. The destination operand is a XMM register, conditionally updated using writemask k1.

The VRSQRT14PS instruction is not affected by the rounding control bits in the MXCSR register. When a source value is a 0.0, an ∞ with the sign of the source value is returned. When the source operand is an +∞ then +ZERO value is returned. A denormal source value is treated as zero only if DAZ bit is set in MXCSR. Otherwise it is treated correctly and performs the approximation with the specified masked response. When a source value is a negative value (other than 0.0) a floating-point QNaN_indefinite is returned. When a source value is an SNaN or QNaN, the SNaN is converted to a QNaN or the source QNaN is returned.

MXCSR exception flags are not affected by this instruction and floating-point exceptions are not reported.

Note: EVEX.vvvv is reserved and must be 1111b, otherwise instructions will #UD.

A numerically exact implementation of VRSQRT14xx can be found at https://software.intel.com/en-us/articles/reference-implementations-for-IA-approximation-instructions-vrcp14-vrsqrt14-vrcp28-vrsqrt28-vexp2.

Operation

VRSQRT14PS (EVEX encoded versions)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j ← 0 TO KL-1
    i ← j * 32
    IF k1[j] OR *no writemask* THEN
            IF (EVEX.b = 1) AND (SRC *is memory*)
                THEN DEST[i+31:i] ← APPROXIMATE(1.0/ SQRT(SRC[31:0]));
                ELSE DEST[i+31:i] ← APPROXIMATE(1.0/ SQRT(SRC[i+31:i]));
            FI;
    ELSE
        IF *merging-masking*                    ; merging-masking
            THEN *DEST[i+31:i] remains unchanged*
            ELSE                                ; zeroing-masking
                DEST[i+31:i] ← 0
        FI;
    FI;
ENDFOR;
DEST[MAXVL-1:VL] ← 0
```

Table 5-25. VRSQRT14PS Special Cases

Input value	Result value	Comments
Any denormal	Normal	Cannot generate overflow
X = 2^{-2n}	2^n	
X < 0	QNaN_Indefinite	Including -INF
X = -0	-INF	
X = +0	+INF	
X = +INF	+0	

Intel C/C++ Compiler Intrinsic Equivalent

VRSQRT14PS __m512 _mm512_rsqrt14_ps(__m512 a);
VRSQRT14PS __m512 _mm512_mask_rsqrt14_ps(__m512 s, __mmask16 k, __m512 a);
VRSQRT14PS __m512 _mm512_maskz_rsqrt14_ps(__mmask16 k, __m512 a);
VRSQRT14PS __m256 _mm256_rsqrt14_ps(__m256 a);
VRSQRT14PS __m256 _mm256_mask_rsqrt14_ps(__m256 s, __mmask8 k, __m256 a);
VRSQRT14PS __m256 _mm256_maskz_rsqrt14_ps(__mmask8 k, __m256 a);
VRSQRT14PS __m128 _mm_rsqrt14_ps(__m128 a);
VRSQRT14PS __m128 _mm_mask_rsqrt14_ps(__m128 s, __mmask8 k, __m128 a);
VRSQRT14PS __m128 _mm_maskz_rsqrt14_ps(__mmask8 k, __m128 a);

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type 4.

VRSQRT14SS—Compute Approximate Reciprocal of Square Root of Scalar Float32 Value

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.NDS.LIG.66.0F38.W0 4F /r VRSQRT14SS xmm1 {k1}{z}, xmm2, xmm3/m32	A	V/V	AVX512F	Computes the approximate reciprocal square root of the scalar single-precision floating-point value in xmm3/m32 and stores the result in the low doubleword element of xmm1 using writemask k1. Bits[127:32] of xmm2 is copied to xmm1[127:32].

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Tuple1 Scalar	ModRM:reg (w)	VEX.vvvv	ModRM:r/m (r)	NA

Description

Computes of the approximate reciprocal of the square root of the scalar single-precision floating-point value in the low doubleword element of the source operand (the second operand) and stores the result in the low doubleword element of the destination operand (the first operand) according to the writemask. The maximum relative error for this approximation is less than 2^{-14}. The source operand can be an XMM register or a 32-bit memory location. The destination operand is an XMM register.

Bits (127:32) of the XMM register destination are copied from corresponding bits in the first source operand. Bits (MAXVL-1:128) of the destination register are zeroed.

The VRSQRT14SS instruction is not affected by the rounding control bits in the MXCSR register. When a source value is a 0.0, an ∞ with the sign of the source value is returned. When the source operand is an ∞, zero with the sign of the source value is returned. A denormal source value is treated as zero only if DAZ bit is set in MXCSR. Otherwise it is treated correctly and performs the approximation with the specified masked response. When a source value is a negative value (other than 0.0) a floating-point indefinite is returned. When a source value is an SNaN or QNaN, the SNaN is converted to a QNaN or the source QNaN is returned.

MXCSR exception flags are not affected by this instruction and floating-point exceptions are not reported.

A numerically exact implementation of VRSQRT14xx can be found at https://software.intel.com/en-us/articles/reference-implementations-for-IA-approximation-instructions-vrcp14-vrsqrt14-vrcp28-vrsqrt28-vexp2.

Operation

VRSQRT14SS (EVEX version)

```
IF k1[0] or *no writemask*
    THEN     DEST[31:0] ← APPROXIMATE(1.0/ SQRT(SRC2[31:0]))
    ELSE
        IF *merging-masking*                    ; merging-masking
            THEN *DEST[31:0] remains unchanged*
            ELSE                                ; zeroing-masking
                THEN DEST[31:0] ← 0
        FI;
FI;
DEST[127:32] ← SRC1[127:32]
DEST[MAXVL-1:128] ← 0
```

Table 5-26. VRSQRT14SS Special Cases

Input value	Result value	Comments
Any denormal	Normal	Cannot generate overflow
X = 2^{-2n}	2^n	
X < 0	QNaN_Indefinite	Including -INF
X = -0	-INF	
X = +0	+INF	
X = +INF	+0	

Intel C/C++ Compiler Intrinsic Equivalent

VRSQRT14SS __m128 _mm_rsqrt14_ss(__m128 a, __m128 b);
VRSQRT14SS __m128 _mm_mask_rsqrt14_ss(__m128 s, __mmask8 k, __m128 a, __m128 b);
VRSQRT14SS __m128 _mm_maskz_rsqrt14_ss(__mmask8 k, __m128 a, __m128 b);

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type E5.

VRSQRT28PD—Approximation to the Reciprocal Square Root of Packed Double-Precision Floating-Point Values with Less Than 2^-28 Relative Error

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.512.66.0F38.W1 CC /r VRSQRT28PD zmm1 {k1}{z}, zmm2/m512/m64bcst {sae}	A	V/V	AVX512ER	Computes approximations to the Reciprocal square root (<2^-28 relative error) of the packed double-precision floating-point values from zmm2/m512/m64bcst and stores result in zmm1with writemask k1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Full Vector	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

Description

Computes the reciprocal square root of the float64 values in the source operand (the second operand) and store the results to the destination operand (the first operand). The approximate reciprocal is evaluated with less than 2^-28 of maximum relative error.

If any source element is NaN, the quietized NaN source value is returned for that element. Negative (non-zero) source numbers, as well as -∞, return the canonical NaN and set the Invalid Flag (#I).

A value of -0 must return -∞ and set the DivByZero flags (#Z). Negative numbers should return NaN and set the Invalid flag (#I). Note however that the instruction flush input denormals to zero of the same sign, so negative denormals return -∞ and set the DivByZero flag.

The source operand is a ZMM register, a 512-bit memory location or a 512-bit vector broadcasted from a 64-bit memory location. The destination operand is a ZMM register, conditionally updated using writemask k1.

EVEX.vvvv is reserved and must be 1111b otherwise instructions will #UD.

A numerically exact implementation of VRSQRT28xx can be found at https://software.intel.com/en-us/articles/reference-implementations-for-IA-approximation-instructions-vrcp14-vrsqrt14-vrcp28-vrsqrt28-vexp2.

Operation

VRSQRT28PD (EVEX encoded versions)

```
(KL, VL) = (8, 512)
FOR j ← 0 TO KL-1
    i ← j * 64
    IF k1[j] OR *no writemask* THEN
            IF (EVEX.b = 1) AND (SRC *is memory*)
                THEN DEST[i+63:i] ← (1.0/ SQRT(SRC[63:0]));
                ELSE DEST[i+63:i] ← (1.0/ SQRT(SRC[i+63:i]));
            FI;
    ELSE
        IF *merging-masking*                ; merging-masking
            THEN *DEST[i+63:i] remains unchanged*
            ELSE                            ; zeroing-masking
                DEST[i+63:i] ← 0
        FI;
    FI;
ENDFOR;
```

Table 5-27. VRSQRT28PD Special Cases

Input value	Result value	Comments
NAN	QNAN(input)	If (SRC = SNaN) then #I
X = 2^{-2n}	2^{n}	
X < 0	QNaN_Indefinite	Including -INF
X = -0 or negative denormal	-INF	#Z
X = +0 or positive denormal	+INF	#Z
X = +INF	+0	

Intel C/C++ Compiler Intrinsic Equivalent

VRSQRT28PD __m512d _mm512_rsqrt28_round_pd(__m512d a, int sae);
VRSQRT28PD __m512d _mm512_mask_rsqrt28_round_pd(__m512d s, __mmask8 m,__m512d a, int sae);
VRSQRT28PD __m512d _mm512_maskz_rsqrt28_round_pd(__mmask8 m,__m512d a, int sae);

SIMD Floating-Point Exceptions

Invalid (if SNaN input), Divide-by-zero

Other Exceptions

See Exceptions Type E2.

VRSQRT28SD—Approximation to the Reciprocal Square Root of Scalar Double-Precision Floating-Point Value with Less Than 2^-28 Relative Error

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.NDS.LIG.66.0F38.W1 CD /r VRSQRT28SD xmm1 {k1}{z}, xmm2, xmm3/m64 {sae}	A	V/V	AVX512ER	Computes approximate reciprocal square root (<2^-28 relative error) of the scalar double-precision floating-point value from xmm3/m64 and stores result in xmm1with writemask k1. Also, upper double-precision floating-point value (bits[127:64]) from xmm2 is copied to xmm1[127:64].

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Tuple1 Scalar	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA

Description

Computes the reciprocal square root of the low float64 value in the second source operand (the third operand) and store the result to the destination operand (the first operand). The approximate reciprocal square root is evaluated with less than 2^-28 of maximum relative error. The result is written into the low float64 element of xmm1 according to the writemask k1. Bits 127:64 of the destination is copied from the corresponding bits of the first source operand (the second operand).

If any source element is NaN, the quietized NaN source value is returned for that element. Negative (non-zero) source numbers, as well as -∞, return the canonical NaN and set the Invalid Flag (#I).

A value of -0 must return -∞ and set the DivByZero flags (#Z). Negative numbers should return NaN and set the Invalid flag (#I). Note however that the instruction flush input denormals to zero of the same sign, so negative denormals return -∞ and set the DivByZero flag.

The first source operand is an XMM register. The second source operand is an XMM register or a 64-bit memory location. The destination operand is a XMM register.

A numerically exact implementation of VRSQRT28xx can be found at https://software.intel.com/en-us/articles/reference-implementations-for-IA-approximation-instructions-vrcp14-vrsqrt14-vrcp28-vrsqrt28-vexp2.

Operation

VRSQRT28SD (EVEX encoded versions)

```
    IF k1[0] OR *no writemask* THEN
                DEST[63: 0] ← (1.0/ SQRT(SRC[63: 0]));
    ELSE
        IF *merging-masking*                    ; merging-masking
            THEN *DEST[63: 0] remains unchanged*
            ELSE                                ; zeroing-masking
                DEST[63: 0] ← 0
        FI;
    FI;
ENDFOR;
DEST[127:64] ← SRC1[127: 64]
DEST[MAXVL-1:128] ← 0
```

Table 5-28. VRSQRT28SD Special Cases

Input value	Result value	Comments
NAN	QNAN(input)	If (SRC = SNaN) then #I
$X = 2^{-2n}$	2^n	
X < 0	QNaN_Indefinite	Including -INF
X = -0 or negative denormal	-INF	#Z
X = +0 or positive denormal	+INF	#Z
X = +INF	+0	

Intel C/C++ Compiler Intrinsic Equivalent

VRSQRT28SD __m128d _mm_rsqrt28_round_sd(__m128d a, __m128b b, int sae);
VRSQRT28SD __m128d _mm_mask_rsqrt28_round_pd(__m128d s, __mmask8 m,__m128d a, __m128d b, int sae);
VRSQRT28SD __m128d _mm_maskz_rsqrt28_round_pd(__mmask8 m,__m128d a, __m128d b, int sae);

SIMD Floating-Point Exceptions

Invalid (if SNaN input), Divide-by-zero

Other Exceptions

See Exceptions Type E3.

VRSQRT28PS—Approximation to the Reciprocal Square Root of Packed Single-Precision Floating-Point Values with Less Than 2^-28 Relative Error

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.512.66.0F38.W0 CC /r VRSQRT28PS zmm1 {k1}{z}, zmm2/m512/m32bcst {sae}	A	V/V	AVX512ER	Computes approximations to the Reciprocal square root (<2^-28 relative error) of the packed single-precision floating-point values from zmm2/m512/m32bcst and stores result in zmm1with writemask k1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Full Vector	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

Description

Computes the reciprocal square root of the float32 values in the source operand (the second operand) and store the results to the destination operand (the first operand). The approximate reciprocal is evaluated with less than 2^-28 of maximum relative error prior to final rounding. The final results is rounded to < 2^-23 relative error before written to the destination.

If any source element is NaN, the quietized NaN source value is returned for that element. Negative (non-zero) source numbers, as well as -∞, return the canonical NaN and set the Invalid Flag (#I).

A value of -0 must return -∞ and set the DivByZero flags (#Z). Negative numbers should return NaN and set the Invalid flag (#I). Note however that the instruction flush input denormals to zero of the same sign, so negative denormals return -∞ and set the DivByZero flag.

The source operand is a ZMM register, a 512-bit memory location, or a 512-bit vector broadcasted from a 32-bit memory location. The destination operand is a ZMM register, conditionally updated using writemask k1.

EVEX.vvvv is reserved and must be 1111b otherwise instructions will #UD.

A numerically exact implementation of VRSQRT28xx can be found at https://software.intel.com/en-us/articles/reference-implementations-for-IA-approximation-instructions-vrcp14-vrsqrt14-vrcp28-vrsqrt28-vexp2.

Operation

VRSQRT28PS (EVEX encoded versions)

```
(KL, VL) = (16, 512)
FOR j ← 0 TO KL-1
    i ← j * 32
    IF k1[j] OR *no writemask* THEN
            IF (EVEX.b = 1) AND (SRC *is memory*)
                THEN DEST[i+31:i] ← (1.0/ SQRT(SRC[31:0]));
                ELSE DEST[i+31:i] ← (1.0/ SQRT(SRC[i+31:i]));
            FI;
    ELSE
        IF *merging-masking*                    ; merging-masking
            THEN *DEST[i+31:i] remains unchanged*
            ELSE                                ; zeroing-masking
                DEST[i+31:i] ← 0
        FI;
    FI;
ENDFOR;
```

Table 5-29. VRSQRT28PS Special Cases

Input value	Result value	Comments
NAN	QNAN(input)	If (SRC = SNaN) then #I
X = 2^{-2n}	2^{n}	
X < 0	QNaN_Indefinite	Including -INF
X = -0 or negative denormal	-INF	#Z
X = +0 or positive denormal	+INF	#Z
X = +INF	+0	

Intel C/C++ Compiler Intrinsic Equivalent

VRSQRT28PS __m512 _mm512_rsqrt28_round_ps(__m512 a, int sae);
VRSQRT28PS __m512 _mm512_mask_rsqrt28_round_ps(__m512 s, __mmask16 m,__m512 a, int sae);
VRSQRT28PS __m512 _mm512_maskz_rsqrt28_round_ps(__mmask16 m,__m512 a, int sae);

SIMD Floating-Point Exceptions

Invalid (if SNaN input), Divide-by-zero

Other Exceptions

See Exceptions Type E2.

VRSQRT28SS—Approximation to the Reciprocal Square Root of Scalar Single-Precision Floating-Point Value with Less Than 2^-28 Relative Error

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.NDS.LIG.66.0F38.W0 CD /r VRSQRT28SS xmm1 {k1}{z}, xmm2, xmm3/m32 {sae}	A	V/V	AVX512ER	Computes approximate reciprocal square root (<2^-28 relative error) of the scalar single-precision floating-point value from xmm3/m32 and stores result in xmm1with writemask k1. Also, upper 3 single-precision floating-point value (bits[127:32]) from xmm2 is copied to xmm1[127:32].

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Tuple1 Scalar	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA

Description

Computes the reciprocal square root of the low float32 value in the second source operand (the third operand) and store the result to the destination operand (the first operand). The approximate reciprocal square root is evaluated with less than 2^-28 of maximum relative error prior to final rounding. The final result is rounded to < 2^-23 relative error before written to the low float32 element of the destination according to the writemask k1. Bits 127:32 of the destination is copied from the corresponding bits of the first source operand (the second operand).

If any source element is NaN, the quietized NaN source value is returned for that element. Negative (non-zero) source numbers, as well as -∞, return the canonical NaN and set the Invalid Flag (#I).

A value of -0 must return -∞ and set the DivByZero flags (#Z). Negative numbers should return NaN and set the Invalid flag (#I). Note however that the instruction flush input denormals to zero of the same sign, so negative denormals return -∞ and set the DivByZero flag.

The first source operand is an XMM register. The second source operand is an XMM register or a 32-bit memory location. The destination operand is a XMM register.

A numerically exact implementation of VRSQRT28xx can be found at https://software.intel.com/en-us/articles/reference-implementations-for-IA-approximation-instructions-vrcp14-vrsqrt14-vrcp28-vrsqrt28-vexp2.

Operation

VRSQRT28SS (EVEX encoded versions)

```
    IF k1[0] OR *no writemask* THEN
                DEST[31: 0] ← (1.0/ SQRT(SRC[31: 0]));
    ELSE
        IF *merging-masking*                    ; merging-masking
            THEN *DEST[31: 0] remains unchanged*
            ELSE                                ; zeroing-masking
                DEST[31: 0] ← 0
        FI;
    FI;
ENDFOR;
DEST[127:32] ← SRC1[127: 32]
DEST[MAXVL-1:128] ← 0
```

Table 5-30. VRSQRT28SS Special Cases

Input value	Result value	Comments
NAN	QNAN(input)	If (SRC = SNaN) then #I
X = 2^{-2n}	2^{n}	
X < 0	QNaN_Indefinite	Including -INF
X = -0 or negative denormal	-INF	#Z
X = +0 or positive denormal	+INF	#Z
X = +INF	+0	

Intel C/C++ Compiler Intrinsic Equivalent

VRSQRT28SS __m128 _mm_rsqrt28_round_ss(__m128 a, __m128 b, int sae);
VRSQRT28SS __m128 _mm512_mask_rsqrt28_round_ss(__m128 s, __mmask8 m,__m128 a,__m128 b, int sae);
VRSQRT28SS __m128 _mm512_maskz_rsqrt28_round_ss(__mmask8 m,__m128 a,__m128 b, int sae);

SIMD Floating-Point Exceptions

Invalid (if SNaN input), Divide-by-zero

Other Exceptions

See Exceptions Type E3.

VSCALEFPD—Scale Packed Float64 Values With Float64 Values

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.NDS.128.66.0F38.W1 2C /r VSCALEFPD xmm1 {k1}{z}, xmm2, xmm3/m128/m64bcst	A	V/V	AVX512VL AVX512F	Scale the packed double-precision floating-point values in xmm2 using values from xmm3/m128/m64bcst. Under writemask k1.
EVEX.NDS.256.66.0F38.W1 2C /r VSCALEFPD ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst	A	V/V	AVX512VL AVX512F	Scale the packed double-precision floating-point values in ymm2 using values from ymm3/m256/m64bcst. Under writemask k1.
EVEX.NDS.512.66.0F38.W1 2C /r VSCALEFPD zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst{er}	A	V/V	AVX512F	Scale the packed double-precision floating-point values in zmm2 using values from zmm3/m512/m64bcst. Under writemask k1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Full Vector	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA

Description

Performs a floating-point scale of the packed double-precision floating-point values in the first source operand by multiplying it by 2 power of the double-precision floating-point values in second source operand.

The equation of this operation is given by:

zmm1 := zmm2*$2^{floor(zmm3)}$.

Floor(zmm3) means maximum integer value ≤ zmm3.

If the result cannot be represented in double precision, then the proper overflow response (for positive scaling operand), or the proper underflow response (for negative scaling operand) is issued. The overflow and underflow responses are dependent on the rounding mode (for IEEE-compliant rounding), as well as on other settings in MXCSR (exception mask bits, FTZ bit), and on the SAE bit.

The first source operand is a ZMM/YMM/XMM register. The second source operand is a ZMM/YMM/XMM register, a 512/256/128-bit memory location or a 512/256/128-bit vector broadcasted from a 64-bit memory location. The destination operand is a ZMM/YMM/XMM register conditionally updated with writemask k1.

Handling of special-case input values are listed in Table 5-31 and Table 5-32.

Table 5-31. \VSCALEFPD/SD/PS/SS Special Cases

		Src2				Set IE
		±NaN	+Inf	-Inf	0/Denorm/Norm	
Src1	±QNaN	QNaN(Src1)	+INF	+0	QNaN(Src1)	IF either source is SNAN
	±SNaN	QNaN(Src1)	QNaN(Src1)	QNaN(Src1)	QNaN(Src1)	YES
	±Inf	QNaN(Src2)	Src1	QNaN_Indefinite	Src1	IF Src2 is SNAN or -INF
	±0	QNaN(Src2)	QNaN_Indefinite	Src1	Src1	IF Src2 is SNAN or +INF
	Denorm/Norm	QNaN(Src2)	±INF (Src1 sign)	±0 (Src1 sign)	Compute Result	IF Src2 is SNAN

Table 5-32. Additional VSCALEFPD/SD Special Cases

Special Case	Returned value	Faults
|result| < 2^{-1074}	±0 or ±Min-Denormal (Src1 sign)	Underflow
|result| ≥ 2^{1024}	±INF (Src1 sign) or ±Max-normal (Src1 sign)	Overflow

Operation

```
SCALE(SRC1, SRC2)
{
TMP_SRC2 ← SRC2
TMP_SRC1 ← SRC1
IF (SRC2 is denormal AND MXCSR.DAZ) THEN TMP_SRC2=0
IF (SRC1 is denormal AND MXCSR.DAZ) THEN TMP_SRC1=0
/* SRC2 is a 64 bits floating-point value */
DEST[63:0] ← TMP_SRC1[63:0] * POW(2, Floor(TMP_SRC2[63:0]))
}
```

VSCALEFPD (EVEX encoded versions)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
IF (VL = 512) AND (EVEX.b = 1) AND (SRC2 *is register*)
    THEN
        SET_RM(EVEX.RC);
    ELSE
        SET_RM(MXCSR.RM);
FI;
FOR j ← 0 TO KL-1
    i ← j * 64
    IF k1[j] OR *no writemask* THEN
            IF (EVEX.b = 1) AND (SRC2 *is memory*)
                THEN DEST[i+63:i] ← SCALE(SRC1[i+63:i], SRC2[63:0]);
                ELSE DEST[i+63:i] ← SCALE(SRC1[i+63:i], SRC2[i+63:i]);
            FI;
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE                                ; zeroing-masking
                    DEST[i+63:i] ← 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VSCALEFPD __m512d _mm512_scalef_round_pd(__m512d a, __m512d b, int);
VSCALEFPD __m512d _mm512_mask_scalef_round_pd(__m512d s, __mmask8 k, __m512d a, __m512d b, int);
VSCALEFPD __m512d _mm512_maskz_scalef_round_pd(__mmask8 k, __m512d a, __m512d b, int);
VSCALEFPD __m256d _mm256_scalef_round_pd(__m256d a, __m256d b, int);
VSCALEFPD __m256d _mm256_mask_scalef_round_pd(__m256d s, __mmask8 k, __m256d a, __m256d b, int);
VSCALEFPD __m256d _mm256_maskz_scalef_round_pd(__mmask8 k, __m256d a, __m256d b, int);
VSCALEFPD __m128d _mm_scalef_round_pd(__m128d a, __m128d b, int);
VSCALEFPD __m128d _mm_mask_scalef_round_pd(__m128d s, __mmask8 k, __m128d a, __m128d b, int);
VSCALEFPD __m128d _mm_maskz_scalef_round_pd(__mmask8 k, __m128d a, __m128d b, int);

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal (for Src1).
Denormal is not reported for Src2.

Other Exceptions

See Exceptions Type E2.

VSCALEFSD—Scale Scalar Float64 Values With Float64 Values

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.NDS.LIG.66.0F38.W1 2D /r VSCALEFSD xmm1 {k1}{z}, xmm2, xmm3/m64{er}	A	V/V	AVX512F	Scale the scalar double-precision floating-point values in xmm2 using the value from xmm3/m64. Under writemask k1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Tuple1 Scalar	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA

Description

Performs a floating-point scale of the packed double-precision floating-point value in the first source operand by multiplying it by 2 power of the double-precision floating-point value in second source operand.

The equation of this operation is given by:

$xmm1 := xmm2*2^{floor(xmm3)}$.

Floor(xmm3) means maximum integer value ≤ xmm3.

If the result cannot be represented in double precision, then the proper overflow response (for positive scaling operand), or the proper underflow response (for negative scaling operand) is issued. The overflow and underflow responses are dependent on the rounding mode (for IEEE-compliant rounding), as well as on other settings in MXCSR (exception mask bits, FTZ bit), and on the SAE bit.

EVEX encoded version: The first source operand is an XMM register. The second source operand is an XMM register or a memory location. The destination operand is an XMM register conditionally updated with writemask k1.

Handling of special-case input values are listed in Table 5-31 and Table 5-32.

Operation

```
SCALE(SRC1, SRC2)
{
    ; Check for denormal operands
TMP_SRC2 ← SRC2
TMP_SRC1 ← SRC1
IF (SRC2 is denormal AND MXCSR.DAZ) THEN TMP_SRC2=0
IF (SRC1 is denormal AND MXCSR.DAZ) THEN TMP_SRC1=0
/* SRC2 is a 64 bits floating-point value */
DEST[63:0] ← TMP_SRC1[63:0] * POW(2, Floor(TMP_SRC2[63:0]))
}
```

VSCALEFSD (EVEX encoded version)

```
IF (EVEX.b= 1) and SRC2 *is a register*
    THEN
        SET_RM(EVEX.RC);
    ELSE
        SET_RM(MXCSR.RM);
FI;
IF k1[0] OR *no writemask*
    THEN DEST[63:0] ← SCALE(SRC1[63:0], SRC2[63:0])
    ELSE
        IF *merging-masking*                    ; merging-masking
            THEN *DEST[63:0] remains unchanged*
            ELSE                                ; zeroing-masking
                DEST[63:0] ← 0
        FI
FI;
DEST[127:64] ← SRC1[127:64]
DEST[MAXVL-1:128] ← 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VSCALEFSD __m128d _mm_scalef_round_sd(__m128d a, __m128d b, int);
VSCALEFSD __m128d _mm_mask_scalef_round_sd(__m128d s, __mmask8 k, __m128d a, __m128d b, int);
VSCALEFSD __m128d _mm_maskz_scalef_round_sd(__mmask8 k, __m128d a, __m128d b, int);

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal (for Src1).
Denormal is not reported for Src2.

Other Exceptions

See Exceptions Type E3.

VSCALEFPS—Scale Packed Float32 Values With Float32 Values

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.NDS.128.66.0F38.W0 2C /r VSCALEFPS xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst	A	V/V	AVX512VL AVX512F	Scale the packed single-precision floating-point values in xmm2 using values from xmm3/m128/m32bcst. Under writemask k1.
EVEX.NDS.256.66.0F38.W0 2C /r VSCALEFPS ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst	A	V/V	AVX512VL AVX512F	Scale the packed single-precision values in ymm2 using floating point values from ymm3/m256/m32bcst. Under writemask k1.
EVEX.NDS.512.66.0F38.W0 2C /r VSCALEFPS zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst{er}	A	V/V	AVX512F	Scale the packed single-precision floating-point values in zmm2 using floating-point values from zmm3/m512/m32bcst. Under writemask k1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Full Vector	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA

Description

Performs a floating-point scale of the packed single-precision floating-point values in the first source operand by multiplying it by 2 power of the float32 values in second source operand.

The equation of this operation is given by:

zmm1 := zmm2*$2^{floor(zmm3)}$.

Floor(zmm3) means maximum integer value ≤ zmm3.

If the result cannot be represented in single precision, then the proper overflow response (for positive scaling operand), or the proper underflow response (for negative scaling operand) is issued. The overflow and underflow responses are dependent on the rounding mode (for IEEE-compliant rounding), as well as on other settings in MXCSR (exception mask bits, FTZ bit), and on the SAE bit.

EVEX.512 encoded version: The first source operand is a ZMM register. The second source operand is a ZMM register, a 512-bit memory location or a 512-bit vector broadcasted from a 32-bit memory location. The destination operand is a ZMM register conditionally updated with writemask k1.

EVEX.256 encoded version: The first source operand is a YMM register. The second source operand is a YMM register, a 256-bit memory location, or a 256-bit vector broadcasted from a 32-bit memory location. The destination operand is a YMM register, conditionally updated using writemask k1.

EVEX.128 encoded version: The first source operand is an XMM register. The second source operand is a XMM register, a 128-bit memory location, or a 128-bit vector broadcasted from a 32-bit memory location. The destination operand is a XMM register, conditionally updated using writemask k1.

Handling of special-case input values are listed in Table 5-31 and Table 5-33.

Table 5-33. Additional VSCALEFPS/SS Special Cases

Special Case	Returned value	Faults
|result| < 2^{-149}	±0 or ±Min-Denormal (Src1 sign)	Underflow
|result| ≥ 2^{128}	±INF (Src1 sign) or ±Max-normal (Src1 sign)	Overflow

Operation

```
SCALE(SRC1, SRC2)
{               ; Check for denormal operands
TMP_SRC2 ← SRC2
TMP_SRC1 ← SRC1
IF (SRC2 is denormal AND MXCSR.DAZ) THEN TMP_SRC2=0
IF (SRC1 is denormal AND MXCSR.DAZ) THEN TMP_SRC1=0
/* SRC2 is a 32 bits floating-point value */
DEST[31:0] ← TMP_SRC1[31:0] * POW(2, Floor(TMP_SRC2[31:0]))
}
```

VSCALEFPS (EVEX encoded versions)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
IF (VL = 512) AND (EVEX.b = 1) AND (SRC2 *is register*)
    THEN
        SET_RM(EVEX.RC);
    ELSE
        SET_RM(MXCSR.RM);
FI;
FOR j ← 0 TO KL-1
    i ← j * 32
    IF k1[j] OR *no writemask* THEN
            IF (EVEX.b = 1) AND (SRC2 *is memory*)
                THEN DEST[i+31:i] ← SCALE(SRC1[i+31:i], SRC2[31:0]);
                ELSE DEST[i+31:i] ← SCALE(SRC1[i+31:i], SRC2[i+31:i]);
            FI;
        ELSE
            IF *merging-masking*                ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE                            ; zeroing-masking
                    DEST[i+31:i] ← 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0;
```

Intel C/C++ Compiler Intrinsic Equivalent

VSCALEFPS __m512 _mm512_scalef_round_ps(__m512 a, __m512 b, int);
VSCALEFPS __m512 _mm512_mask_scalef_round_ps(__m512 s, __mmask16 k, __m512 a, __m512 b, int);
VSCALEFPS __m512 _mm512_maskz_scalef_round_ps(__mmask16 k, __m512 a, __m512 b, int);
VSCALEFPS __m256 _mm256_scalef_round_ps(__m256 a, __m256 b, int);
VSCALEFPS __m256 _mm256_mask_scalef_round_ps(__m256 s, __mmask8 k, __m256 a, __m256 b, int);
VSCALEFPS __m256 _mm256_maskz_scalef_round_ps(__mmask8 k, __m256 a, __m256 b, int);
VSCALEFPS __m128 _mm_scalef_round_ps(__m128 a, __m128 b, int);
VSCALEFPS __m128 _mm_mask_scalef_round_ps(__m128 s, __mmask8 k, __m128 a, __m128 b, int);
VSCALEFPS __m128 _mm_maskz_scalef_round_ps(__mmask8 k, __m128 a, __m128 b, int);

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal (for Src1).
Denormal is not reported for Src2.

Other Exceptions

See Exceptions Type E2.

VSCALEFSS—Scale Scalar Float32 Value With Float32 Value

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.NDS.LIG.66.0F38.W0 2D /r VSCALEFSS xmm1 {k1}{z}, xmm2, xmm3/m32{er}	A	V/V	AVX512F	Scale the scalar single-precision floating-point value in xmm2 using floating-point value from xmm3/m32. Under writemask k1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Tuple1 Scalar	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA

Description

Performs a floating-point scale of the scalar single-precision floating-point value in the first source operand by multiplying it by 2 power of the float32 value in second source operand.

The equation of this operation is given by:

$xmm1 := xmm2*2^{floor(xmm3)}$.

Floor(xmm3) means maximum integer value ≤ xmm3.

If the result cannot be represented in single precision, then the proper overflow response (for positive scaling operand), or the proper underflow response (for negative scaling operand) is issued. The overflow and underflow responses are dependent on the rounding mode (for IEEE-compliant rounding), as well as on other settings in MXCSR (exception mask bits, FTZ bit), and on the SAE bit.

EVEX encoded version: The first source operand is an XMM register. The second source operand is an XMM register or a memory location. The destination operand is an XMM register conditionally updated with writemask k1.

Handling of special-case input values are listed in Table 5-31 and Table 5-33.

Operation

```
SCALE(SRC1, SRC2)
{
            ; Check for denormal operands
TMP_SRC2 ← SRC2
TMP_SRC1 ← SRC1
IF (SRC2 is denormal AND MXCSR.DAZ) THEN TMP_SRC2=0
IF (SRC1 is denormal AND MXCSR.DAZ) THEN TMP_SRC1=0
/* SRC2 is a 32 bits floating-point value */
DEST[31:0] ← TMP_SRC1[31:0] * POW(2, Floor(TMP_SRC2[31:0]))
}

VSCALEFSS (EVEX encoded version)
IF (EVEX.b= 1) and SRC2 *is a register*
    THEN
        SET_RM(EVEX.RC);
    ELSE
        SET_RM(MXCSR.RM);
FI;
IF k1[0] OR *no writemask*
    THEN DEST[31:0] ← SCALE(SRC1[31:0], SRC2[31:0])
    ELSE
        IF *merging-masking*                ; merging-masking
            THEN *DEST[31:0] remains unchanged*
            ELSE                            ; zeroing-masking
                DEST[31:0] ← 0
        FI
FI;
DEST[127:32] ← SRC1[127:32]
DEST[MAXVL-1:128] ← 0
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VSCALEFSS __m128 _mm_scalef_round_ss(__m128 a, __m128 b, int);
VSCALEFSS __m128 _mm_mask_scalef_round_ss(__m128 s, __mmask8 k, __m128 a, __m128 b, int);
VSCALEFSS __m128 _mm_maskz_scalef_round_ss(__mmask8 k, __m128 a, __m128 b, int);
```

SIMD Floating-Point Exceptions

Overflow, Underflow, Invalid, Precision, Denormal (for Src1).
Denormal is not reported for Src2.

Other Exceptions

See Exceptions Type E3.

VSCATTERDPS/VSCATTERDPD/VSCATTERQPS/VSCATTERQPD—Scatter Packed Single, Packed Double with Signed Dword and Qword Indices

Opcode/ Instruction	Op/En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.128.66.0F38.W0 A2 /vsib VSCATTERDPS vm32x {k1}, xmm1	A	V/V	AVX512VL AVX512F	Using signed dword indices, scatter single-precision floating-point values to memory using writemask k1.
EVEX.256.66.0F38.W0 A2 /vsib VSCATTERDPS vm32y {k1}, ymm1	A	V/V	AVX512VL AVX512F	Using signed dword indices, scatter single-precision floating-point values to memory using writemask k1.
EVEX.512.66.0F38.W0 A2 /vsib VSCATTERDPS vm32z {k1}, zmm1	A	V/V	AVX512F	Using signed dword indices, scatter single-precision floating-point values to memory using writemask k1.
EVEX.128.66.0F38.W1 A2 /vsib VSCATTERDPD vm32x {k1}, xmm1	A	V/V	AVX512VL AVX512F	Using signed dword indices, scatter double-precision floating-point values to memory using writemask k1.
EVEX.256.66.0F38.W1 A2 /vsib VSCATTERDPD vm32x {k1}, ymm1	A	V/V	AVX512VL AVX512F	Using signed dword indices, scatter double-precision floating-point values to memory using writemask k1.
EVEX.512.66.0F38.W1 A2 /vsib VSCATTERDPD vm32y {k1}, zmm1	A	V/V	AVX512F	Using signed dword indices, scatter double-precision floating-point values to memory using writemask k1.
EVEX.128.66.0F38.W0 A3 /vsib VSCATTERQPS vm64x {k1}, xmm1	A	V/V	AVX512VL AVX512F	Using signed qword indices, scatter single-precision floating-point values to memory using writemask k1.
EVEX.256.66.0F38.W0 A3 /vsib VSCATTERQPS vm64y {k1}, xmm1	A	V/V	AVX512VL AVX512F	Using signed qword indices, scatter single-precision floating-point values to memory using writemask k1.
EVEX.512.66.0F38.W0 A3 /vsib VSCATTERQPS vm64z {k1}, ymm1	A	V/V	AVX512F	Using signed qword indices, scatter single-precision floating-point values to memory using writemask k1.
EVEX.128.66.0F38.W1 A3 /vsib VSCATTERQPD vm64x {k1}, xmm1	A	V/V	AVX512VL AVX512F	Using signed qword indices, scatter double-precision floating-point values to memory using writemask k1.
EVEX.256.66.0F38.W1 A3 /vsib VSCATTERQPD vm64y {k1}, ymm1	A	V/V	AVX512VL AVX512F	Using signed qword indices, scatter double-precision floating-point values to memory using writemask k1.
EVEX.512.66.0F38.W1 A3 /vsib VSCATTERQPD vm64z {k1}, zmm1	A	V/V	AVX512F	Using signed qword indices, scatter double-precision floating-point values to memory using writemask k1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Tuple1 Scalar	BaseReg (R): VSIB:base, VectorReg(R): VSIB:index	ModRM:reg (r)	NA	NA

Description

Stores up to 16 elements (or 8 elements) in doubleword/quadword vector zmm1 to the memory locations pointed by base address BASE_ADDR and index vector VINDEX, with scale SCALE. The elements are specified via the VSIB (i.e., the index register is a vector register, holding packed indices). Elements will only be stored if their corresponding mask bit is one. The entire mask register will be set to zero by this instruction unless it triggers an exception.

This instruction can be suspended by an exception if at least one element is already scattered (i.e., if the exception is triggered by an element other than the rightmost one with its mask bit set). When this happens, the destination register and the mask register (k1) are partially updated. If any traps or interrupts are pending from already scattered elements, they will be delivered in lieu of the exception; in this case, EFLAG.RF is set to one so an instruction breakpoint is not re-triggered when the instruction is continued.

Note that:

- Only writes to overlapping vector indices are guaranteed to be ordered with respect to each other (from LSB to MSB of the source registers). Note that this also include partially overlapping vector indices. Writes that are not overlapped may happen in any order. Memory ordering with other instructions follows the Intel-64 memory ordering model. Note that this does not account for non-overlapping indices that map into the same physical address locations.

- If two or more destination indices completely overlap, the "earlier" write(s) may be skipped.
- Faults are delivered in a right-to-left manner. That is, if a fault is triggered by an element and delivered, all elements closer to the LSB of the destination zmm will be completed (and non-faulting). Individual elements closer to the MSB may or may not be completed. If a given element triggers multiple faults, they are delivered in the conventional order.
- Elements may be scattered in any order, but faults must be delivered in a right-to left order; thus, elements to the left of a faulting one may be gathered before the fault is delivered. A given implementation of this instruction is repeatable - given the same input values and architectural state, the same set of elements to the left of the faulting one will be gathered.
- This instruction does not perform AC checks, and so will never deliver an AC fault.
- Not valid with 16-bit effective addresses. Will deliver a #UD fault.
- If this instruction overwrites itself and then takes a fault, only a subset of elements may be completed before the fault is delivered (as described above). If the fault handler completes and attempts to re-execute this instruction, the new instruction will be executed, and the scatter will not complete.

Note that the presence of VSIB byte is enforced in this instruction. Hence, the instruction will #UD fault if ModRM.rm is different than 100b.

This instruction has special disp8*N and alignment rules. N is considered to be the size of a single vector element.

The scaled index may require more bits to represent than the address bits used by the processor (e.g., in 32-bit mode, if the scale is greater than one). In this case, the most significant bits beyond the number of address bits are ignored.

The instruction will #UD fault if the k0 mask register is specified.

Operation

BASE_ADDR stands for the memory operand base address (a GPR); may not exist
VINDEX stands for the memory operand vector of indices (a ZMM register)
SCALE stands for the memory operand scalar (1, 2, 4 or 8)
DISP is the optional 1, 2 or 4 byte displacement

VSCATTERDPS (EVEX encoded versions)

```
(KL, VL)= (4, 128), (8, 256), (16, 512)
FOR j ← 0 TO KL-1
    i ← j * 32
    IF k1[j] OR *no writemask*
        THEN MEM[BASE_ADDR +SignExtend(VINDEX[i+31:i]) * SCALE + DISP] ←
            SRC[i+31:i]
            k1[j] ← 0
    FI;
ENDFOR
k1[MAX_KL-1:KL] ← 0
```

VSCATTERDPD (EVEX encoded versions)

```
(KL, VL)= (2, 128), (4, 256), (8, 512)
FOR j ← 0 TO KL-1
    i ← j * 64
    k ← j * 32
    IF k1[j] OR *no writemask*
        THEN MEM[BASE_ADDR +SignExtend(VINDEX[k+31:k]) * SCALE + DISP] ←
            SRC[i+63:i]
            k1[j] ← 0
    FI;
ENDFOR
k1[MAX_KL-1:KL] ← 0
```

VSCATTERQPS (EVEX encoded versions)

```
(KL, VL)= (2, 128), (4, 256), (8, 512)
FOR j ← 0 TO KL-1
    i ← j * 32
    k ← j * 64
    IF k1[j] OR *no writemask*
        THEN MEM[BASE_ADDR + (VINDEX[k+63:k]) * SCALE + DISP] ←
            SRC[i+31:i]
            k1[j] ← 0
    FI;
ENDFOR
k1[MAX_KL-1:KL] ← 0
```

VSCATTERQPD (EVEX encoded versions)

```
(KL, VL)= (2, 128), (4, 256), (8, 512)
FOR j ← 0 TO KL-1
    i ← j * 64
    IF k1[j] OR *no writemask*
        THEN MEM[BASE_ADDR + (VINDEX[i+63:i]) * SCALE + DISP] ←
            SRC[i+63:i]
            k1[j] ← 0
    FI;
ENDFOR
k1[MAX_KL-1:KL] ← 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VSCATTERDPD void _mm512_i32scatter_pd(void * base, __m256i vdx, __m512d a, int scale);
VSCATTERDPD void _mm512_mask_i32scatter_pd(void * base, __mmask8 k, __m256i vdx, __m512d a, int scale);
VSCATTERDPS void _mm512_i32scatter_ps(void * base, __m512i vdx, __m512 a, int scale);
VSCATTERDPS void _mm512_mask_i32scatter_ps(void * base, __mmask16 k, __m512i vdx, __m512 a, int scale);
VSCATTERQPD void _mm512_i64scatter_pd(void * base, __m512i vdx, __m512d a, int scale);
VSCATTERQPD void _mm512_mask_i64scatter_pd(void * base, __mmask8 k, __m512i vdx, __m512d a, int scale);
VSCATTERQPS void _mm512_i64scatter_ps(void * base, __m512i vdx, __m256 a, int scale);
VSCATTERQPS void _mm512_mask_i64scatter_ps(void * base, __mmask8 k, __m512i vdx, __m256 a, int scale);
VSCATTERDPD void _mm256_i32scatter_pd(void * base, __m128i vdx, __m256d a, int scale);
VSCATTERDPD void _mm256_mask_i32scatter_pd(void * base, __mmask8 k, __m128i vdx, __m256d a, int scale);
VSCATTERDPS void _mm256_i32scatter_ps(void * base, __m256i vdx, __m256 a, int scale);
VSCATTERDPS void _mm256_mask_i32scatter_ps(void * base, __mmask8 k, __m256i vdx, __m256 a, int scale);
VSCATTERQPD void _mm256_i64scatter_pd(void * base, __m256i vdx, __m256d a, int scale);
VSCATTERQPD void _mm256_mask_i64scatter_pd(void * base, __mmask8 k, __m256i vdx, __m256d a, int scale);
VSCATTERQPS void _mm256_i64scatter_ps(void * base, __m256i vdx, __m128 a, int scale);
VSCATTERQPS void _mm256_mask_i64scatter_ps(void * base, __mmask8 k, __m256i vdx, __m128 a, int scale);
VSCATTERDPD void _mm_i32scatter_pd(void * base, __m128i vdx, __m128d a, int scale);
VSCATTERDPD void _mm_mask_i32scatter_pd(void * base, __mmask8 k, __m128i vdx, __m128d a, int scale);
VSCATTERDPS void _mm_i32scatter_ps(void * base, __m128i vdx, __m128 a, int scale);
VSCATTERDPS void _mm_mask_i32scatter_ps(void * base, __mmask8 k, __m128i vdx, __m128 a, int scale);
VSCATTERQPD void _mm_i64scatter_pd(void * base, __m128i vdx, __m128d a, int scale);
VSCATTERQPD void _mm_mask_i64scatter_pd(void * base, __mmask8 k, __m128i vdx, __m128d a, int scale);
VSCATTERQPS void _mm_i64scatter_ps(void * base, __m128i vdx, __m128 a, int scale);
VSCATTERQPS void _mm_mask_i64scatter_ps(void * base, __mmask8 k, __m128i vdx, __m128 a, int scale);

SIMD Floating-Point Exceptions

Invalid, Overflow, Underflow, Precision, Denormal

Other Exceptions

See Exceptions Type E12.

VSCATTERPF0DPS/VSCATTERPF0QPS/VSCATTERPF0DPD/VSCATTERPF0QPD—Sparse Prefetch Packed SP/DP Data Values with Signed Dword, Signed Qword Indices Using T0 Hint with Intent to Write

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.512.66.0F38.W0 C6 /5 /vsib VSCATTERPF0DPS vm32z {k1}	A	V/V	AVX512PF	Using signed dword indices, prefetch sparse byte memory locations containing single-precision data using writemask k1 and T0 hint with intent to write.
EVEX.512.66.0F38.W0 C7 /5 /vsib VSCATTERPF0QPS vm64z {k1}	A	V/V	AVX512PF	Using signed qword indices, prefetch sparse byte memory locations containing single-precision data using writemask k1 and T0 hint with intent to write.
EVEX.512.66.0F38.W1 C6 /5 /vsib VSCATTERPF0DPD vm32y {k1}	A	V/V	AVX512PF	Using signed dword indices, prefetch sparse byte memory locations containing double-precision data using writemask k1 and T0 hint with intent to write.
EVEX.512.66.0F38.W1 C7 /5 /vsib VSCATTERPF0QPD vm64z {k1}	A	V/V	AVX512PF	Using signed qword indices, prefetch sparse byte memory locations containing double-precision data using writemask k1 and T0 hint with intent to write.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Tuple1 Scalar	BaseReg (R): VSIB:base, VectorReg(R): VSIB:index	NA	NA	NA

Description

The instruction conditionally prefetches up to sixteen 32-bit or eight 64-bit integer byte data elements. The elements are specified via the VSIB (i.e., the index register is an zmm, holding packed indices). Elements will only be prefetched if their corresponding mask bit is one.

cache lines will be brought into exclusive state (RFO) specified by a locality hint (T0):

• T0 (temporal data)—prefetch data into the first level cache.

[PS data] For dword indices, the instruction will prefetch sixteen memory locations. For qword indices, the instruction will prefetch eight values.

[PD data] For dword and qword indices, the instruction will prefetch eight memory locations.

Note that:

(1) The prefetches may happen in any order (or not at all). The instruction is a hint.

(2) The mask is left unchanged.

(3) Not valid with 16-bit effective addresses. Will deliver a #UD fault.

(4) No FP nor memory faults may be produced by this instruction.

(5) Prefetches do not handle cache line splits

(6) A #UD is signaled if the memory operand is encoded without the SIB byte.

Operation

BASE_ADDR stands for the memory operand base address (a GPR); may not exist
VINDEX stands for the memory operand vector of indices (a vector register)
SCALE stands for the memory operand scalar (1, 2, 4 or 8)
DISP is the optional 1, 2 or 4 byte displacement
PREFETCH(mem, Level, State) Prefetches a byte memory location pointed by 'mem' into the cache level specified by 'Level'; a request for exclusive/ownership is done if 'State' is 1. Note that the memory location ignore cache line splits. This operation is considered a hint for the processor and may be skipped depending on implementation.

VSCATTERPF0DPS (EVEX encoded version)

```
(KL, VL) = (16, 512)
FOR j ← 0 TO KL-1
    i ← j * 32
    IF k1[j]
        Prefetch( [BASE_ADDR + SignExtend(VINDEX[i+31:i]) * SCALE + DISP], Level=0, RFO = 1)
    FI;
ENDFOR
```

VSCATTERPF0DPD (EVEX encoded version)

```
(KL, VL) = (8, 512)
FOR j ← 0 TO KL-1
    i ← j * 64
    k ← j * 32
    IF k1[j]
        Prefetch( [BASE_ADDR + SignExtend(VINDEX[k+31:k]) * SCALE + DISP], Level=0, RFO = 1)
    FI;
ENDFOR
```

VSCATTERPF0QPS (EVEX encoded version)

```
(KL, VL) = (8, 256)
FOR j ← 0 TO KL-1
    i ← j * 64
    IF k1[j]
        Prefetch( [BASE_ADDR + SignExtend(VINDEX[i+63:i]) * SCALE + DISP], Level=0, RFO = 1)
    FI;
ENDFOR
```

VSCATTERPF0QPD (EVEX encoded version)

```
(KL, VL) = (8, 512)
FOR j ← 0 TO KL-1
    i ← j * 64
    k ← j * 64
    IF k1[j]
        Prefetch( [BASE_ADDR + SignExtend(VINDEX[k+63:k]) * SCALE + DISP], Level=0, RFO = 1)
    FI;
ENDFOR
```

Intel C/C++ Compiler Intrinsic Equivalent

VSCATTERPF0DPD void _mm512_prefetch_i32scatter_pd(void *base, __m256i vdx, int scale, int hint);
VSCATTERPF0DPD void _mm512_mask_prefetch_i32scatter_pd(void *base, __mmask8 m, __m256i vdx, int scale, int hint);
VSCATTERPF0DPS void _mm512_prefetch_i32scatter_ps(void *base, __m512i vdx, int scale, int hint);
VSCATTERPF0DPS void _mm512_mask_prefetch_i32scatter_ps(void *base, __mmask16 m, __m512i vdx, int scale, int hint);
VSCATTERPF0QPD void _mm512_prefetch_i64scatter_pd(void * base, __m512i vdx, int scale, int hint);
VSCATTERPF0QPD void _mm512_mask_prefetch_i64scatter_pd(void * base, __mmask8 m, __m512i vdx, int scale, int hint);
VSCATTERPF0QPS void _mm512_prefetch_i64scatter_ps(void * base, __m512i vdx, int scale, int hint);
VSCATTERPF0QPS void _mm512_mask_prefetch_i64scatter_ps(void * base, __mmask8 m, __m512i vdx, int scale, int hint);

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type E12NP.

VSCATTERPF1DPS/VSCATTERPF1QPS/VSCATTERPF1DPD/VSCATTERPF1QPD—Sparse Prefetch Packed SP/DP Data Values with Signed Dword, Signed Qword Indices Using T1 Hint with Intent to Write

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.512.66.0F38.W0 C6 /6 /vsib VSCATTERPF1DPS vm32z {k1}	A	V/V	AVX512PF	Using signed dword indices, prefetch sparse byte memory locations containing single-precision data using writemask k1 and T1 hint with intent to write.
EVEX.512.66.0F38.W0 C7 /6 /vsib VSCATTERPF1QPS vm64z {k1}	A	V/V	AVX512PF	Using signed qword indices, prefetch sparse byte memory locations containing single-precision data using writemask k1 and T1 hint with intent to write.
EVEX.512.66.0F38.W1 C6 /6 /vsib VSCATTERPF1DPD vm32y {k1}	A	V/V	AVX512PF	Using signed dword indices, prefetch sparse byte memory locations containing double-precision data using writemask k1 and T1 hint with intent to write.
EVEX.512.66.0F38.W1 C7 /6 /vsib VSCATTERPF1QPD vm64z {k1}	A	V/V	AVX512PF	Using signed qword indices, prefetch sparse byte memory locations containing double-precision data using writemask k1 and T1 hint with intent to write.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Tuple1 Scalar	BaseReg (R): VSIB:base, VectorReg(R): VSIB:index	NA	NA	NA

Description

The instruction conditionally prefetches up to sixteen 32-bit or eight 64-bit integer byte data elements. The elements are specified via the VSIB (i.e., the index register is an zmm, holding packed indices). Elements will only be prefetched if their corresponding mask bit is one.

cache lines will be brought into exclusive state (RFO) specified by a locality hint (T1):

• T1 (temporal data)—prefetch data into the second level cache.

[PS data] For dword indices, the instruction will prefetch sixteen memory locations. For qword indices, the instruction will prefetch eight values.

[PD data] For dword and qword indices, the instruction will prefetch eight memory locations.

Note that:

(1) The prefetches may happen in any order (or not at all). The instruction is a hint.

(2) The mask is left unchanged.

(3) Not valid with 16-bit effective addresses. Will deliver a #UD fault.

(4) No FP nor memory faults may be produced by this instruction.

(5) Prefetches do not handle cache line splits

(6) A #UD is signaled if the memory operand is encoded without the SIB byte.

Operation

BASE_ADDR stands for the memory operand base address (a GPR); may not exist
VINDEX stands for the memory operand vector of indices (a vector register)
SCALE stands for the memory operand scalar (1, 2, 4 or 8)
DISP is the optional 1, 2 or 4 byte displacement
PREFETCH(mem, Level, State) Prefetches a byte memory location pointed by 'mem' into the cache level specified by 'Level'; a request for exclusive/ownership is done if 'State' is 1. Note that the memory location ignore cache line splits. This operation is considered a hint for the processor and may be skipped depending on implementation.

VSCATTERPF1DPS (EVEX encoded version)

```
(KL, VL) = (16, 512)
FOR j ← 0 TO KL-1
    i ← j * 32
    IF k1[j]
        Prefetch( [BASE_ADDR + SignExtend(VINDEX[i+31:i]) * SCALE + DISP], Level=1, RFO = 1)
    FI;
ENDFOR
```

VSCATTERPF1DPD (EVEX encoded version)

```
(KL, VL) = (8, 512)
FOR j ← 0 TO KL-1
    i ← j * 64
    k ← j * 32
    IF k1[j]
        Prefetch( [BASE_ADDR + SignExtend(VINDEX[k+31:k]) * SCALE + DISP], Level=1, RFO = 1)
    FI;
ENDFOR
```

VSCATTERPF1QPS (EVEX encoded version)

```
(KL, VL) = (8, 512)
FOR j ← 0 TO KL-1
    i ← j * 64
    IF k1[j]
        Prefetch( [BASE_ADDR + SignExtend(VINDEX[i+63:i]) * SCALE + DISP], Level=1, RFO = 1)
    FI;
ENDFOR
```

VSCATTERPF1QPD (EVEX encoded version)

```
(KL, VL) = (8, 512)
FOR j ← 0 TO KL-1
    i ← j * 64
    k ← j * 64
    IF k1[j]
        Prefetch( [BASE_ADDR + SignExtend(VINDEX[k+63:k]) * SCALE + DISP], Level=1, RFO = 1)
    FI;
ENDFOR
```

Intel C/C++ Compiler Intrinsic Equivalent

VSCATTERPF1DPD void _mm512_prefetch_i32scatter_pd(void *base, __m256i vdx, int scale, int hint);
VSCATTERPF1DPD void _mm512_mask_prefetch_i32scatter_pd(void *base, __mmask8 m, __m256i vdx, int scale, int hint);
VSCATTERPF1DPS void _mm512_prefetch_i32scatter_ps(void *base, __m512i vdx, int scale, int hint);
VSCATTERPF1DPS void _mm512_mask_prefetch_i32scatter_ps(void *base, __mmask16 m, __m512i vdx, int scale, int hint);
VSCATTERPF1QPD void _mm512_prefetch_i64scatter_pd(void * base, __m512i vdx, int scale, int hint);
VSCATTERPF1QPD void _mm512_mask_prefetch_i64scatter_pd(void * base, __mmask8 m, __m512i vdx, int scale, int hint);
VSCATTERPF1QPS void _mm512_prefetch_i64scatter_ps(void *base, __m512i vdx, int scale, int hint);
VSCATTERPF1QPS void _mm512_mask_prefetch_i64scatter_ps(void *base, __mmask8 m, __m512i vdx, int scale, int hint);

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type E12NP.

VSHUFF32x4/VSHUFF64x2/VSHUFI32x4/VSHUFI64x2—Shuffle Packed Values at 128-bit Granularity

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
EVEX.NDS.256.66.0F3A.W0 23 /r ib VSHUFF32X4 ymm1{k1}{z}, ymm2, ymm3/m256/m32bcst, imm8	A	V/V	AVX512VL AVX512F	Shuffle 128-bit packed single-precision floating-point values selected by imm8 from ymm2 and ymm3/m256/m32bcst and place results in ymm1 subject to writemask k1.
EVEX.NDS.512.66.0F3A.W0 23 /r ib VSHUFF32x4 zmm1{k1}{z}, zmm2, zmm3/m512/m32bcst, imm8	A	V/V	AVX512F	Shuffle 128-bit packed single-precision floating-point values selected by imm8 from zmm2 and zmm3/m512/m32bcst and place results in zmm1 subject to writemask k1.
EVEX.NDS.256.66.0F3A.W1 23 /r ib VSHUFF64X2 ymm1{k1}{z}, ymm2, ymm3/m256/m64bcst, imm8	A	V/V	AVX512VL AVX512F	Shuffle 128-bit packed double-precision floating-point values selected by imm8 from ymm2 and ymm3/m256/m64bcst and place results in ymm1 subject to writemask k1.
EVEX.NDS.512.66.0F3A.W1 23 /r ib VSHUFF64x2 zmm1{k1}{z}, zmm2, zmm3/m512/m64bcst, imm8	A	V/V	AVX512F	Shuffle 128-bit packed double-precision floating-point values selected by imm8 from zmm2 and zmm3/m512/m64bcst and place results in zmm1 subject to writemask k1.
EVEX.NDS.256.66.0F3A.W0 43 /r ib VSHUFI32X4 ymm1{k1}{z}, ymm2, ymm3/m256/m32bcst, imm8	A	V/V	AVX512VL AVX512F	Shuffle 128-bit packed double-word values selected by imm8 from ymm2 and ymm3/m256/m32bcst and place results in ymm1 subject to writemask k1.
EVEX.NDS.512.66.0F3A.W0 43 /r ib VSHUFI32x4 zmm1{k1}{z}, zmm2, zmm3/m512/m32bcst, imm8	A	V/V	AVX512F	Shuffle 128-bit packed double-word values selected by imm8 from zmm2 and zmm3/m512/m32bcst and place results in zmm1 subject to writemask k1.
EVEX.NDS.256.66.0F3A.W1 43 /r ib VSHUFI64X2 ymm1{k1}{z}, ymm2, ymm3/m256/m64bcst, imm8	A	V/V	AVX512VL AVX512F	Shuffle 128-bit packed quad-word values selected by imm8 from ymm2 and ymm3/m256/m64bcst and place results in ymm1 subject to writemask k1.
EVEX.NDS.512.66.0F3A.W1 43 /r ib VSHUFI64x2 zmm1{k1}{z}, zmm2, zmm3/m512/m64bcst, imm8	A	V/V	AVX512F	Shuffle 128-bit packed quad-word values selected by imm8 from zmm2 and zmm3/m512/m64bcst and place results in zmm1 subject to writemask k1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	Full Vector	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA

Description

256-bit Version: Moves one of the two 128-bit packed single-precision floating-point values from the first source operand (second operand) into the low 128-bit of the destination operand (first operand); moves one of the two packed 128-bit floating-point values from the second source operand (third operand) into the high 128-bit of the destination operand. The selector operand (third operand) determines which values are moved to the destination operand.

512-bit Version: Moves two of the four 128-bit packed single-precision floating-point values from the first source operand (second operand) into the low 256-bit of each double qword of the destination operand (first operand); moves two of the four packed 128-bit floating-point values from the second source operand (third operand) into the high 256-bit of the destination operand. The selector operand (third operand) determines which values are moved to the destination operand.

The first source operand is a vector register. The second source operand can be a ZMM register, a 512-bit memory location or a 512-bit vector broadcasted from a 32/64-bit memory location. The destination operand is a vector register.

The writemask updates the destination operand with the granularity of 32/64-bit data elements.

Operation

```
Select2(SRC, control) {
CASE (control[0]) OF
    0:    TMP ← SRC[127:0];
    1:    TMP ← SRC[255:128];
ESAC;
RETURN TMP
}

Select4(SRC, control) {
CASE (control[1:0]) OF
    0:    TMP ← SRC[127:0];
    1:    TMP ← SRC[255:128];
    2:    TMP ← SRC[383:256];
    3:    TMP ← SRC[511:384];
ESAC;
RETURN TMP
}
```

VSHUFF32x4 (EVEX versions)

```
(KL, VL) = (8, 256), (16, 512)
FOR j ← 0 TO KL-1
    i ← j * 32
    IF (EVEX.b = 1) AND (SRC2 *is memory*)
        THEN TMP_SRC2[i+31:i] ← SRC2[31:0]
        ELSE TMP_SRC2[i+31:i] ← SRC2[i+31:i]
    FI;
ENDFOR;
IF VL = 256
    TMP_DEST[127:0] ← Select2(SRC1[255:0], imm8[0]);
    TMP_DEST[255:128] ← Select2(SRC2[255:0], imm8[1]);
FI;
IF VL = 512
    TMP_DEST[127:0] ← Select4(SRC1[511:0], imm8[1:0]);
    TMP_DEST[255:128] ← Select4(SRC1[511:0], imm8[3:2]);
    TMP_DEST[383:256] ← Select4(TMP_SRC2[511:0], imm8[5:4]);
    TMP_DEST[511:384] ← Select4(TMP_SRC2[511:0], imm8[7:6]);
FI;
FOR j ← 0 TO KL-1
    i ← j * 32
    IF k1[j] OR *no writemask*
        THEN DEST[i+31:i] ← TMP_DEST[i+31:i]
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE *zeroing-masking*                ; zeroing-masking
                    THEN DEST[i+31:i] ← 0
            FI;
    FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0
```

VSHUFF64x2 (EVEX 512-bit version)

```
(KL, VL) = (4, 256), (8, 512)
FOR j ← 0 TO KL-1
    i ← j * 64
    IF (EVEX.b = 1) AND (SRC2 *is memory*)
        THEN TMP_SRC2[i+63:i] ← SRC2[63:0]
        ELSE TMP_SRC2[i+63:i] ← SRC2[i+63:i]
    FI;
ENDFOR;
IF VL = 256
    TMP_DEST[127:0] ← Select2(SRC1[255:0], imm8[0]);
    TMP_DEST[255:128] ← Select2(SRC2[255:0], imm8[1]);
FI;
IF VL = 512
    TMP_DEST[127:0] ← Select4(SRC1[511:0], imm8[1:0]);
    TMP_DEST[255:128] ← Select4(SRC1[511:0], imm8[3:2]);
    TMP_DEST[383:256] ← Select4(TMP_SRC2[511:0], imm8[5:4]);
    TMP_DEST[511:384] ← Select4(TMP_SRC2[511:0], imm8[7:6]);
FI;
FOR j ← 0 TO KL-1
    i ← j * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+63:i] ← TMP_DEST[i+63:i]
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE *zeroing-masking*              ; zeroing-masking
                    THEN DEST[i+63:i] ←    0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0
```

VSHUFI32x4 (EVEX 512-bit version)

```
(KL, VL) = (8, 256), (16, 512)
FOR j ← 0 TO KL-1
    i ← j * 32
    IF (EVEX.b = 1) AND (SRC2 *is memory*)
        THEN TMP_SRC2[i+31:i] ← SRC2[31:0]
        ELSE TMP_SRC2[i+31:i] ← SRC2[i+31:i]
    FI;
ENDFOR;
IF VL = 256
    TMP_DEST[127:0] ← Select2(SRC1[255:0], imm8[0]);
    TMP_DEST[255:128] ← Select2(SRC2[255:0], imm8[1]);
FI;
IF VL = 512
    TMP_DEST[127:0] ← Select4(SRC1[511:0], imm8[1:0]);
    TMP_DEST[255:128] ← Select4(SRC1[511:0], imm8[3:2]);
    TMP_DEST[383:256] ← Select4(TMP_SRC2[511:0], imm8[5:4]);
    TMP_DEST[511:384] ← Select4(TMP_SRC2[511:0], imm8[7:6]);
FI;
FOR j ← 0 TO KL-1
    i ← j * 32
```

```
    IF k1[j] OR *no writemask*
        THEN DEST[i+31:i] ← TMP_DEST[i+31:i]
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE *zeroing-masking*              ; zeroing-masking
                    THEN DEST[i+31:i] ← 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0
```

VSHUFI64x2 (EVEX 512-bit version)

```
(KL, VL) = (4, 256), (8, 512)
FOR j ← 0 TO KL-1
    i ← j * 64
    IF (EVEX.b = 1) AND (SRC2 *is memory*)
        THEN TMP_SRC2[i+63:i] ← SRC2[63:0]
        ELSE TMP_SRC2[i+63:i] ← SRC2[i+63:i]
    FI;
ENDFOR;
IF VL = 256
    TMP_DEST[127:0] ← Select2(SRC1[255:0], imm8[0]);
    TMP_DEST[255:128] ← Select2(SRC2[255:0], imm8[1]);
FI;
IF VL = 512
    TMP_DEST[127:0] ← Select4(SRC1[511:0], imm8[1:0]);
    TMP_DEST[255:128] ← Select4(SRC1[511:0], imm8[3:2]);
    TMP_DEST[383:256] ← Select4(TMP_SRC2[511:0], imm8[5:4]);
    TMP_DEST[511:384] ← Select4(TMP_SRC2[511:0], imm8[7:6]);
FI;
FOR j ← 0 TO KL-1
    i ← j * 64
    IF k1[j] OR *no writemask*
        THEN DEST[i+63:i] ← TMP_DEST[i+63:i]
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE *zeroing-masking*              ; zeroing-masking
                    THEN DEST[i+63:i] ←    0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VSHUFI32x4 __m512i _mm512_shuffle_i32x4(__m512i a, __m512i b, int imm);
VSHUFI32x4 __m512i _mm512_mask_shuffle_i32x4(__m512i s, __mmask16 k, __m512i a, __m512i b, int imm);
VSHUFI32x4 __m512i _mm512_maskz_shuffle_i32x4(__mmask16 k, __m512i a, __m512i b, int imm);
VSHUFI32x4 __m256i _mm256_shuffle_i32x4(__m256i a, __m256i b, int imm);
VSHUFI32x4 __m256i _mm256_mask_shuffle_i32x4(__m256i s, __mmask8 k, __m256i a, __m256i b, int imm);
VSHUFI32x4 __m256i _mm256_maskz_shuffle_i32x4(__mmask8 k, __m256i a, __m256i b, int imm);
VSHUFF32x4 __m512 _mm512_shuffle_f32x4(__m512 a, __m512 b, int imm);
VSHUFF32x4 __m512 _mm512_mask_shuffle_f32x4(__m512 s, __mmask16 k, __m512 a, __m512 b, int imm);
VSHUFF32x4 __m512 _mm512_maskz_shuffle_f32x4(__mmask16 k, __m512 a, __m512 b, int imm);
VSHUFI64x2 __m512i _mm512_shuffle_i64x2(__m512i a, __m512i b, int imm);
VSHUFI64x2 __m512i _mm512_mask_shuffle_i64x2(__m512i s, __mmask8 k, __m512i b, __m512i b, int imm);
VSHUFI64x2 __m512i _mm512_maskz_shuffle_i64x2(__mmask8 k, __m512i a, __m512i b, int imm);
VSHUFF64x2 __m512d _mm512_shuffle_f64x2(__m512d a, __m512d b, int imm);
VSHUFF64x2 __m512d _mm512_mask_shuffle_f64x2(__m512d s, __mmask8 k, __m512d a, __m512d b, int imm);
VSHUFF64x2 __m512d _mm512_maskz_shuffle_f64x2(__mmask8 k, __m512d a, __m512d b, int imm);

SIMD Floating-Point Exceptions

None

Other Exceptions

See Exceptions Type E4NF.

#UD	If EVEX.L'L = 0 for VSHUFF32x4/VSHUFF64x2.

VTESTPD/VTESTPS—Packed Bit Test

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
VEX.128.66.0F38.W0 0E /r VTESTPS *xmm1, xmm2/m128*	RM	V/V	AVX	Set ZF and CF depending on sign bit AND and ANDN of packed single-precision floating-point sources.
VEX.256.66.0F38.W0 0E /r VTESTPS *ymm1, ymm2/m256*	RM	V/V	AVX	Set ZF and CF depending on sign bit AND and ANDN of packed single-precision floating-point sources.
VEX.128.66.0F38.W0 0F /r VTESTPD *xmm1, xmm2/m128*	RM	V/V	AVX	Set ZF and CF depending on sign bit AND and ANDN of packed double-precision floating-point sources.
VEX.256.66.0F38.W0 0F /r VTESTPD *ymm1, ymm2/m256*	RM	V/V	AVX	Set ZF and CF depending on sign bit AND and ANDN of packed double-precision floating-point sources.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
RM	ModRM:reg (r)	ModRM:r/m (r)	NA	NA

Description

VTESTPS performs a bitwise comparison of all the sign bits of the packed single-precision elements in the first source operation and corresponding sign bits in the second source operand. If the AND of the source sign bits with the dest sign bits produces all zeros, the ZF is set else the ZF is clear. If the AND of the source sign bits with the inverted dest sign bits produces all zeros the CF is set else the CF is clear. An attempt to execute VTESTPS with VEX.W=1 will cause #UD.

VTESTPD performs a bitwise comparison of all the sign bits of the double-precision elements in the first source operation and corresponding sign bits in the second source operand. If the AND of the source sign bits with the dest sign bits produces all zeros, the ZF is set else the ZF is clear. If the AND the source sign bits with the inverted dest sign bits produces all zeros the CF is set else the CF is clear. An attempt to execute VTESTPS with VEX.W=1 will cause #UD.

The first source register is specified by the ModR/M *reg* field.

128-bit version: The first source register is an XMM register. The second source register can be an XMM register or a 128-bit memory location. The destination register is not modified.

VEX.256 encoded version: The first source register is a YMM register. The second source register can be a YMM register or a 256-bit memory location. The destination register is not modified.

Note: In VEX-encoded versions, VEX.vvvv is reserved and must be 1111b, otherwise instructions will #UD.

Operation

VTESTPS (128-bit version)

```
TEMP[127:0] ← SRC[127:0] AND DEST[127:0]
IF (TEMP[31] = TEMP[63] = TEMP[95] = TEMP[127] = 0)
    THEN ZF ←1;
    ELSE ZF ← 0;

TEMP[127:0] ← SRC[127:0] AND NOT DEST[127:0]
IF (TEMP[31] = TEMP[63] = TEMP[95] = TEMP[127] = 0)
    THEN CF ←1;
    ELSE CF ← 0;
DEST (unmodified)
AF ← OF ← PF ← SF ← 0;
```

VTESTPS (VEX.256 encoded version)

```
TEMP[255:0] ← SRC[255:0] AND DEST[255:0]
IF (TEMP[31] = TEMP[63] = TEMP[95] = TEMP[127]= TEMP[160] =TEMP[191] = TEMP[224] = TEMP[255] = 0)
    THEN ZF ←1;
    ELSE ZF ← 0;

TEMP[255:0] ← SRC[255:0] AND NOT DEST[255:0]
IF (TEMP[31] = TEMP[63] = TEMP[95] = TEMP[127]= TEMP[160] =TEMP[191] = TEMP[224] = TEMP[255] = 0)
    THEN CF ←1;
    ELSE CF ← 0;
DEST (unmodified)
AF ← OF ← PF ← SF ← 0;
```

VTESTPD (128-bit version)

```
TEMP[127:0] ← SRC[127:0] AND DEST[127:0]
IF ( TEMP[63] = TEMP[127] = 0)
    THEN ZF ←1;
    ELSE ZF ← 0;

TEMP[127:0] ← SRC[127:0] AND NOT DEST[127:0]
IF ( TEMP[63] = TEMP[127] = 0)
    THEN CF ←1;
    ELSE CF ← 0;
DEST (unmodified)
AF ← OF ← PF ← SF ← 0;
```

VTESTPD (VEX.256 encoded version)

```
TEMP[255:0] ← SRC[255:0] AND DEST[255:0]
IF (TEMP[63] = TEMP[127] = TEMP[191] = TEMP[255] = 0)
    THEN ZF ←1;
    ELSE ZF ← 0;

TEMP[255:0] ← SRC[255:0] AND NOT DEST[255:0]
IF (TEMP[63] = TEMP[127] = TEMP[191] = TEMP[255] = 0)
    THEN CF ←1;
    ELSE CF ← 0;
DEST (unmodified)
AF ← OF ← PF ← SF ← 0;
```

Intel C/C++ Compiler Intrinsic Equivalent

VTESTPS

int _mm256_testz_ps (__m256 s1, __m256 s2);

int _mm256_testc_ps (__m256 s1, __m256 s2);

int _mm256_testnzc_ps (__m256 s1, __m128 s2);

int _mm_testz_ps (__m128 s1, __m128 s2);

int _mm_testc_ps (__m128 s1, __m128 s2);

int _mm_testnzc_ps (__m128 s1, __m128 s2);

VTESTPD

int _mm256_testz_pd (__m256d s1, __m256d s2);

int _mm256_testc_pd (__m256d s1, __m256d s2);

int _mm256_testnzc_pd (__m256d s1, __m256d s2);

int _mm_testz_pd (__m128d s1, __m128d s2);

int _mm_testc_pd (__m128d s1, __m128d s2);

int _mm_testnzc_pd (__m128d s1, __m128d s2);

Flags Affected

The 0F, AF, PF, SF flags are cleared and the ZF, CF flags are set according to the operation.

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Exceptions Type 4; additionally

#UD	If VEX.vvvv ≠ 1111B.
	If VEX.W = 1 for VTESTPS or VTESTPD.

VZEROALL—Zero All YMM Registers

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
VEX.256.0F.WIG 77 VZEROALL	ZO	V/V	AVX	Zero all YMM registers.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
ZO	NA	NA	NA	NA

Description

The instruction zeros contents of all XMM or YMM registers.

Note: VEX.vvvv is reserved and must be 1111b, otherwise instructions will #UD. In Compatibility and legacy 32-bit mode only the lower 8 registers are modified.

Operation

simd_reg_file[][] is a two dimensional array representing the SIMD register file containing all the overlapping xmm, ymm and zmm registers present in that implementation. The major dimension is the register number: 0 for xmm0, ymm0 and zmm0; 1 for xmm1, ymm1, and zmm1; etc. The minor dimension size is the width of the implemented SIMD state measured in bits. On a machine supporting Intel AVX-512, the width is 512. On a machine supporting Intel AVX but not Intel AVX-512, the width is "MAXVL".

VZEROALL (VEX.256 encoded version)

```
IF (64-bit mode)
    limit ←15
ELSE
    limit ← 7
FOR i in 0 .. limit:
    simd_reg_file[i][MAXVL-1:0] ← 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VZEROALL: _mm256_zeroall()

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Exceptions Type 8.

VZEROUPPER—Zero Upper Bits of YMM Registers

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
VEX.128.0F.WIG 77 VZEROUPPER	ZO	V/V	AVX	Zero upper 128 bits of all YMM registers.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
ZO	NA	NA	NA	NA

Description

The instruction zeros the bits in position 128 and higher of all YMM registers. The lower 128-bits of the registers (the corresponding XMM registers) are unmodified.

This instruction is recommended when transitioning between AVX and legacy SSE code - it will eliminate performance penalties caused by false dependencies.

Note: VEX.vvvv is reserved and must be 1111b otherwise instructions will #UD. In Compatibility and legacy 32-bit mode only the lower 8 registers are modified.

Operation

simd_reg_file[][] is a two dimensional array representing the SIMD register file containing all the overlapping xmm, ymm and zmm registers present in that implementation. The major dimension is the register number: 0 for xmm0, ymm0 and zmm0; 1 for xmm1, ymm1, and zmm1; etc. The minor dimension size is the width of the implemented SIMD state measured in bits. On a machine supporting Intel AVX-512, the width is 512. On a machine supporting Intel AVX but not Intel AVX-512, the width is "MAXVL".

VZEROUPPER

```
IF (64-bit mode)
    limit ←15
ELSE
    limit ← 7
FOR i in 0 .. limit:
    simd_reg_file[i][MAXVL-1:128] ← 0
```

Intel C/C++ Compiler Intrinsic Equivalent

VZEROUPPER: _mm256_zeroupper()

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Exceptions Type 8.

WAIT/FWAIT—Wait

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
9B	WAIT	ZO	Valid	Valid	Check pending unmasked floating-point exceptions.
9B	FWAIT	ZO	Valid	Valid	Check pending unmasked floating-point exceptions.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
ZO	NA	NA	NA	NA

Description

Causes the processor to check for and handle pending, unmasked, floating-point exceptions before proceeding. (FWAIT is an alternate mnemonic for WAIT.)

This instruction is useful for synchronizing exceptions in critical sections of code. Coding a WAIT instruction after a floating-point instruction ensures that any unmasked floating-point exceptions the instruction may raise are handled before the processor can modify the instruction's results. See the section titled "Floating-Point Exception Synchronization" in Chapter 8 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*, for more information on using the WAIT/FWAIT instruction.

This instruction's operation is the same in non-64-bit modes and 64-bit mode.

Operation

```
CheckForPendingUnmaskedFloatingPointExceptions;
```

FPU Flags Affected

The C0, C1, C2, and C3 flags are undefined.

Floating-Point Exceptions

None.

Protected Mode Exceptions

#NM	If CR0.MP[bit 1] = 1 and CR0.TS[bit 3] = 1.
#UD	If the LOCK prefix is used.

Real-Address Mode Exceptions

Same exceptions as in protected mode.

Virtual-8086 Mode Exceptions

Same exceptions as in protected mode.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

Same exceptions as in protected mode.

WBINVD—Write Back and Invalidate Cache

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
0F 09	WBINVD	ZO	Valid	Valid	Write back and flush Internal caches; initiate writing-back and flushing of external caches.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
ZO	NA	NA	NA	NA

Description

Writes back all modified cache lines in the processor's internal cache to main memory and invalidates (flushes) the internal caches. The instruction then issues a special-function bus cycle that directs external caches to also write back modified data and another bus cycle to indicate that the external caches should be invalidated.

After executing this instruction, the processor does not wait for the external caches to complete their write-back and flushing operations before proceeding with instruction execution. It is the responsibility of hardware to respond to the cache write-back and flush signals. The amount of time or cycles for WBINVD to complete will vary due to size and other factors of different cache hierarchies. As a consequence, the use of the WBINVD instruction can have an impact on logical processor interrupt/event response time. Additional information of WBINVD behavior in a cache hierarchy with hierarchical sharing topology can be found in Chapter 2 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*.

The WBINVD instruction is a privileged instruction. When the processor is running in protected mode, the CPL of a program or procedure must be 0 to execute this instruction. This instruction is also a serializing instruction (see "Serializing Instructions" in Chapter 8 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*).

In situations where cache coherency with main memory is not a concern, software can use the INVD instruction.

This instruction's operation is the same in non-64-bit modes and 64-bit mode.

IA-32 Architecture Compatibility

The WBINVD instruction is implementation dependent, and its function may be implemented differently on future Intel 64 and IA-32 processors. The instruction is not supported on IA-32 processors earlier than the Intel486 processor.

Operation

```
WriteBack(InternalCaches);
Flush(InternalCaches);
SignalWriteBack(ExternalCaches);
SignalFlush(ExternalCaches);
Continue; (* Continue execution *)
```

Flags Affected

None.

Protected Mode Exceptions

#GP(0)	If the current privilege level is not 0.
#UD	If the LOCK prefix is used.

Real-Address Mode Exceptions

#UD	If the LOCK prefix is used.

Virtual-8086 Mode Exceptions

#GP(0)	WBINVD cannot be executed at the virtual-8086 mode.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

Same exceptions as in protected mode.

WRFSBASE/WRGSBASE—Write FS/GS Segment Base

Opcode/ Instruction	Op/ En	64/32-bit Mode	CPUID Feature Flag	Description
F3 0F AE /2 WRFSBASE *r32*	M	V/I	FSGSBASE	Load the FS base address with the 32-bit value in the source register.
F3 REX.W 0F AE /2 WRFSBASE *r64*	M	V/I	FSGSBASE	Load the FS base address with the 64-bit value in the source register.
F3 0F AE /3 WRGSBASE *r32*	M	V/I	FSGSBASE	Load the GS base address with the 32-bit value in the source register.
F3 REX.W 0F AE /3 WRGSBASE *r64*	M	V/I	FSGSBASE	Load the GS base address with the 64-bit value in the source register.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
M	ModRM:r/m (r)	NA	NA	NA

Description

Loads the FS or GS segment base address with the general-purpose register indicated by the modR/M:r/m field.

The source operand may be either a 32-bit or a 64-bit general-purpose register. The REX.W prefix indicates the operand size is 64 bits. If no REX.W prefix is used, the operand size is 32 bits; the upper 32 bits of the source register are ignored and upper 32 bits of the base address (for FS or GS) are cleared.

This instruction is supported only in 64-bit mode.

Operation

```
FS/GS segment base address ← SRC;
```

Flags Affected

None

C/C++ Compiler Intrinsic Equivalent

WRFSBASE: void _writefsbase_u32(unsigned int);

WRFSBASE: _writefsbase_u64(unsigned __int64);

WRGSBASE: void _writegsbase_u32(unsigned int);

WRGSBASE: _writegsbase_u64(unsigned __int64);

Protected Mode Exceptions

#UD	The WRFSBASE and WRGSBASE instructions are not recognized in protected mode.

Real-Address Mode Exceptions

#UD	The WRFSBASE and WRGSBASE instructions are not recognized in real-address mode.

Virtual-8086 Mode Exceptions

#UD	The WRFSBASE and WRGSBASE instructions are not recognized in virtual-8086 mode.

Compatibility Mode Exceptions

#UD	The WRFSBASE and WRGSBASE instructions are not recognized in compatibility mode.

64-Bit Mode Exceptions

#UD	If the LOCK prefix is used. If CR4.FSGSBASE[bit 16] = 0. If CPUID.07H.0H:EBX.FSGSBASE[bit 0] = 0
#GP(0)	If the source register contains a non-canonical address.

WRMSR—Write to Model Specific Register

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
0F 30	WRMSR	ZO	Valid	Valid	Write the value in EDX:EAX to MSR specified by ECX.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
ZO	NA	NA	NA	NA

Description

Writes the contents of registers EDX:EAX into the 64-bit model specific register (MSR) specified in the ECX register. (On processors that support the Intel 64 architecture, the high-order 32 bits of RCX are ignored.) The contents of the EDX register are copied to high-order 32 bits of the selected MSR and the contents of the EAX register are copied to low-order 32 bits of the MSR. (On processors that support the Intel 64 architecture, the high-order 32 bits of each of RAX and RDX are ignored.) Undefined or reserved bits in an MSR should be set to values previously read.

This instruction must be executed at privilege level 0 or in real-address mode; otherwise, a general protection exception #GP(0) is generated. Specifying a reserved or unimplemented MSR address in ECX will also cause a general protection exception. The processor will also generate a general protection exception if software attempts to write to bits in a reserved MSR.

When the WRMSR instruction is used to write to an MTRR, the TLBs are invalidated. This includes global entries (see "Translation Lookaside Buffers (TLBs)" in Chapter 3 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*).

MSRs control functions for testability, execution tracing, performance-monitoring and machine check errors. Chapter 2, "Model-Specific Registers (MSRs)" of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 4*, lists all MSRs that can be written with this instruction and their addresses. Note that each processor family has its own set of MSRs.

The WRMSR instruction is a serializing instruction (see "Serializing Instructions" in Chapter 8 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*). Note that WRMSR to the IA32_TSC_DEADLINE MSR (MSR index 6E0H) and the X2APIC MSRs (MSR indices 802H to 83FH) are not serializing.

The CPUID instruction should be used to determine whether MSRs are supported (CPUID.01H:EDX[5] = 1) before using this instruction.

IA-32 Architecture Compatibility

The MSRs and the ability to read them with the WRMSR instruction were introduced into the IA-32 architecture with the Pentium processor. Execution of this instruction by an IA-32 processor earlier than the Pentium processor results in an invalid opcode exception #UD.

Operation

MSR[ECX] ← EDX:EAX;

Flags Affected

None.

Protected Mode Exceptions

#GP(0)	If the current privilege level is not 0.
	If the value in ECX specifies a reserved or unimplemented MSR address.
	If the value in EDX:EAX sets bits that are reserved in the MSR specified by ECX.
	If the source register contains a non-canonical address and ECX specifies one of the following MSRs: IA32_DS_AREA, IA32_FS_BASE, IA32_GS_BASE, IA32_KERNEL_GS_BASE, IA32_LSTAR, IA32_SYSENTER_EIP, IA32_SYSENTER_ESP.
#UD	If the LOCK prefix is used.

Real-Address Mode Exceptions

#GP	If the value in ECX specifies a reserved or unimplemented MSR address.
	If the value in EDX:EAX sets bits that are reserved in the MSR specified by ECX.
	If the source register contains a non-canonical address and ECX specifies one of the following MSRs: IA32_DS_AREA, IA32_FS_BASE, IA32_GS_BASE, IA32_KERNEL_GS_BASE, IA32_LSTAR, IA32_SYSENTER_EIP, IA32_SYSENTER_ESP.
#UD	If the LOCK prefix is used.

Virtual-8086 Mode Exceptions

#GP(0)	The WRMSR instruction is not recognized in virtual-8086 mode.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

Same exceptions as in protected mode.

WRPKRU—Write Data to User Page Key Register

Opcode*	Instruction	Op/ En	64/32bit Mode Support	CPUID Feature Flag	Description
NP 0F 01 EF	WRPKRU	ZO	V/V	OSPKE	Writes EAX into PKRU.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
ZO	NA	NA	NA	NA

Description

Writes the value of EAX into PKRU. ECX and EDX must be 0 when WRPKRU is executed; otherwise, a general-protection exception (#GP) occurs.

WRPKRU can be executed only if CR4.PKE = 1; otherwise, an invalid-opcode exception (#UD) occurs. Software can discover the value of CR4.PKE by examining CPUID.(EAX=07H,ECX=0H):ECX.OSPKE [bit 4].

On processors that support the Intel 64 Architecture, the high-order 32-bits of RCX, RDX and RAX are ignored.

Operation

```
IF (ECX = 0 AND EDX = 0)
    THEN PKRU ← EAX;
    ELSE #GP(0);
FI;
```

Flags Affected

None.

C/C++ Compiler Intrinsic Equivalent

WRPKRU: void _wrpkru(uint32_t);

Protected Mode Exceptions

#GP(0)	If ECX ≠ 0.
	If EDX ≠ 0.
#UD	If the LOCK prefix is used.
	If CR4.PKE = 0.

Real-Address Mode Exceptions

Same exceptions as in protected mode.

Virtual-8086 Mode Exceptions

Same exceptions as in protected mode.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

Same exceptions as in protected mode.

XACQUIRE/XRELEASE — Hardware Lock Elision Prefix Hints

Opcode/Instruction	64/32bit Mode Support	CPUID Feature Flag	Description
F2 XACQUIRE	V/V	HLE[1]	A hint used with an "XACQUIRE-enabled" instruction to start lock elision on the instruction memory operand address.
F3 XRELEASE	V/V	HLE	A hint used with an "XRELEASE-enabled" instruction to end lock elision on the instruction memory operand address.

NOTES:

1. Software is not required to check the HLE feature flag to use XACQUIRE or XRELEASE, as they are treated as regular prefix if HLE feature flag reports 0.

Description

The XACQUIRE prefix is a hint to start lock elision on the memory address specified by the instruction and the XRELEASE prefix is a hint to end lock elision on the memory address specified by the instruction.

The XACQUIRE prefix hint can only be used with the following instructions (these instructions are also referred to as XACQUIRE-enabled when used with the XACQUIRE prefix):

- Instructions with an explicit LOCK prefix (F0H) prepended to forms of the instruction where the destination operand is a memory operand: ADD, ADC, AND, BTC, BTR, BTS, CMPXCHG, CMPXCHG8B, DEC, INC, NEG, NOT, OR, SBB, SUB, XOR, XADD, and XCHG.
- The XCHG instruction either with or without the presence of the LOCK prefix.

The XRELEASE prefix hint can only be used with the following instructions (also referred to as XRELEASE-enabled when used with the XRELEASE prefix):

- Instructions with an explicit LOCK prefix (F0H) prepended to forms of the instruction where the destination operand is a memory operand: ADD, ADC, AND, BTC, BTR, BTS, CMPXCHG, CMPXCHG8B, DEC, INC, NEG, NOT, OR, SBB, SUB, XOR, XADD, and XCHG.
- The XCHG instruction either with or without the presence of the LOCK prefix.
- The "MOV mem, reg" (Opcode 88H/89H) and "MOV mem, imm" (Opcode C6H/C7H) instructions. In these cases, the XRELEASE is recognized without the presence of the LOCK prefix.

The lock variables must satisfy the guidelines described in *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*, Section 16.3.3, for elision to be successful, otherwise an HLE abort may be signaled.

If an encoded byte sequence that meets XACQUIRE/XRELEASE requirements includes both prefixes, then the HLE semantic is determined by the prefix byte that is placed closest to the instruction opcode. For example, an F3F2C6 will not be treated as a XRELEASE-enabled instruction since the F2H (XACQUIRE) is closest to the instruction opcode C6. Similarly, an F2F3F0 prefixed instruction will be treated as a XRELEASE-enabled instruction since F3H (XRELEASE) is closest to the instruction opcode.

Intel 64 and IA-32 Compatibility

The effect of the XACQUIRE/XRELEASE prefix hint is the same in non-64-bit modes and in 64-bit mode.

For instructions that do not support the XACQUIRE hint, the presence of the F2H prefix behaves the same way as prior hardware, according to

- REPNE/REPNZ semantics for string instructions,
- Serve as SIMD prefix for legacy SIMD instructions operating on XMM register
- Cause #UD if prepending the VEX prefix.
- Undefined for non-string instructions or other situations.

For instructions that do not support the XRELEASE hint, the presence of the F3H prefix behaves the same way as in prior hardware, according to

- REP/REPE/REPZ semantics for string instructions,
- Serve as SIMD prefix for legacy SIMD instructions operating on XMM register
- Cause #UD if prepending the VEX prefix.
- Undefined for non-string instructions or other situations.

Operation

XACQUIRE

```
IF XACQUIRE-enabled instruction
    THEN
        IF (HLE_NEST_COUNT < MAX_HLE_NEST_COUNT) THEN
            HLE_NEST_COUNT++
            IF (HLE_NEST_COUNT = 1) THEN
                HLE_ACTIVE ← 1
                IF 64-bit mode
                    THEN
                        restartRIP ← instruction pointer of the XACQUIRE-enabled instruction
                    ELSE
                        restartEIP ← instruction pointer of the XACQUIRE-enabled instruction
                FI;
                Enter HLE Execution (* record register state, start tracking memory state *)
            FI; (* HLE_NEST_COUNT = 1*)
            IF ElisionBufferAvailable
                THEN
                    Allocate elision buffer
                    Record address and data for forwarding and commit checking
                    Perform elision
                ELSE
                    Perform lock acquire operation transactionally but without elision
            FI;
        ELSE (* HLE_NEST_COUNT = MAX_HLE_NEST_COUNT *)
                GOTO HLE_ABORT_PROCESSING
        FI;
    ELSE
        Treat instruction as non-XACQUIRE F2H prefixed legacy instruction
FI;
```

XRELEASE

```
IF XRELEASE-enabled instruction
    THEN
        IF (HLE_NEST_COUNT > 0)
            THEN
                HLE_NEST_COUNT--
                IF lock address matches in elision buffer THEN
                    IF lock satisfies address and value requirements THEN
                        Deallocate elision buffer
                    ELSE
                        GOTO HLE_ABORT_PROCESSING
                    FI;
                FI;
                IF (HLE_NEST_COUNT = 0)
                    THEN
                        IF NoAllocatedElisionBuffer
                            THEN
                                Try to commit transactional execution
                                IF fail to commit transactional execution
                                    THEN
                                        GOTO HLE_ABORT_PROCESSING;
                                    ELSE (* commit success *)
                                        HLE_ACTIVE ← 0
                                FI;
                            ELSE
                                GOTO HLE_ABORT_PROCESSING
                        FI;
                FI;
        FI; (* HLE_NEST_COUNT > 0 *)
    ELSE
        Treat instruction as non-XRELEASE F3H prefixed legacy instruction
FI;

(* For any HLE abort condition encountered during HLE execution *)
HLE_ABORT_PROCESSING:
    HLE_ACTIVE ← 0
    HLE_NEST_COUNT ← 0
    Restore architectural register state
    Discard memory updates performed in transaction
    Free any allocated lock elision buffers
    IF 64-bit mode
        THEN
            RIP ← restartRIP
        ELSE
            EIP ← restartEIP
    FI;
    Execute and retire instruction at RIP (or EIP) and ignore any HLE hint
END
```

SIMD Floating-Point Exceptions

None

Other Exceptions

#GP(0)	If the use of prefix causes instruction length to exceed 15 bytes.

XABORT — Transactional Abort

Opcode/Instruction	Op/ En	64/32bit Mode Support	CPUID Feature Flag	Description
C6 F8 ib XABORT imm8	A	V/V	RTM	Causes an RTM abort if in RTM execution

Instruction Operand Encoding

Op/En	Operand 1	Operand2	Operand3	Operand4
A	imm8	NA	NA	NA

Description

XABORT forces an RTM abort. Following an RTM abort, the logical processor resumes execution at the fallback address computed through the outermost XBEGIN instruction. The EAX register is updated to reflect an XABORT instruction caused the abort, and the imm8 argument will be provided in bits 31:24 of EAX.

Operation

XABORT

```
IF RTM_ACTIVE = 0
    THEN
        Treat as NOP;
    ELSE
        GOTO RTM_ABORT_PROCESSING;
FI;

(* For any RTM abort condition encountered during RTM execution *)
RTM_ABORT_PROCESSING:
    Restore architectural register state;
    Discard memory updates performed in transaction;
    Update EAX with status and XABORT argument;
    RTM_NEST_COUNT ← 0;
    RTM_ACTIVE ← 0;
    IF 64-bit Mode
        THEN
            RIP ← fallbackRIP;
        ELSE
            EIP ← fallbackEIP;
    FI;
END
```

Flags Affected

None

Intel C/C++ Compiler Intrinsic Equivalent

XABORT: void _xabort(unsigned int);

SIMD Floating-Point Exceptions

None

Other Exceptions

#UD	CPUID.(EAX=7, ECX=0):EBX.RTM[bit 11] = 0. If LOCK prefix is used.

XADD—Exchange and Add

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
0F C0 /r	XADD *r/m8, r8*	MR	Valid	Valid	Exchange *r8* and *r/m8*; load sum into *r/m8*.
REX + 0F C0 /r	XADD *r/m8*, r8**	MR	Valid	N.E.	Exchange *r8* and *r/m8*; load sum into *r/m8*.
0F C1 /r	XADD *r/m16, r16*	MR	Valid	Valid	Exchange *r16* and *r/m16*; load sum into *r/m16*.
0F C1 /r	XADD *r/m32, r32*	MR	Valid	Valid	Exchange *r32* and *r/m32*; load sum into *r/m32*.
REX.W + 0F C1 /r	XADD *r/m64, r64*	MR	Valid	N.E.	Exchange *r64* and *r/m64*; load sum into *r/m64*.

NOTES:

* In 64-bit mode, r/m8 can not be encoded to access the following byte registers if a REX prefix is used: AH, BH, CH, DH.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
MR	ModRM:r/m (r, w)	ModRM:reg (r, w)	NA	NA

Description

Exchanges the first operand (destination operand) with the second operand (source operand), then loads the sum of the two values into the destination operand. The destination operand can be a register or a memory location; the source operand is a register.

In 64-bit mode, the instruction's default operation size is 32 bits. Using a REX prefix in the form of REX.R permits access to additional registers (R8-R15). Using a REX prefix in the form of REX.W promotes operation to 64 bits. See the summary chart at the beginning of this section for encoding data and limits.

This instruction can be used with a LOCK prefix to allow the instruction to be executed atomically.

IA-32 Architecture Compatibility

IA-32 processors earlier than the Intel486 processor do not recognize this instruction. If this instruction is used, you should provide an equivalent code sequence that runs on earlier processors.

Operation

```
TEMP ← SRC + DEST;
SRC ← DEST;
DEST ← TEMP;
```

Flags Affected

The CF, PF, AF, SF, ZF, and OF flags are set according to the result of the addition, which is stored in the destination operand.

Protected Mode Exceptions

#GP(0)	If the destination is located in a non-writable segment.
	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
	If the DS, ES, FS, or GS register contains a NULL segment selector.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If the LOCK prefix is used but the destination is not a memory operand.

Real-Address Mode Exceptions

#GP	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS	If a memory operand effective address is outside the SS segment limit.
#UD	If the LOCK prefix is used but the destination is not a memory operand.

Virtual-8086 Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.
#UD	If the LOCK prefix is used but the destination is not a memory operand.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If the LOCK prefix is used but the destination is not a memory operand.

XBEGIN — Transactional Begin

Opcode/Instruction	Op/ En	64/32bit Mode Support	CPUID Feature Flag	Description
C7 F8 XBEGIN rel16	A	V/V	RTM	Specifies the start of an RTM region. Provides a 16-bit relative offset to compute the address of the fallback instruction address at which execution resumes following an RTM abort.
C7 F8 XBEGIN rel32	A	V/V	RTM	Specifies the start of an RTM region. Provides a 32-bit relative offset to compute the address of the fallback instruction address at which execution resumes following an RTM abort.

Instruction Operand Encoding

Op/En	Operand 1	Operand2	Operand3	Operand4
A	Offset	NA	NA	NA

Description

The XBEGIN instruction specifies the start of an RTM code region. If the logical processor was not already in transactional execution, then the XBEGIN instruction causes the logical processor to transition into transactional execution. The XBEGIN instruction that transitions the logical processor into transactional execution is referred to as the outermost XBEGIN instruction. The instruction also specifies a relative offset to compute the address of the fallback code path following a transactional abort.

On an RTM abort, the logical processor discards all architectural register and memory updates performed during the RTM execution and restores architectural state to that corresponding to the outermost XBEGIN instruction. The fallback address following an abort is computed from the outermost XBEGIN instruction.

Operation

XBEGIN

```
IF RTM_NEST_COUNT < MAX_RTM_NEST_COUNT
    THEN
        RTM_NEST_COUNT++
        IF RTM_NEST_COUNT = 1 THEN
            IF 64-bit Mode
                THEN
                        fallbackRIP ← RIP + SignExtend64(IMM)
                                (* RIP is instruction following XBEGIN instruction *)
                    ELSE
                        fallbackEIP ← EIP + SignExtend32(IMM)
                                (* EIP is instruction following XBEGIN instruction *)
            FI;

            IF (64-bit mode)
                THEN IF (fallbackRIP is not canonical)
                    THEN #GP(0)
                FI;
                ELSE IF (fallbackEIP outside code segment limit)
                    THEN #GP(0)
                FI;
            FI;

            RTM_ACTIVE ← 1
            Enter RTM Execution (* record register state, start tracking memory state*)
        FI; (* RTM_NEST_COUNT = 1 *)
```

```
    ELSE (* RTM_NEST_COUNT = MAX_RTM_NEST_COUNT *)
        GOTO RTM_ABORT_PROCESSING
FI;

(* For any RTM abort condition encountered during RTM execution *)
RTM_ABORT_PROCESSING:
    Restore architectural register state
    Discard memory updates performed in transaction
    Update EAX with status
    RTM_NEST_COUNT ← 0
    RTM_ACTIVE ← 0
    IF 64-bit mode
        THEN
            RIP ← fallbackRIP
        ELSE
            EIP ← fallbackEIP
    FI;
END
```

Flags Affected

None

Intel C/C++ Compiler Intrinsic Equivalent

XBEGIN: unsigned int _xbegin(void);

SIMD Floating-Point Exceptions

None

Protected Mode Exceptions

#UD	CPUID.(EAX=7, ECX=0):EBX.RTM[bit 11]=0.
	If LOCK prefix is used.
#GP(0)	If the fallback address is outside the CS segment.

Real-Address Mode Exceptions

#GP(0)	If the fallback address is outside the address space 0000H and FFFFH.
#UD	CPUID.(EAX=7, ECX=0):EBX.RTM[bit 11]=0.
	If LOCK prefix is used.

Virtual-8086 Mode Exceptions

#GP(0)	If the fallback address is outside the address space 0000H and FFFFH.
#UD	CPUID.(EAX=7, ECX=0):EBX.RTM[bit 11]=0.
	If LOCK prefix is used.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-bit Mode Exceptions

#UD	CPUID.(EAX=7, ECX=0):EBX.RTM[bit 11] = 0. If LOCK prefix is used.
#GP(0)	If the fallback address is non-canonical.

XCHG—Exchange Register/Memory with Register

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
90+*rw*	XCHG AX, *r16*	O	Valid	Valid	Exchange *r16* with AX.
90+*rw*	XCHG *r16*, AX	O	Valid	Valid	Exchange AX with *r16.*
90+*rd*	XCHG EAX, *r32*	O	Valid	Valid	Exchange *r32* with EAX.
REX.W + 90+*rd*	XCHG RAX, *r64*	O	Valid	N.E.	Exchange *r64* with RAX.
90+*rd*	XCHG *r32*, EAX	O	Valid	Valid	Exchange EAX with *r32.*
REX.W + 90+*rd*	XCHG *r64*, RAX	O	Valid	N.E.	Exchange RAX with *r64.*
86 /*r*	XCHG *r/m8, r8*	MR	Valid	Valid	Exchange *r8* (byte register) with byte from *r/m8.*
REX + 86 /*r*	XCHG *r/m8*, r8**	MR	Valid	N.E.	Exchange *r8* (byte register) with byte from *r/m8.*
86 /*r*	XCHG *r8, r/m8*	RM	Valid	Valid	Exchange byte from *r/m8* with *r8* (byte register).
REX + 86 /*r*	XCHG *r8*, r/m8**	RM	Valid	N.E.	Exchange byte from *r/m8* with *r8* (byte register).
87 /*r*	XCHG *r/m16, r16*	MR	Valid	Valid	Exchange *r16* with word from *r/m16.*
87 /*r*	XCHG *r16, r/m16*	RM	Valid	Valid	Exchange word from *r/m16* with *r16.*
87 /*r*	XCHG *r/m32, r32*	MR	Valid	Valid	Exchange *r32* with doubleword from *r/m32.*
REX.W + 87 /*r*	XCHG *r/m64, r64*	MR	Valid	N.E.	Exchange *r64* with quadword from *r/m64.*
87 /*r*	XCHG *r32, r/m32*	RM	Valid	Valid	Exchange doubleword from *r/m32* with *r32.*
REX.W + 87 /*r*	XCHG *r64, r/m64*	RM	Valid	N.E.	Exchange quadword from *r/m64* with *r64.*

NOTES:

* In 64-bit mode, r/m8 can not be encoded to access the following byte registers if a REX prefix is used: AH, BH, CH, DH.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
O	AX/EAX/RAX (r, w)	opcode + rd (r, w)	NA	NA
O	opcode + rd (r, w)	AX/EAX/RAX (r, w)	NA	NA
MR	ModRM:r/m (r, w)	ModRM:reg (r)	NA	NA
RM	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

Description

Exchanges the contents of the destination (first) and source (second) operands. The operands can be two general-purpose registers or a register and a memory location. If a memory operand is referenced, the processor's locking protocol is automatically implemented for the duration of the exchange operation, regardless of the presence or absence of the LOCK prefix or of the value of the IOPL. (See the LOCK prefix description in this chapter for more information on the locking protocol.)

This instruction is useful for implementing semaphores or similar data structures for process synchronization. (See "Bus Locking" in Chapter 8 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*, for more information on bus locking.)

The XCHG instruction can also be used instead of the BSWAP instruction for 16-bit operands.

In 64-bit mode, the instruction's default operation size is 32 bits. Using a REX prefix in the form of REX.R permits access to additional registers (R8-R15). Using a REX prefix in the form of REX.W promotes operation to 64 bits. See the summary chart at the beginning of this section for encoding data and limits.

NOTE

XCHG (E)AX, (E)AX (encoded instruction byte is 90H) is an alias for NOP regardless of data size prefixes, including REX.W.

Operation

```
TEMP ← DEST;
DEST ← SRC;
SRC ← TEMP;
```

Flags Affected

None.

Protected Mode Exceptions

#GP(0)	If either operand is in a non-writable segment.
	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
	If the DS, ES, FS, or GS register contains a NULL segment selector.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If the LOCK prefix is used but the destination is not a memory operand.

Real-Address Mode Exceptions

#GP	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS	If a memory operand effective address is outside the SS segment limit.
#UD	If the LOCK prefix is used but the destination is not a memory operand.

Virtual-8086 Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.
#UD	If the LOCK prefix is used but the destination is not a memory operand.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If the LOCK prefix is used but the destination is not a memory operand.

XEND — Transactional End

Opcode/Instruction	Op/ En	64/32bit Mode Support	CPUID Feature Flag	Description
NP 0F 01 D5 XEND	A	V/V	RTM	Specifies the end of an RTM code region.

Instruction Operand Encoding

Op/En	Operand 1	Operand2	Operand3	Operand4
A	NA	NA	NA	NA

Description

The instruction marks the end of an RTM code region. If this corresponds to the outermost scope (that is, including this XEND instruction, the number of XBEGIN instructions is the same as number of XEND instructions), the logical processor will attempt to commit the logical processor state atomically. If the commit fails, the logical processor will rollback all architectural register and memory updates performed during the RTM execution. The logical processor will resume execution at the fallback address computed from the outermost XBEGIN instruction. The EAX register is updated to reflect RTM abort information.

XEND executed outside a transactional region will cause a #GP (General Protection Fault).

Operation

XEND

```
IF (RTM_ACTIVE = 0) THEN
    SIGNAL #GP
ELSE
    RTM_NEST_COUNT--
    IF (RTM_NEST_COUNT = 0) THEN
        Try to commit transaction
        IF fail to commit transactional execution
            THEN
                GOTO RTM_ABORT_PROCESSING;
            ELSE (* commit success *)
                RTM_ACTIVE ← 0
        FI;
    FI;
FI;

(* For any RTM abort condition encountered during RTM execution *)
RTM_ABORT_PROCESSING:
    Restore architectural register state
    Discard memory updates performed in transaction
    Update EAX with status
    RTM_NEST_COUNT ← 0
    RTM_ACTIVE ← 0
    IF 64-bit Mode
        THEN
            RIP ← fallbackRIP
        ELSE
            EIP ← fallbackEIP
    FI;
END
```

Flags Affected

None

Intel C/C++ Compiler Intrinsic Equivalent

XEND: void _xend(void);

SIMD Floating-Point Exceptions

None

Other Exceptions

#UD	CPUID.(EAX=7, ECX=0):EBX.RTM[bit 11] = 0. If LOCK or 66H or F2H or F3H prefix is used.
#GP(0)	If RTM_ACTIVE = 0.

XGETBV—Get Value of Extended Control Register

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
NP 0F 01 D0	XGETBV	ZO	Valid	Valid	Reads an XCR specified by ECX into EDX:EAX.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
ZO	NA	NA	NA	NA

Description

Reads the contents of the extended control register (XCR) specified in the ECX register into registers EDX:EAX. (On processors that support the Intel 64 architecture, the high-order 32 bits of RCX are ignored.) The EDX register is loaded with the high-order 32 bits of the XCR and the EAX register is loaded with the low-order 32 bits. (On processors that support the Intel 64 architecture, the high-order 32 bits of each of RAX and RDX are cleared.) If fewer than 64 bits are implemented in the XCR being read, the values returned to EDX:EAX in unimplemented bit locations are undefined.

XCR0 is supported on any processor that supports the XGETBV instruction. If CPUID.(EAX=0DH,ECX=1):EAX.XG1[bit 2] = 1, executing XGETBV with ECX = 1 returns in EDX:EAX the logical-AND of XCR0 and the current value of the XINUSE state-component bitmap. This allows software to discover the state of the init optimization used by XSAVEOPT and XSAVES. See Chapter 13, "Managing State Using the XSAVE Feature Set," in *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*.

Use of any other value for ECX results in a general-protection (#GP) exception.

Operation

```
EDX:EAX ← XCR[ECX];
```

Flags Affected

None.

Intel C/C++ Compiler Intrinsic Equivalent

XGETBV: unsigned __int64 _xgetbv(unsigned int);

Protected Mode Exceptions

#GP(0)	If an invalid XCR is specified in ECX (includes ECX = 1 if CPUID.(EAX=0DH,ECX=1):EAX.XG1[bit 2] = 0).
#UD	If CPUID.01H:ECX.XSAVE[bit 26] = 0.
	If CR4.OSXSAVE[bit 18] = 0.
	If the LOCK prefix is used.

Real-Address Mode Exceptions

#GP(0)	If an invalid XCR is specified in ECX (includes ECX = 1 if CPUID.(EAX=0DH,ECX=1):EAX.XG1[bit 2] = 0).
#UD	If CPUID.01H:ECX.XSAVE[bit 26] = 0.
	If CR4.OSXSAVE[bit 18] = 0.
	If the LOCK prefix is used.

Virtual-8086 Mode Exceptions

Same exceptions as in protected mode.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

Same exceptions as in protected mode.

XLAT/XLATB—Table Look-up Translation

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
D7	XLAT *m8*	ZO	Valid	Valid	Set AL to memory byte DS:[(E)BX + unsigned AL].
D7	XLATB	ZO	Valid	Valid	Set AL to memory byte DS:[(E)BX + unsigned AL].
REX.W + D7	XLATB	ZO	Valid	N.E.	Set AL to memory byte [RBX + unsigned AL].

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
ZO	NA	NA	NA	NA

Description

Locates a byte entry in a table in memory, using the contents of the AL register as a table index, then copies the contents of the table entry back into the AL register. The index in the AL register is treated as an unsigned integer. The XLAT and XLATB instructions get the base address of the table in memory from either the DS:EBX or the DS:BX registers (depending on the address-size attribute of the instruction, 32 or 16, respectively). (The DS segment may be overridden with a segment override prefix.)

At the assembly-code level, two forms of this instruction are allowed: the "explicit-operand" form and the "no-operand" form. The explicit-operand form (specified with the XLAT mnemonic) allows the base address of the table to be specified explicitly with a symbol. This explicit-operands form is provided to allow documentation; however, note that the documentation provided by this form can be misleading. That is, the symbol does not have to specify the correct base address. The base address is always specified by the DS:(E)BX registers, which must be loaded correctly before the XLAT instruction is executed.

The no-operands form (XLATB) provides a "short form" of the XLAT instructions. Here also the processor assumes that the DS:(E)BX registers contain the base address of the table.

In 64-bit mode, operation is similar to that in legacy or compatibility mode. AL is used to specify the table index (the operand size is fixed at 8 bits). RBX, however, is used to specify the table's base address. See the summary chart at the beginning of this section for encoding data and limits.

Operation

```
IF AddressSize = 16
    THEN
        AL ← (DS:BX + ZeroExtend(AL));
    ELSE IF (AddressSize = 32)
        AL ← (DS:EBX + ZeroExtend(AL)); FI;
    ELSE (AddressSize = 64)
        AL ← (RBX + ZeroExtend(AL));
FI;
```

Flags Affected

None.

Protected Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
	If the DS, ES, FS, or GS register contains a NULL segment selector.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.

#UD	If the LOCK prefix is used.

Real-Address Mode Exceptions

#GP	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS	If a memory operand effective address is outside the SS segment limit.
#UD	If the LOCK prefix is used.

Virtual-8086 Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#UD	If the LOCK prefix is used.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form.
#PF(fault-code)	If a page fault occurs.
#UD	If the LOCK prefix is used.

XOR—Logical Exclusive OR

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
34 *ib*	XOR AL, i*mm8*	I	Valid	Valid	AL XOR *imm8.*
35 *iw*	XOR AX, i*mm16*	I	Valid	Valid	AX XOR *imm16.*
35 *id*	XOR EAX, i*mm32*	I	Valid	Valid	EAX XOR *imm32.*
REX.W + 35 *id*	XOR RAX, i*mm32*	I	Valid	N.E.	RAX XOR *imm32 (sign-extended).*
80 /6 *ib*	XOR *r/m8, imm8*	MI	Valid	Valid	*r/m8* XOR *imm8.*
REX + 80 /6 *ib*	XOR *r/m8*, imm8*	MI	Valid	N.E.	*r/m8* XOR *imm8.*
81 /6 *iw*	XOR *r/m16, imm16*	MI	Valid	Valid	*r/m16* XOR *imm16.*
81 /6 *id*	XOR *r/m32, imm32*	MI	Valid	Valid	*r/m32* XOR *imm32.*
REX.W + 81 /6 *id*	XOR *r/m64, imm32*	MI	Valid	N.E.	*r/m64* XOR *imm32 (sign-extended).*
83 /6 *ib*	XOR *r/m16, imm8*	MI	Valid	Valid	*r/m16* XOR *imm8 (sign-extended).*
83 /6 *ib*	XOR *r/m32, imm8*	MI	Valid	Valid	*r/m32* XOR *imm8 (sign-extended).*
REX.W + 83 /6 *ib*	XOR *r/m64, imm8*	MI	Valid	N.E.	*r/m64* XOR *imm8 (sign-extended).*
30 /*r*	XOR *r/m8, r8*	MR	Valid	Valid	*r/m8* XOR *r8.*
REX + 30 /*r*	XOR *r/m8*, r8**	MR	Valid	N.E.	*r/m8* XOR *r8.*
31 /*r*	XOR *r/m16, r16*	MR	Valid	Valid	*r/m16* XOR *r16.*
31 /*r*	XOR *r/m32, r32*	MR	Valid	Valid	*r/m32* XOR *r32.*
REX.W + 31 /*r*	XOR *r/m64, r64*	MR	Valid	N.E.	*r/m64* XOR *r64.*
32 /*r*	XOR *r8, r/m8*	RM	Valid	Valid	*r8* XOR *r/m8.*
REX + 32 /*r*	XOR *r8*, r/m8**	RM	Valid	N.E.	*r8* XOR *r/m8.*
33 /*r*	XOR *r16, r/m16*	RM	Valid	Valid	*r16* XOR *r/m16.*
33 /*r*	XOR *r32, r/m32*	RM	Valid	Valid	*r32* XOR *r/m32.*
REX.W + 33 /*r*	XOR *r64, r/m64*	RM	Valid	N.E.	*r64* XOR *r/m64.*

NOTES:

* In 64-bit mode, r/m8 can not be encoded to access the following byte registers if a REX prefix is used: AH, BH, CH, DH.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
I	AL/AX/EAX/RAX	imm8/16/32	NA	NA
MI	ModRM:r/m (r, w)	imm8/16/32	NA	NA
MR	ModRM:r/m (r, w)	ModRM:reg (r)	NA	NA
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA

Description

Performs a bitwise exclusive OR (XOR) operation on the destination (first) and source (second) operands and stores the result in the destination operand location. The source operand can be an immediate, a register, or a memory location; the destination operand can be a register or a memory location. (However, two memory operands cannot be used in one instruction.) Each bit of the result is 1 if the corresponding bits of the operands are different; each bit is 0 if the corresponding bits are the same.

This instruction can be used with a LOCK prefix to allow the instruction to be executed atomically.

In 64-bit mode, using a REX prefix in the form of REX.R permits access to additional registers (R8-R15). Using a REX prefix in the form of REX.W promotes operation to 64 bits. See the summary chart at the beginning of this section for encoding data and limits.

Operation

```
DEST ← DEST XOR SRC;
```

Flags Affected

The OF and CF flags are cleared; the SF, ZF, and PF flags are set according to the result. The state of the AF flag is undefined.

Protected Mode Exceptions

#GP(0)	If the destination operand points to a non-writable segment.
	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
	If the DS, ES, FS, or GS register contains a NULL segment selector.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If the LOCK prefix is used but the destination is not a memory operand.

Real-Address Mode Exceptions

#GP	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS	If a memory operand effective address is outside the SS segment limit.
#UD	If the LOCK prefix is used but the destination is not a memory operand.

Virtual-8086 Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.
#UD	If the LOCK prefix is used but the destination is not a memory operand.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If the LOCK prefix is used but the destination is not a memory operand.

XORPD—Bitwise Logical XOR of Packed Double Precision Floating-Point Values

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
66 0F 57/r XORPD xmm1, xmm2/m128	A	V/V	SSE2	Return the bitwise logical XOR of packed double-precision floating-point values in xmm1 and xmm2/mem.
VEX.NDS.128.66.0F.WIG 57 /r VXORPD xmm1,xmm2, xmm3/m128	B	V/V	AVX	Return the bitwise logical XOR of packed double-precision floating-point values in xmm2 and xmm3/mem.
VEX.NDS.256.66.0F.WIG 57 /r VXORPD ymm1, ymm2, ymm3/m256	B	V/V	AVX	Return the bitwise logical XOR of packed double-precision floating-point values in ymm2 and ymm3/mem.
EVEX.NDS.128.66.0F.W1 57 /r VXORPD xmm1 {k1}{z}, xmm2, xmm3/m128/m64bcst	C	V/V	AVX512VL AVX512DQ	Return the bitwise logical XOR of packed double-precision floating-point values in xmm2 and xmm3/m128/m64bcst subject to writemask k1.
EVEX.NDS.256.66.0F.W1 57 /r VXORPD ymm1 {k1}{z}, ymm2, ymm3/m256/m64bcst	C	V/V	AVX512VL AVX512DQ	Return the bitwise logical XOR of packed double-precision floating-point values in ymm2 and ymm3/m256/m64bcst subject to writemask k1.
EVEX.NDS.512.66.0F.W1 57 /r VXORPD zmm1 {k1}{z}, zmm2, zmm3/m512/m64bcst	C	V/V	AVX512DQ	Return the bitwise logical XOR of packed double-precision floating-point values in zmm2 and zmm3/m512/m64bcst subject to writemask k1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	NA	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
B	NA	ModRM:reg (w)	VEX.vvvv (r)	ModRM:r/m (r)	NA
C	Full Vector	ModRM:reg (w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA

Description

Performs a bitwise logical XOR of the two, four or eight packed double-precision floating-point values from the first source operand and the second source operand, and stores the result in the destination operand

EVEX.512 encoded version: The first source operand is a ZMM register. The second source operand can be a ZMM register or a vector memory location. The destination operand is a ZMM register conditionally updated with writemask k1.

VEX.256 and EVEX.256 encoded versions: The first source operand is a YMM register. The second source operand is a YMM register or a 256-bit memory location. The destination operand is a YMM register (conditionally updated with writemask k1 in case of EVEX). The upper bits (MAXVL-1:256) of the corresponding ZMM register destination are zeroed.

VEX.128 and EVEX.128 encoded versions: The first source operand is an XMM register. The second source operand is an XMM register or 128-bit memory location. The destination operand is an XMM register (conditionally updated with writemask k1 in case of EVEX). The upper bits (MAXVL-1:128) of the corresponding ZMM register destination are zeroed.

128-bit Legacy SSE version: The second source can be an XMM register or an 128-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (MAXVL-1:128) of the corresponding register destination are unmodified.

Operation

VXORPD (EVEX encoded versions)

```
(KL, VL) = (2, 128), (4, 256), (8, 512)
FOR j ← 0 TO KL-1
    i ← j * 64
    IF k1[j] OR *no writemask* THEN
            IF (EVEX.b == 1) AND (SRC2 *is memory*)
                THEN DEST[i+63:i] ← SRC1[i+63:i] BITWISE XOR SRC2[63:0];
                ELSE DEST[i+63:i] ← SRC1[i+63:i] BITWISE XOR SRC2[i+63:i];
            FI;
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+63:i] remains unchanged*
                ELSE *zeroing-masking*                  ; zeroing-masking
                    DEST[i+63:i] = 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0
```

VXORPD (VEX.256 encoded version)

```
DEST[63:0] ← SRC1[63:0] BITWISE XOR SRC2[63:0]
DEST[127:64] ← SRC1[127:64] BITWISE XOR SRC2[127:64]
DEST[191:128] ← SRC1[191:128] BITWISE XOR SRC2[191:128]
DEST[255:192] ← SRC1[255:192] BITWISE XOR SRC2[255:192]
DEST[MAXVL-1:256] ← 0
```

VXORPD (VEX.128 encoded version)

```
DEST[63:0] ← SRC1[63:0] BITWISE XOR SRC2[63:0]
DEST[127:64] ← SRC1[127:64] BITWISE XOR SRC2[127:64]
DEST[MAXVL-1:128] ← 0
```

XORPD (128-bit Legacy SSE version)

```
DEST[63:0] ← DEST[63:0] BITWISE XOR SRC[63:0]
DEST[127:64] ← DEST[127:64] BITWISE XOR SRC[127:64]
DEST[MAXVL-1:128] (Unmodified)
```

Intel C/C++ Compiler Intrinsic Equivalent

VXORPD __m512d _mm512_xor_pd (__m512d a, __m512d b);
VXORPD __m512d _mm512_mask_xor_pd (__m512d a, __mmask8 m, __m512d b);
VXORPD __m512d _mm512_maskz_xor_pd (__mmask8 m, __m512d a);
VXORPD __m256d _mm256_xor_pd (__m256d a, __m256d b);
VXORPD __m256d _mm256_mask_xor_pd (__m256d a, __mmask8 m, __m256d b);
VXORPD __m256d _mm256_maskz_xor_pd (__mmask8 m, __m256d a);
XORPD __m128d _mm_xor_pd (__m128d a, __m128d b);
VXORPD __m128d _mm_mask_xor_pd (__m128d a, __mmask8 m, __m128d b);
VXORPD __m128d _mm_maskz_xor_pd (__mmask8 m, __m128d a);

SIMD Floating-Point Exceptions

None

Other Exceptions

Non-EVEX-encoded instructions, see Exceptions Type 4.
EVEX-encoded instructions, see Exceptions Type E4.

XORPS—Bitwise Logical XOR of Packed Single Precision Floating-Point Values

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F 57 /r XORPS xmm1, xmm2/m128	A	V/V	SSE	Return the bitwise logical XOR of packed single-precision floating-point values in xmm1 and xmm2/mem.
VEX.NDS.128.0F.WIG 57 /r VXORPS xmm1,xmm2, xmm3/m128	B	V/V	AVX	Return the bitwise logical XOR of packed single-precision floating-point values in xmm2 and xmm3/mem.
VEX.NDS.256.0F.WIG 57 /r VXORPS ymm1, ymm2, ymm3/m256	B	V/V	AVX	Return the bitwise logical XOR of packed single-precision floating-point values in ymm2 and ymm3/mem.
EVEX.NDS.128.0F.W0 57 /r VXORPS xmm1 {k1}{z}, xmm2, xmm3/m128/m32bcst	C	V/V	AVX512VL AVX512DQ	Return the bitwise logical XOR of packed single-precision floating-point values in xmm2 and xmm3/m128/m32bcst subject to writemask k1.
EVEX.NDS.256.0F.W0 57 /r VXORPS ymm1 {k1}{z}, ymm2, ymm3/m256/m32bcst	C	V/V	AVX512VL AVX512DQ	Return the bitwise logical XOR of packed single-precision floating-point values in ymm2 and ymm3/m256/m32bcst subject to writemask k1.
EVEX.NDS.512.0F.W0 57 /r VXORPS zmm1 {k1}{z}, zmm2, zmm3/m512/m32bcst	C	V/V	AVX512DQ	Return the bitwise logical XOR of packed single-precision floating-point values in zmm2 and zmm3/m512/m32bcst subject to writemask k1.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
A	NA	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA
B	NA	ModRM:reg (w)	VEX.vvvv	ModRM:r/m (r)	NA
C	Full Vector	ModRM:reg (w)	EVEX.vvvv	ModRM:r/m (r)	NA

Description

Performs a bitwise logical XOR of the four, eight or sixteen packed single-precision floating-point values from the first source operand and the second source operand, and stores the result in the destination operand

EVEX.512 encoded version: The first source operand is a ZMM register. The second source operand can be a ZMM register or a vector memory location. The destination operand is a ZMM register conditionally updated with writemask k1.

VEX.256 and EVEX.256 encoded versions: The first source operand is a YMM register. The second source operand is a YMM register or a 256-bit memory location. The destination operand is a YMM register (conditionally updated with writemask k1 in case of EVEX). The upper bits (MAXVL-1:256) of the corresponding ZMM register destination are zeroed.

VEX.128 and EVEX.128 encoded versions: The first source operand is an XMM register. The second source operand is an XMM register or 128-bit memory location. The destination operand is an XMM register (conditionally updated with writemask k1 in case of EVEX). The upper bits (MAXVL-1:128) of the corresponding ZMM register destination are zeroed.

128-bit Legacy SSE version: The second source can be an XMM register or an 128-bit memory location. The destination is not distinct from the first source XMM register and the upper bits (MAXVL-1:128) of the corresponding register destination are unmodified.

Operation

VXORPS (EVEX encoded versions)

```
(KL, VL) = (4, 128), (8, 256), (16, 512)
FOR j ← 0 TO KL-1
    i ← j * 32
    IF k1[j] OR *no writemask* THEN
            IF (EVEX.b == 1) AND (SRC2 *is memory*)
                THEN DEST[i+31:i] ← SRC1[i+31:i] BITWISE XOR SRC2[31:0];
                ELSE DEST[i+31:i] ← SRC1[i+31:i] BITWISE XOR SRC2[i+31:i];
            FI;
        ELSE
            IF *merging-masking*                    ; merging-masking
                THEN *DEST[i+31:i] remains unchanged*
                ELSE *zeroing-masking*                  ; zeroing-masking
                    DEST[i+31:i] = 0
            FI
    FI;
ENDFOR
DEST[MAXVL-1:VL] ← 0
```

VXORPS (VEX.256 encoded version)

```
DEST[31:0] ← SRC1[31:0] BITWISE XOR SRC2[31:0]
DEST[63:32] ← SRC1[63:32] BITWISE XOR SRC2[63:32]
DEST[95:64] ← SRC1[95:64] BITWISE XOR SRC2[95:64]
DEST[127:96] ← SRC1[127:96] BITWISE XOR SRC2[127:96]
DEST[159:128] ← SRC1[159:128] BITWISE XOR SRC2[159:128]
DEST[191:160] ← SRC1[191:160] BITWISE XOR SRC2[191:160]
DEST[223:192] ← SRC1[223:192] BITWISE XOR SRC2[223:192]
DEST[255:224] ← SRC1[255:224] BITWISE XOR SRC2[255:224].
DEST[MAXVL-1:256] ← 0
```

VXORPS (VEX.128 encoded version)

```
DEST[31:0] ← SRC1[31:0] BITWISE XOR SRC2[31:0]
DEST[63:32] ← SRC1[63:32] BITWISE XOR SRC2[63:32]
DEST[95:64] ← SRC1[95:64] BITWISE XOR SRC2[95:64]
DEST[127:96] ← SRC1[127:96] BITWISE XOR SRC2[127:96]
DEST[MAXVL-1:128] ← 0
```

XORPS (128-bit Legacy SSE version)

```
DEST[31:0] ← SRC1[31:0] BITWISE XOR SRC2[31:0]
DEST[63:32] ← SRC1[63:32] BITWISE XOR SRC2[63:32]
DEST[95:64] ← SRC1[95:64] BITWISE XOR SRC2[95:64]
DEST[127:96] ← SRC1[127:96] BITWISE XOR SRC2[127:96]
DEST[MAXVL-1:128] (Unmodified)
```

Intel C/C++ Compiler Intrinsic Equivalent

```
VXORPS __m512 _mm512_xor_ps (__m512 a, __m512 b);
VXORPS __m512 _mm512_mask_xor_ps (__m512 a, __mmask16 m, __m512 b);
VXORPS __m512 _mm512_maskz_xor_ps (__mmask16 m, __m512 a);
VXORPS __m256 _mm256_xor_ps (__m256 a, __m256 b);
VXORPS __m256 _mm256_mask_xor_ps (__m256 a, __mmask8 m, __m256 b);
VXORPS __m256 _mm256_maskz_xor_ps (__mmask8 m, __m256 a);
XORPS __m128 _mm_xor_ps (__m128 a, __m128 b);
VXORPS __m128 _mm_mask_xor_ps (__m128 a, __mmask8 m, __m128 b);
```

VXORPS __m128 _mm_maskz_xor_ps (__mmask8 m, __m128 a);

SIMD Floating-Point Exceptions

None

Other Exceptions

Non-EVEX-encoded instructions, see Exceptions Type 4.
EVEX-encoded instructions, see Exceptions Type E4.

XRSTOR—Restore Processor Extended States

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
NP 0F AE /5	XRSTOR *mem*	M	Valid	Valid	Restore state components specified by EDX:EAX from *mem*.
NP REX.W + 0F AE /5	XRSTOR64 *mem*	M	Valid	N.E.	Restore state components specified by EDX:EAX from *mem*.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
M	ModRM:r/m (r)	NA	NA	NA

Description

Performs a full or partial restore of processor state components from the XSAVE area located at the memory address specified by the source operand. The implicit EDX:EAX register pair specifies a 64-bit instruction mask. The specific state components restored correspond to the bits set in the requested-feature bitmap (RFBM), which is the logical-AND of EDX:EAX and XCR0.

The format of the XSAVE area is detailed in Section 13.4, "XSAVE Area," of *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*.

Section 13.8, "Operation of XRSTOR," of *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1* provides a detailed description of the operation of the XRSTOR instruction. The following items provide a high-level outline:

- Execution of XRSTOR may take one of two forms: standard and compacted. Bit 63 of the XCOMP_BV field in the XSAVE header determines which form is used: value 0 specifies the standard form, while value 1 specifies the compacted form.
- If RFBM[*i*] = 0, XRSTOR does not update state component *i*.[1]
- If RFBM[*i*] = 1 and bit *i* is clear in the XSTATE_BV field in the XSAVE header, XRSTOR initializes state component *i*.
- If RFBM[*i*] = 1 and XSTATE_BV[*i*] = 1, XRSTOR loads state component *i* from the XSAVE area.
- The standard form of XRSTOR treats MXCSR (which is part of state component 1 — SSE) differently from the XMM registers. If either form attempts to load MXCSR with an illegal value, a general-protection exception (#GP) occurs.
- XRSTOR loads the internal value XRSTOR_INFO, which may be used to optimize a subsequent execution of XSAVEOPT or XSAVES.
- Immediately following an execution of XRSTOR, the processor tracks as in-use (not in initial configuration) any state component *i* for which RFBM[*i*] = 1 and XSTATE_BV[*i*] = 1; it tracks as modified any state component *i* for which RFBM[*i*] = 0.

Use of a source operand not aligned to 64-byte boundary (for 64-bit and 32-bit modes) results in a general-protection (#GP) exception. In 64-bit mode, the upper 32 bits of RDX and RAX are ignored.

See Section 13.6, "Processor Tracking of XSAVE-Managed State," of *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1* for discussion of the bitmaps XINUSE and XMODIFIED and of the quantity XRSTOR_INFO.

Operation

```
RFBM ← XCR0 AND EDX:EAX;   /* bitwise logical AND */
COMPMASK ← XCOMP_BV field from XSAVE header;
```

1. There is an exception if RFBM[1] = 0 and RFBM[2] = 1. In this case, the standard form of XRSTOR will load MXCSR from memory, even though MXCSR is part of state component 1 — SSE. The compacted form of XRSTOR does not make this exception.

```
RSTORMASK ← XSTATE_BV field from XSAVE header;

IF COMPMASK[63] = 0
    THEN
        /* Standard form of XRSTOR */
        TO_BE_RESTORED ← RFBM AND RSTORMASK;
        TO_BE_INITIALIZED ← RFBM AND NOT RSTORMASK;

        IF TO_BE_RESTORED[0] = 1
            THEN
                load x87 state from legacy region of XSAVE area;
                XINUSE[0] ← 1;
        ELSIF TO_BE_INITIALIZED[0] = 1
            THEN
                initialize x87 state;
                XINUSE[0] ← 0;
        FI;

        IF RFBM[1] = 1 OR RFBM[2] = 1
            THEN load MXCSR from legacy region of XSAVE area;
        FI;

        IF TO_BE_RESTORED[1] = 1
            THEN
                load XMM registers from legacy region of XSAVE area; // this step does not load MXCSR
                XINUSE[1] ← 1;
        ELSIF TO_BE_INITIALIZED[1] = 1
            THEN
                set all XMM registers to 0; // this step does not initialize MXCSR
                XINUSE[1] ← 0;
        FI;

        FOR i ← 2 TO 62
            IF TO_BE_RESTORED[i] = 1
                THEN
                    load XSAVE state component i at offset n from base of XSAVE area;
                        // n enumerated by CPUID(EAX=0DH,ECX=i):EBX)
                    XINUSE[i] ← 1;
            ELSIF TO_BE_INITIALIZED[i] = 1
                THEN
                    initialize XSAVE state component i;
                    XINUSE[i] ← 0;
            FI;
        ENDFOR;

    ELSE
        /* Compacted form of XRSTOR */
        IF CPUID.(EAX=0DH,ECX=1):EAX.XSAVEC[bit 1] = 0
            THEN    /* compacted form not supported */
                #GP(0);
        FI;

        FORMAT = COMPMASK AND 7FFFFFFF_FFFFFFFFH;
        RESTORE_FEATURES = FORMAT AND RFBM;
```

```
        TO_BE_RESTORED ← RESTORE_FEATURES AND RSTORMASK;
        FORCE_INIT ← RFBM AND NOT FORMAT;
        TO_BE_INITIALIZED = (RFBM AND NOT RSTORMASK) OR FORCE_INIT;

        IF TO_BE_RESTORED[0] = 1
              THEN
                    load x87 state from legacy region of XSAVE area;
                    XINUSE[0] ← 1;
        ELSIF TO_BE_INITIALIZED[0] = 1
              THEN
                    initialize x87 state;
                    XINUSE[0] ← 0;
        FI;

        IF TO_BE_RESTORED[1] = 1
              THEN
                    load SSE state from legacy region of XSAVE area; // this step loads the XMM registers and MXCSR
                    XINUSE[1] ← 1;
        ELSIF TO_BE_INITIALIZED[1] = 1
              THEN
                    set all XMM registers to 0;
                    MXCSR ← 1F80H;
                    XINUSE[1] ← 0;
        FI;

        NEXT_FEATURE_OFFSET = 576;          // Legacy area and XSAVE header consume 576 bytes
        FOR i ← 2 TO 62
              IF FORMAT[i] = 1
                    THEN
                          IF TO_BE_RESTORED[i] = 1
                                THEN
                                      load XSAVE state component i at offset NEXT_FEATURE_OFFSET from base of XSAVE area;
                                      XINUSE[i] ← 1;
                          FI;
                          NEXT_FEATURE_OFFSET = NEXT_FEATURE_OFFSET + n (n enumerated by CPUID(EAX=0DH,ECX=i):EAX);
              FI;
              IF TO_BE_INITIALIZED[i] = 1
                    THEN
                          initialize XSAVE state component i;
                          XINUSE[i] ← 0;
              FI;
        ENDFOR;
FI;

XMODIFIED_BV ← NOT RFBM;

IF in VMX non-root operation
     THEN VMXNR ← 1;
     ELSE VMXNR ← 0;
FI;
LAXA ← linear address of XSAVE area;
XRSTOR_INFO ← ⟨CPL,VMXNR,LAXA,COMPMASK⟩;
```

Flags Affected

None.

Intel C/C++ Compiler Intrinsic Equivalent

XRSTOR: void _xrstor(void * , unsigned __int64);

XRSTOR: void _xrstor64(void * , unsigned __int64);

Protected Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
	If a memory operand is not aligned on a 64-byte boundary, regardless of segment.
	If bit 63 of the XCOMP_BV field of the XSAVE header is 1 and CPUID.(EAX=0DH,ECX=1):EAX.XSAVEC[bit 1] = 0.
	If the standard form is executed and a bit in XCR0 is 0 and the corresponding bit in the XSTATE_BV field of the XSAVE header is 1.
	If the standard form is executed and bytes 23:8 of the XSAVE header are not all zero.
	If the compacted form is executed and a bit in XCR0 is 0 and the corresponding bit in the XCOMP_BV field of the XSAVE header is 1.
	If the compacted form is executed and a bit in the XCOMP_BV field in the XSAVE header is 0 and the corresponding bit in the XSTATE_BV field is 1.
	If the compacted form is executed and bytes 63:16 of the XSAVE header are not all zero.
	If attempting to write any reserved bits of the MXCSR register with 1.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#NM	If CR0.TS[bit 3] = 1.
#UD	If CPUID.01H:ECX.XSAVE[bit 26] = 0.
	If CR4.OSXSAVE[bit 18] = 0.
	If the LOCK prefix is used.
#AC	If this exception is disabled a general protection exception (#GP) is signaled if the memory operand is not aligned on a 16-byte boundary, as described above. If the alignment check exception (#AC) is enabled (and the CPL is 3), signaling of #AC is not guaranteed and may vary with implementation, as follows. In all implementations where #AC is not signaled, a general protection exception is signaled in its place. In addition, the width of the alignment check may also vary with implementation. For instance, for a given implementation, an alignment check exception might be signaled for a 2-byte misalignment, whereas a general protection exception might be signaled for all other misalignments (4-, 8-, or 16-byte misalignments).

Real-Address Mode Exceptions

#GP	If a memory operand is not aligned on a 64-byte boundary, regardless of segment.
	If any part of the operand lies outside the effective address space from 0 to FFFFH.
	If bit 63 of the XCOMP_BV field of the XSAVE header is 1 and CPUID.(EAX=0DH,ECX=1):EAX.XSAVEC[bit 1] = 0.
	If the standard form is executed and a bit in XCR0 is 0 and the corresponding bit in the XSTATE_BV field of the XSAVE header is 1.
	If the standard form is executed and bytes 23:8 of the XSAVE header are not all zero.
	If the compacted form is executed and a bit in XCR0 is 0 and the corresponding bit in the XCOMP_BV field of the XSAVE header is 1.
	If the compacted form is executed and a bit in the XCOMP_BV field in the XSAVE header is 0 and the corresponding bit in the XSTATE_BV field is 1.
	If the compacted form is executed and bytes 63:16 of the XSAVE header are not all zero.

	If attempting to write any reserved bits of the MXCSR register with 1.
#NM	If CR0.TS[bit 3] = 1.
#UD	If CPUID.01H:ECX.XSAVE[bit 26] = 0.
	If CR4.OSXSAVE[bit 18] = 0.
	If the LOCK prefix is used.

Virtual-8086 Mode Exceptions

Same exceptions as in protected mode

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#GP(0)	If a memory address is in a non-canonical form.
	If a memory operand is not aligned on a 64-byte boundary, regardless of segment.
	If bit 63 of the XCOMP_BV field of the XSAVE header is 1 and CPUID.(EAX=0DH,ECX=1):EAX.XSAVEC[bit 1] = 0.
	If the standard form is executed and a bit in XCR0 is 0 and the corresponding bit in the XSTATE_BV field of the XSAVE header is 1.
	If the standard form is executed and bytes 23:8 of the XSAVE header are not all zero.
	If the compacted form is executed and a bit in XCR0 is 0 and the corresponding bit in the XCOMP_BV field of the XSAVE header is 1.
	If the compacted form is executed and a bit in the XCOMP_BV field in the XSAVE header is 0 and the corresponding bit in the XSTATE_BV field is 1.
	If the compacted form is executed and bytes 63:16 of the XSAVE header are not all zero.
	If attempting to write any reserved bits of the MXCSR register with 1.
#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#PF(fault-code)	If a page fault occurs.
#NM	If CR0.TS[bit 3] = 1.
#UD	If CPUID.01H:ECX.XSAVE[bit 26] = 0.
	If CR4.OSXSAVE[bit 18] = 0.
	If the LOCK prefix is used.
#AC	If this exception is disabled a general protection exception (#GP) is signaled if the memory operand is not aligned on a 16-byte boundary, as described above. If the alignment check exception (#AC) is enabled (and the CPL is 3), signaling of #AC is not guaranteed and may vary with implementation, as follows. In all implementations where #AC is not signaled, a general protection exception is signaled in its place. In addition, the width of the alignment check may also vary with implementation. For instance, for a given implementation, an alignment check exception might be signaled for a 2-byte misalignment, whereas a general protection exception might be signaled for all other misalignments (4-, 8-, or 16-byte misalignments).

XRSTORS—Restore Processor Extended States Supervisor

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
NP 0F C7 /3	XRSTORS *mem*	M	Valid	Valid	Restore state components specified by EDX:EAX from *mem*.
NP REX.W + 0F C7 /3	XRSTORS64 *mem*	M	Valid	N.E.	Restore state components specified by EDX:EAX from *mem*.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
M	ModRM:r/m (r)	NA	NA	NA

Description

Performs a full or partial restore of processor state components from the XSAVE area located at the memory address specified by the source operand. The implicit EDX:EAX register pair specifies a 64-bit instruction mask. The specific state components restored correspond to the bits set in the requested-feature bitmap (RFBM), which is the logical-AND of EDX:EAX and the logical-OR of XCR0 with the IA32_XSS MSR. XRSTORS may be executed only if CPL = 0.

The format of the XSAVE area is detailed in Section 13.4, "XSAVE Area," of *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*.

Section 13.12, "Operation of XRSTORS," of *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1* provides a detailed description of the operation of the XRSTOR instruction. The following items provide a high-level outline:

- Execution of XRSTORS is similar to that of the compacted form of XRSTOR; XRSTORS cannot restore from an XSAVE area in which the extended region is in the standard format (see Section 13.4.3, "Extended Region of an XSAVE Area").
- XRSTORS differs from XRSTOR in that it can restore state components corresponding to bits set in the IA32_XSS MSR.
- If RFBM[*i*] = 0, XRSTORS does not update state component *i*.
- If RFBM[*i*] = 1 and bit *i* is clear in the XSTATE_BV field in the XSAVE header, XRSTORS initializes state component *i*.
- If RFBM[*i*] = 1 and XSTATE_BV[*i*] = 1, XRSTORS loads state component *i* from the XSAVE area.
- If XRSTORS attempts to load MXCSR with an illegal value, a general-protection exception (#GP) occurs.
- XRSTORS loads the internal value XRSTOR_INFO, which may be used to optimize a subsequent execution of XSAVEOPT or XSAVES.
- Immediately following an execution of XRSTORS, the processor tracks as in-use (not in initial configuration) any state component *i* for which RFBM[*i*] = 1 and XSTATE_BV[*i*] = 1; it tracks as modified any state component *i* for which RFBM[*i*] = 0.

Use of a source operand not aligned to 64-byte boundary (for 64-bit and 32-bit modes) results in a general-protection (#GP) exception. In 64-bit mode, the upper 32 bits of RDX and RAX are ignored.

See Section 13.6, "Processor Tracking of XSAVE-Managed State," of *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1* for discussion of the bitmaps XINUSE and XMODIFIED and of the quantity XRSTOR_INFO.

Operation

```
RFBM ← (XCR0 OR IA32_XSS) AND EDX:EAX;          /* bitwise logical OR and AND */
COMPMASK ← XCOMP_BV field from XSAVE header;
RSTORMASK ← XSTATE_BV field from XSAVE header;
```

```
FORMAT = COMPMASK AND 7FFFFFFF_FFFFFFFFH;
RESTORE_FEATURES = FORMAT AND RFBM;
TO_BE_RESTORED ← RESTORE_FEATURES AND RSTORMASK;
FORCE_INIT ← RFBM AND NOT FORMAT;
TO_BE_INITIALIZED = (RFBM AND NOT RSTORMASK) OR FORCE_INIT;

IF TO_BE_RESTORED[0] = 1
    THEN
        load x87 state from legacy region of XSAVE area;
        XINUSE[0] ← 1;
ELSIF TO_BE_INITIALIZED[0] = 1
    THEN
        initialize x87 state;
        XINUSE[0] ← 0;
FI;

IF TO_BE_RESTORED[1] = 1
    THEN
        load SSE state from legacy region of XSAVE area; // this step loads the XMM registers and MXCSR
        XINUSE[1] ← 1;
ELSIF TO_BE_INITIALIZED[1] = 1
    THEN
        set all XMM registers to 0;
        MXCSR ← 1F80H;
        XINUSE[1] ← 0;
FI;

NEXT_FEATURE_OFFSET = 576;          // Legacy area and XSAVE header consume 576 bytes
FOR i ← 2 TO 62
    IF FORMAT[i] = 1
        THEN
            IF TO_BE_RESTORED[i] = 1
                THEN
                    load XSAVE state component i at offset NEXT_FEATURE_OFFSET from base of XSAVE area;
                    XINUSE[i] ← 1;
            FI;
            NEXT_FEATURE_OFFSET = NEXT_FEATURE_OFFSET + n (n enumerated by CPUID(EAX=0DH,ECX=i):EAX);
    FI;
    IF TO_BE_INITIALIZED[i] = 1
        THEN
            initialize XSAVE state component i;
            XINUSE[i] ← 0;
    FI;
ENDFOR;

XMODIFIED_BV ← NOT RFBM;

IF in VMX non-root operation
    THEN VMXNR ← 1;
    ELSE VMXNR ← 0;
FI;
LAXA ← linear address of XSAVE area;
XRSTOR_INFO ← ⟨CPL,VMXNR,LAXA,COMPMASK⟩;
```

Flags Affected

None.

Intel C/C++ Compiler Intrinsic Equivalent

XRSTORS: void _xrstors(void * , unsigned __int64);

XRSTORS64: void _xrstors64(void * , unsigned __int64);

Protected Mode Exceptions

#GP(0)	If CPL > 0.
	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
	If a memory operand is not aligned on a 64-byte boundary, regardless of segment.
	If bit 63 of the XCOMP_BV field of the XSAVE header is 0.
	If a bit in XCR0 is 0 and the corresponding bit in the XCOMP_BV field of the XSAVE header is 1.
	If a bit in the XCOMP_BV field in the XSAVE header is 0 and the corresponding bit in the XSTATE_BV field is 1.
	If bytes 63:16 of the XSAVE header are not all zero.
	If attempting to write any reserved bits of the MXCSR register with 1.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#NM	If CR0.TS[bit 3] = 1.
#UD	If CPUID.01H:ECX.XSAVE[bit 26] = 0 or CPUID.(EAX=0DH,ECX=1):EAX.XSS[bit 3] = 0.
	If CR4.OSXSAVE[bit 18] = 0.
	If the LOCK prefix is used.
#AC	If this exception is disabled a general protection exception (#GP) is signaled if the memory operand is not aligned on a 16-byte boundary, as described above. If the alignment check exception (#AC) is enabled (and the CPL is 3), signaling of #AC is not guaranteed and may vary with implementation, as follows. In all implementations where #AC is not signaled, a #GP is signaled in its place. In addition, the width of the alignment check may also vary with implementation. For instance, for a given implementation, an alignment check exception might be signaled for a 2-byte misalignment, whereas a #GP might be signaled for all other misalignments (4-, 8-, or 16-byte misalignments).

Real-Address Mode Exceptions

#GP	If a memory operand is not aligned on a 64-byte boundary, regardless of segment.
	If any part of the operand lies outside the effective address space from 0 to FFFFH.
	If bit 63 of the XCOMP_BV field of the XSAVE header is 0.
	If a bit in XCR0 is 0 and the corresponding bit in the XCOMP_BV field of the XSAVE header is 1.
	If a bit in the XCOMP_BV field in the XSAVE header is 0 and the corresponding bit in the XSTATE_BV field is 1.
	If bytes 63:16 of the XSAVE header are not all zero.
	If attempting to write any reserved bits of the MXCSR register with 1.
#NM	If CR0.TS[bit 3] = 1.
#UD	If CPUID.01H:ECX.XSAVE[bit 26] = 0 or CPUID.(EAX=0DH,ECX=1):EAX.XSS[bit 3] = 0.
	If CR4.OSXSAVE[bit 18] = 0.
	If the LOCK prefix is used.

Virtual-8086 Mode Exceptions

Same exceptions as in protected mode.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#GP(0)	If CPL > 0.
	If a memory address is in a non-canonical form.
	If a memory operand is not aligned on a 64-byte boundary, regardless of segment.
	If bit 63 of the XCOMP_BV field of the XSAVE header is 0.
	If a bit in XCR0 is 0 and the corresponding bit in the XCOMP_BV field of the XSAVE header is 1.
	If a bit in the XCOMP_BV field in the XSAVE header is 0 and the corresponding bit in the XSTATE_BV field is 1.
	If bytes 63:16 of the XSAVE header are not all zero.
	If attempting to write any reserved bits of the MXCSR register with 1.
#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#PF(fault-code)	If a page fault occurs.
#NM	If CR0.TS[bit 3] = 1.
#UD	If CPUID.01H:ECX.XSAVE[bit 26] = 0 or CPUID.(EAX=0DH,ECX=1):EAX.XSS[bit 3] = 0.
	If CR4.OSXSAVE[bit 18] = 0.
	If the LOCK prefix is used.
#AC	If this exception is disabled a general protection exception (#GP) is signaled if the memory operand is not aligned on a 16-byte boundary, as described above. If the alignment check exception (#AC) is enabled (and the CPL is 3), signaling of #AC is not guaranteed and may vary with implementation, as follows. In all implementations where #AC is not signaled, a general protection exception is signaled in its place. In addition, the width of the alignment check may also vary with implementation. For instance, for a given implementation, an alignment check exception might be signaled for a 2-byte misalignment, whereas a general protection exception might be signaled for all other misalignments (4-, 8-, or 16-byte misalignments).

XSAVE—Save Processor Extended States

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
NP 0F AE /4	XSAVE *mem*	M	Valid	Valid	Save state components specified by EDX:EAX to *mem*.
NP REX.W + 0F AE /4	XSAVE64 *mem*	M	Valid	N.E.	Save state components specified by EDX:EAX to *mem*.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
M	ModRM:r/m (w)	NA	NA	NA

Description

Performs a full or partial save of processor state components to the XSAVE area located at the memory address specified by the destination operand. The implicit EDX:EAX register pair specifies a 64-bit instruction mask. The specific state components saved correspond to the bits set in the requested-feature bitmap (RFBM), which is the logical-AND of EDX:EAX and XCR0.

The format of the XSAVE area is detailed in Section 13.4, "XSAVE Area," of *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*.

Section 13.7, "Operation of XSAVE," of *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1* provides a detailed description of the operation of the XSAVE instruction. The following items provide a high-level outline:

- XSAVE saves state component *i* if and only if RFBM[*i*] = 1.[1]
- XSAVE does not modify bytes 511:464 of the legacy region of the XSAVE area (see Section 13.4.1, "Legacy Region of an XSAVE Area").
- XSAVE reads the XSTATE_BV field of the XSAVE header (see Section 13.4.2, "XSAVE Header") and writes a modified value back to memory as follows. If RFBM[*i*] = 1, XSAVE writes XSTATE_BV[*i*] with the value of XINUSE[*i*]. (XINUSE is a bitmap by which the processor tracks the status of various state components. See Section 13.6, "Processor Tracking of XSAVE-Managed State.") If RFBM[*i*] = 0, XSAVE writes XSTATE_BV[*i*] with the value that it read from memory (it does not modify the bit). XSAVE does not write to any part of the XSAVE header other than the XSTATE_BV field.
- XSAVE always uses the standard format of the extended region of the XSAVE area (see Section 13.4.3, "Extended Region of an XSAVE Area").

Use of a destination operand not aligned to 64-byte boundary (in either 64-bit or 32-bit modes) results in a general-protection (#GP) exception. In 64-bit mode, the upper 32 bits of RDX and RAX are ignored.

Operation

```
RFBM ← XCR0 AND EDX:EAX;   /* bitwise logical AND */
OLD_BV ← XSTATE_BV field from XSAVE header;

IF RFBM[0] = 1
    THEN store x87 state into legacy region of XSAVE area;
FI;

IF RFBM[1] = 1
    THEN store XMM registers into legacy region of XSAVE area; // this step does not save MXCSR or MXCSR_MASK
```

1. An exception is made for MXCSR and MXCSR_MASK, which belong to state component 1 — SSE. XSAVE saves these values to memory if either RFBM[1] or RFBM[2] is 1.

```
FI;

IF RFBM[1] = 1 OR RFBM[2] = 1
    THEN store MXCSR and MXCSR_MASK into legacy region of XSAVE area;
FI;

FOR i ← 2 TO 62
    IF RFBM[i] = 1
        THEN save XSAVE state component i at offset n from base of XSAVE area (n enumerated by CPUID(EAX=0DH,ECX=i):EBX);
    FI;
ENDFOR;

XSTATE_BV field in XSAVE header ← (OLD_BV AND NOT RFBM) OR (XINUSE AND RFBM);
```

Flags Affected

None.

Intel C/C++ Compiler Intrinsic Equivalent

XSAVE: void _xsave(void * , unsigned __int64);

XSAVE: void _xsave64(void * , unsigned __int64);

Protected Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
	If a memory operand is not aligned on a 64-byte boundary, regardless of segment.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#NM	If CR0.TS[bit 3] = 1.
#UD	If CPUID.01H:ECX.XSAVE[bit 26] = 0.
	If CR4.OSXSAVE[bit 18] = 0.
	If the LOCK prefix is used.
#AC	If this exception is disabled a general protection exception (#GP) is signaled if the memory operand is not aligned on a 16-byte boundary, as described above. If the alignment check exception (#AC) is enabled (and the CPL is 3), signaling of #AC is not guaranteed and may vary with implementation, as follows. In all implementations where #AC is not signaled, a general protection exception is signaled in its place. In addition, the width of the alignment check may also vary with implementation. For instance, for a given implementation, an alignment check exception might be signaled for a 2-byte misalignment, whereas a general protection exception might be signaled for all other misalignments (4-, 8-, or 16-byte misalignments).

Real-Address Mode Exceptions

#GP	If a memory operand is not aligned on a 64-byte boundary, regardless of segment.
	If any part of the operand lies outside the effective address space from 0 to FFFFH.
#NM	If CR0.TS[bit 3] = 1.
#UD	If CPUID.01H:ECX.XSAVE[bit 26] = 0.
	If CR4.OSXSAVE[bit 18] = 0.
	If the LOCK prefix is used.

Virtual-8086 Mode Exceptions

Same exceptions as in protected mode.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#GP(0)	If the memory address is in a non-canonical form.
	If a memory operand is not aligned on a 64-byte boundary, regardless of segment.
#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#PF(fault-code)	If a page fault occurs.
#NM	If CR0.TS[bit 3] = 1.
#UD	If CPUID.01H:ECX.XSAVE[bit 26] = 0.
	If CR4.OSXSAVE[bit 18] = 0.
	If the LOCK prefix is used.
#AC	If this exception is disabled a general protection exception (#GP) is signaled if the memory operand is not aligned on a 16-byte boundary, as described above. If the alignment check exception (#AC) is enabled (and the CPL is 3), signaling of #AC is not guaranteed and may vary with implementation, as follows. In all implementations where #AC is not signaled, a general protection exception is signaled in its place. In addition, the width of the alignment check may also vary with implementation. For instance, for a given implementation, an alignment check exception might be signaled for a 2-byte misalignment, whereas a general protection exception might be signaled for all other misalignments (4-, 8-, or 16-byte misalignments).

XSAVEC—Save Processor Extended States with Compaction

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
NP 0F C7 /4	XSAVEC *mem*	M	Valid	Valid	Save state components specified by EDX:EAX to *mem* with compaction.
NP REX.W + 0F C7 /4	XSAVEC64 *mem*	M	Valid	N.E.	Save state components specified by EDX:EAX to *mem* with compaction.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
M	ModRM:r/m (w)	NA	NA	NA

Description

Performs a full or partial save of processor state components to the XSAVE area located at the memory address specified by the destination operand. The implicit EDX:EAX register pair specifies a 64-bit instruction mask. The specific state components saved correspond to the bits set in the requested-feature bitmap (RFBM), which is the logical-AND of EDX:EAX and XCR0.

The format of the XSAVE area is detailed in Section 13.4, "XSAVE Area," of *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*.

Section 13.10, "Operation of XSAVEC," of *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1* provides a detailed description of the operation of the XSAVEC instruction. The following items provide a high-level outline:

- Execution of XSAVEC is similar to that of XSAVE. XSAVEC differs from XSAVE in that it uses compaction and that it may use the init optimization.
- XSAVEC saves state component *i* if and only if RFBM[*i*] = 1 and XINUSE[*i*] = 1.[1] (XINUSE is a bitmap by which the processor tracks the status of various state components. See Section 13.6, "Processor Tracking of XSAVE-Managed State.")
- XSAVEC does not modify bytes 511:464 of the legacy region of the XSAVE area (see Section 13.4.1, "Legacy Region of an XSAVE Area").
- XSAVEC writes the logical AND of RFBM and XINUSE to the XSTATE_BV field of the XSAVE header.[2,3] (See Section 13.4.2, "XSAVE Header.") XSAVEC sets bit 63 of the XCOMP_BV field and sets bits 62:0 of that field to RFBM[62:0]. XSAVEC does not write to any parts of the XSAVE header other than the XSTATE_BV and XCOMP_BV fields.
- XSAVEC always uses the compacted format of the extended region of the XSAVE area (see Section 13.4.3, "Extended Region of an XSAVE Area").

Use of a destination operand not aligned to 64-byte boundary (in either 64-bit or 32-bit modes) results in a general-protection (#GP) exception. In 64-bit mode, the upper 32 bits of RDX and RAX are ignored.

Operation

```
RFBM ← XCR0 AND EDX:EAX;              /* bitwise logical AND */
TO_BE_SAVED ← RFBM AND XINUSE;        /* bitwise logical AND */
If MXCSR ≠ 1F80H AND RFBM[1]
    TO_BE_SAVED[1] = 1;
```

1. There is an exception for state component 1 (SSE). MXCSR is part of SSE state, but XINUSE[1] may be 0 even if MXCSR does not have its initial value of 1F80H. In this case, XSAVEC saves SSE state as long as RFBM[1] = 1.
2. Unlike XSAVE and XSAVEOPT, XSAVEC clears bits in the XSTATE_BV field that correspond to bits that are clear in RFBM.
3. There is an exception for state component 1 (SSE). MXCSR is part of SSE state, but XINUSE[1] may be 0 even if MXCSR does not have its initial value of 1F80H. In this case, XSAVEC sets XSTATE_BV[1] to 1 as long as RFBM[1] = 1.

```
FI;

IF TO_BE_SAVED[0] = 1
    THEN store x87 state into legacy region of XSAVE area;
FI;

IF TO_BE_SAVED[1] = 1
    THEN store SSE state into legacy region of XSAVE area; // this step saves the XMM registers, MXCSR, and MXCSR_MASK
FI;

NEXT_FEATURE_OFFSET = 576;          // Legacy area and XSAVE header consume 576 bytes
FOR i ← 2 TO 62
    IF RFBM[i] = 1
        THEN
            IF TO_BE_SAVED[i]
                THEN save XSAVE state component i at offset NEXT_FEATURE_OFFSET from base of XSAVE area;
            FI;
            NEXT_FEATURE_OFFSET = NEXT_FEATURE_OFFSET + n (n enumerated by CPUID(EAX=0DH,ECX=i):EAX);
    FI;
ENDFOR;

XSTATE_BV field in XSAVE header ← TO_BE_SAVED;
XCOMP_BV field in XSAVE header ← RFBM OR 80000000_00000000H;
```

Flags Affected

None.

Intel C/C++ Compiler Intrinsic Equivalent

XSAVEC: void _xsavec(void * , unsigned __int64);

XSAVEC64: void _xsavec64(void * , unsigned __int64);

Protected Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
	If a memory operand is not aligned on a 64-byte boundary, regardless of segment.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#NM	If CR0.TS[bit 3] = 1.
#UD	If CPUID.01H:ECX.XSAVE[bit 26] = 0 or CPUID.(EAX=0DH,ECX=1):EAX.XSAVEC[bit 1] = 0.
	If CR4.OSXSAVE[bit 18] = 0.
	If the LOCK prefix is used.
#AC	If this exception is disabled a general protection exception (#GP) is signaled if the memory operand is not aligned on a 16-byte boundary, as described above. If the alignment check exception (#AC) is enabled (and the CPL is 3), signaling of #AC is not guaranteed and may vary with implementation, as follows. In all implementations where #AC is not signaled, a general protection exception is signaled in its place. In addition, the width of the alignment check may also vary with implementation. For instance, for a given implementation, an alignment check exception might be signaled for a 2-byte misalignment, whereas a general protection exception might be signaled for all other misalignments (4-, 8-, or 16-byte misalignments).

Real-Address Mode Exceptions

#GP	If a memory operand is not aligned on a 64-byte boundary, regardless of segment. If any part of the operand lies outside the effective address space from 0 to FFFFH.
#NM	If CR0.TS[bit 3] = 1.
#UD	If CPUID.01H:ECX.XSAVE[bit 26] = 0 or CPUID.(EAX=0DH,ECX=1):EAX.XSAVEC[bit 1] = 0. If CR4.OSXSAVE[bit 18] = 0. If the LOCK prefix is used.

Virtual-8086 Mode Exceptions

Same exceptions as in protected mode.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#GP(0)	If the memory address is in a non-canonical form. If a memory operand is not aligned on a 64-byte boundary, regardless of segment.
#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#PF(fault-code)	If a page fault occurs.
#NM	If CR0.TS[bit 3] = 1.
#UD	If CPUID.01H:ECX.XSAVE[bit 26] = 0 or CPUID.(EAX=0DH,ECX=1):EAX.XSAVEC[bit 1] = 0. If CR4.OSXSAVE[bit 18] = 0. If the LOCK prefix is used.
#AC	If this exception is disabled a general protection exception (#GP) is signaled if the memory operand is not aligned on a 16-byte boundary, as described above. If the alignment check exception (#AC) is enabled (and the CPL is 3), signaling of #AC is not guaranteed and may vary with implementation, as follows. In all implementations where #AC is not signaled, a general protection exception is signaled in its place. In addition, the width of the alignment check may also vary with implementation. For instance, for a given implementation, an alignment check exception might be signaled for a 2-byte misalignment, whereas a general protection exception might be signaled for all other misalignments (4-, 8-, or 16-byte misalignments).

XSAVEOPT—Save Processor Extended States Optimized

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F AE /6 XSAVEOPT *mem*	M	V/V	XSAVEOPT	Save state components specified by EDX:EAX to *mem*, optimizing if possible.
NP REX.W + 0F AE /6 XSAVEOPT64 *mem*	M	V/V	XSAVEOPT	Save state components specified by EDX:EAX to *mem*, optimizing if possible.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
M	ModRM:r/m (w)	NA	NA	NA

Description

Performs a full or partial save of processor state components to the XSAVE area located at the memory address specified by the destination operand. The implicit EDX:EAX register pair specifies a 64-bit instruction mask. The specific state components saved correspond to the bits set in the requested-feature bitmap (RFBM), which is the logical-AND of EDX:EAX and XCR0.

The format of the XSAVE area is detailed in Section 13.4, "XSAVE Area," of *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*.

Section 13.9, "Operation of XSAVEOPT," of *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1* provides a detailed description of the operation of the XSAVEOPT instruction. The following items provide a high-level outline:

- Execution of XSAVEOPT is similar to that of XSAVE. XSAVEOPT differs from XSAVE in that it may use the init and modified optimizations. The performance of XSAVEOPT will be equal to or better than that of XSAVE.
- XSAVEOPT saves state component *i* only if RFBM[*i*] = 1 and XINUSE[*i*] = 1.[1] (XINUSE is a bitmap by which the processor tracks the status of various state components. See Section 13.6, "Processor Tracking of XSAVE-Managed State.") Even if both bits are 1, XSAVEOPT may optimize and not save state component *i* if (1) state component *i* has not been modified since the last execution of XRSTOR or XRSTORS; and (2) this execution of XSAVES corresponds to that last execution of XRSTOR or XRSTORS as determined by the internal value XRSTOR_INFO (see the Operation section below).
- XSAVEOPT does not modify bytes 511:464 of the legacy region of the XSAVE area (see Section 13.4.1, "Legacy Region of an XSAVE Area").
- XSAVEOPT reads the XSTATE_BV field of the XSAVE header (see Section 13.4.2, "XSAVE Header") and writes a modified value back to memory as follows. If RFBM[*i*] = 1, XSAVEOPT writes XSTATE_BV[*i*] with the value of XINUSE[*i*]. If RFBM[*i*] = 0, XSAVEOPT writes XSTATE_BV[*i*] with the value that it read from memory (it does not modify the bit). XSAVEOPT does not write to any part of the XSAVE header other than the XSTATE_BV field.
- XSAVEOPT always uses the standard format of the extended region of the XSAVE area (see Section 13.4.3, "Extended Region of an XSAVE Area").

Use of a destination operand not aligned to 64-byte boundary (in either 64-bit or 32-bit modes) will result in a general-protection (#GP) exception. In 64-bit mode, the upper 32 bits of RDX and RAX are ignored.

See Section 13.6, "Processor Tracking of XSAVE-Managed State," of *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1* for discussion of the bitmap XMODIFIED and of the quantity XRSTOR_INFO.

Operation

```
RFBM ← XCR0 AND EDX:EAX;   /* bitwise logical AND */
```

1. There is an exception made for MXCSR and MXCSR_MASK, which belong to state component 1 — SSE. XSAVEOPT always saves these to memory if RFBM[1] = 1 or RFBM[2] = 1, regardless of the value of XINUSE.

```
OLD_BV ← XSTATE_BV field from XSAVE header;
TO_BE_SAVED ← RFBM AND XINUSE;

IF in VMX non-root operation
    THEN VMXNR ← 1;
    ELSE VMXNR ← 0;
FI;
LAXA ← linear address of XSAVE area;
IF XRSTOR_INFO = ⟨CPL,VMXNR,LAXA,00000000_00000000H⟩
    THEN TO_BE_SAVED ← TO_BE_SAVED AND XMODIFIED;
FI;

IF TO_BE_SAVED[0] = 1
    THEN store x87 state into legacy region of XSAVE area;
FI;

IF TO_BE_SAVED[1]
    THEN store XMM registers into legacy region of XSAVE area; // this step does not save MXCSR or MXCSR_MASK
FI;

IF RFBM[1] = 1 or RFBM[2] = 1
    THEN store MXCSR and MXCSR_MASK into legacy region of XSAVE area;
FI;

FOR i ← 2 TO 62
    IF TO_BE_SAVED[i] = 1
        THEN save XSAVE state component i at offset n from base of XSAVE area (n enumerated by CPUID(EAX=0DH,ECX=i):EBX);
    FI;
ENDFOR;

XSTATE_BV field in XSAVE header ← (OLD_BV AND NOT RFBM) OR (XINUSE AND RFBM);
```

Flags Affected

None.

Intel C/C++ Compiler Intrinsic Equivalent

XSAVEOPT: void _xsaveopt(void * , unsigned __int64);

XSAVEOPT: void _xsaveopt64(void * , unsigned __int64);

Protected Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
	If a memory operand is not aligned on a 64-byte boundary, regardless of segment.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#NM	If CR0.TS[bit 3] = 1.
#UD	If CPUID.01H:ECX.XSAVE[bit 26] = 0 or CPUID.(EAX=0DH,ECX=1):EAX.XSAVEOPT[bit 0] = 0.
	If CR4.OSXSAVE[bit 18] = 0.
	If the LOCK prefix is used.

Real-Address Mode Exceptions

#GP	If a memory operand is not aligned on a 64-byte boundary, regardless of segment.
	If any part of the operand lies outside the effective address space from 0 to FFFFH.
#NM	If CR0.TS[bit 3] = 1.
#UD	If CPUID.01H:ECX.XSAVE[bit 26] = 0 or CPUID.(EAX=0DH,ECX=1):EAX.XSAVEOPT[bit 0] = 0.
	If CR4.OSXSAVE[bit 18] = 0.
	If the LOCK prefix is used.

Virtual-8086 Mode Exceptions

Same exceptions as in protected mode.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#GP(0)	If the memory address is in a non-canonical form.
	If a memory operand is not aligned on a 64-byte boundary, regardless of segment.
#PF(fault-code)	If a page fault occurs.
#NM	If CR0.TS[bit 3] = 1.
#UD	If CPUID.01H:ECX.XSAVE[bit 26] = 0 or CPUID.(EAX=0DH,ECX=1):EAX.XSAVEOPT[bit 0] = 0.
	If CR4.OSXSAVE[bit 18] = 0.
	If the LOCK prefix is used.

XSAVES—Save Processor Extended States Supervisor

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
NP 0F C7 /5	XSAVES *mem*	M	Valid	Valid	Save state components specified by EDX:EAX to *mem* with compaction, optimizing if possible.
NP REX.W + 0F C7 /5	XSAVES64 *mem*	M	Valid	N.E.	Save state components specified by EDX:EAX to *mem* with compaction, optimizing if possible.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
M	ModRM:r/m (w)	NA	NA	NA

Description

Performs a full or partial save of processor state components to the XSAVE area located at the memory address specified by the destination operand. The implicit EDX:EAX register pair specifies a 64-bit instruction mask. The specific state components saved correspond to the bits set in the requested-feature bitmap (RFBM), the logical-AND of EDX:EAX and the logical-OR of XCR0 with the IA32_XSS MSR. XSAVES may be executed only if CPL = 0.

The format of the XSAVE area is detailed in Section 13.4, "XSAVE Area," of *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*.

Section 13.11, "Operation of XSAVES," of *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1* provides a detailed description of the operation of the XSAVES instruction. The following items provide a high-level outline:

- Execution of XSAVES is similar to that of XSAVEC. XSAVES differs from XSAVEC in that it can save state components corresponding to bits set in the IA32_XSS MSR and that it may use the modified optimization.
- XSAVES saves state component *i* only if RFBM[*i*] = 1 and XINUSE[*i*] = 1.[1] (XINUSE is a bitmap by which the processor tracks the status of various state components. See Section 13.6, "Processor Tracking of XSAVE-Managed State.") Even if both bits are 1, XSAVES may optimize and not save state component *i* if (1) state component *i* has not been modified since the last execution of XRSTOR or XRSTORS; and (2) this execution of XSAVES correspond to that last execution of XRSTOR or XRSTORS as determined by XRSTOR_INFO (see the Operation section below).
- XSAVES does not modify bytes 511:464 of the legacy region of the XSAVE area (see Section 13.4.1, "Legacy Region of an XSAVE Area").
- XSAVES writes the logical AND of RFBM and XINUSE to the XSTATE_BV field of the XSAVE header.[2] (See Section 13.4.2, "XSAVE Header.") XSAVES sets bit 63 of the XCOMP_BV field and sets bits 62:0 of that field to RFBM[62:0]. XSAVES does not write to any parts of the XSAVE header other than the XSTATE_BV and XCOMP_BV fields.
- XSAVES always uses the compacted format of the extended region of the XSAVE area (see Section 13.4.3, "Extended Region of an XSAVE Area").

Use of a destination operand not aligned to 64-byte boundary (in either 64-bit or 32-bit modes) results in a general-protection (#GP) exception. In 64-bit mode, the upper 32 bits of RDX and RAX are ignored.

See Section 13.6, "Processor Tracking of XSAVE-Managed State," of *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1* for discussion of the bitmap XMODIFIED and of the quantity XRSTOR_INFO.

---

1. There is an exception for state component 1 (SSE). MXCSR is part of SSE state, but XINUSE[1] may be 0 even if MXCSR does not have its initial value of 1F80H. In this case, the init optimization does not apply and XSAVEC will save SSE state as long as RFBM[1] = 1 and the modified optimization is not being applied.
2. There is an exception for state component 1 (SSE). MXCSR is part of SSE state, but XINUSE[1] may be 0 even if MXCSR does not have its initial value of 1F80H. In this case, XSAVES sets XSTATE_BV[1] to 1 as long as RFBM[1] = 1.

Operation

```
RFBM ← (XCR0 OR IA32_XSS) AND EDX:EAX;              /* bitwise logical OR and AND */
IF in VMX non-root operation
    THEN VMXNR ← 1;
    ELSE VMXNR ← 0;
FI;
LAXA ← linear address of XSAVE area;
COMPMASK ← RFBM OR 80000000_00000000H;
TO_BE_SAVED ← RFBM AND XINUSE;
IF XRSTOR_INFO = ⟨CPL,VMXNR,LAXA,COMPMASK⟩
    THEN TO_BE_SAVED ← TO_BE_SAVED AND XMODIFIED;
FI;
If MXCSR ≠ 1F80H AND RFBM[1]
    TO_BE_SAVED[1] = 1;
FI;

IF TO_BE_SAVED[0] = 1
    THEN store x87 state into legacy region of XSAVE area;
FI;

IF TO_BE_SAVED[1] = 1
    THEN store SSE state into legacy region of XSAVE area; // this step saves the XMM registers, MXCSR, and MXCSR_MASK
FI;

NEXT_FEATURE_OFFSET = 576;          // Legacy area and XSAVE header consume 576 bytes
FOR i ← 2 TO 62
    IF RFBM[i] = 1
        THEN
            IF TO_BE_SAVED[i]
                THEN
                    save XSAVE state component i at offset NEXT_FEATURE_OFFSET from base of XSAVE area;
                    IF i = 8        // state component 8 is for PT state
                        THEN IA32_RTIT_CTL.TraceEn[bit 0] ← 0;
                    FI;
            FI;
            NEXT_FEATURE_OFFSET = NEXT_FEATURE_OFFSET + n (n enumerated by CPUID(EAX=0DH,ECX=i):EAX);
    FI;
ENDFOR;

XSTATE_BV field in XSAVE header ← TO_BE_SAVED;
XCOMP_BV field in XSAVE header ← COMPMASK;
```

Flags Affected

None.

Intel C/C++ Compiler Intrinsic Equivalent

XSAVES: void _xsaves(void * , unsigned __int64);

XSAVES64: void _xsaves64(void * , unsigned __int64);

Protected Mode Exceptions

#GP(0)	If CPL > 0.
	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
	If a memory operand is not aligned on a 64-byte boundary, regardless of segment.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#NM	If CR0.TS[bit 3] = 1.
#UD	If CPUID.01H:ECX.XSAVE[bit 26] = 0 or CPUID.(EAX=0DH,ECX=1):EAX.XSS[bit 3] = 0.
	If CR4.OSXSAVE[bit 18] = 0.
	If the LOCK prefix is used.
#AC	If this exception is disabled a general protection exception (#GP) is signaled if the memory operand is not aligned on a 16-byte boundary, as described above. If the alignment check exception (#AC) is enabled (and the CPL is 3), signaling of #AC is not guaranteed and may vary with implementation, as follows. In all implementations where #AC is not signaled, a general protection exception is signaled in its place. In addition, the width of the alignment check may also vary with implementation. For instance, for a given implementation, an alignment check exception might be signaled for a 2-byte misalignment, whereas a general protection exception might be signaled for all other misalignments (4-, 8-, or 16-byte misalignments).

Real-Address Mode Exceptions

#GP	If a memory operand is not aligned on a 64-byte boundary, regardless of segment.
	If any part of the operand lies outside the effective address space from 0 to FFFFH.
#NM	If CR0.TS[bit 3] = 1.
#UD	If CPUID.01H:ECX.XSAVE[bit 26] = 0 or CPUID.(EAX=0DH,ECX=1):EAX.XSS[bit 3] = 0.
	If CR4.OSXSAVE[bit 18] = 0.
	If the LOCK prefix is used.

Virtual-8086 Mode Exceptions

Same exceptions as in protected mode.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#GP(0)	If CPL > 0.
	If the memory address is in a non-canonical form.
	If a memory operand is not aligned on a 64-byte boundary, regardless of segment.
#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#PF(fault-code)	If a page fault occurs.
#NM	If CR0.TS[bit 3] = 1.
#UD	If CPUID.01H:ECX.XSAVE[bit 26] = 0 or CPUID.(EAX=0DH,ECX=1):EAX.XSS[bit 3] = 0.
	If CR4.OSXSAVE[bit 18] = 0.
	If the LOCK prefix is used.
#AC	If this exception is disabled a general protection exception (#GP) is signaled if the memory operand is not aligned on a 16-byte boundary, as described above. If the alignment check exception (#AC) is enabled (and the CPL is 3), signaling of #AC is not guaranteed and may vary with implementation, as follows. In all implementations where #AC is not signaled, a general protection exception is signaled in its place. In addition, the width of the alignment check may also vary with implementation. For instance, for a given implementation, an align-

ment check exception might be signaled for a 2-byte misalignment, whereas a general protection exception might be signaled for all other misalignments (4-, 8-, or 16-byte misalignments).

XSETBV—Set Extended Control Register

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
NP 0F 01 D1	XSETBV	ZO	Valid	Valid	Write the value in EDX:EAX to the XCR specified by ECX.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
ZO	NA	NA	NA	NA

Description

Writes the contents of registers EDX:EAX into the 64-bit extended control register (XCR) specified in the ECX register. (On processors that support the Intel 64 architecture, the high-order 32 bits of RCX are ignored.) The contents of the EDX register are copied to high-order 32 bits of the selected XCR and the contents of the EAX register are copied to low-order 32 bits of the XCR. (On processors that support the Intel 64 architecture, the high-order 32 bits of each of RAX and RDX are ignored.) Undefined or reserved bits in an XCR should be set to values previously read.

This instruction must be executed at privilege level 0 or in real-address mode; otherwise, a general protection exception #GP(0) is generated. Specifying a reserved or unimplemented XCR in ECX will also cause a general protection exception. The processor will also generate a general protection exception if software attempts to write to reserved bits in an XCR.

Currently, only XCR0 is supported. Thus, all other values of ECX are reserved and will cause a #GP(0). Note that bit 0 of XCR0 (corresponding to x87 state) must be set to 1; the instruction will cause a #GP(0) if an attempt is made to clear this bit. In addition, the instruction causes a #GP(0) if an attempt is made to set XCR0[2] (AVX state) while clearing XCR0[1] (SSE state); it is necessary to set both bits to use AVX instructions; Section 13.3, "Enabling the XSAVE Feature Set and XSAVE-Enabled Features," of *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*.

Operation

```
XCR[ECX] ← EDX:EAX;
```

Flags Affected

None.

Intel C/C++ Compiler Intrinsic Equivalent

XSETBV: void _xsetbv(unsigned int, unsigned __int64);

Protected Mode Exceptions

#GP(0)	If the current privilege level is not 0.
	If an invalid XCR is specified in ECX.
	If the value in EDX:EAX sets bits that are reserved in the XCR specified by ECX.
	If an attempt is made to clear bit 0 of XCR0.
	If an attempt is made to set XCR0[2:1] to 10b.
#UD	If CPUID.01H:ECX.XSAVE[bit 26] = 0.
	If CR4.OSXSAVE[bit 18] = 0.
	If the LOCK prefix is used.

Real-Address Mode Exceptions

#GP	If an invalid XCR is specified in ECX.
	If the value in EDX:EAX sets bits that are reserved in the XCR specified by ECX.
	If an attempt is made to clear bit 0 of XCR0.
	If an attempt is made to set XCR0[2:1] to 10b.
#UD	If CPUID.01H:ECX.XSAVE[bit 26] = 0.
	If CR4.OSXSAVE[bit 18] = 0.
	If the LOCK prefix is used.

Virtual-8086 Mode Exceptions

#GP(0)	The XSETBV instruction is not recognized in virtual-8086 mode.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

Same exceptions as in protected mode.

XTEST — Test If In Transactional Execution

Opcode/Instruction	Op/ En	64/32bit Mode Support	CPUID Feature Flag	Description
NP 0F 01 D6 XTEST	A	V/V	HLE or RTM	Test if executing in a transactional region

Instruction Operand Encoding

Op/En	Operand 1	Operand2	Operand3	Operand4
A	NA	NA	NA	NA

Description

The XTEST instruction queries the transactional execution status. If the instruction executes inside a transactionally executing RTM region or a transactionally executing HLE region, then the ZF flag is cleared, else it is set.

Operation

XTEST

```
IF (RTM_ACTIVE = 1 OR HLE_ACTIVE = 1)
    THEN
        ZF ← 0
    ELSE
        ZF ← 1
FI;
```

Flags Affected

The ZF flag is cleared if the instruction is executed transactionally; otherwise it is set to 1. The CF, OF, SF, PF, and AF, flags are cleared.

Intel C/C++ Compiler Intrinsic Equivalent

XTEST: int _xtest(void);

SIMD Floating-Point Exceptions

None

Other Exceptions

#UD	CPUID.(EAX=7, ECX=0):HLE[bit 4] = 0 and CPUID.(EAX=7, ECX=0):RTM[bit 11] = 0. If LOCK prefix is used.

10.Updates to Chapter 1, Volume 3A

Change bars show changes to Chapter 1 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A:* System Programming Guide, Part 1.

Change to this chapter: Updates to list of processors supported and minor typo correction in Figure 1-2 "Syntax for CPUID, CR, and MSR Data Presentation".

CHAPTER 1
ABOUT THIS MANUAL

The *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A: System Programming Guide, Part 1* (order number 253668), the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3B: System Programming Guide, Part 2* (order number 253669), the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3C: System Programming Guide, Part 3* (order number 326019), and the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3D:System Programming Guide, Part 4* (order number 332831) are part of a set that describes the architecture and programming environment of Intel 64 and IA-32 Architecture processors. The other volumes in this set are:

- *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1: Basic Architecture* (order number 253665).
- *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volumes 2A, 2B, 2C & 2D: Instruction Set Reference* (order numbers 253666, 253667, 326018 and 334569).
- The *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 4: Model-Specific Registers* (order number 335592).

The *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*, describes the basic architecture and programming environment of Intel 64 and IA-32 processors. The *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volumes 2A, 2B, 2C & 2D*, describe the instruction set of the processor and the opcode structure. These volumes apply to application programmers and to programmers who write operating systems or executives. The *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volumes 3A, 3B, 3C & 3D*, describe the operating-system support environment of Intel 64 and IA-32 processors. These volumes target operating-system and BIOS designers. In addition, *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3B*, and *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3C* address the programming environment for classes of software that host operating systems. The *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 4*, describes the model-specific registers of Intel 64 and IA-32 processors.

1.1 INTEL® 64 AND IA-32 PROCESSORS COVERED IN THIS MANUAL

This manual set includes information pertaining primarily to the most recent Intel 64 and IA-32 processors, which include:

- Pentium® processors
- P6 family processors
- Pentium® 4 processors
- Pentium® M processors
- Intel® Xeon® processors
- Pentium® D processors
- Pentium® processor Extreme Editions
- 64-bit Intel® Xeon® processors
- Intel® Core™ Duo processor
- Intel® Core™ Solo processor
- Dual-Core Intel® Xeon® processor LV
- Intel® Core™2 Duo processor
- Intel® Core™2 Quad processor Q6000 series
- Intel® Xeon® processor 3000, 3200 series
- Intel® Xeon® processor 5000 series
- Intel® Xeon® processor 5100, 5300 series

- Intel® Core™2 Extreme processor X7000 and X6800 series
- Intel® Core™2 Extreme QX6000 series
- Intel® Xeon® processor 7100 series
- Intel® Pentium® Dual-Core processor
- Intel® Xeon® processor 7200, 7300 series
- Intel® Core™2 Extreme QX9000 series
- Intel® Xeon® processor 5200, 5400, 7400 series
- Intel® Core™2 Extreme processor QX9000 and X9000 series
- Intel® Core™2 Quad processor Q9000 series
- Intel® Core™2 Duo processor E8000, T9000 series
- Intel® Atom™ processors 200, 300, D400, D500, D2000, N200, N400, N2000, E2000, Z500, Z600, Z2000, C1000 series are built from 45 nm and 32 nm processes.
- Intel® Core™ i7 processor
- Intel® Core™ i5 processor
- Intel® Xeon® processor E7-8800/4800/2800 product families
- Intel® Core™ i7-3930K processor
- 2nd generation Intel® Core™ i7-2xxx, Intel® Core™ i5-2xxx, Intel® Core™ i3-2xxx processor series
- Intel® Xeon® processor E3-1200 product family
- Intel® Xeon® processor E5-2400/1400 product family
- Intel® Xeon® processor E5-4600/2600/1600 product family
- 3rd generation Intel® Core™ processors
- Intel® Xeon® processor E3-1200 v2 product family
- Intel® Xeon® processor E5-2400/1400 v2 product families
- Intel® Xeon® processor E5-4600/2600/1600 v2 product families
- Intel® Xeon® processor E7-8800/4800/2800 v2 product families
- 4th generation Intel® Core™ processors
- The Intel® Core™ M processor family
- Intel® Core™ i7-59xx Processor Extreme Edition
- Intel® Core™ i7-49xx Processor Extreme Edition
- Intel® Xeon® processor E3-1200 v3 product family
- Intel® Xeon® processor E5-2600/1600 v3 product families
- 5th generation Intel® Core™ processors
- Intel® Xeon® processor D-1500 product family
- Intel® Xeon® processor E5 v4 family
- Intel® Atom™ processor X7-Z8000 and X5-Z8000 series
- Intel® Atom™ processor Z3400 series
- Intel® Atom™ processor Z3500 series
- 6th generation Intel® Core™ processors
- Intel® Xeon® processor E3-1500m v5 product family
- 7th generation Intel® Core™ processors
- Intel® Xeon Phi™ Processor 3200, 5200, 7200 Series
- Intel® Xeon® Processor Scalable Family

The Pentium® 4, Pentium® D, and Pentium® processor Extreme Editions are based on the Intel NetBurst® microarchitecture. Most early Intel® Xeon® processors are based on the Intel NetBurst® microarchitecture. Intel Xeon processor 5000, 7100 series are based on the Intel NetBurst® microarchitecture.

The Intel® Core™ Duo, Intel® Core™ Solo and dual-core Intel® Xeon® processor LV are based on an improved Pentium® M processor microarchitecture.

The Intel® Xeon® processor 3000, 3200, 5100, 5300, 7200, and 7300 series, Intel® Pentium® dual-core, Intel® Core™2 Duo, Intel® Core™2 Quad, and Intel® Core™2 Extreme processors are based on Intel® Core™ microarchitecture.

The Intel® Xeon® processor 5200, 5400, 7400 series, Intel® Core™2 Quad processor Q9000 series, and Intel® Core™2 Extreme processors QX9000, X9000 series, Intel® Core™2 processor E8000 series are based on Enhanced Intel® Core™ microarchitecture.

The Intel® Atom™ processors 200, 300, D400, D500, D2000, N200, N400, N2000, E2000, Z500, Z600, Z2000, C1000 series are based on the Intel® Atom™ microarchitecture and supports Intel 64 architecture.

P6 family, Pentium® M, Intel® Core™ Solo, Intel® Core™ Duo processors, dual-core Intel® Xeon® processor LV, and early generations of Pentium 4 and Intel Xeon processors support IA-32 architecture. The Intel® Atom™ processor Z5xx series support IA-32 architecture.

The Intel® Xeon® processor 3000, 3200, 5000, 5100, 5200, 5300, 5400, 7100, 7200, 7300, 7400 series, Intel® Core™2 Duo, Intel® Core™2 Extreme, Intel® Core™2 Quad processors, Pentium® D processors, Pentium® Dual-Core processor, newer generations of Pentium 4 and Intel Xeon processor family support Intel® 64 architecture.

The Intel® Core™ i7 processor and Intel® Xeon® processor 3400, 5500, 7500 series are based on 45 nm Intel® microarchitecture code name Nehalem. Intel® microarchitecture code name Westmere is a 32 nm version of Intel® microarchitecture code name Nehalem. Intel® Xeon® processor 5600 series, Intel Xeon processor E7 and various Intel Core i7, i5, i3 processors are based on Intel® microarchitecture code name Westmere. These processors support Intel 64 architecture.

The Intel® Xeon® processor E5 family, Intel® Xeon® processor E3-1200 family, Intel® Xeon® processor E7-8800/4800/2800 product families, Intel® Core™ i7-3930K processor, and 2nd generation Intel® Core™ i7-2xxx, Intel® Core™ i5-2xxx, Intel® Core™ i3-2xxx processor series are based on the Intel® microarchitecture code name Sandy Bridge and support Intel 64 architecture.

The Intel® Xeon® processor E7-8800/4800/2800 v2 product families, Intel® Xeon® processor E3-1200 v2 product family and 3rd generation Intel® Core™ processors are based on the Intel® microarchitecture code name Ivy Bridge and support Intel 64 architecture.

The Intel® Xeon® processor E5-4600/2600/1600 v2 product families, Intel® Xeon® processor E5-2400/1400 v2 product families and Intel® Core™ i7-49xx Processor Extreme Edition are based on the Intel® microarchitecture code name Ivy Bridge-E and support Intel 64 architecture.

The Intel® Xeon® processor E3-1200 v3 product family and 4th Generation Intel® Core™ processors are based on the Intel® microarchitecture code name Haswell and support Intel 64 architecture.

The Intel® Xeon® processor E5-2600/1600 v3 product families and the Intel® Core™ i7-59xx Processor Extreme Edition are based on the Intel® microarchitecture code name Haswell-E and support Intel 64 architecture.

The Intel® Atom™ processor Z8000 series is based on the Intel microarchitecture code name Airmont.

The Intel® Atom™ processor Z3400 series and the Intel® Atom™ processor Z3500 series are based on the Intel microarchitecture code name Silvermont.

The Intel® Core™ M processor family, 5th generation Intel® Core™ processors, Intel® Xeon® processor D-1500 product family and the Intel® Xeon® processor E5 v4 family are based on the Intel® microarchitecture code name Broadwell and support Intel 64 architecture.

The Intel® Xeon® Processor Scalable Family, Intel® Xeon® processor E3-1500m v5 product family and 6th generation Intel® Core™ processors are based on the Intel® microarchitecture code name Skylake and support Intel 64 architecture.

The 7th generation Intel® Core™ processors are based on the Intel® microarchitecture code name Kaby Lake and support Intel 64 architecture.

The Intel® Xeon Phi™ Processor 3200, 5200, 7200 Series is based on the Intel® microarchitecture code name Knights Landing and supports Intel 64 architecture.

IA-32 architecture is the instruction set architecture and programming environment for Intel's 32-bit microprocessors. Intel® 64 architecture is the instruction set architecture and programming environment which is the superset of Intel's 32-bit and 64-bit architectures. It is compatible with the IA-32 architecture.

1.2 OVERVIEW OF THE SYSTEM PROGRAMMING GUIDE

A description of this manual's content follows[1]:

Chapter 1 — About This Manual. Gives an overview of all eight volumes of the *Intel® 64 and IA-32 Architectures Software Developer's Manual*. It also describes the notational conventions in these manuals and lists related Intel manuals and documentation of interest to programmers and hardware designers.

Chapter 2 — System Architecture Overview. Describes the modes of operation used by Intel 64 and IA-32 processors and the mechanisms provided by the architectures to support operating systems and executives, including the system-oriented registers and data structures and the system-oriented instructions. The steps necessary for switching between real-address and protected modes are also identified.

Chapter 3 — Protected-Mode Memory Management. Describes the data structures, registers, and instructions that support segmentation and paging. The chapter explains how they can be used to implement a "flat" (unsegmented) memory model or a segmented memory model.

Chapter 4 — Paging. Describes the paging modes supported by Intel 64 and IA-32 processors.

Chapter 5 — Protection. Describes the support for page and segment protection provided in the Intel 64 and IA-32 architectures. This chapter also explains the implementation of privilege rules, stack switching, pointer validation, user and supervisor modes.

Chapter 6 — Interrupt and Exception Handling. Describes the basic interrupt mechanisms defined in the Intel 64 and IA-32 architectures, shows how interrupts and exceptions relate to protection, and describes how the architecture handles each exception type. Reference information for each exception is given in this chapter. Includes programming the LINT0 and LINT1 inputs and gives an example of how to program the LINT0 and LINT1 pins for specific interrupt vectors.

Chapter 7 — Task Management. Describes mechanisms the Intel 64 and IA-32 architectures provide to support multitasking and inter-task protection.

Chapter 8 — Multiple-Processor Management. Describes the instructions and flags that support multiple processors with shared memory, memory ordering, and Intel® Hyper-Threading Technology. Includes MP initialization for P6 family processors and gives an example of how to use the MP protocol to boot P6 family processors in an MP system.

Chapter 9 — Processor Management and Initialization. Defines the state of an Intel 64 or IA-32 processor after reset initialization. This chapter also explains how to set up an Intel 64 or IA-32 processor for real-address mode operation and protected- mode operation, and how to switch between modes.

Chapter 10 — Advanced Programmable Interrupt Controller (APIC). Describes the programming interface to the local APIC and gives an overview of the interface between the local APIC and the I/O APIC. Includes APIC bus message formats and describes the message formats for messages transmitted on the APIC bus for P6 family and Pentium processors.

Chapter 11 — Memory Cache Control. Describes the general concept of caching and the caching mechanisms supported by the Intel 64 or IA-32 architectures. This chapter also describes the memory type range registers (MTRRs) and how they can be used to map memory types of physical memory. Information on using the new cache control and memory streaming instructions introduced with the Pentium III, Pentium 4, and Intel Xeon processors is also given.

Chapter 12 — Intel® MMX™ Technology System Programming. Describes those aspects of the Intel® MMX™ technology that must be handled and considered at the system programming level, including: task switching, exception handling, and compatibility with existing system environments.

1. Model-Specific Registers have been moved out of this volume and into a separate volume: *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 4*.

Chapter 13 — System Programming For Instruction Set Extensions And Processor Extended States. Describes the operating system requirements to support SSE/SSE2/SSE3/SSSE3/SSE4 extensions, including task switching, exception handling, and compatibility with existing system environments. The latter part of this chapter describes the extensible framework of operating system requirements to support processor extended states. Processor extended state may be required by instruction set extensions beyond those of SSE/SSE2/SSE3/SSSE3/SSE4 extensions.

Chapter 14 — Power and Thermal Management. Describes facilities of Intel 64 and IA-32 architecture used for power management and thermal monitoring.

Chapter 15 — Machine-Check Architecture. Describes the machine-check architecture and machine-check exception mechanism found in the Pentium 4, Intel Xeon, and P6 family processors. Additionally, a signaling mechanism for software to respond to hardware corrected machine check error is covered.

Chapter 16 — Interpreting Machine-Check Error Codes. Gives an example of how to interpret the error codes for a machine-check error that occurred on a P6 family processor.

Chapter 17 — Debug, Branch Profile, TSC, and Resource Monitoring Features. Describes the debugging registers and other debug mechanism provided in Intel 64 or IA-32 processors. This chapter also describes the time-stamp counter.

Chapter 18 — Performance Monitoring. Describes the Intel 64 and IA-32 architectures' facilities for monitoring performance.

Chapter 19 — Performance-Monitoring Events. Lists architectural performance events. Non-architectural performance events (i.e. model-specific events) are listed for each generation of microarchitecture.

Chapter 20 — 8086 Emulation. Describes the real-address and virtual-8086 modes of the IA-32 architecture.

Chapter 21 — Mixing 16-Bit and 32-Bit Code. Describes how to mix 16-bit and 32-bit code modules within the same program or task.

Chapter 22 — IA-32 Architecture Compatibility. Describes architectural compatibility among IA-32 processors.

Chapter 23 — Introduction to Virtual Machine Extensions. Describes the basic elements of virtual machine architecture and the virtual machine extensions for Intel 64 and IA-32 Architectures.

Chapter 24 — Virtual Machine Control Structures. Describes components that manage VMX operation. These include the working-VMCS pointer and the controlling-VMCS pointer.

Chapter 25 — VMX Non-Root Operation. Describes the operation of a VMX non-root operation. Processor operation in VMX non-root mode can be restricted programmatically such that certain operations, events or conditions can cause the processor to transfer control from the guest (running in VMX non-root mode) to the monitor software (running in VMX root mode).

Chapter 26 — VM Entries. Describes VM entries. VM entry transitions the processor from the VMM running in VMX root-mode to a VM running in VMX non-root mode. VM-Entry is performed by the execution of VMLAUNCH or VMRESUME instructions.

Chapter 27 — VM Exits. Describes VM exits. Certain events, operations or situations while the processor is in VMX non-root operation may cause VM-exit transitions. In addition, VM exits can also occur on failed VM entries.

Chapter 28 — VMX Support for Address Translation. Describes virtual-machine extensions that support address translation and the virtualization of physical memory.

Chapter 29 — APIC Virtualization and Virtual Interrupts. Describes the VMCS including controls that enable the virtualization of interrupts and the Advanced Programmable Interrupt Controller (APIC).

Chapter 30 — VMX Instruction Reference. Describes the virtual-machine extensions (VMX). VMX is intended for a system executive to support virtualization of processor hardware and a system software layer acting as a host to multiple guest software environments.

Chapter 31 — Virtual-Machine Monitor Programming Considerations. Describes programming considerations for VMMs. VMMs manage virtual machines (VMs).

Chapter 32 — Virtualization of System Resources. Describes the virtualization of the system resources. These include: debugging facilities, address translation, physical memory, and microcode update facilities.

Chapter 33 — Handling Boundary Conditions in a Virtual Machine Monitor. Describes what a VMM must consider when handling exceptions, interrupts, error conditions, and transitions between activity states.

Chapter 34 — System Management Mode. Describes Intel 64 and IA-32 architectures' system management mode (SMM) facilities.

Chapter 35 — Intel® Processor Trace. Describes details of Intel® Processor Trace.

Chapter 36 — Introduction to Intel® Software Guard Extensions. Provides an overview of the Intel® Software Guard Extensions (Intel® SGX) set of instructions.

Chapter 37 — Enclave Access Control and Data Structures. Describes Enclave Access Control procedures and defines various Intel SGX data structures.

Chapter 38 — Enclave Operation. Describes enclave creation and initialization, adding pages and measuring an enclave, and enclave entry and exit.

Chapter 39 — Enclave Exiting Events. Describes enclave-exiting events (EEE) and asynchronous enclave exit (AEX).

Chapter 40 — SGX Instruction References. Describes the supervisor and user level instructions provided by Intel SGX.

Chapter 41 — Intel® SGX Interactions with IA32 and Intel® 64 Architecture. Describes the Intel SGX collection of enclave instructions for creating protected execution environments on processors supporting IA32 and Intel 64 architectures.

Chapter 42 — Enclave Code Debug and Profiling. Describes enclave code debug processes and options.

Appendix A — VMX Capability Reporting Facility. Describes the VMX capability MSRs. Support for specific VMX features is determined by reading capability MSRs.

Appendix B — Field Encoding in VMCS. Enumerates all fields in the VMCS and their encodings. Fields are grouped by width (16-bit, 32-bit, etc.) and type (guest-state, host-state, etc.).

Appendix C — VM Basic Exit Reasons. Describes the 32-bit fields that encode reasons for a VM exit. Examples of exit reasons include, but are not limited to: software interrupts, processor exceptions, software traps, NMIs, external interrupts, and triple faults.

1.3 NOTATIONAL CONVENTIONS

This manual uses specific notation for data-structure formats, for symbolic representation of instructions, and for hexadecimal and binary numbers. A review of this notation makes the manual easier to read.

1.3.1 Bit and Byte Order

In illustrations of data structures in memory, smaller addresses appear toward the bottom of the figure; addresses increase toward the top. Bit positions are numbered from right to left. The numerical value of a set bit is equal to two raised to the power of the bit position. Intel 64 and IA-32 processors are "little endian" machines; this means the bytes of a word are numbered starting from the least significant byte. Figure 1-1 illustrates these conventions.

1.3.2 Reserved Bits and Software Compatibility

In many register and memory layout descriptions, certain bits are marked as **reserved**. When bits are marked as reserved, it is essential for compatibility with future processors that software treat these bits as having a future, though unknown, effect. The behavior of reserved bits should be regarded as not only undefined, but unpredictable. Software should follow these guidelines in dealing with reserved bits:

- Do not depend on the states of any reserved bits when testing the values of registers which contain such bits. Mask out the reserved bits before testing.
- Do not depend on the states of any reserved bits when storing to memory or to a register.
- Do not depend on the ability to retain information written into any reserved bits.
- When loading a register, always load the reserved bits with the values indicated in the documentation, if any, or reload them with values previously read from the same register.

NOTE

Avoid any software dependence upon the state of reserved bits in Intel 64 and IA-32 registers. Depending upon the values of reserved register bits will make software dependent upon the unspecified manner in which the processor handles these bits. Programs that depend upon reserved values risk incompatibility with future processors.

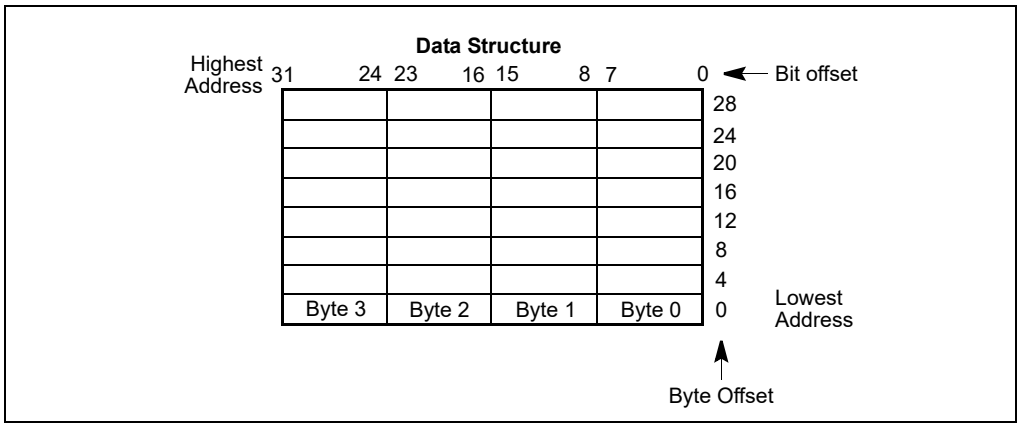


Figure 1-1. Bit and Byte Order

1.3.3 Instruction Operands

When instructions are represented symbolically, a subset of assembly language is used. In this subset, an instruction has the following format:

```
label: mnemonic argument1, argument2, argument3
```

where:

- A **label** is an identifier which is followed by a colon.
- A **mnemonic** is a reserved name for a class of instruction opcodes which have the same function.
- The operands **argument1**, **argument2**, and **argument3** are optional. There may be from zero to three operands, depending on the opcode. When present, they take the form of either literals or identifiers for data items. Operand identifiers are either reserved names of registers or are assumed to be assigned to data items declared in another part of the program (which may not be shown in the example).

When two operands are present in an arithmetic or logical instruction, the right operand is the source and the left operand is the destination.

For example:

```
LOADREG: MOV EAX, SUBTOTAL
```

In this example LOADREG is a label, MOV is the mnemonic identifier of an opcode, EAX is the destination operand, and SUBTOTAL is the source operand. Some assembly languages put the source and destination in reverse order.

1.3.4 Hexadecimal and Binary Numbers

Base 16 (hexadecimal) numbers are represented by a string of hexadecimal digits followed by the character H (for example, F82EH). A hexadecimal digit is a character from the following set: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, and F.

Base 2 (binary) numbers are represented by a string of 1s and 0s, sometimes followed by the character B (for example, 1010B). The "B" designation is only used in situations where confusion as to the type of number might arise.

1.3.5 Segmented Addressing

The processor uses byte addressing. This means memory is organized and accessed as a sequence of bytes. Whether one or more bytes are being accessed, a byte address is used to locate the byte or bytes memory. The range of memory that can be addressed is called an **address space**.

The processor also supports segmented addressing. This is a form of addressing where a program may have many independent address spaces, called **segments**. For example, a program can keep its code (instructions) and stack in separate segments. Code addresses would always refer to the code space, and stack addresses would always refer to the stack space. The following notation is used to specify a byte address within a segment:

Segment-register:Byte-address

For example, the following segment address identifies the byte at address FF79H in the segment pointed by the DS register:

```
DS:FF79H
```

The following segment address identifies an instruction address in the code segment. The CS register points to the code segment and the EIP register contains the address of the instruction.

```
CS:EIP
```

1.3.6 Syntax for CPUID, CR, and MSR Values

Obtain feature flags, status, and system information by using the CPUID instruction, by checking control register bits, and by reading model-specific registers. We are moving toward a single syntax to represent this type of information. See Figure 1-2.

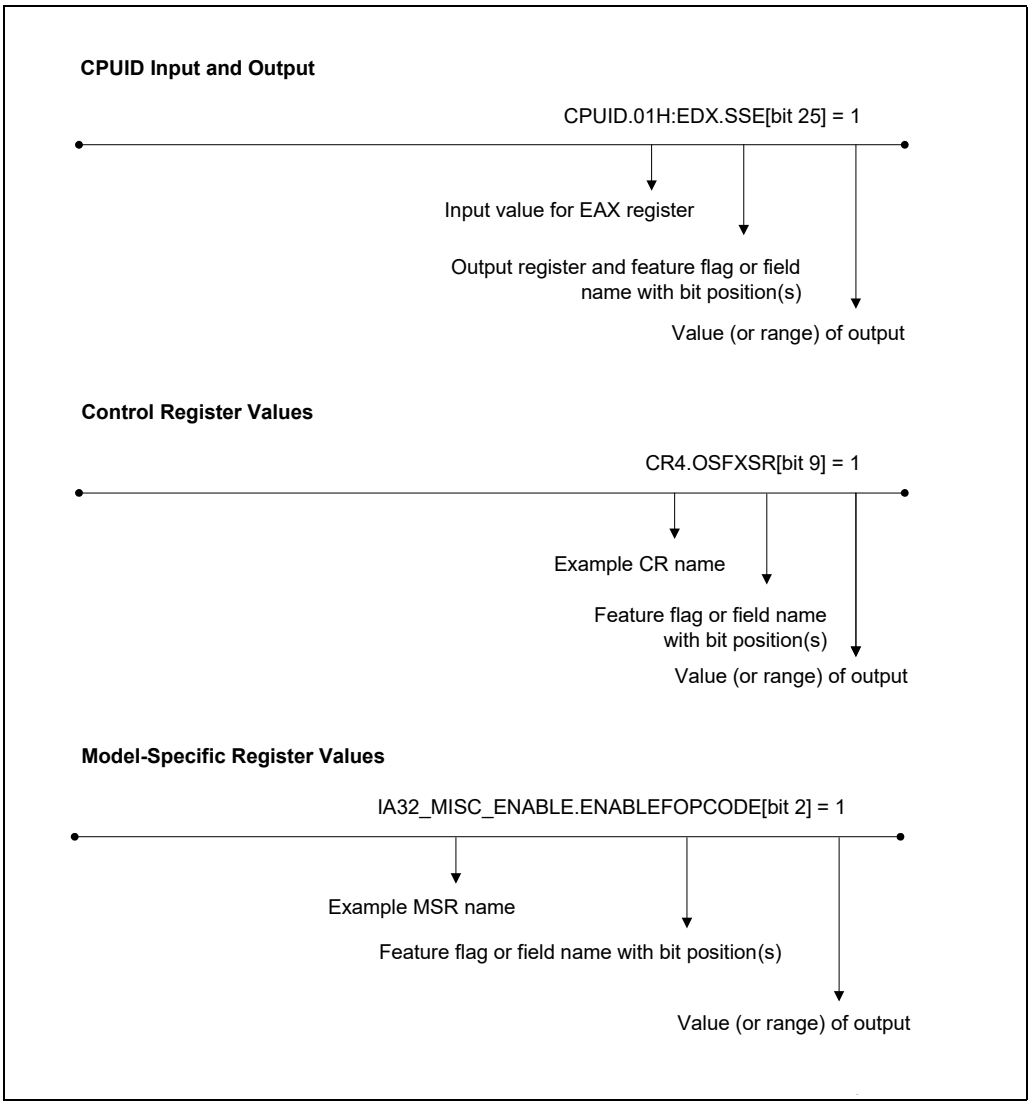


Figure 1-2. Syntax for CPUID, CR, and MSR Data Presentation

1.3.7 Exceptions

An exception is an event that typically occurs when an instruction causes an error. For example, an attempt to divide by zero generates an exception. However, some exceptions, such as breakpoints, occur under other conditions. Some types of exceptions may provide error codes. An error code reports additional information about the error. An example of the notation used to show an exception and error code is shown below:

```
#PF(fault code)
```

This example refers to a page-fault exception under conditions where an error code naming a type of fault is reported. Under some conditions, exceptions which produce error codes may not be able to report an accurate code. In this case, the error code is zero, as shown below for a general-protection exception:

```
#GP(0)
```

1.4 RELATED LITERATURE

Literature related to Intel 64 and IA-32 processors is listed and viewable on-line at:

http://www.intel.com/content/www/us/en/processors/architectures-software-developer-manuals.html

See also:

- The data sheet for a particular Intel 64 or IA-32 processor
- The specification update for a particular Intel 64 or IA-32 processor
- Intel® C++ Compiler documentation and online help:
  http://software.intel.com/en-us/articles/intel-compilers/
- Intel® Fortran Compiler documentation and online help:
  http://software.intel.com/en-us/articles/intel-compilers/
- Intel® Software Development Tools:
  https://software.intel.com/en-us/intel-sdp-home
- Intel® 64 and IA-32 Architectures Software Developer's Manual (in one, four or ten volumes):
  https://software.intel.com/en-us/articles/intel-sdm
- Intel® 64 and IA-32 Architectures Optimization Reference Manual:
  https://software.intel.com/en-us/articles/intel-sdm#optimization
- Intel 64 Architecture x2APIC Specification:

  http://www.intel.com/content/www/us/en/architecture-and-technology/64-architecture-x2apic-specification.html
- Intel® Trusted Execution Technology Measured Launched Environment Programming Guide:

  http://www.intel.com/content/www/us/en/software-developers/intel-txt-software-development-guide.html
- Developing Multi-threaded Applications: A Platform Consistent Approach:
  https://software.intel.com/sites/default/files/article/147714/51534-developing-multithreaded-applications.pdf
- Using Spin-Loops on Intel® Pentium® 4 Processor and Intel® Xeon® Processor:
  https://software.intel.com/sites/default/files/22/30/25602
- Performance Monitoring Unit Sharing Guide
  http://software.intel.com/file/30388

Literature related to selected features in future Intel processors are available at:

- Intel® Architecture Instruction Set Extensions Programming Reference
  https://software.intel.com/en-us/isa-extensions
- Intel® Software Guard Extensions (Intel® SGX) Programming Reference
  https://software.intel.com/en-us/isa-extensions/intel-sgx

More relevant links are:

- Intel® Developer Zone:

  https://software.intel.com/en-us
- Developer centers:

  http://www.intel.com/content/www/us/en/hardware-developers/developer-centers.html
- Processor support general link:

  http://www.intel.com/support/processors/
- Intel® Hyper-Threading Technology (Intel® HT Technology):

  http://www.intel.com/technology/platform-technology/hyper-threading/index.htm

11. Updates to Chapter 7, Volume 3A

Change bars show changes to Chapter 7 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A:* System Programming Guide, Part 1.

Changes to this chapter: Update to table 7-1 "Exception Conditions Checked During a Task Switch".

CHAPTER 7
TASK MANAGEMENT

This chapter describes the IA-32 architecture's task management facilities. These facilities are only available when the processor is running in protected mode.

This chapter focuses on 32-bit tasks and the 32-bit TSS structure. For information on 16-bit tasks and the 16-bit TSS structure, see Section 7.6, "16-Bit Task-State Segment (TSS)." For information specific to task management in 64-bit mode, see Section 7.7, "Task Management in 64-bit Mode."

7.1 TASK MANAGEMENT OVERVIEW

A task is a unit of work that a processor can dispatch, execute, and suspend. It can be used to execute a program, a task or process, an operating-system service utility, an interrupt or exception handler, or a kernel or executive utility.

The IA-32 architecture provides a mechanism for saving the state of a task, for dispatching tasks for execution, and for switching from one task to another. When operating in protected mode, all processor execution takes place from within a task. Even simple systems must define at least one task. More complex systems can use the processor's task management facilities to support multitasking applications.

7.1.1 Task Structure

A task is made up of two parts: a task execution space and a task-state segment (TSS). The task execution space consists of a code segment, a stack segment, and one or more data segments (see Figure 7-1). If an operating system or executive uses the processor's privilege-level protection mechanism, the task execution space also provides a separate stack for each privilege level.

The TSS specifies the segments that make up the task execution space and provides a storage place for task state information. In multitasking systems, the TSS also provides a mechanism for linking tasks.

A task is identified by the segment selector for its TSS. When a task is loaded into the processor for execution, the segment selector, base address, limit, and segment descriptor attributes for the TSS are loaded into the task register (see Section 2.4.4, "Task Register (TR)").

If paging is implemented for the task, the base address of the page directory used by the task is loaded into control register CR3.

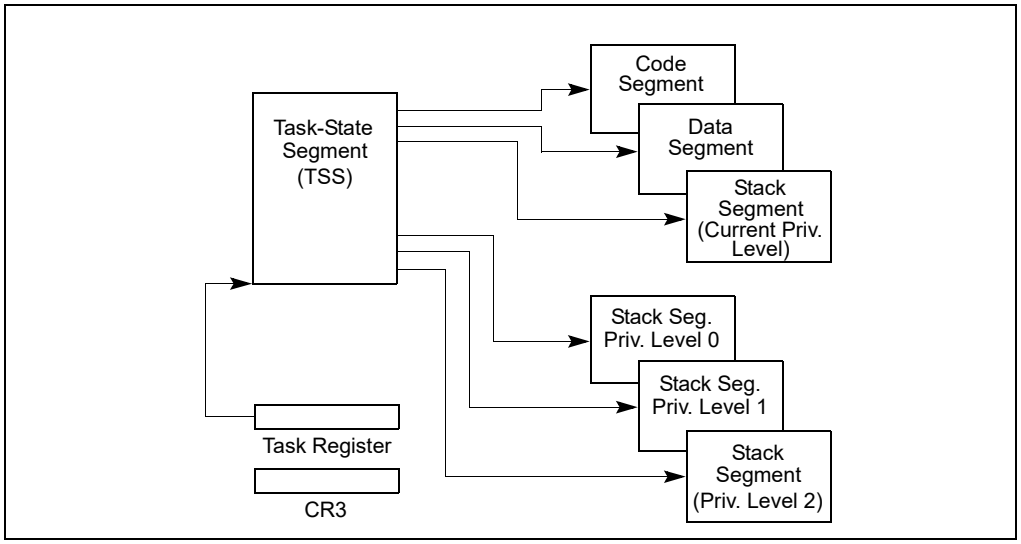


Figure 7-1. Structure of a Task

7.1.2 Task State

The following items define the state of the currently executing task:

- The task's current execution space, defined by the segment selectors in the segment registers (CS, DS, SS, ES, FS, and GS).
- The state of the general-purpose registers.
- The state of the EFLAGS register.
- The state of the EIP register.
- The state of control register CR3.
- The state of the task register.
- The state of the LDTR register.
- The I/O map base address and I/O map (contained in the TSS).
- Stack pointers to the privilege 0, 1, and 2 stacks (contained in the TSS).
- Link to previously executed task (contained in the TSS).

Prior to dispatching a task, all of these items are contained in the task's TSS, except the state of the task register. Also, the complete contents of the LDTR register are not contained in the TSS, only the segment selector for the LDT.

7.1.3 Executing a Task

Software or the processor can dispatch a task for execution in one of the following ways:

- A explicit call to a task with the CALL instruction.
- A explicit jump to a task with the JMP instruction.
- An implicit call (by the processor) to an interrupt-handler task.
- An implicit call to an exception-handler task.
- A return (initiated with an IRET instruction) when the NT flag in the EFLAGS register is set.

All of these methods for dispatching a task identify the task to be dispatched with a segment selector that points to a task gate or the TSS for the task. When dispatching a task with a CALL or JMP instruction, the selector in the instruction may select the TSS directly or a task gate that holds the selector for the TSS. When dispatching a task

to handle an interrupt or exception, the IDT entry for the interrupt or exception must contain a task gate that holds the selector for the interrupt- or exception-handler TSS.

When a task is dispatched for execution, a task switch occurs between the currently running task and the dispatched task. During a task switch, the execution environment of the currently executing task (called the task's state or **context**) is saved in its TSS and execution of the task is suspended. The context for the dispatched task is then loaded into the processor and execution of that task begins with the instruction pointed to by the newly loaded EIP register. If the task has not been run since the system was last initialized, the EIP will point to the first instruction of the task's code; otherwise, it will point to the next instruction after the last instruction that the task executed when it was last active.

If the currently executing task (the calling task) called the task being dispatched (the called task), the TSS segment selector for the calling task is stored in the TSS of the called task to provide a link back to the calling task.

For all IA-32 processors, tasks are not recursive. A task cannot call or jump to itself.

Interrupts and exceptions can be handled with a task switch to a handler task. Here, the processor performs a task switch to handle the interrupt or exception and automatically switches back to the interrupted task upon returning from the interrupt-handler task or exception-handler task. This mechanism can also handle interrupts that occur during interrupt tasks.

As part of a task switch, the processor can also switch to another LDT, allowing each task to have a different logical-to-physical address mapping for LDT-based segments. The page-directory base register (CR3) also is reloaded on a task switch, allowing each task to have its own set of page tables. These protection facilities help isolate tasks and prevent them from interfering with one another.

If protection mechanisms are not used, the processor provides no protection between tasks. This is true even with operating systems that use multiple privilege levels for protection. A task running at privilege level 3 that uses the same LDT and page tables as other privilege-level-3 tasks can access code and corrupt data and the stack of other tasks.

Use of task management facilities for handling multitasking applications is optional. Multitasking can be handled in software, with each software defined task executed in the context of a single IA-32 architecture task.

7.2 TASK MANAGEMENT DATA STRUCTURES

The processor defines five data structures for handling task-related activities:

- Task-state segment (TSS).
- Task-gate descriptor.
- TSS descriptor.
- Task register.
- NT flag in the EFLAGS register.

When operating in protected mode, a TSS and TSS descriptor must be created for at least one task, and the segment selector for the TSS must be loaded into the task register (using the LTR instruction).

7.2.1 Task-State Segment (TSS)

The processor state information needed to restore a task is saved in a system segment called the task-state segment (TSS). Figure 7-2 shows the format of a TSS for tasks designed for 32-bit CPUs. The fields of a TSS are divided into two main categories: dynamic fields and static fields.

For information about 16-bit Intel 286 processor task structures, see Section 7.6, "16-Bit Task-State Segment (TSS)." For information about 64-bit mode task structures, see Section 7.7, "Task Management in 64-bit Mode."

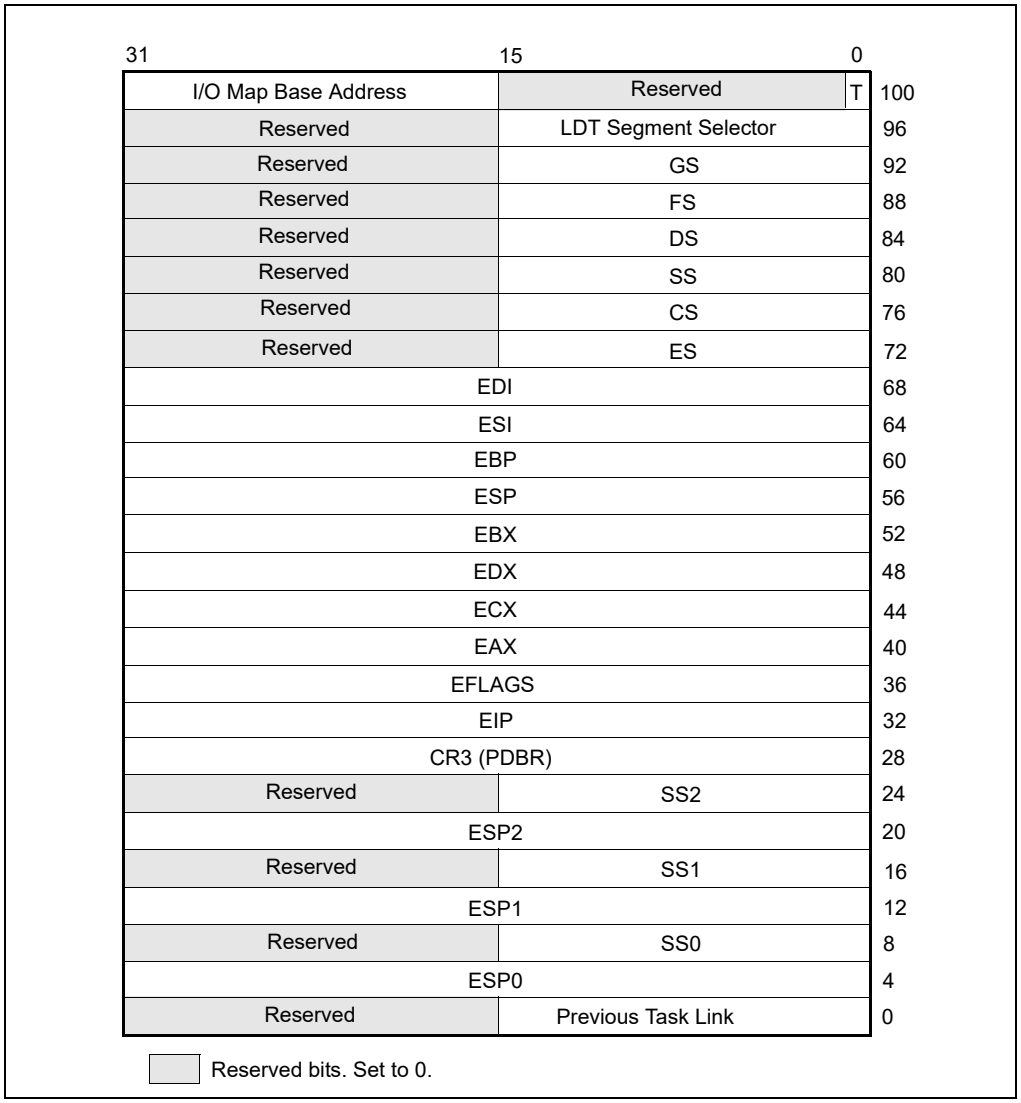


Figure 7-2. 32-Bit Task-State Segment (TSS)

The processor updates dynamic fields when a task is suspended during a task switch. The following are dynamic fields:

- **General-purpose register fields** — State of the EAX, ECX, EDX, EBX, ESP, EBP, ESI, and EDI registers prior to the task switch.
- **Segment selector fields** — Segment selectors stored in the ES, CS, SS, DS, FS, and GS registers prior to the task switch.
- **EFLAGS register field** — State of the EFAGS register prior to the task switch.
- **EIP (instruction pointer) field** — State of the EIP register prior to the task switch.
- **Previous task link field** — Contains the segment selector for the TSS of the previous task (updated on a task switch that was initiated by a call, interrupt, or exception). This field (which is sometimes called the back link field) permits a task switch back to the previous task by using the IRET instruction.

The processor reads the static fields, but does not normally change them. These fields are set up when a task is created. The following are static fields:

- **LDT segment selector field** — Contains the segment selector for the task's LDT.

- **CR3 control register field** — Contains the base physical address of the page directory to be used by the task. Control register CR3 is also known as the page-directory base register (PDBR).
- **Privilege level-0, -1, and -2 stack pointer fields** — These stack pointers consist of a logical address made up of the segment selector for the stack segment (SS0, SS1, and SS2) and an offset into the stack (ESP0, ESP1, and ESP2). Note that the values in these fields are static for a particular task; whereas, the SS and ESP values will change if stack switching occurs within the task.
- **T (debug trap) flag (byte 100, bit 0)** — When set, the T flag causes the processor to raise a debug exception when a task switch to this task occurs (see Section 17.3.1.5, "Task-Switch Exception Condition").
- **I/O map base address field** — Contains a 16-bit offset from the base of the TSS to the I/O permission bit map and interrupt redirection bitmap. When present, these maps are stored in the TSS at higher addresses. The I/O map base address points to the beginning of the I/O permission bit map and the end of the interrupt redirection bit map. See Chapter 18, "Input/Output," in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*, for more information about the I/O permission bit map. See Section 20.3, "Interrupt and Exception Handling in Virtual-8086 Mode," for a detailed description of the interrupt redirection bit map.

If paging is used:

- Avoid placing a page boundary in the part of the TSS that the processor reads during a task switch (the first 104 bytes). The processor may not correctly perform address translations if a boundary occurs in this area. During a task switch, the processor reads and writes into the first 104 bytes of each TSS (using contiguous physical addresses beginning with the physical address of the first byte of the TSS). So, after TSS access begins, if part of the 104 bytes is not physically contiguous, the processor will access incorrect information without generating a page-fault exception.
- Pages corresponding to the previous task's TSS, the current task's TSS, and the descriptor table entries for each all should be marked as read/write.
- Task switches are carried out faster if the pages containing these structures are present in memory before the task switch is initiated.

7.2.2 TSS Descriptor

The TSS, like all other segments, is defined by a segment descriptor. Figure 7-3 shows the format of a TSS descriptor. TSS descriptors may only be placed in the GDT; they cannot be placed in an LDT or the IDT.

An attempt to access a TSS using a segment selector with its TI flag set (which indicates the current LDT) causes a general-protection exception (#GP) to be generated during CALLs and JMPs; it causes an invalid TSS exception (#TS) during IRETs. A general-protection exception is also generated if an attempt is made to load a segment selector for a TSS into a segment register.

The busy flag (B) in the type field indicates whether the task is busy. A busy task is currently running or suspended. A type field with a value of 1001B indicates an inactive task; a value of 1011B indicates a busy task. Tasks are not recursive. The processor uses the busy flag to detect an attempt to call a task whose execution has been interrupted. To insure that there is only one busy flag is associated with a task, each TSS should have only one TSS descriptor that points to it.

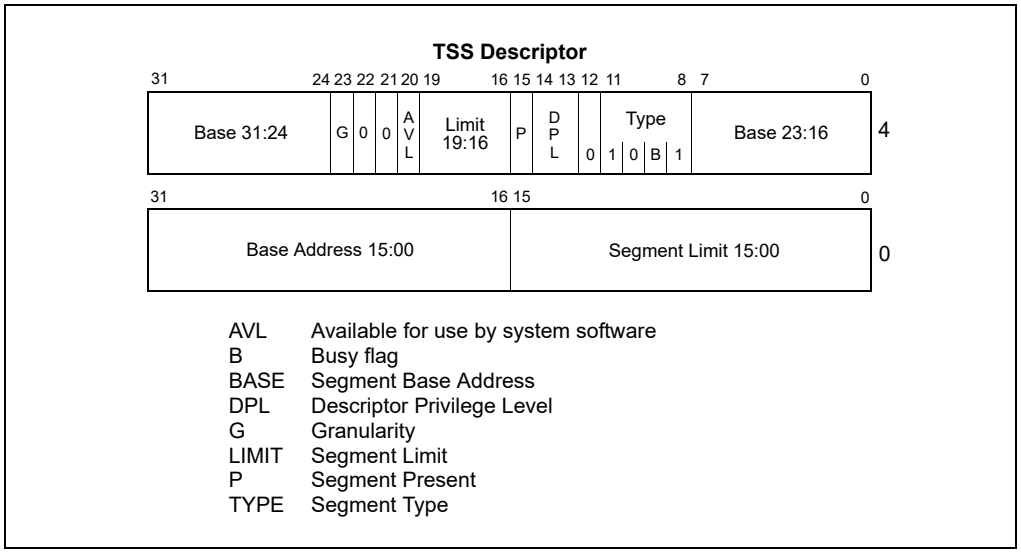


Figure 7-3. TSS Descriptor

The base, limit, and DPL fields and the granularity and present flags have functions similar to their use in data-segment descriptors (see Section 3.4.5, "Segment Descriptors"). When the G flag is 0 in a TSS descriptor for a 32-bit TSS, the limit field must have a value equal to or greater than 67H, one byte less than the minimum size of a TSS. Attempting to switch to a task whose TSS descriptor has a limit less than 67H generates an invalid-TSS exception (#TS). A larger limit is required if an I/O permission bit map is included or if the operating system stores additional data. The processor does not check for a limit greater than 67H on a task switch; however, it does check when accessing the I/O permission bit map or interrupt redirection bit map.

Any program or procedure with access to a TSS descriptor (that is, whose CPL is numerically equal to or less than the DPL of the TSS descriptor) can dispatch the task with a call or a jump.

In most systems, the DPLs of TSS descriptors are set to values less than 3, so that only privileged software can perform task switching. However, in multitasking applications, DPLs for some TSS descriptors may be set to 3 to allow task switching at the application (or user) privilege level.

7.2.3 TSS Descriptor in 64-bit mode

In 64-bit mode, task switching is not supported, but TSS descriptors still exist. The format of a 64-bit TSS is described in Section 7.7.

In 64-bit mode, the TSS descriptor is expanded to 16 bytes (see Figure 7-4). This expansion also applies to an LDT descriptor in 64-bit mode. Table 3-2 provides the encoding information for the segment type field.

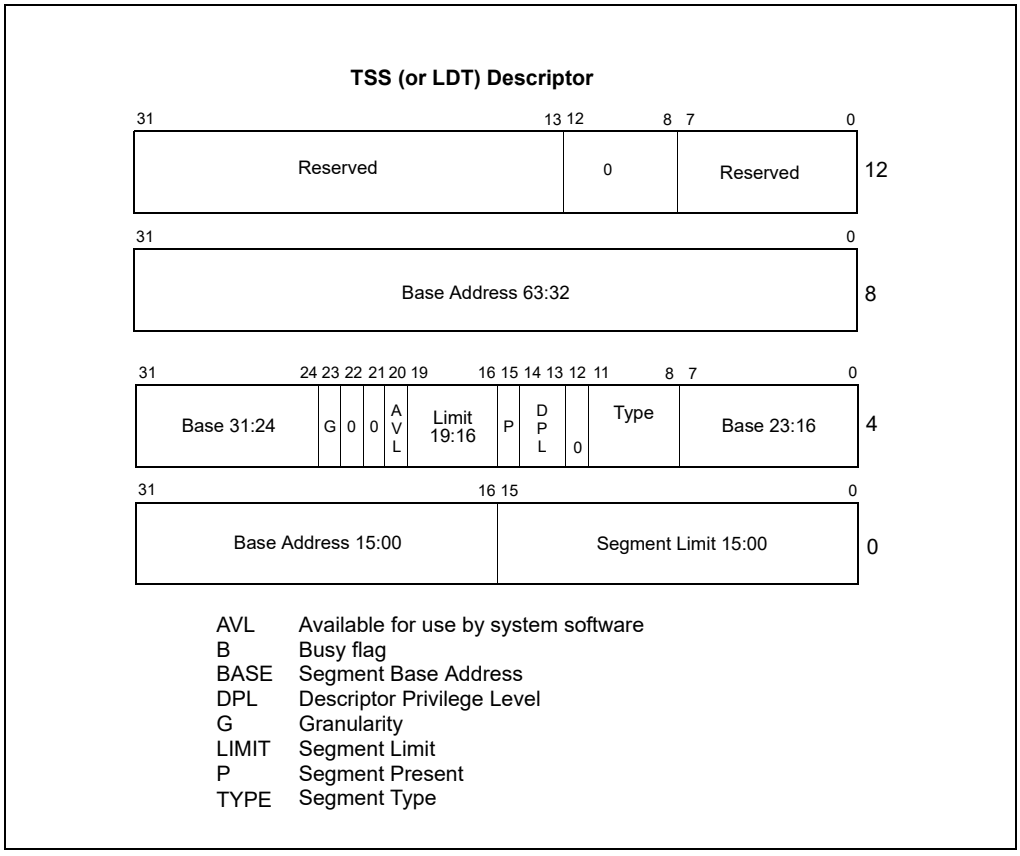


Figure 7-4. Format of TSS and LDT Descriptors in 64-bit Mode

7.2.4 Task Register

The task register holds the 16-bit segment selector and the entire segment descriptor (32-bit base address (64 bits in IA-32e mode), 16-bit segment limit, and descriptor attributes) for the TSS of the current task (see Figure 2-6). This information is copied from the TSS descriptor in the GDT for the current task. Figure 7-5 shows the path the processor uses to access the TSS (using the information in the task register).

The task register has a visible part (that can be read and changed by software) and an invisible part (maintained by the processor and is inaccessible by software). The segment selector in the visible portion points to a TSS descriptor in the GDT. The processor uses the invisible portion of the task register to cache the segment descriptor for the TSS. Caching these values in a register makes execution of the task more efficient. The LTR (load task register) and STR (store task register) instructions load and read the visible portion of the task register:

The LTR instruction loads a segment selector (source operand) into the task register that points to a TSS descriptor in the GDT. It then loads the invisible portion of the task register with information from the TSS descriptor. LTR is a privileged instruction that may be executed only when the CPL is 0. It's used during system initialization to put an initial value in the task register. Afterwards, the contents of the task register are changed implicitly when a task switch occurs.

The STR (store task register) instruction stores the visible portion of the task register in a general-purpose register or memory. This instruction can be executed by code running at any privilege level in order to identify the currently running task. However, it is normally used only by operating system software. (If CR4.UMIP = 1, STR can be executed only when CPL = 0.)

On power up or reset of the processor, segment selector and base address are set to the default value of 0; the limit is set to FFFFH.

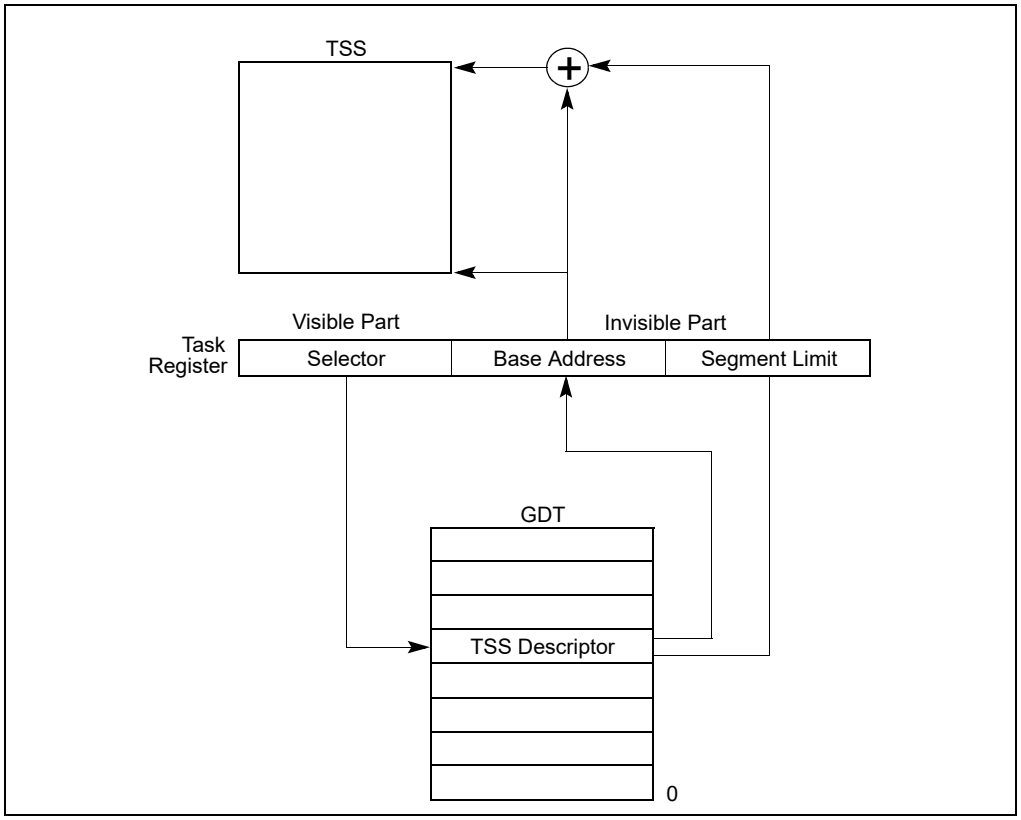


Figure 7-5. Task Register

7.2.5 Task-Gate Descriptor

A task-gate descriptor provides an indirect, protected reference to a task (see Figure 7-6). It can be placed in the GDT, an LDT, or the IDT. The TSS segment selector field in a task-gate descriptor points to a TSS descriptor in the GDT. The RPL in this segment selector is not used.

The DPL of a task-gate descriptor controls access to the TSS descriptor during a task switch. When a program or procedure makes a call or jump to a task through a task gate, the CPL and the RPL field of the gate selector pointing to the task gate must be less than or equal to the DPL of the task-gate descriptor. Note that when a task gate is used, the DPL of the destination TSS descriptor is not used.

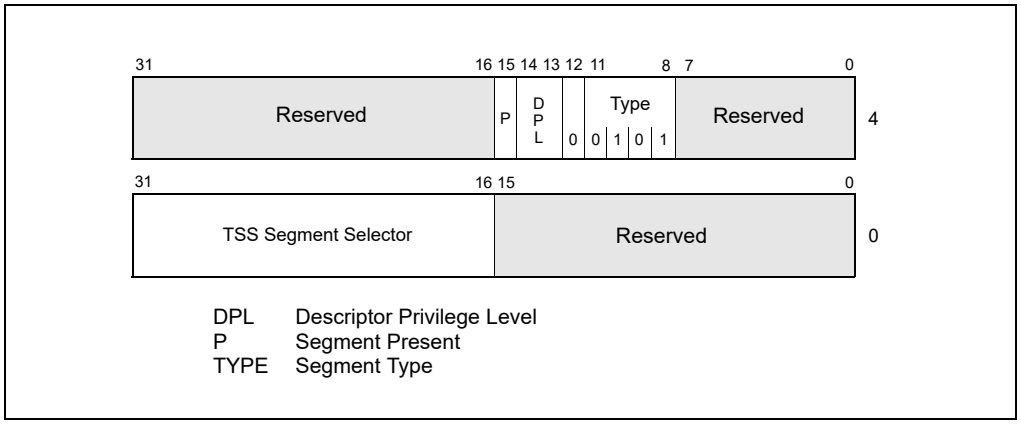


Figure 7-6. Task-Gate Descriptor

A task can be accessed either through a task-gate descriptor or a TSS descriptor. Both of these structures satisfy the following needs:

- **Need for a task to have only one busy flag** — Because the busy flag for a task is stored in the TSS descriptor, each task should have only one TSS descriptor. There may, however, be several task gates that reference the same TSS descriptor.
- **Need to provide selective access to tasks** — Task gates fill this need, because they can reside in an LDT and can have a DPL that is different from the TSS descriptor's DPL. A program or procedure that does not have sufficient privilege to access the TSS descriptor for a task in the GDT (which usually has a DPL of 0) may be allowed access to the task through a task gate with a higher DPL. Task gates give the operating system greater latitude for limiting access to specific tasks.
- **Need for an interrupt or exception to be handled by an independent task** — Task gates may also reside in the IDT, which allows interrupts and exceptions to be handled by handler tasks. When an interrupt or exception vector points to a task gate, the processor switches to the specified task.

Figure 7-7 illustrates how a task gate in an LDT, a task gate in the GDT, and a task gate in the IDT can all point to the same task.

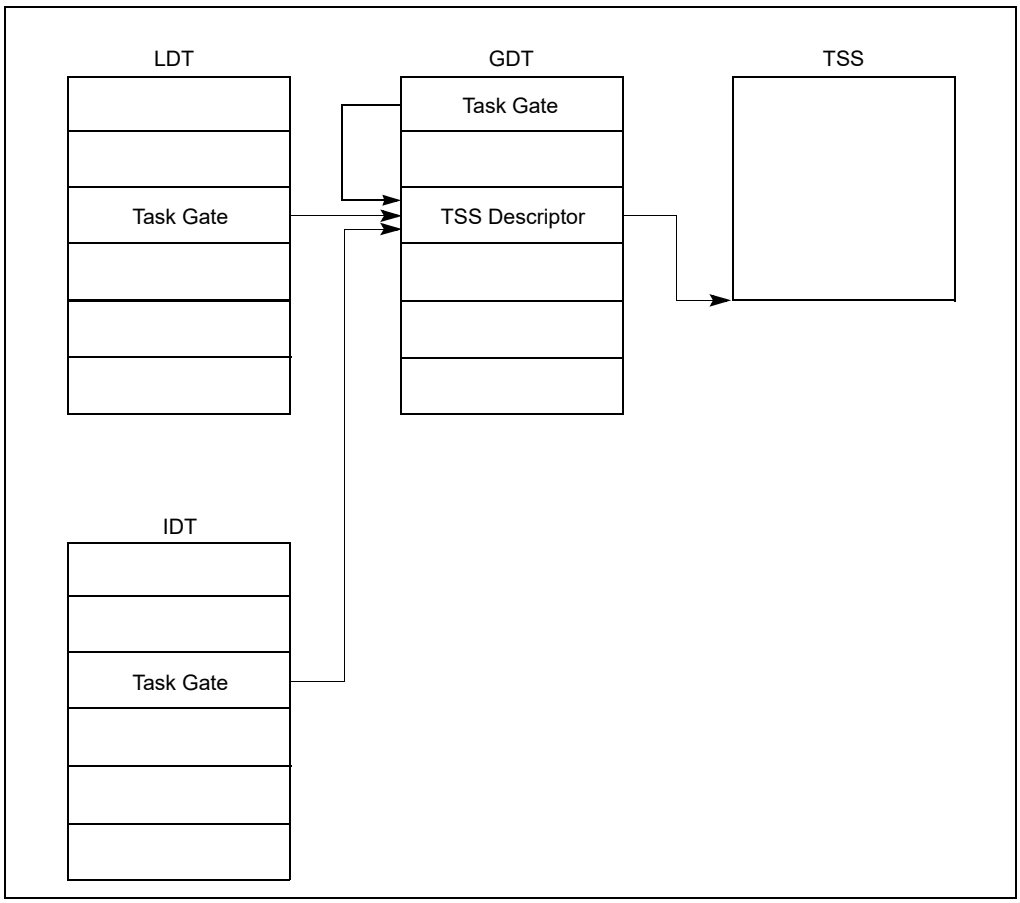


Figure 7-7. Task Gates Referencing the Same Task

7.3 TASK SWITCHING

The processor transfers execution to another task in one of four cases:

- The current program, task, or procedure executes a JMP or CALL instruction to a TSS descriptor in the GDT.
- The current program, task, or procedure executes a JMP or CALL instruction to a task-gate descriptor in the GDT or the current LDT.

- An interrupt or exception vector points to a task-gate descriptor in the IDT.
- The current task executes an IRET when the NT flag in the EFLAGS register is set.

JMP, CALL, and IRET instructions, as well as interrupts and exceptions, are all mechanisms for redirecting a program. The referencing of a TSS descriptor or a task gate (when calling or jumping to a task) or the state of the NT flag (when executing an IRET instruction) determines whether a task switch occurs.

The processor performs the following operations when switching to a new task:

1. Obtains the TSS segment selector for the new task as the operand of the JMP or CALL instruction, from a task gate, or from the previous task link field (for a task switch initiated with an IRET instruction).
2. Checks that the current (old) task is allowed to switch to the new task. Data-access privilege rules apply to JMP and CALL instructions. The CPL of the current (old) task and the RPL of the segment selector for the new task must be less than or equal to the DPL of the TSS descriptor or task gate being referenced. Exceptions, interrupts (except for interrupts generated by the INT *n* instruction), and the IRET instruction are permitted to switch tasks regardless of the DPL of the destination task-gate or TSS descriptor. For interrupts generated by the INT *n* instruction, the DPL is checked.
3. Checks that the TSS descriptor of the new task is marked present and has a valid limit (greater than or equal to 67H).
4. Checks that the new task is available (call, jump, exception, or interrupt) or busy (IRET return).
5. Checks that the current (old) TSS, new TSS, and all segment descriptors used in the task switch are paged into system memory.
6. If the task switch was initiated with a JMP or IRET instruction, the processor clears the busy (B) flag in the current (old) task's TSS descriptor; if initiated with a CALL instruction, an exception, or an interrupt: the busy (B) flag is left set. (See Table 7-2.)
7. If the task switch was initiated with an IRET instruction, the processor clears the NT flag in a temporarily saved image of the EFLAGS register; if initiated with a CALL or JMP instruction, an exception, or an interrupt, the NT flag is left unchanged in the saved EFLAGS image.
8. Saves the state of the current (old) task in the current task's TSS. The processor finds the base address of the current TSS in the task register and then copies the states of the following registers into the current TSS: all the general-purpose registers, segment selectors from the segment registers, the temporarily saved image of the EFLAGS register, and the instruction pointer register (EIP).
9. If the task switch was initiated with a CALL instruction, an exception, or an interrupt, the processor will set the NT flag in the EFLAGS loaded from the new task. If initiated with an IRET instruction or JMP instruction, the NT flag will reflect the state of NT in the EFLAGS loaded from the new task (see Table 7-2).
10. If the task switch was initiated with a CALL instruction, JMP instruction, an exception, or an interrupt, the processor sets the busy (B) flag in the new task's TSS descriptor; if initiated with an IRET instruction, the busy (B) flag is left set.
11. Loads the task register with the segment selector and descriptor for the new task's TSS.
12. The TSS state is loaded into the processor. This includes the LDTR register, the PDBR (control register CR3), the EFLAGS register, the EIP register, the general-purpose registers, and the segment selectors. A fault during the load of this state may corrupt architectural state. (If paging is not enabled, a PDBR value is read from the new task's TSS, but it is not loaded into CR3.)
13. The descriptors associated with the segment selectors are loaded and qualified. Any errors associated with this loading and qualification occur in the context of the new task and may corrupt architectural state.

NOTES

If all checks and saves have been carried out successfully, the processor commits to the task switch. If an unrecoverable error occurs in steps 1 through 11, the processor does not complete the task switch and insures that the processor is returned to its state prior to the execution of the instruction that initiated the task switch.

If an unrecoverable error occurs in step 12, architectural state may be corrupted, but an attempt will be made to handle the error in the prior execution environment. If an unrecoverable error

occurs after the commit point (in step 13), the processor completes the task switch (without performing additional access and segment availability checks) and generates the appropriate exception prior to beginning execution of the new task.

If exceptions occur after the commit point, the exception handler must finish the task switch itself before allowing the processor to begin executing the new task. See Chapter 6, "Interrupt 10—Invalid TSS Exception (#TS)," for more information about the affect of exceptions on a task when they occur after the commit point of a task switch.

14. Begins executing the new task. (To an exception handler, the first instruction of the new task appears not to have been executed.)

The state of the currently executing task is always saved when a successful task switch occurs. If the task is resumed, execution starts with the instruction pointed to by the saved EIP value, and the registers are restored to the values they held when the task was suspended.

When switching tasks, the privilege level of the new task does not inherit its privilege level from the suspended task. The new task begins executing at the privilege level specified in the CPL field of the CS register, which is loaded from the TSS. Because tasks are isolated by their separate address spaces and TSSs and because privilege rules control access to a TSS, software does not need to perform explicit privilege checks on a task switch.

Table 7-1 shows the exception conditions that the processor checks for when switching tasks. It also shows the exception that is generated for each check if an error is detected and the segment that the error code references. (The order of the checks in the table is the order used in the P6 family processors. The exact order is model specific and may be different for other IA-32 processors.) Exception handlers designed to handle these exceptions may be subject to recursive calls if they attempt to reload the segment selector that generated the exception. The cause of the exception (or the first of multiple causes) should be fixed before reloading the selector.

Table 7-1. Exception Conditions Checked During a Task Switch

Condition Checked	Exception[1]	Error Code Reference[2]
Segment selector for a TSS descriptor references the GDT and is within the limits of the table.	#GP #TS (for IRET)	New Task's TSS
P bit is set in TSS descriptor.	#NP	New Task's TSS
TSS descriptor is not busy (for task switch initiated by a call, interrupt, or exception).	#GP (for JMP, CALL, INT)	Task's back-link TSS
TSS descriptor is not busy (for task switch initiated by an IRET instruction).	#TS (for IRET)	New Task's TSS
TSS segment limit greater than or equal to 108 (for 32-bit TSS) or 44 (for 16-bit TSS).	#TS	New Task's TSS
Registers are loaded from the values in the TSS.		
LDT segment selector of new task is valid [3].	#TS	New Task's LDT
If code segment is non-conforming, its DPL should equal its RPL.	#TS	New Code Segment
If code segment is conforming, its DPL should be less than or equal to its RPL.	#TS	New Code Segment
SS segment selector is valid [2].	#TS	New Stack Segment
P bit is set in stack segment descriptor.	#SS	New Stack Segment
Stack segment DPL should equal CPL.	#TS	New stack segment
P bit is set in new task's LDT descriptor.	#TS	New Task's LDT
CS segment selector is valid [3].	#TS	New Code Segment
P bit is set in code segment descriptor.	#NP	New Code Segment
Stack segment DPL should equal its RPL.	#TS	New Stack Segment
DS, ES, FS, and GS segment selectors are valid [3].	#TS	New Data Segment
DS, ES, FS, and GS segments are readable.	#TS	New Data Segment

Table 7-1. Exception Conditions Checked During a Task Switch (Contd.)

Condition Checked	Exception[1]	Error Code Reference[2]
P bits are set in descriptors of DS, ES, FS, and GS segments.	#NP	New Data Segment
DS, ES, FS, and GS segment DPL greater than or equal to CPL (unless these are conforming segments).	#TS	New Data Segment

NOTES:

1. #NP is segment-not-present exception, #GP is general-protection exception, #TS is invalid-TSS exception, and #SS is stack-fault exception.
2. The error code contains an index to the segment descriptor referenced in this column.
3. A segment selector is valid if it is in a compatible type of table (GDT or LDT), occupies an address within the table's segment limit, and refers to a compatible type of descriptor (for example, a segment selector in the CS register only is valid when it points to a code-segment descriptor).

The TS (task switched) flag in the control register CR0 is set every time a task switch occurs. System software uses the TS flag to coordinate the actions of floating-point unit when generating floating-point exceptions with the rest of the processor. The TS flag indicates that the context of the floating-point unit may be different from that of the current task. See Section 2.5, "Control Registers", for a detailed description of the function and use of the TS flag.

7.4 TASK LINKING

The previous task link field of the TSS (sometimes called the "backlink") and the NT flag in the EFLAGS register are used to return execution to the previous task. EFLAGS.NT = 1 indicates that the currently executing task is nested within the execution of another task.

When a CALL instruction, an interrupt, or an exception causes a task switch: the processor copies the segment selector for the current TSS to the previous task link field of the TSS for the new task; it then sets EFLAGS.NT = 1. If software uses an IRET instruction to suspend the new task, the processor checks for EFLAGS.NT = 1; it then uses the value in the previous task link field to return to the previous task. See Figures 7-8.

When a JMP instruction causes a task switch, the new task is not nested. The previous task link field is not used and EFLAGS.NT = 0. Use a JMP instruction to dispatch a new task when nesting is not desired.

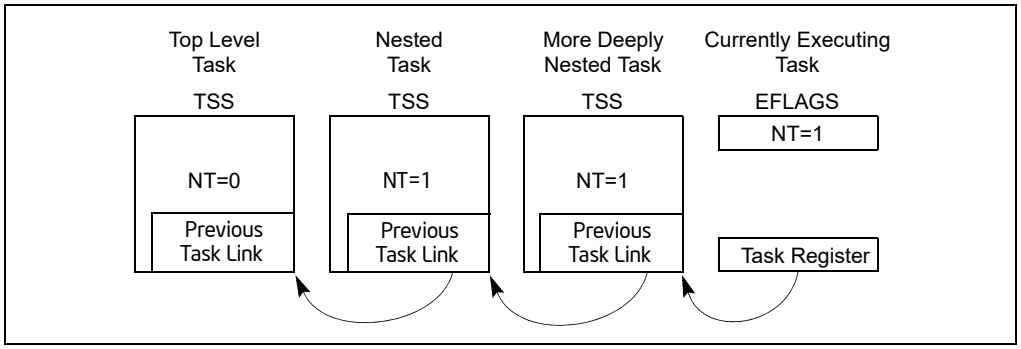


Figure 7-8. Nested Tasks

Table 7-2 shows the busy flag (in the TSS segment descriptor), the NT flag, the previous task link field, and TS flag (in control register CR0) during a task switch.

The NT flag may be modified by software executing at any privilege level. It is possible for a program to set the NT flag and execute an IRET instruction. This might randomly invoke the task specified in the previous link field of the current task's TSS. To keep such spurious task switches from succeeding, the operating system should initialize the previous task link field in every TSS that it creates to 0.

Table 7-2. Effect of a Task Switch on Busy Flag, NT Flag, Previous Task Link Field, and TS Flag

Flag or Field	Effect of JMP instruction	Effect of CALL Instruction or Interrupt	Effect of IRET Instruction
Busy (B) flag of new task.	Flag is set. Must have been clear before.	Flag is set. Must have been clear before.	No change. Must have been set.
Busy flag of old task.	Flag is cleared.	No change. Flag is currently set.	Flag is cleared.
NT flag of new task.	Set to value from TSS of new task.	Flag is set.	Set to value from TSS of new task.
NT flag of old task.	No change.	No change.	Flag is cleared.
Previous task link field of new task.	No change.	Loaded with selector for old task's TSS.	No change.
Previous task link field of old task.	No change.	No change.	No change.
TS flag in control register CR0.	Flag is set.	Flag is set.	Flag is set.

7.4.1 Use of Busy Flag To Prevent Recursive Task Switching

A TSS allows only one context to be saved for a task; therefore, once a task is called (dispatched), a recursive (or re-entrant) call to the task would cause the current state of the task to be lost. The busy flag in the TSS segment descriptor is provided to prevent re-entrant task switching and a subsequent loss of task state information. The processor manages the busy flag as follows:

1. When dispatching a task, the processor sets the busy flag of the new task.
2. If during a task switch, the current task is placed in a nested chain (the task switch is being generated by a CALL instruction, an interrupt, or an exception), the busy flag for the current task remains set.
3. When switching to the new task (initiated by a CALL instruction, interrupt, or exception), the processor generates a general-protection exception (#GP) if the busy flag of the new task is already set. If the task switch is initiated with an IRET instruction, the exception is not raised because the processor expects the busy flag to be set.
4. When a task is terminated by a jump to a new task (initiated with a JMP instruction in the task code) or by an IRET instruction in the task code, the processor clears the busy flag, returning the task to the "not busy" state.

The processor prevents recursive task switching by preventing a task from switching to itself or to any task in a nested chain of tasks. The chain of nested suspended tasks may grow to any length, due to multiple calls, interrupts, or exceptions. The busy flag prevents a task from being invoked if it is in this chain.

The busy flag may be used in multiprocessor configurations, because the processor follows a LOCK protocol (on the bus or in the cache) when it sets or clears the busy flag. This lock keeps two processors from invoking the same task at the same time. See Section 8.1.2.1, "Automatic Locking," for more information about setting the busy flag in a multiprocessor applications.

7.4.2 Modifying Task Linkages

In a uniprocessor system, in situations where it is necessary to remove a task from a chain of linked tasks, use the following procedure to remove the task:

1. Disable interrupts.
2. Change the previous task link field in the TSS of the pre-empting task (the task that suspended the task to be removed). It is assumed that the pre-empting task is the next task (newer task) in the chain from the task to be removed. Change the previous task link field to point to the TSS of the next oldest task in the chain or to an even older task in the chain.
3. Clear the busy (B) flag in the TSS segment descriptor for the task being removed from the chain. If more than one task is being removed from the chain, the busy flag for each task being remove must be cleared.

4. Enable interrupts.

In a multiprocessing system, additional synchronization and serialization operations must be added to this procedure to insure that the TSS and its segment descriptor are both locked when the previous task link field is changed and the busy flag is cleared.

7.5 TASK ADDRESS SPACE

The address space for a task consists of the segments that the task can access. These segments include the code, data, stack, and system segments referenced in the TSS and any other segments accessed by the task code. The segments are mapped into the processor's linear address space, which is in turn mapped into the processor's physical address space (either directly or through paging).

The LDT segment field in the TSS can be used to give each task its own LDT. Giving a task its own LDT allows the task address space to be isolated from other tasks by placing the segment descriptors for all the segments associated with the task in the task's LDT.

It also is possible for several tasks to use the same LDT. This is a memory-efficient way to allow specific tasks to communicate with or control each other, without dropping the protection barriers for the entire system.

Because all tasks have access to the GDT, it also is possible to create shared segments accessed through segment descriptors in this table.

If paging is enabled, the CR3 register (PDBR) field in the TSS allows each task to have its own set of page tables for mapping linear addresses to physical addresses. Or, several tasks can share the same set of page tables.

7.5.1 Mapping Tasks to the Linear and Physical Address Spaces

Tasks can be mapped to the linear address space and physical address space in one of two ways:

- **One linear-to-physical address space mapping is shared among all tasks.** — When paging is not enabled, this is the only choice. Without paging, all linear addresses map to the same physical addresses. When paging is enabled, this form of linear-to-physical address space mapping is obtained by using one page directory for all tasks. The linear address space may exceed the available physical space if demand-paged virtual memory is supported.
- **Each task has its own linear address space that is mapped to the physical address space.** — This form of mapping is accomplished by using a different page directory for each task. Because the PDBR (control register CR3) is loaded on task switches, each task may have a different page directory.

The linear address spaces of different tasks may map to completely distinct physical addresses. If the entries of different page directories point to different page tables and the page tables point to different pages of physical memory, then the tasks do not share physical addresses.

With either method of mapping task linear address spaces, the TSSs for all tasks must lie in a shared area of the physical space, which is accessible to all tasks. This mapping is required so that the mapping of TSS addresses does not change while the processor is reading and updating the TSSs during a task switch. The linear address space mapped by the GDT also should be mapped to a shared area of the physical space; otherwise, the purpose of the GDT is defeated. Figure 7-9 shows how the linear address spaces of two tasks can overlap in the physical space by sharing page tables.

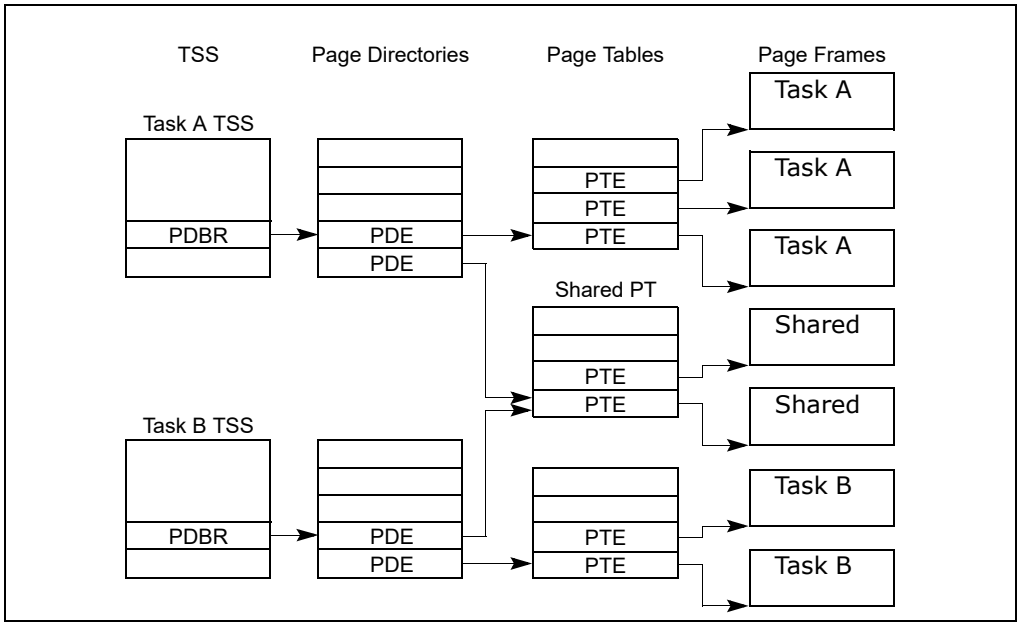


Figure 7-9. Overlapping Linear-to-Physical Mappings

7.5.2 Task Logical Address Space

To allow the sharing of data among tasks, use the following techniques to create shared logical-to-physical address-space mappings for data segments:

- **Through the segment descriptors in the GDT** — All tasks must have access to the segment descriptors in the GDT. If some segment descriptors in the GDT point to segments in the linear-address space that are mapped into an area of the physical-address space common to all tasks, then all tasks can share the data and code in those segments.
- **Through a shared LDT** — Two or more tasks can use the same LDT if the LDT fields in their TSSs point to the same LDT. If some segment descriptors in a shared LDT point to segments that are mapped to a common area of the physical address space, the data and code in those segments can be shared among the tasks that share the LDT. This method of sharing is more selective than sharing through the GDT, because the sharing can be limited to specific tasks. Other tasks in the system may have different LDTs that do not give them access to the shared segments.
- **Through segment descriptors in distinct LDTs that are mapped to common addresses in linear address space** — If this common area of the linear address space is mapped to the same area of the physical address space for each task, these segment descriptors permit the tasks to share segments. Such segment descriptors are commonly called aliases. This method of sharing is even more selective than those listed above, because, other segment descriptors in the LDTs may point to independent linear addresses which are not shared.

7.6 16-BIT TASK-STATE SEGMENT (TSS)

The 32-bit IA-32 processors also recognize a 16-bit TSS format like the one used in Intel 286 processors (see Figure 7-10). This format is supported for compatibility with software written to run on earlier IA-32 processors.

The following information is important to know about the 16-bit TSS.

- Do not use a 16-bit TSS to implement a virtual-8086 task.
- The valid segment limit for a 16-bit TSS is 2CH.

- The 16-bit TSS does not contain a field for the base address of the page directory, which is loaded into control register CR3. A separate set of page tables for each task is not supported for 16-bit tasks. If a 16-bit task is dispatched, the page-table structure for the previous task is used.
- The I/O base address is not included in the 16-bit TSS. None of the functions of the I/O map are supported.
- When task state is saved in a 16-bit TSS, the upper 16 bits of the EFLAGS register and the EIP register are lost.
- When the general-purpose registers are loaded or saved from a 16-bit TSS, the upper 16 bits of the registers are modified and not maintained.

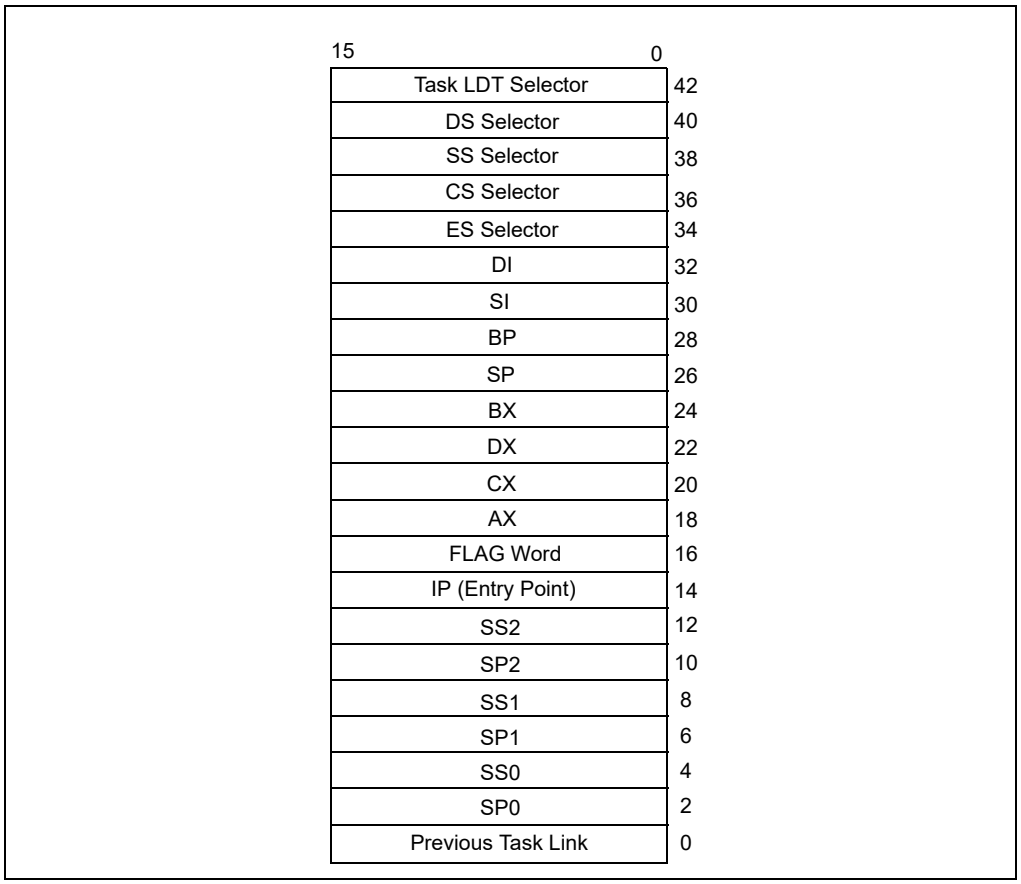


Figure 7-10. 16-Bit TSS Format

7.7 TASK MANAGEMENT IN 64-BIT MODE

In 64-bit mode, task structure and task state are similar to those in protected mode. However, the task switching mechanism available in protected mode is not supported in 64-bit mode. Task management and switching must be performed by software. The processor issues a general-protection exception (#GP) if the following is attempted in 64-bit mode:

- Control transfer to a TSS or a task gate using JMP, CALL, INTn, or interrupt.
- An IRET with EFLAGS.NT (nested task) set to 1.

Although hardware task-switching is not supported in 64-bit mode, a 64-bit task state segment (TSS) must exist. Figure 7-11 shows the format of a 64-bit TSS. The TSS holds information important to 64-bit mode and that is not directly related to the task-switch mechanism. This information includes:

- **RSPn** — The full 64-bit canonical forms of the stack pointers (RSP) for privilege levels 0-2.
- **ISTn** — The full 64-bit canonical forms of the interrupt stack table (IST) pointers.
- **I/O map base address** — The 16-bit offset to the I/O permission bit map from the 64-bit TSS base.

The operating system must create at least one 64-bit TSS after activating IA-32e mode. It must execute the LTR instruction (in 64-bit mode) to load the TR register with a pointer to the 64-bit TSS responsible for both 64-bit-mode programs and compatibility-mode programs.

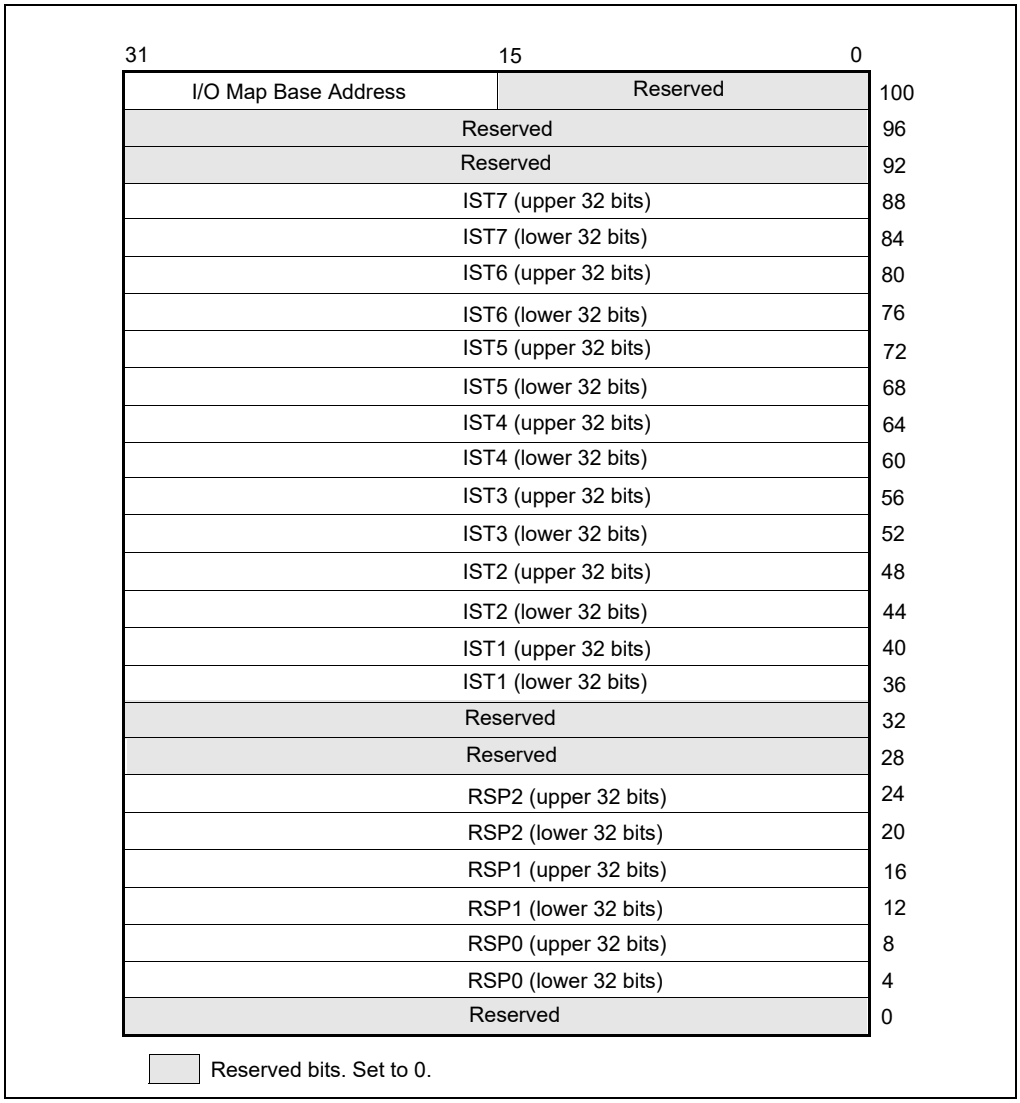


Figure 7-11. 64-Bit TSS Format

12.Updates to Chapter 9, Volume 3A

Change bars show changes to Chapter 9 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A:* System Programming Guide, Part 1.

Changes to this chapter: Updates to change references to "the BIOS" to "software"; this reflects the expectations of software, which is more than just the BIOS. Update to requirements listed in Section 9.11.6 "Microcode Update Loader".

CHAPTER 9
PROCESSOR MANAGEMENT AND INITIALIZATION

This chapter describes the facilities provided for managing processor wide functions and for initializing the processor. The subjects covered include: processor initialization, x87 FPU initialization, processor configuration, feature determination, mode switching, the MSRs (in the Pentium, P6 family, Pentium 4, and Intel Xeon processors), and the MTRRs (in the P6 family, Pentium 4, and Intel Xeon processors).

9.1 INITIALIZATION OVERVIEW

Following power-up or an assertion of the RESET# pin, each processor on the system bus performs a hardware initialization of the processor (known as a hardware reset) and an optional built-in self-test (BIST). A hardware reset sets each processor's registers to a known state and places the processor in real-address mode. It also invalidates the internal caches, translation lookaside buffers (TLBs) and the branch target buffer (BTB). At this point, the action taken depends on the processor family:

- **Pentium 4 processors (CPUID DisplayFamily 0FH)** — All the processors on the system bus (including a single processor in a uniprocessor system) execute the multiple processor (MP) initialization protocol. The processor that is selected through this protocol as the bootstrap processor (BSP) then immediately starts executing software-initialization code in the current code segment beginning at the offset in the EIP register. The application (non-BSP) processors (APs) go into a Wait For Startup IPI (SIPI) state while the BSP is executing initialization code. See Section 8.4, "Multiple-Processor (MP) Initialization," for more details. Note that in a uniprocessor system, the single Pentium 4 or Intel Xeon processor automatically becomes the BSP.
- **IA-32 and Intel 64 processors (CPUID DisplayFamily 06H)** — The action taken is the same as for the Pentium 4 processors (as described in the previous paragraph).
- **Pentium processors** — In either a single- or dual- processor system, a single Pentium processor is always pre-designated as the primary processor. Following a reset, the primary processor behaves as follows in both single- and dual-processor systems. Using the dual-processor (DP) ready initialization protocol, the primary processor immediately starts executing software-initialization code in the current code segment beginning at the offset in the EIP register. The secondary processor (if there is one) goes into a halt state.
- **Intel486 processor** — The primary processor (or single processor in a uniprocessor system) immediately starts executing software-initialization code in the current code segment beginning at the offset in the EIP register. (The Intel486 does not automatically execute a DP or MP initialization protocol to determine which processor is the primary processor.)

The software-initialization code performs all system-specific initialization of the BSP or primary processor and the system logic.

At this point, for MP (or DP) systems, the BSP (or primary) processor wakes up each AP (or secondary) processor to enable those processors to execute self-configuration code.

When all processors are initialized, configured, and synchronized, the BSP or primary processor begins executing an initial operating-system or executive task.

The x87 FPU is also initialized to a known state during hardware reset. x87 FPU software initialization code can then be executed to perform operations such as setting the precision of the x87 FPU and the exception masks. No special initialization of the x87 FPU is required to switch operating modes.

Asserting the INIT# pin on the processor invokes a similar response to a hardware reset. The major difference is that during an INIT, the internal caches, MSRs, MTRRs, and x87 FPU state are left unchanged (although, the TLBs and BTB are invalidated as with a hardware reset). An INIT provides a method for switching from protected to real-address mode while maintaining the contents of the internal caches.

9.1.1 Processor State After Reset

Following power-up, The state of control register CR0 is 60000010H (see Figure 9-1). This places the processor is in real-address mode with paging disabled.

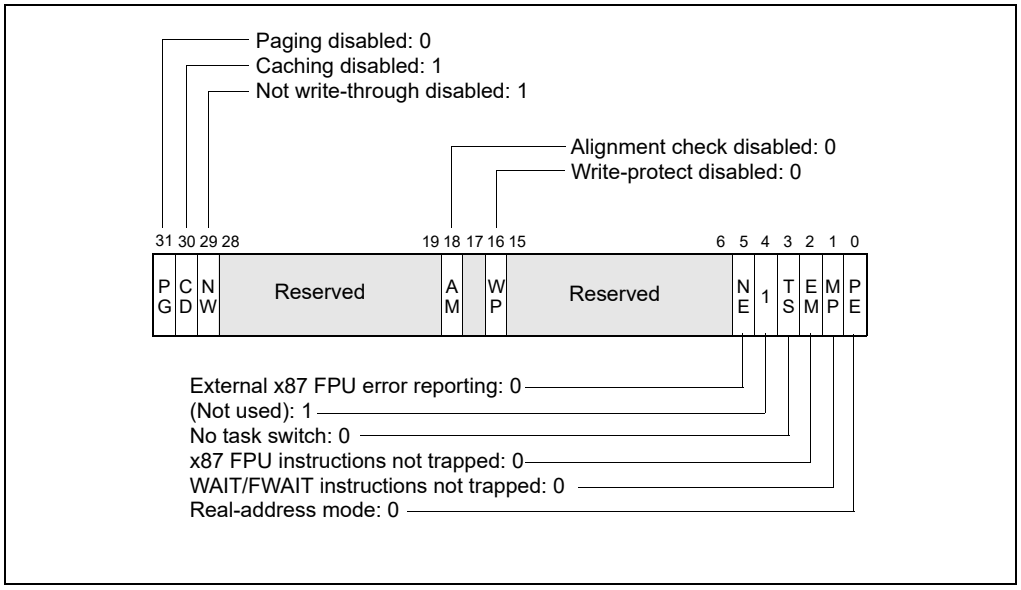


Figure 9-1. Contents of CR0 Register after Reset

The state of the flags and other registers following power-up for the Pentium 4, Pentium Pro, and Pentium processors are shown in Section 22.39, "Initial State of Pentium, Pentium Pro and Pentium 4 Processors" of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3B*.

Table 9-1 shows processor states of IA-32 and Intel 64 processors with CPUID DisplayFamily signature of 06H at the following events: power-up, RESET, and INIT. In a few cases, the behavior of some registers behave slightly different across warm RESET, the variant cases are marked in Table 9-1 and described in more detail in Table 9-2.

Table 9-1. IA-32 and Intel 64 Processor States Following Power-up, Reset, or INIT

Register	Power up	Reset	INIT
EFLAGS[1]	00000002H	00000002H	00000002H
EIP	0000FFF0H	0000FFF0H	0000FFF0H
CR0	60000010H[2]	60000010H[2]	60000010H[2]
CR2, CR3, CR4	00000000H	00000000H	00000000H
CS	Selector = F000H Base = FFFF0000H Limit = FFFFH AR = Present, R/W, Accessed	Selector = F000H Base = FFFF0000H Limit = FFFFH AR = Present, R/W, Accessed	Selector = F000H Base = FFFF0000H Limit = FFFFH AR = Present, R/W, Accessed
SS, DS, ES, FS, GS	Selector = 0000H Base = 00000000H Limit = FFFFH AR = Present, R/W, Accessed	Selector = 0000H Base = 00000000H Limit = FFFFH AR = Present, R/W, Accessed	Selector = 0000H Base = 00000000H Limit = FFFFH AR = Present, R/W, Accessed
EDX	000n06xxH[3]	000n06xxH[3]	000n06xxH[3]
EAX	0[4]	0[4]	0[4]
EBX, ECX, ESI, EDI, EBP, ESP	00000000H	00000000H	00000000H
ST0 through ST7[5]	+0.0	+0.0	FINIT/FNINIT: Unchanged

Table 9-1. IA-32 and Intel 64 Processor States Following Power-up, Reset, or INIT (Contd.)

Register	Power up	Reset	INIT
x87 FPU Control Word[5]	0040H	0040H	FINIT/FNINIT: 037FH
x87 FPU Status Word[5]	0000H	0000H	FINIT/FNINIT: 0000H
x87 FPU Tag Word[5]	5555H	5555H	FINIT/FNINIT: FFFFH
x87 FPU Data Operand and CS Seg. Selectors[5]	0000H	0000H	FINIT/FNINIT: 0000H
x87 FPU Data Operand and Inst. Pointers[5]	00000000H	00000000H	FINIT/FNINIT: 00000000H
MM0 through MM7[5]	0000000000000000H	0000000000000000H	INIT or FINIT/FNINIT: Unchanged
XMM0 through XMM7	0H	0H	Unchanged
MXCSR	1F80H	1F80H	Unchanged
GDTR, IDTR	Base = 00000000H Limit = FFFFH AR = Present, R/W	Base = 00000000H Limit = FFFFH AR = Present, R/W	Base = 00000000H Limit = FFFFH AR = Present, R/W
LDTR, Task Register	Selector = 0000H Base = 00000000H Limit = FFFFH AR = Present, R/W	Selector = 0000H Base = 00000000H Limit = FFFFH AR = Present, R/W	Selector = 0000H Base = 00000000H Limit = FFFFH AR = Present, R/W
DR0, DR1, DR2, DR3	00000000H	00000000H	00000000H
DR6	FFFF0FF0H	FFFF0FF0H	FFFF0FF0H
DR7	00000400H	00000400H	00000400H
R8-R15	0000000000000000H	0000000000000000H	0000000000000000H
XMM8-XMM15	0H	0H	Unchanged
XCR0	1H	1H	Unchanged
IA32_XSS	0H	0H	0H
YMM_H[255:128]	0H	0H	Unchanged
BNDCFGU	0H	0H	0H
BND0-BND3	0H	0H	0H
IA32_BNDCFGS	0H	0H	0H
OPMASK	0H	0H	Unchanged
ZMM_H[511:256]	0H	0H	Unchanged
ZMMHi16[511:0]	0H	0H	Unchanged
PKRU	0H	0H	Unchanged
Intel Processor Trace MSRs	0H	0H[W]	Unchanged
Time-Stamp Counter	0H	0H[W]	Unchanged
IA32_TSC_AUX	0H	0H	Unchanged
IA32_TSC_ADJUST	0H	0H	Unchanged
IA32_TSC_DEADLINE	0H	0H	Unchanged
IA32_SYSENTER_CS/ESP/EIP	0H	0H	Unchanged
IA32_EFER	0000000000000000H	0000000000000000H	0000000000000000H
IA32_STAR/LSTAR	0H	0H	Unchanged
IA32_FS_BASE/GS_BASE	0H	0H	0H

Table 9-1. IA-32 and Intel 64 Processor States Following Power-up, Reset, or INIT (Contd.)

Register	Power up	Reset	INIT
IA32_PMCx, IA32_PERFEVTSELx	0H	0H	Unchanged
IA32_FIXED_CTRx, IA32_FIXED_CTR_CTRL, Global Perf Counter Controls	0H	0H	Unchanged
Data and Code Cache, TLBs	Invalid[6]	Invalid[6]	Unchanged
Fixed MTRRs	Disabled	Disabled	Unchanged
Variable MTRRs	Disabled	Disabled	Unchanged
Machine-Check Banks	Undefined	Undefined[W]	Unchanged
Last Branch Record Stack	0	0[W]	Unchanged
APIC	Enabled	Enabled	Unchanged
X2APIC	Disabled	Disabled	Unchanged
IA32_DEBUG_INTERFACE	0	0[W]	Unchanged

NOTES:

1. The 10 most-significant bits of the EFLAGS register are undefined following a reset. Software should not depend on the states of any of these bits.
2. The CD and NW flags are unchanged, bit 4 is set to 1, all other bits are cleared.
3. Where "n" is the Extended Model Value for the respective processor, and "xx" = don't care.
4. If Built-In Self-Test (BIST) is invoked on power up or reset, EAX is 0 only if all tests passed. (BIST cannot be invoked during an INIT.)
5. The state of the x87 FPU and MMX registers is not changed by the execution of an INIT.
6. Internal caches are invalid after power-up and RESET, but left unchanged with an INIT.

W: Warm RESET behavior differs from power-on RESET with details listed in Table 9-2.

Table 9-2. Variance of RESET Values in Selected Intel Architecture Processors

State	XREF	Value	Feature Flag or DisplayFamily_DisplayModel Signatures
Time-Stamp Counter	Warm RESET	Unmodified across warm Reset	06_2DH, 06_3EH
Machine-Check Banks	Warm RESET	IA32_MCi_Status banks are unmodified across warm Reset	06_2DH, 06_3EH, 06_3FH, 06_4FH, 06_56H
Last Branch Record Stack	Warm RESET	LBR stack MSRs are unmodified across warm Reset	06_1AH, 06_1CH, DisplayFamiy= 06 and DisplayModel >1DH
Intel Processor Trace MSRs	Warm RESET	Clears IA32_RTIT_CTL.TraceEn, the rest of MSRs are unmodified	If CPUID.(EAX=14H, ECX=0H):EBX[bit 2] = 1
IA32_DEBUG_INTERFACE	Warm RESET	Unmodified across warm Reset	If CPUID.01H:ECX.[11] = 1

9.1.2 Processor Built-In Self-Test (BIST)

Hardware may request that the BIST be performed at power-up. The EAX register is cleared (0H) if the processor passes the BIST. A nonzero value in the EAX register after the BIST indicates that a processor fault was detected. If the BIST is not requested, the contents of the EAX register after a hardware reset is 0H.

The overhead for performing a BIST varies between processor families. For example, the BIST takes approximately 30 million processor clock periods to execute on the Pentium 4 processor. This clock count is model-specific; Intel reserves the right to change the number of periods for any Intel 64 or IA-32 processor, without notification.

9.1.3 Model and Stepping Information

Following a hardware reset, the EDX register contains component identification and revision information (see Figure 9-2). For example, the model, family, and processor type returned for the first processor in the Intel Pentium 4 family is as follows: model (0000B), family (1111B), and processor type (00B).

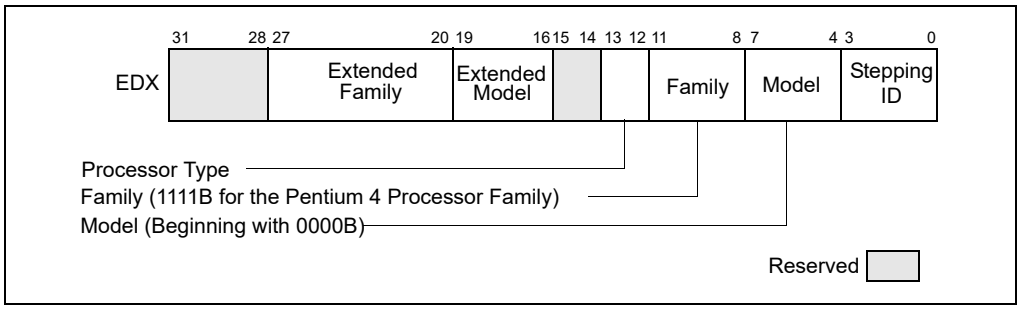


Figure 9-2. Version Information in the EDX Register after Reset

The stepping ID field contains a unique identifier for the processor's stepping ID or revision level. The extended family and extended model fields were added to the IA-32 architecture in the Pentium 4 processors.

9.1.4 First Instruction Executed

The first instruction that is fetched and executed following a hardware reset is located at physical address FFFFFFF0H. This address is 16 bytes below the processor's uppermost physical address. The EPROM containing the software-initialization code must be located at this address.

The address FFFFFFF0H is beyond the 1-MByte addressable range of the processor while in real-address mode. The processor is initialized to this starting address as follows. The CS register has two parts: the visible segment selector part and the hidden base address part. In real-address mode, the base address is normally formed by shifting the 16-bit segment selector value 4 bits to the left to produce a 20-bit base address. However, during a hardware reset, the segment selector in the CS register is loaded with F000H and the base address is loaded with FFFF0000H. The starting address is thus formed by adding the base address to the value in the EIP register (that is, FFFF0000 + FFF0H = FFFFFFF0H).

The first time the CS register is loaded with a new value after a hardware reset, the processor will follow the normal rule for address translation in real-address mode (that is, [CS base address = CS segment selector * 16]). To insure that the base address in the CS register remains unchanged until the EPROM based software-initialization code is completed, the code must not contain a far jump or far call or allow an interrupt to occur (which would cause the CS selector value to be changed).

9.2 X87 FPU INITIALIZATION

Software-initialization code can determine the whether the processor contains an x87 FPU by using the CPUID instruction. The code must then initialize the x87 FPU and set flags in control register CR0 to reflect the state of the x87 FPU environment.

A hardware reset places the x87 FPU in the state shown in Table 9-1. This state is different from the state the x87 FPU is placed in following the execution of an FINIT or FNINIT instruction (also shown in Table 9-1). If the x87 FPU is to be used, the software-initialization code should execute an FINIT/FNINIT instruction following a hardware reset. These instructions, tag all data registers as empty, clear all the exception masks, set the TOP-of-stack value to 0, and select the default rounding and precision controls setting (round to nearest and 64-bit precision).

If the processor is reset by asserting the INIT# pin, the x87 FPU state is not changed.

9.2.1 Configuring the x87 FPU Environment

Initialization code must load the appropriate values into the MP, EM, and NE flags of control register CR0. These bits are cleared on hardware reset of the processor. Figure 9-3 shows the suggested settings for these flags, depending on the IA-32 processor being initialized. Initialization code can test for the type of processor present before setting or clearing these flags.

Table 9-3. Recommended Settings of EM and MP Flags on IA-32 Processors

EM	MP	NE	IA-32 processor
1	0	1	Intel486™ SX, Intel386™ DX, and Intel386™ SX processors only, without the presence of a math coprocessor.
0	1	1 or 0*	Pentium 4, Intel Xeon, P6 family, Pentium, Intel486™ DX, and Intel 487 SX processors, and Intel386 DX and Intel386 SX processors when a companion math coprocessor is present.
0	1	1 or 0*	More recent Intel 64 or IA-32 processors

NOTE:

* The setting of the NE flag depends on the operating system being used.

The EM flag determines whether floating-point instructions are executed by the x87 FPU (EM is cleared) or a device-not-available exception (#NM) is generated for all floating-point instructions so that an exception handler can emulate the floating-point operation (EM = 1). Ordinarily, the EM flag is cleared when an x87 FPU or math coprocessor is present and set if they are not present. If the EM flag is set and no x87 FPU, math coprocessor, or floating-point emulator is present, the processor will hang when a floating-point instruction is executed.

The MP flag determines whether WAIT/FWAIT instructions react to the setting of the TS flag. If the MP flag is clear, WAIT/FWAIT instructions ignore the setting of the TS flag; if the MP flag is set, they will generate a device-not-available exception (#NM) if the TS flag is set. Generally, the MP flag should be set for processors with an integrated x87 FPU and clear for processors without an integrated x87 FPU and without a math coprocessor present. However, an operating system can choose to save the floating-point context at every context switch, in which case there would be no need to set the MP bit.

Table 2-2 shows the actions taken for floating-point and WAIT/FWAIT instructions based on the settings of the EM, MP, and TS flags.

The NE flag determines whether unmasked floating-point exceptions are handled by generating a floating-point error exception internally (NE is set, native mode) or through an external interrupt (NE is cleared). In systems where an external interrupt controller is used to invoke numeric exception handlers (such as MS-DOS-based systems), the NE bit should be cleared.

9.2.2 Setting the Processor for x87 FPU Software Emulation

Setting the EM flag causes the processor to generate a device-not-available exception (#NM) and trap to a software exception handler whenever it encounters a floating-point instruction. (Table 9-3 shows when it is appropriate to use this flag.) Setting this flag has two functions:

- It allows x87 FPU code to run on an IA-32 processor that has neither an integrated x87 FPU nor is connected to an external math coprocessor, by using a floating-point emulator.
- It allows floating-point code to be executed using a special or nonstandard floating-point emulator, selected for a particular application, regardless of whether an x87 FPU or math coprocessor is present.

To emulate floating-point instructions, the EM, MP, and NE flag in control register CR0 should be set as shown in Table 9-4.

Table 9-4. Software Emulation Settings of EM, MP, and NE Flags

CR0 Bit	Value
EM	1
MP	0
NE	1

Regardless of the value of the EM bit, the Intel486 SX processor generates a device-not-available exception (#NM) upon encountering any floating-point instruction.

9.3 CACHE ENABLING

IA-32 processors (beginning with the Intel486 processor) and Intel 64 processors contain internal instruction and data caches. These caches are enabled by clearing the CD and NW flags in control register CR0. (They are set during a hardware reset.) Because all internal cache lines are invalid following reset initialization, it is not necessary to invalidate the cache before enabling caching. Any external caches may require initialization and invalidation using a system-specific initialization and invalidation code sequence.

Depending on the hardware and operating system or executive requirements, additional configuration of the processor's caching facilities will probably be required. Beginning with the Intel486 processor, page-level caching can be controlled with the PCD and PWT flags in page-directory and page-table entries. Beginning with the P6 family processors, the memory type range registers (MTRRs) control the caching characteristics of the regions of physical memory. (For the Intel486 and Pentium processors, external hardware can be used to control the caching characteristics of regions of physical memory.) See Chapter 11, "Memory Cache Control," for detailed information on configuration of the caching facilities in the Pentium 4, Intel Xeon, and P6 family processors and system memory.

9.4 MODEL-SPECIFIC REGISTERS (MSRS)

Most IA-32 processors (starting from Pentium processors) and Intel 64 processors contain a model-specific registers (MSRs). A given MSR may not be supported across all families and models for Intel 64 and IA-32 processors. Some MSRs are designated as architectural to simplify software programming; a feature introduced by an architectural MSR is expected to be supported in future processors. Non-architectural MSRs are not guaranteed to be supported or to have the same functions on future processors.

MSRs that provide control for a number of hardware and software-related features, include:

- Performance-monitoring counters (see Chapter 23, "Introduction to Virtual Machine Extensions").
- Debug extensions (see Chapter 23, "Introduction to Virtual Machine Extensions.").
- Machine-check exception capability and its accompanying machine-check architecture (see Chapter 15, "Machine-Check Architecture").
- MTRRs (see Section 11.11, "Memory Type Range Registers (MTRRs)").
- Thermal and power management.
- Instruction-specific support (for example: SYSENTER, SYSEXIT, SWAPGS, etc.).
- Processor feature/mode support (for example: IA32_EFER, IA32_FEATURE_CONTROL).

The MSRs can be read and written to using the RDMSR and WRMSR instructions, respectively.

When performing software initialization of an IA-32 or Intel 64 processor, many of the MSRs will need to be initialized to set up things like performance-monitoring events, run-time machine checks, and memory types for physical memory.

Lists of available performance-monitoring events are given in Chapter 19, "Performance Monitoring Events", and lists of available MSRs are given in Chapter 2, "Model-Specific Registers (MSRs)" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 4*. The references earlier in this section show where the functions of the various groups of MSRs are described in this manual.

9.5 MEMORY TYPE RANGE REGISTERS (MTRRS)

Memory type range registers (MTRRs) were introduced into the IA-32 architecture with the Pentium Pro processor. They allow the type of caching (or no caching) to be specified in system memory for selected physical address ranges. They allow memory accesses to be optimized for various types of memory such as RAM, ROM, frame buffer memory, and memory-mapped I/O devices.

In general, initializing the MTRRs is normally handled by the software initialization code or BIOS and is not an operating system or executive function. At the very least, all the MTRRs must be cleared to 0, which selects the uncached (UC) memory type. See Section 11.11, "Memory Type Range Registers (MTRRs)," for detailed information on the MTRRs.

9.6 INITIALIZING SSE/SSE2/SSE3/SSSE3 EXTENSIONS

For processors that contain SSE/SSE2/SSE3/SSSE3 extensions, steps must be taken when initializing the processor to allow execution of these instructions.

1. Check the CPUID feature flags for the presence of the SSE/SSE2/SSE3/SSSE3 extensions (respectively: EDX bits 25 and 26, ECX bit 0 and 9) and support for the FXSAVE and FXRSTOR instructions (EDX bit 24). Also check for support for the CLFLUSH instruction (EDX bit 19). The CPUID feature flags are loaded in the EDX and ECX registers when the CPUID instruction is executed with a 1 in the EAX register.
2. Set the OSFXSR flag (bit 9 in control register CR4) to indicate that the operating system supports saving and restoring the SSE/SSE2/SSE3/SSSE3 execution environment (XMM and MXCSR registers) with the FXSAVE and FXRSTOR instructions, respectively. See Section 2.5, "Control Registers," for a description of the OSFXSR flag.
3. Set the OSXMMEXCPT flag (bit 10 in control register CR4) to indicate that the operating system supports the handling of SSE/SSE2/SSE3 SIMD floating-point exceptions (#XM). See Section 2.5, "Control Registers," for a description of the OSXMMEXCPT flag.
4. Set the mask bits and flags in the MXCSR register according to the mode of operation desired for SSE/SSE2/SSE3 SIMD floating-point instructions. See "MXCSR Control and Status Register" in Chapter 10, "Programming with Streaming SIMD Extensions (SSE)," of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*, for a detailed description of the bits and flags in the MXCSR register.

9.7 SOFTWARE INITIALIZATION FOR REAL-ADDRESS MODE OPERATION

Following a hardware reset (either through a power-up or the assertion of the RESET# pin) the processor is placed in real-address mode and begins executing software initialization code from physical address FFFFFFF0H. Software initialization code must first set up the necessary data structures for handling basic system functions, such as a real-mode IDT for handling interrupts and exceptions. If the processor is to remain in real-address mode, software must then load additional operating-system or executive code modules and data structures to allow reliable execution of application programs in real-address mode.

If the processor is going to operate in protected mode, software must load the necessary data structures to operate in protected mode and then switch to protected mode. The protected-mode data structures that must be loaded are described in Section 9.8, "Software Initialization for Protected-Mode Operation."

9.7.1 Real-Address Mode IDT

In real-address mode, the only system data structure that must be loaded into memory is the IDT (also called the "interrupt vector table"). By default, the address of the base of the IDT is physical address 0H. This address can be

changed by using the LIDT instruction to change the base address value in the IDTR. Software initialization code needs to load interrupt- and exception-handler pointers into the IDT before interrupts can be enabled.

The actual interrupt- and exception-handler code can be contained either in EPROM or RAM; however, the code must be located within the 1-MByte addressable range of the processor in real-address mode. If the handler code is to be stored in RAM, it must be loaded along with the IDT.

9.7.2 NMI Interrupt Handling

The NMI interrupt is always enabled (except when multiple NMIs are nested). If the IDT and the NMI interrupt handler need to be loaded into RAM, there will be a period of time following hardware reset when an NMI interrupt cannot be handled. During this time, hardware must provide a mechanism to prevent an NMI interrupt from halting code execution until the IDT and the necessary NMI handler software is loaded. Here are two examples of how NMIs can be handled during the initial states of processor initialization:

- A simple IDT and NMI interrupt handler can be provided in EPROM. This allows an NMI interrupt to be handled immediately after reset initialization.
- The system hardware can provide a mechanism to enable and disable NMIs by passing the NMI# signal through an AND gate controlled by a flag in an I/O port. Hardware can clear the flag when the processor is reset, and software can set the flag when it is ready to handle NMI interrupts.

9.8 SOFTWARE INITIALIZATION FOR PROTECTED-MODE OPERATION

The processor is placed in real-address mode following a hardware reset. At this point in the initialization process, some basic data structures and code modules must be loaded into physical memory to support further initialization of the processor, as described in Section 9.7, "Software Initialization for Real-Address Mode Operation." Before the processor can be switched to protected mode, the software initialization code must load a minimum number of protected mode data structures and code modules into memory to support reliable operation of the processor in protected mode. These data structures include the following:

- A IDT.
- A GDT.
- A TSS.
- (Optional) An LDT.
- If paging is to be used, at least one page directory and one page table.
- A code segment that contains the code to be executed when the processor switches to protected mode.
- One or more code modules that contain the necessary interrupt and exception handlers.

Software initialization code must also initialize the following system registers before the processor can be switched to protected mode:

- The GDTR.
- (Optional.) The IDTR. This register can also be initialized immediately after switching to protected mode, prior to enabling interrupts.
- Control registers CR1 through CR4.
- (Pentium 4, Intel Xeon, and P6 family processors only.) The memory type range registers (MTRRs).

With these data structures, code modules, and system registers initialized, the processor can be switched to protected mode by loading control register CR0 with a value that sets the PE flag (bit 0).

9.8.1 Protected-Mode System Data Structures

The contents of the protected-mode system data structures loaded into memory during software initialization, depend largely on the type of memory management the protected-mode operating-system or executive is going to support: flat, flat with paging, segmented, or segmented with paging.

To implement a flat memory model without paging, software initialization code must at a minimum load a GDT with one code and one data-segment descriptor. A null descriptor in the first GDT entry is also required. The stack can be placed in a normal read/write data segment, so no dedicated descriptor for the stack is required. A flat memory model with paging also requires a page directory and at least one page table (unless all pages are 4 MBytes in which case only a page directory is required). See Section 9.8.3, "Initializing Paging."

Before the GDT can be used, the base address and limit for the GDT must be loaded into the GDTR register using an LGDT instruction.

A multi-segmented model may require additional segments for the operating system, as well as segments and LDTs for each application program. LDTs require segment descriptors in the GDT. Some operating systems allocate new segments and LDTs as they are needed. This provides maximum flexibility for handling a dynamic programming environment. However, many operating systems use a single LDT for all tasks, allocating GDT entries in advance. An embedded system, such as a process controller, might pre-allocate a fixed number of segments and LDTs for a fixed number of application programs. This would be a simple and efficient way to structure the software environment of a real-time system.

9.8.2 Initializing Protected-Mode Exceptions and Interrupts

Software initialization code must at a minimum load a protected-mode IDT with gate descriptor for each exception vector that the processor can generate. If interrupt or trap gates are used, the gate descriptors can all point to the same code segment, which contains the necessary exception handlers. If task gates are used, one TSS and accompanying code, data, and task segments are required for each exception handler called with a task gate.

If hardware allows interrupts to be generated, gate descriptors must be provided in the IDT for one or more interrupt handlers.

Before the IDT can be used, the base address and limit for the IDT must be loaded into the IDTR register using an LIDT instruction. This operation is typically carried out immediately after switching to protected mode.

9.8.3 Initializing Paging

Paging is controlled by the PG flag in control register CR0. When this flag is clear (its state following a hardware reset), the paging mechanism is turned off; when it is set, paging is enabled. Before setting the PG flag, the following data structures and registers must be initialized:

- Software must load at least one page directory and one page table into physical memory. The page table can be eliminated if the page directory contains a directory entry pointing to itself (here, the page directory and page table reside in the same page), or if only 4-MByte pages are used.
- Control register CR3 (also called the PDBR register) is loaded with the physical base address of the page directory.
- (Optional) Software may provide one set of code and data descriptors in the GDT or in an LDT for supervisor mode and another set for user mode.

With this paging initialization complete, paging is enabled and the processor is switched to protected mode at the same time by loading control register CR0 with an image in which the PG and PE flags are set. (Paging cannot be enabled before the processor is switched to protected mode.)

9.8.4 Initializing Multitasking

If the multitasking mechanism is not going to be used and changes between privilege levels are not allowed, it is not necessary load a TSS into memory or to initialize the task register.

If the multitasking mechanism is going to be used and/or changes between privilege levels are allowed, software initialization code must load at least one TSS and an accompanying TSS descriptor. (A TSS is required to change privilege levels because pointers to the privileged-level 0, 1, and 2 stack segments and the stack pointers for these stacks are obtained from the TSS.) TSS descriptors must not be marked as busy when they are created; they should be marked busy by the processor only as a side-effect of performing a task switch. As with descriptors for LDTs, TSS descriptors reside in the GDT.

After the processor has switched to protected mode, the LTR instruction can be used to load a segment selector for a TSS descriptor into the task register. This instruction marks the TSS descriptor as busy, but does not perform a task switch. The processor can, however, use the TSS to locate pointers to privilege-level 0, 1, and 2 stacks. The segment selector for the TSS must be loaded before software performs its first task switch in protected mode, because a task switch copies the current task state into the TSS.

After the LTR instruction has been executed, further operations on the task register are performed by task switching. As with other segments and LDTs, TSSs and TSS descriptors can be either pre-allocated or allocated as needed.

9.8.5 Initializing IA-32e Mode

On Intel 64 processors, the IA32_EFER MSR is cleared on system reset. The operating system must be in protected mode with paging enabled before attempting to initialize IA-32e mode. IA-32e mode operation also requires physical-address extensions with four levels of enhanced paging structures (see Section 4.5, “4-Level Paging”).

Operating systems should follow this sequence to initialize IA-32e mode:

1. Starting from protected mode, disable paging by setting CR0.PG = 0. Use the MOV CR0 instruction to disable paging (the instruction must be located in an identity-mapped page).
2. Enable physical-address extensions (PAE) by setting CR4.PAE = 1. Failure to enable PAE will result in a #GP fault when an attempt is made to initialize IA-32e mode.
3. Load CR3 with the physical base address of the Level 4 page map table (PML4).
4. Enable IA-32e mode by setting IA32_EFER.LME = 1.
5. Enable paging by setting CR0.PG = 1. This causes the processor to set the IA32_EFER.LMA bit to 1. The MOV CR0 instruction that enables paging and the following instructions must be located in an identity-mapped page (until such time that a branch to non-identity mapped pages can be effected).

64-bit mode paging tables must be located in the first 4 GBytes of physical-address space prior to activating IA-32e mode. This is necessary because the MOV CR3 instruction used to initialize the page-directory base must be executed in legacy mode prior to activating IA-32e mode (setting CR0.PG = 1 to enable paging). Because MOV CR3 is executed in protected mode, only the lower 32 bits of the register are written, limiting the table location to the low 4 GBytes of memory. Software can relocate the page tables anywhere in physical memory after IA-32e mode is activated.

The processor performs 64-bit mode consistency checks whenever software attempts to modify any of the enable bits directly involved in activating IA-32e mode (IA32_EFER.LME, CR0.PG, and CR4.PAE). It will generate a general protection fault (#GP) if consistency checks fail. 64-bit mode consistency checks ensure that the processor does not enter an undefined mode or state with unpredictable behavior.

64-bit mode consistency checks fail in the following circumstances:

- An attempt is made to enable or disable IA-32e mode while paging is enabled.
- IA-32e mode is enabled and an attempt is made to enable paging prior to enabling physical-address extensions (PAE).
- IA-32e mode is active and an attempt is made to disable physical-address extensions (PAE).
- If the current CS has the L-bit set on an attempt to activate IA-32e mode.
- If the TR contains a 16-bit TSS.

9.8.5.1 IA-32e Mode System Data Structures

After activating IA-32e mode, the system-descriptor-table registers (GDTR, LDTR, IDTR, TR) continue to reference legacy protected-mode descriptor tables. Tables referenced by the descriptors all reside in the lower 4 GBytes of linear-address space. After activating IA-32e mode, 64-bit operating-systems should use the LGDT, LLDT, LIDT, and LTR instructions to load the system-descriptor-table registers with references to 64-bit descriptor tables.

9.8.5.2 IA-32e Mode Interrupts and Exceptions

Software must not allow exceptions or interrupts to occur between the time IA-32e mode is activated and the update of the interrupt-descriptor-table register (IDTR) that establishes references to a 64-bit interrupt-descriptor table (IDT). This is because the IDT remains in legacy form immediately after IA-32e mode is activated.

If an interrupt or exception occurs prior to updating the IDTR, a legacy 32-bit interrupt gate will be referenced and interpreted as a 64-bit interrupt gate with unpredictable results. External interrupts can be disabled by using the CLI instruction.

Non-maskable interrupts (NMI) must be disabled using external hardware.

9.8.5.3 64-bit Mode and Compatibility Mode Operation

IA-32e mode uses two code segment-descriptor bits (CS.L and CS.D, see Figure 3-8) to control the operating modes after IA-32e mode is initialized. If CS.L = 1 and CS.D = 0, the processor is running in 64-bit mode. With this encoding, the default operand size is 32 bits and default address size is 64 bits. Using instruction prefixes, operand size can be changed to 64 bits or 16 bits; address size can be changed to 32 bits.

When IA-32e mode is active and CS.L = 0, the processor operates in compatibility mode. In this mode, CS.D controls default operand and address sizes exactly as it does in the IA-32 architecture. Setting CS.D = 1 specifies default operand and address size as 32 bits. Clearing CS.D to 0 specifies default operand and address size as 16 bits (the CS.L = 1, CS.D = 1 bit combination is reserved).

Compatibility mode execution is selected on a code-segment basis. This mode allows legacy applications to coexist with 64-bit applications running in 64-bit mode. An operating system running in IA-32e mode can execute existing 16-bit and 32-bit applications by clearing their code-segment descriptor's CS.L bit to 0.

In compatibility mode, the following system-level mechanisms continue to operate using the IA-32e-mode architectural semantics:

- Linear-to-physical address translation uses the 64-bit mode extended page-translation mechanism.
- Interrupts and exceptions are handled using the 64-bit mode mechanisms.
- System calls (calls through call gates and SYSENTER/SYSEXIT) are handled using the IA-32e mode mechanisms.

9.8.5.4 Switching Out of IA-32e Mode Operation

To return from IA-32e mode to paged-protected mode operation operating systems must use the following sequence:

1. Switch to compatibility mode.
2. Deactivate IA-32e mode by clearing CR0.PG = 0. This causes the processor to set IA32_EFER.LMA = 0. The MOV CR0 instruction used to disable paging and subsequent instructions must be located in an identity-mapped page.
3. Load CR3 with the physical base address of the legacy page-table-directory base address.
4. Disable IA-32e mode by setting IA32_EFER.LME = 0.
5. Enable legacy paged-protected mode by setting CR0.PG = 1
6. A branch instruction must follow the MOV CR0 that enables paging. Both the MOV CR0 and the branch instruction must be located in an identity-mapped page.

Registers only available in 64-bit mode (R8-R15 and XMM8-XMM15) are preserved across transitions from 64-bit mode into compatibility mode then back into 64-bit mode. However, values of R8-R15 and XMM8-XMM15 are undefined after transitions from 64-bit mode through compatibility mode to legacy or real mode and then back through compatibility mode to 64-bit mode.

9.9 MODE SWITCHING

To use the processor in protected mode after hardware or software reset, a mode switch must be performed from real-address mode. Once in protected mode, software generally does not need to return to real-address mode. To run software written to run in real-address mode (8086 mode), it is generally more convenient to run the software in virtual-8086 mode, than to switch back to real-address mode.

9.9.1 Switching to Protected Mode

Before switching to protected mode from real mode, a minimum set of system data structures and code modules must be loaded into memory, as described in Section 9.8, "Software Initialization for Protected-Mode Operation." Once these tables are created, software initialization code can switch into protected mode.

Protected mode is entered by executing a MOV CR0 instruction that sets the PE flag in the CR0 register. (In the same instruction, the PG flag in register CR0 can be set to enable paging.) Execution in protected mode begins with a CPL of 0.

Intel 64 and IA-32 processors have slightly different requirements for switching to protected mode. To insure upwards and downwards code compatibility with Intel 64 and IA-32 processors, we recommend that you follow these steps:

1. Disable interrupts. A CLI instruction disables maskable hardware interrupts. NMI interrupts can be disabled with external circuitry. (Software must guarantee that no exceptions or interrupts are generated during the mode switching operation.)
2. Execute the LGDT instruction to load the GDTR register with the base address of the GDT.
3. Execute a MOV CR0 instruction that sets the PE flag (and optionally the PG flag) in control register CR0.
4. Immediately following the MOV CR0 instruction, execute a far JMP or far CALL instruction. (This operation is typically a far jump or call to the next instruction in the instruction stream.)
5. The JMP or CALL instruction immediately after the MOV CR0 instruction changes the flow of execution and serializes the processor.
6. If paging is enabled, the code for the MOV CR0 instruction and the JMP or CALL instruction must come from a page that is identity mapped (that is, the linear address before the jump is the same as the physical address after paging and protected mode is enabled). The target instruction for the JMP or CALL instruction does not need to be identity mapped.
7. If a local descriptor table is going to be used, execute the LLDT instruction to load the segment selector for the LDT in the LDTR register.
8. Execute the LTR instruction to load the task register with a segment selector to the initial protected-mode task or to a writable area of memory that can be used to store TSS information on a task switch.
9. After entering protected mode, the segment registers continue to hold the contents they had in real-address mode. The JMP or CALL instruction in step 4 resets the CS register. Perform one of the following operations to update the contents of the remaining segment registers.
   - — Reload segment registers DS, SS, ES, FS, and GS. If the ES, FS, and/or GS registers are not going to be used, load them with a null selector.
   - — Perform a JMP or CALL instruction to a new task, which automatically resets the values of the segment registers and branches to a new code segment.
10. Execute the LIDT instruction to load the IDTR register with the address and limit of the protected-mode IDT.
11. Execute the STI instruction to enable maskable hardware interrupts and perform the necessary hardware operation to enable NMI interrupts.

Random failures can occur if other instructions exist between steps 3 and 4 above. Failures will be readily seen in some situations, such as when instructions that reference memory are inserted between steps 3 and 4 while in system management mode.

9.9.2 Switching Back to Real-Address Mode

The processor switches from protected mode back to real-address mode if software clears the PE bit in the CR0 register with a MOV CR0 instruction. A procedure that re-enters real-address mode should perform the following steps:

1. Disable interrupts. A CLI instruction disables maskable hardware interrupts. NMI interrupts can be disabled with external circuitry.
2. If paging is enabled, perform the following operations:
   - Transfer program control to linear addresses that are identity mapped to physical addresses (that is, linear addresses equal physical addresses).
   - Insure that the GDT and IDT are in identity mapped pages.
   - Clear the PG bit in the CR0 register.
   - Move 0H into the CR3 register to flush the TLB.
3. Transfer program control to a readable segment that has a limit of 64 KBytes (FFFFH). This operation loads the CS register with the segment limit required in real-address mode.
4. Load segment registers SS, DS, ES, FS, and GS with a selector for a descriptor containing the following values, which are appropriate for real-address mode:
   - Limit = 64 KBytes (0FFFFH)
   - Byte granular (G = 0)
   - Expand up (E = 0)
   - Writable (W = 1)
   - Present (P = 1)
   - Base = any value

   The segment registers must be loaded with non-null segment selectors or the segment registers will be unusable in real-address mode. Note that if the segment registers are not reloaded, execution continues using the descriptor attributes loaded during protected mode.
5. Execute an LIDT instruction to point to a real-address mode interrupt table that is within the 1-MByte real-address mode address range.
6. Clear the PE flag in the CR0 register to switch to real-address mode.
7. Execute a far JMP instruction to jump to a real-address mode program. This operation flushes the instruction queue and loads the appropriate base-address value in the CS register.
8. Load the SS, DS, ES, FS, and GS registers as needed by the real-address mode code. If any of the registers are not going to be used in real-address mode, write 0s to them.
9. Execute the STI instruction to enable maskable hardware interrupts and perform the necessary hardware operation to enable NMI interrupts.

NOTE

All the code that is executed in steps 1 through 9 must be in a single page and the linear addresses in that page must be identity mapped to physical addresses.

9.10 INITIALIZATION AND MODE SWITCHING EXAMPLE

This section provides an initialization and mode switching example that can be incorporated into an application. This code was originally written to initialize the Intel386 processor, but it will execute successfully on the Pentium 4, Intel Xeon, P6 family, Pentium, and Intel486 processors. The code in this example is intended to reside in EPROM and to run following a hardware reset of the processor. The function of the code is to do the following:

- Establish a basic real-address mode operating environment.

- Load the necessary protected-mode system data structures into RAM.
- Load the system registers with the necessary pointers to the data structures and the appropriate flag settings for protected-mode operation.
- Switch the processor to protected mode.

Figure 9-3 shows the physical memory layout for the processor following a hardware reset and the starting point of this example. The EPROM that contains the initialization code resides at the upper end of the processor's physical memory address range, starting at address FFFFFFFFH and going down from there. The address of the first instruction to be executed is at FFFFFFF0H, the default starting address for the processor following a hardware reset.

The main steps carried out in this example are summarized in Table 9-5. The source listing for the example (with the filename STARTUP.ASM) is given in Example 9-1. The line numbers given in Table 9-5 refer to the source listing.

The following are some additional notes concerning this example:

- When the processor is switched into protected mode, the original code segment base-address value of FFFF0000H (located in the hidden part of the CS register) is retained and execution continues from the current offset in the EIP register. The processor will thus continue to execute code in the EPROM until a far jump or call is made to a new code segment, at which time, the base address in the CS register will be changed.
- Maskable hardware interrupts are disabled after a hardware reset and should remain disabled until the necessary interrupt handlers have been installed. The NMI interrupt is not disabled following a reset. The NMI# pin must thus be inhibited from being asserted until an NMI handler has been loaded and made available to the processor.
- The use of a temporary GDT allows simple transfer of tables from the EPROM to anywhere in the RAM area. A GDT entry is constructed with its base pointing to address 0 and a limit of 4 GBytes. When the DS and ES registers are loaded with this descriptor, the temporary GDT is no longer needed and can be replaced by the application GDT.
- This code loads one TSS and no LDTs. If more TSSs exist in the application, they must be loaded into RAM. If there are LDTs they may be loaded as well.

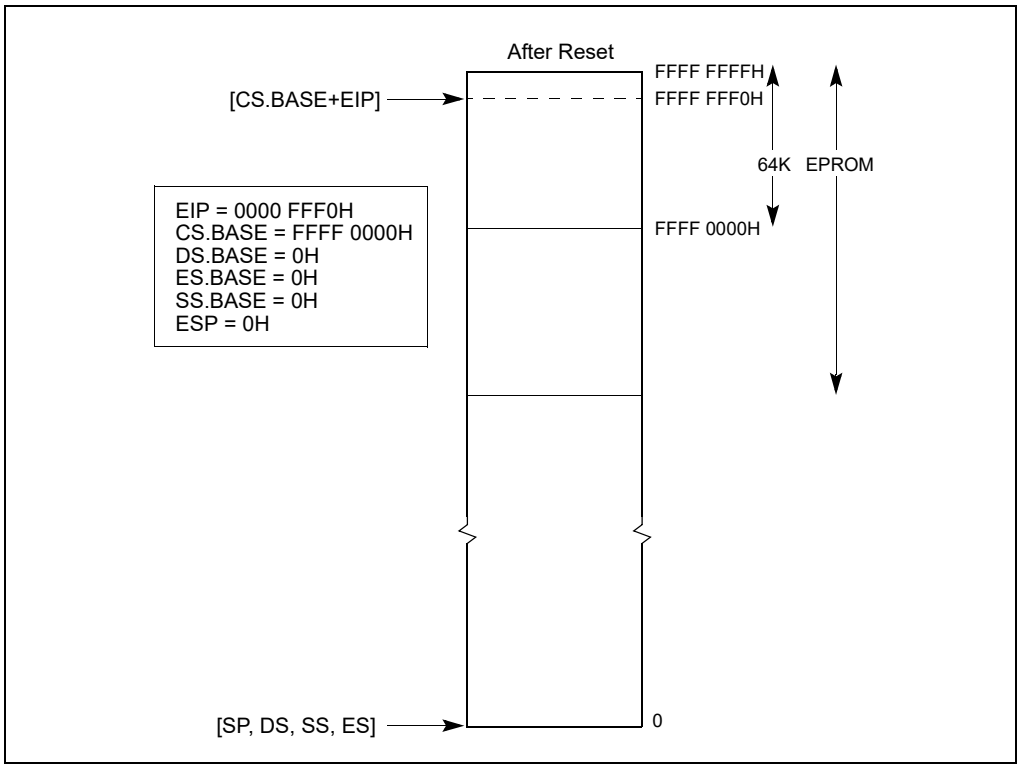


Figure 9-3. Processor State After Reset

Table 9-5. Main Initialization Steps in STARTUP.ASM Source Listing

STARTUP.ASM Line Numbers		Description
From	To	
157	157	Jump (short) to the entry code in the EPROM
162	169	Construct a temporary GDT in RAM with one entry: 0 - null 1 - R/W data segment, base = 0, limit = 4 GBytes
171	172	Load the GDTR to point to the temporary GDT
174	177	Load CR0 with PE flag set to switch to protected mode
179	181	Jump near to clear real mode instruction queue
184	186	Load DS, ES registers with GDT[1] descriptor, so both point to the entire physical memory space
188	195	Perform specific board initialization that is imposed by the new protected mode
196	218	Copy the application's GDT from ROM into RAM
220	238	Copy the application's IDT from ROM into RAM
241	243	Load application's GDTR
244	245	Load application's IDTR
247	261	Copy the application's TSS from ROM into RAM
263	267	Update TSS descriptor and other aliases in GDT (GDT alias or IDT alias)
277	277	Load the task register (without task switch) using LTR instruction
282	286	Load SS, ESP with the value found in the application's TSS
287	287	Push EFLAGS value found in the application's TSS
288	288	Push CS value found in the application's TSS
289	289	Push EIP value found in the application's TSS
290	293	Load DS, ES with the value found in the application's TSS
296	296	Perform IRET; pop the above values and enter the application code

9.10.1 Assembler Usage

In this example, the Intel assembler ASM386 and build tools BLD386 are used to assemble and build the initialization code module. The following assumptions are used when using the Intel ASM386 and BLD386 tools.

- The ASM386 will generate the right operand size opcodes according to the code-segment attribute. The attribute is assigned either by the ASM386 invocation controls or in the code-segment definition.
- If a code segment that is going to run in real-address mode is defined, it must be set to a USE 16 attribute. If a 32-bit operand is used in an instruction in this code segment (for example, MOV EAX, EBX), the assembler automatically generates an operand prefix for the instruction that forces the processor to execute a 32-bit operation, even though its default code-segment attribute is 16-bit.
- Intel's ASM386 assembler allows specific use of the 16- or 32-bit instructions, for example, LGDTW, LGDTD, IRETD. If the generic instruction LGDT is used, the default- segment attribute will be used to generate the right opcode.

9.10.2 STARTUP.ASM Listing

Example 9-1 provides high-level sample code designed to move the processor into protected mode. This listing does not include any opcode and offset information.

Example 9-1. STARTUP.ASM

```
MS-DOS* 5.0(045-N) 386(TM) MACRO ASSEMBLER STARTUP  09:44:51 08/19/92 PAGE 1

MS-DOS 5.0(045-N) 386(TM) MACRO ASSEMBLER V4.0, ASSEMBLY OF MODULE STARTUP
OBJECT MODULE PLACED IN startup.obj
ASSEMBLER INVOKED BY: f:\386tools\ASM386.EXE startup.a58 pw (132 )

LINE     SOURCE

   1      NAME    STARTUP
   2
   3  ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
   4  ;
   5  ;   ASSUMPTIONS:
   6  ;
   7  ;      1.  The bottom 64K of memory is ram, and can be used for
   8  ;          scratch space by this module.
   9  ;
  10  ;      2.  The system has sufficient free usable ram to copy the
  11  ;          initial GDT, IDT, and TSS
  12  ;
  13  ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
  14
  15  ; configuration data - must match with build definition
  16
  17  CS_BASE       EQU     0FFFF0000H
  18
  19   ; CS_BASE is the linear address of the segment STARTUP_CODE
  20   ; - this is specified in the build language file
  21
  22  RAM_START     EQU     400H
  23
  24  ; RAM_START  is the start of free, usable ram in the linear
  25  ; memory  space.   The GDT,  IDT, and  initial TSS  will be
  26  ; copied above this space, and a small data segment will be
  27  ; discarded at  this linear  address.   The 32-bit  word at
  28  ; RAM_START will contain  the linear  address of  the first
  29  ; free byte above the copied tables - this may be useful if
  30  ; a memory manager is used.
  31
  32  TSS_INDEX    EQU     10
  33
  34  ; TSS_INDEX is the  index of the  TSS of the  first task to
  35  ; run after startup
  36
  37
  38   ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
  39
  40  ; ------------------------- STRUCTURES and EQU ---------------
  41  ; structures for system data
  42
  43  ; TSS structure
  44  TASK_STATE  STRUC
  45      link     DW ?
```

```
46      link_h    DW ?
47      ESP0      DD ?
48      SS0       DW ?
49      SS0_h     DW ?
50      ESP1      DD ?
51      SS1       DW ?
52      SS1_h     DW ?
53      ESP2      DD ?
54      SS2       DW ?
55      SS2_h     DW ?
56      CR3_reg   DD ?
57      EIP_reg   DD ?
58      EFLAGS_regDD ?
59      EAX_reg   DD ?
60      ECX_reg   DD ?
61      EDX_reg   DD ?
62      EBX_reg   DD ?
63      ESP_reg   DD ?
64      EBP_reg   DD ?
65      ESI_reg   DD ?
66      EDI_reg   DD ?
67      ES_reg    DW ?
68      ES_h      DW ?
69      CS_reg    DW ?
70      CS_h      DW ?
71      SS_reg    DW ?
72      SS_h      DW ?
73      DS_reg    DW ?
74      DS_h      DW ?
75      FS_reg    DW ?
76      FS_h      DW ?
77      GS_reg    DW ?
78      GS_h      DW ?
79      LDT_reg   DW ?
80      LDT_h     DW ?
81      TRAP_reg  DW ?
82      IO_map_baseDW ?
83  TASK_STATE   ENDS
84
85  ; basic structure of a descriptor
86  DESC    STRUC
87      lim_0_15  DW ?
88      bas_0_15  DW ?
89      bas_16_23 DB ?
90      access    DB ?
91      gran      DB ?
92      bas_24_31 DB ?
93  DESC    ENDS
94
95  ; structure for use with LGDT and LIDT instructions
96  TABLE_REG   STRUC
97      table_limDW ?
98      table_linearDD ?
99  TABLE_REG   ENDS
```

```
100
101  ; offset of GDT and IDT descriptors in builder generated GDT
102  GDT_DESC_OFF    EQU 1*SIZE(DESC)
103  IDT_DESC_OFF    EQU 2*SIZE(DESC)
104
105  ; equates for building temporary GDT in RAM
106  LINEAR_SEL          EQU     1*SIZE (DESC)
107  LINEAR_PROTO_LO     EQU     00000FFFFH  ; LINEAR_ALIAS
108  LINEAR_PROTO_HI     EQU     000CF9200H
109
110  ; Protection Enable Bit in CR0
111  PE_BIT  EQU 1B
112
113  ; -------------------------------------------------------------
114
115  ; ------------------------- DATA SEGMENT----------------------
116
117  ; Initially, this  data segment starts at linear 0, according
118  ; to the processor's power-up state.
119
120  STARTUP_DATA    SEGMENT RW
121
122  free_mem_linear_base    LABEL   DWORD
123  TEMP_GDT                LABEL   BYTE  ; must be first in segment
124  TEMP_GDT_NULL_DESC   DESC    <>
125  TEMP_GDT_LINEAR_DESC DESC    <>
126
127  ; scratch areas for LGDT and LIDT instructions
128  TEMP_GDT_SCRATCH TABLE_REG   <>
129  APP_GDT_RAM     TABLE_REG    <>
130  APP_IDT_RAM     TABLE_REG    <>
131          ; align end_data
132  fill    DW      ?
133
134  ; last thing in this segment - should be on a dword boundary
135  end_data    LABEL   BYTE
136
137  STARTUP_DATA    ENDS
138  ; -------------------------------------------------------------
139
140
141  ; ------------------------- CODE SEGMENT----------------------
142  STARTUP_CODE SEGMENT ER PUBLIC USE16
143
144  ; filled in by builder
145      PUBLIC  GDT_EPROM
146  GDT_EPROM   TABLE_REG   <>
147
148  ; filled in by builder
149      PUBLIC  IDT_EPROM
150  IDT_EPROM   TABLE_REG   <>
151
152  ; entry point into startup code - the bootstrap will vector
153  ; here  with a  near JMP  generated by  the builder.   This
```

```
154  ; label must be in the top 64K of linear memory.
155
156      PUBLIC  STARTUP
157  STARTUP:
158
159  ; DS,ES address the bottom 64K of flat linear memory
160      ASSUME  DS:STARTUP_DATA, ES:STARTUP_DATA
161  ; See Figure 9-4
162  ; load GDTR with temporary GDT
163          LEA     EBX,TEMP_GDT  ; build the TEMP_GDT in low ram,
164          MOV     DWORD PTR [EBX],0   ; where we can address
165          MOV     DWORD PTR [EBX]+4,0
166          MOV     DWORD PTR [EBX]+8, LINEAR_PROTO_LO
167          MOV     DWORD PTR [EBX]+12, LINEAR_PROTO_HI
168          MOV     TEMP_GDT_scratch.table_linear,EBX
169          MOV     TEMP_GDT_scratch.table_lim,15
170
171          DB 66H; execute a 32 bit LGDT
172          LGDT    TEMP_GDT_scratch
173
174  ; enter protected mode
175          MOV     EBX,CR0
176          OR      EBX,PE_BIT
177          MOV     CR0,EBX
178

179   ; clear prefetch queue
180          JMP     CLEAR_LABEL
181  CLEAR_LABEL:
182
183   ; make DS and ES address 4G of linear memory
184          MOV     CX,LINEAR_SEL
185          MOV     DS,CX
186          MOV     ES,CX
187
188    ; do board specific initialization
189    ;
190                  ;
191                  ; ......
192                  ;
193
194
195          ; See Figure 9-5
196          ; copy EPROM GDT to ram at:
197          ;                RAM_START + size (STARTUP_DATA)
198          MOV     EAX,RAM_START
199          ADD     EAX,OFFSET (end_data)
200          MOV     EBX,RAM_START
201          MOV     ECX, CS_BASE
202          ADD     ECX, OFFSET (GDT_EPROM)
203          MOV     ESI, [ECX].table_linear
204          MOV     EDI,EAX
205          MOVZX   ECX, [ECX].table_lim
206          MOV     APP_GDT_ram[EBX].table_lim,CX
```

```
207          INC     ECX
208          MOV     EDX,EAX
209          MOV     APP_GDT_ram[EBX].table_linear,EAX
210          ADD     EAX,ECX
211      REP MOVS    BYTE PTR ES:[EDI],BYTE PTR DS:[ESI]
212
213          ; fixup GDT base in descriptor
214          MOV     ECX,EDX
215          MOV     [EDX].bas_0_15+GDT_DESC_OFF,CX
216          ROR     ECX,16
217          MOV     [EDX].bas_16_23+GDT_DESC_OFF,CL
218          MOV     [EDX].bas_24_31+GDT_DESC_OFF,CH
219
220          ; copy EPROM IDT to ram at:
221          ; RAM_START+size(STARTUP_DATA)+SIZE (EPROM GDT)
222          MOV     ECX, CS_BASE
223          ADD     ECX, OFFSET (IDT_EPROM)
224          MOV     ESI, [ECX].table_linear
225          MOV     EDI,EAX
226          MOVZX   ECX, [ECX].table_lim
227          MOV     APP_IDT_ram[EBX].table_lim,CX
228          INC     ECX
229          MOV     APP_IDT_ram[EBX].table_linear,EAX
230          MOV     EBX,EAX
231          ADD     EAX,ECX
232      REP MOVS    BYTE PTR ES:[EDI],BYTE PTR DS:[ESI]
233
234                  ; fixup IDT pointer in GDT
235          MOV     [EDX].bas_0_15+IDT_DESC_OFF,BX
236          ROR     EBX,16
237          MOV     [EDX].bas_16_23+IDT_DESC_OFF,BL
238          MOV     [EDX].bas_24_31+IDT_DESC_OFF,BH
239
240                  ; load GDTR and IDTR
241          MOV     EBX,RAM_START
242                  DB      66H         ; execute a 32 bit LGDT
243          LGDT    APP_GDT_ram[EBX]
244                  DB      66H         ; execute a 32 bit LIDT
245          LIDT    APP_IDT_ram[EBX]
246
247                  ; move the TSS
248          MOV     EDI,EAX
249          MOV     EBX,TSS_INDEX*SIZE(DESC)
250          MOV     ECX,GDT_DESC_OFF ;build linear address for TSS
251          MOV     GS,CX
252          MOV     DH,GS:[EBX].bas_24_31
253          MOV     DL,GS:[EBX].bas_16_23
254          ROL     EDX,16
255          MOV     DX,GS:[EBX].bas_0_15
256          MOV     ESI,EDX
257          LSL     ECX,EBX
258          INC     ECX
259          MOV     EDX,EAX
260          ADD     EAX,ECX
```

```
 261      REP MOVS    BYTE PTR ES:[EDI],BYTE PTR DS:[ESI]
 262
 263                  ; fixup TSS pointer
 264          MOV     GS:[EBX].bas_0_15,DX
 265          ROL     EDX,16
 266          MOV     GS:[EBX].bas_24_31,DH
 267          MOV     GS:[EBX].bas_16_23,DL
 268          ROL     EDX,16
 269      ;save start of free ram at linear location RAMSTART
 270          MOV     free_mem_linear_base+RAM_START,EAX
 271
 272      ;assume no  LDT used in  the initial task  - if necessary,
 273      ;code  to move the LDT could be added, and should resemble
 274      ;that used to move the TSS
 275
 276      ; load task register
 277          LTR     BX   ; No task switch, only descriptor loading
 278      ; See Figure 9-6
 279      ; load minimal set of registers necessary to simulate task
 280      ; switch
 281
 282
 283          MOV     AX,[EDX].SS_reg     ; start loading registers
 284          MOV     EDI,[EDX].ESP_reg
 285          MOV     SS,AX
 286          MOV     ESP,EDI             ; stack now valid
 287          PUSH    DWORD PTR [EDX].EFLAGS_reg
 288          PUSH    DWORD PTR [EDX].CS_reg
 289          PUSH    DWORD PTR [EDX].EIP_reg
 290          MOV     AX,[EDX].DS_reg
 291          MOV     BX,[EDX].ES_reg
 292          MOV     DS,AX     ; DS and ES no longer linear memory
 293          MOV     ES,BX
 294
 295          ; simulate far jump to initial task
 296          IRETD
 297
 298  STARTUP_CODE  ENDS
*** WARNING #377 IN 298, (PASS 2) SEGMENT CONTAINS PRIVILEGED INSTRUCTION(S)
 299
 300  END STARTUP, DS:STARTUP_DATA, SS:STARTUP_DATA
 301
 302

ASSEMBLY COMPLETE,   1 WARNING,   NO ERRORS.
```

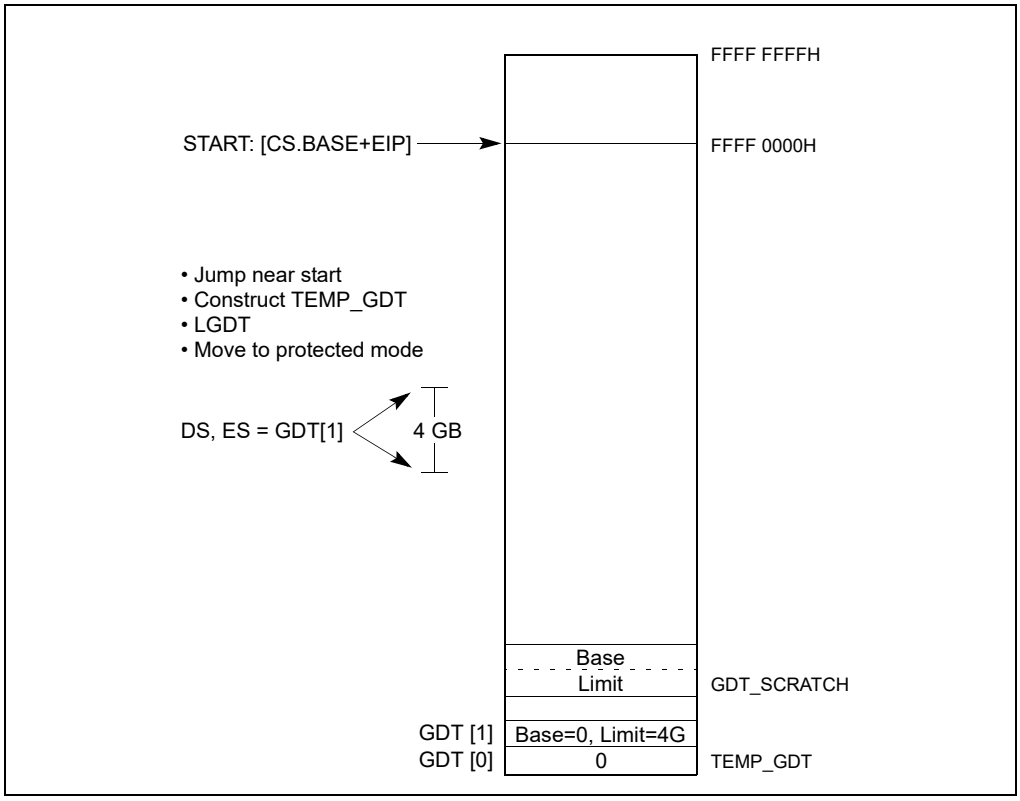


Figure 9-4. Constructing Temporary GDT and Switching to Protected Mode (Lines 162-172 of List File)

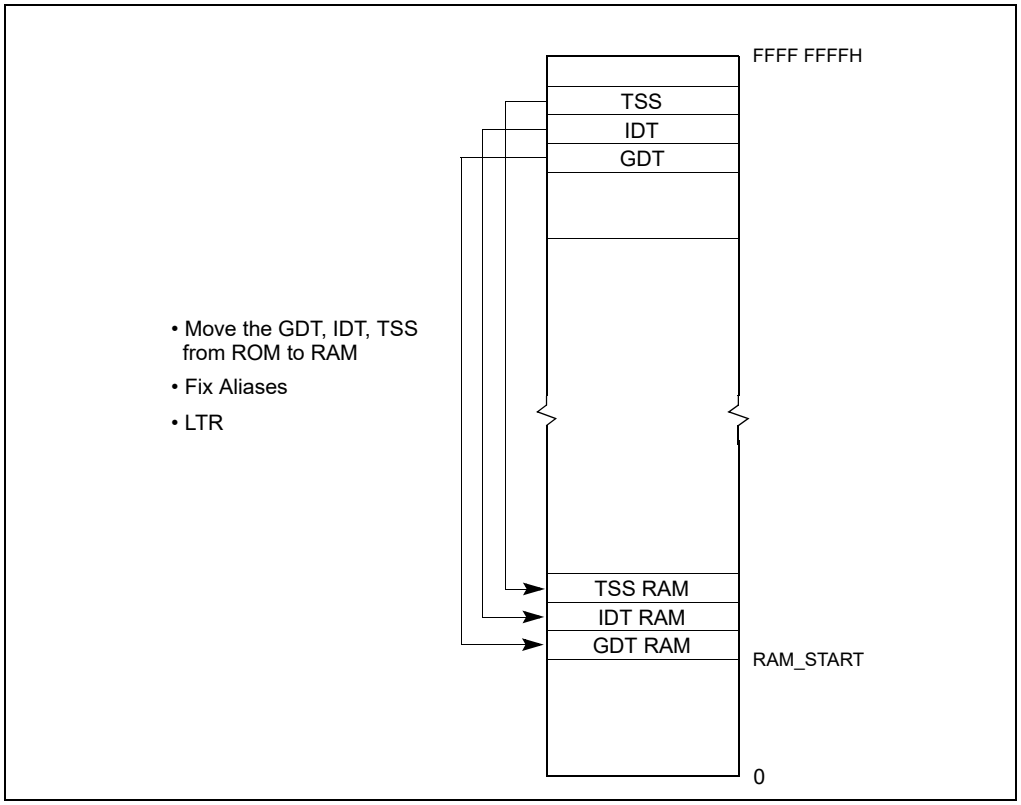


Figure 9-5. Moving the GDT, IDT, and TSS from ROM to RAM (Lines 196-261 of List File)

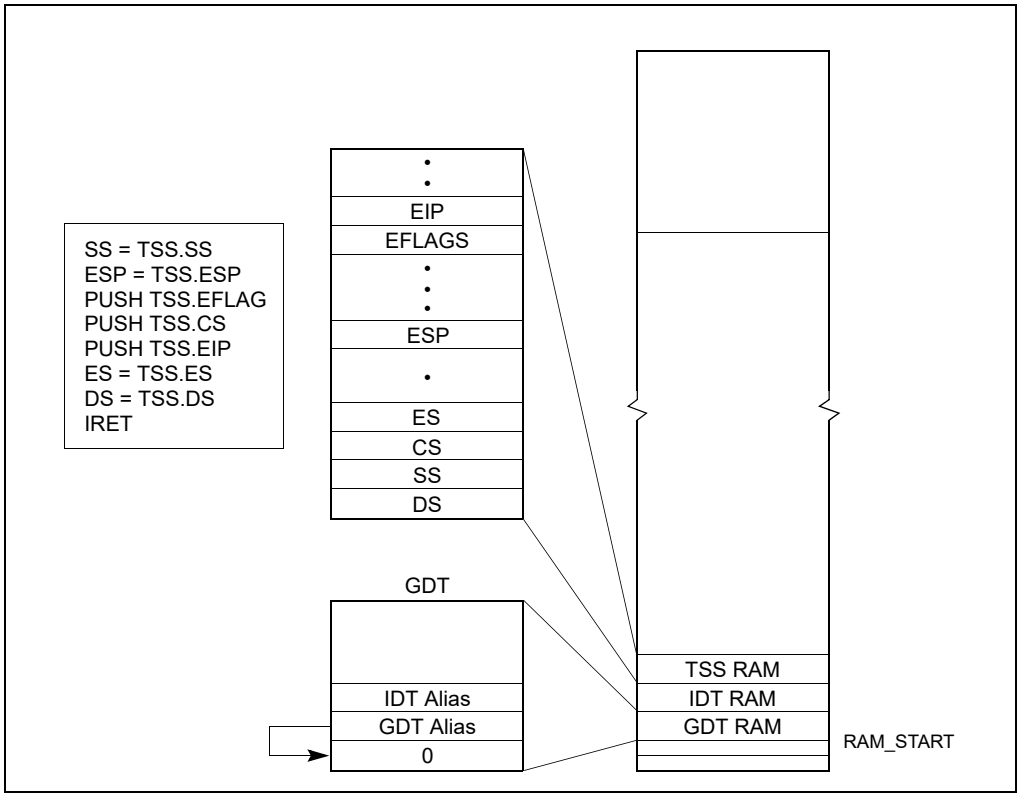


Figure 9-6. Task Switching (Lines 282-296 of List File)

9.10.3 MAIN.ASM Source Code

The file MAIN.ASM shown in Example 9-2 defines the data and stack segments for this application and can be substituted with the main module task written in a high-level language that is invoked by the IRET instruction executed by STARTUP.ASM.

Example 9-2. MAIN.ASM

```
NAME    main_module
data    SEGMENT RW
   dw 1000 dup(?)
DATA    ENDS
stack stackseg 800
CODE SEGMENT ER  use32 PUBLIC
main_start:
   nop
   nop
   nop
CODE  ENDS
END main_start, ds:data, ss:stack
```

9.10.4 Supporting Files

The batch file shown in Example 9-3 can be used to assemble the source code files STARTUP.ASM and MAIN.ASM and build the final application.

Example 9-3. Batch File to Assemble and Build the Application

```
ASM386 STARTUP.ASM
ASM386 MAIN.ASM
BLD386 STARTUP.OBJ, MAIN.OBJ buildfile(EPROM.BLD) bootstrap(STARTUP) Bootload

BLD386 performs several operations in this example:
It allocates physical memory location to segments and tables.
It generates tables using the build file and the input files.
It links object files and resolves references.
It generates a boot-loadable file to be programmed into the EPROM.
```

Example 9-4 shows the build file used as an input to BLD386 to perform the above functions.

Example 9-4. Build File

```
INIT_BLD_EXAMPLE;

SEGMENT
        *SEGMENTS(DPL = 0)
    ,   startup.startup_code(BASE = 0FFFF0000H)
    ;

TASK
        BOOT_TASK(OBJECT = startup, INITIAL,DPL = 0,
                    NOT INTENABLED)
,       PROTECTED_MODE_TASK(OBJECT = main_module,DPL = 0,
                    NOT INTENABLED)
    ;

TABLE
    GDT (
        LOCATION = GDT_EPROM
    ,   ENTRY = (
            10:   PROTECTED_MODE_TASK
    ,        startup.startup_code
    ,       startup.startup_data
    ,       main_module.data
    ,       main_module.code
    ,       main_module.stack
             )
        ),

    IDT (
        LOCATION = IDT_EPROM
        );

MEMORY
    (
        RESERVE = (0..3FFFH
                -- Area for the GDT, IDT, TSS copied from ROM
    ,              60000H..0FFFEFFFFH)
    ,   RANGE = (ROM_AREA = ROM (0FFFF0000H..0FFFFFFFFH))
                   -- Eprom size 64K
    ,   RANGE = (RAM_AREA = RAM (4000H..05FFFFH))
```

```
    );

END
```

Table 9-6 shows the relationship of each build item with an ASM source file.

Table 9-6. Relationship Between BLD Item and ASM Source File

Item	ASM386 and Startup.A58	BLD386 Controls and BLD file	Effect
Bootstrap	public startup startup:	bootstrap start(startup)	Near jump at 0FFFFFFF0H to start.
GDT location	public GDT_EPROM GDT_EPROM TABLE_REG <>	TABLE GDT(location = GDT_EPROM)	The location of the GDT will be programmed into the GDT_EPROM location.
IDT location	public IDT_EPROM IDT_EPROM TABLE_REG <>	TABLE IDT(location = IDT_EPROM	The location of the IDT will be programmed into the IDT_EPROM location.
RAM start	RAM_START equ 400H	memory (reserve = (0..3FFFH))	RAM_START is used as the ram destination for moving the tables. It must be excluded from the application's segment area.
Location of the application TSS in the GDT	TSS_INDEX EQU 10	TABLE GDT(ENTRY = (10: PROTECTED_MODE_ TASK))	Put the descriptor of the application TSS in GDT entry 10.
EPROM size and location	size and location of the initialization code	SEGMENT startup.code (base = 0FFFF0000H) ...memory (RANGE(ROM_AREA = ROM(x..y))	Initialization code size must be less than 64K and resides at upper most 64K of the 4-GByte memory space.

9.11 MICROCODE UPDATE FACILITIES

The P6 family and later processors have the capability to correct errata by loading an Intel-supplied data block into the processor. The data block is called a microcode update. This section describes the mechanisms the BIOS needs to provide in order to use this feature during system initialization. It also describes a specification that permits the incorporation of future updates into a system BIOS.

Intel considers the release of a microcode update for a silicon revision to be the equivalent of a processor stepping and completes a full-stepping level validation for releases of microcode updates.

A microcode update is used to correct errata in the processor. The BIOS, which has an update loader, is responsible for loading the update on processors during system initialization (Figure 9-7). There are two steps to this process: the first is to incorporate the necessary update data blocks into the BIOS; the second is to load update data blocks into the processor.

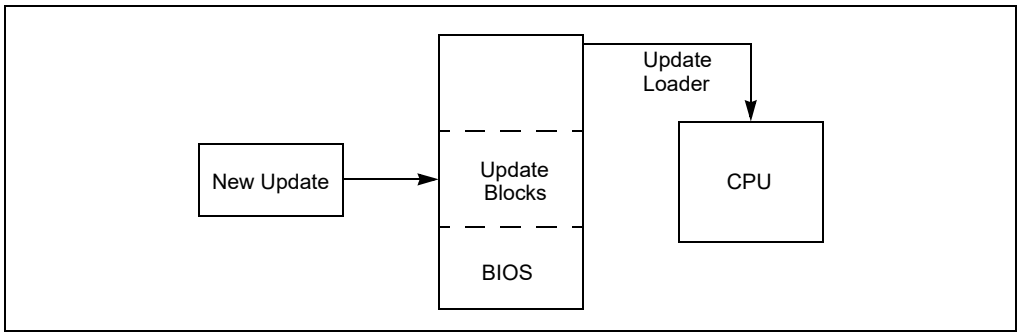


Figure 9-7. Applying Microcode Updates

9.11.1 Microcode Update

A microcode update consists of an Intel-supplied binary that contains a descriptive header and data. No executable code resides within the update. Each microcode update is tailored for a specific list of processor signatures. A mismatch of the processor's signature with the signature contained in the update will result in a failure to load. A processor signature includes the extended family, extended model, type, family, model, and stepping of the processor (starting with processor family 0fH, model 03H, a given microcode update may be associated with one of multiple processor signatures; see Section 9.11.2 for details).

Microcode updates are composed of a multi-byte header, followed by encrypted data and then by an optional extended signature table. Table 9-7 provides a definition of the fields; Table 9-8 shows the format of an update.

The header is 48 bytes. The first 4 bytes of the header contain the header version. The update header and its reserved fields are interpreted by software based upon the header version. An encoding scheme guards against tampering and provides a means for determining the authenticity of any given update. For microcode updates with a data size field equal to 00000000H, the size of the microcode update is 2048 bytes. The first 48 bytes contain the microcode update header. The remaining 2000 bytes contain encrypted data.

For microcode updates with a data size not equal to 00000000H, the total size field specifies the size of the micro-code update. The first 48 bytes contain the microcode update header. The second part of the microcode update is the encrypted data. The data size field of the microcode update header specifies the encrypted data size, its value must be a multiple of the size of DWORD. The total size field of the microcode update header specifies the encrypted data size plus the header size; its value must be in multiples of 1024 bytes (1 KBytes). The optional extended signature table if implemented follows the encrypted data, and its size is calculated by (Total Size – (Data Size + 48)).

NOTE

The optional extended signature table is supported starting with processor family 0FH, model 03H.

.

Table 9-7. Microcode Update Field Definitions

Field Name	Offset (bytes)	Length (bytes)	Description
Header Version	0	4	Version number of the update header.
Update Revision	4	4	Unique version number for the update, the basis for the update signature provided by the processor to indicate the current update functioning within the processor. Used by the BIOS to authenticate the update and verify that the processor loads successfully. The value in this field cannot be used for processor stepping identification alone. This is a signed 32-bit number.
Date	8	4	Date of the update creation in binary format: mmddyyyy (e.g. 07/18/98 is 07181998H).

Table 9-7. Microcode Update Field Definitions (Contd.)

Field Name	Offset (bytes)	Length (bytes)	Description
Processor Signature	12	4	*Extended family, extended model, type, family, model*, and *stepping* of processor that requires this particular update revision (e.g., 00000650H). Each microcode update is designed specifically for a given extended family, extended model, *type, family, model*, and *stepping* of the processor. Software should use the processor signature field in conjunction with the CPUID instruction to determine whether or not an update is appropriate to load on a processor. The information encoded within this field exactly corresponds to the bit representations returned by the CPUID instruction.
Checksum	16	4	Checksum of Update Data and Header. Used to verify the integrity of the update header and data. Checksum is correct when the summation of all the DWORDs (including the extended Processor Signature Table) that comprise the microcode update result in 00000000H.
Loader Revision	20	4	Version number of the loader program needed to correctly load this update. The initial version is 00000001H.
Processor Flags	24	4	Platform type information is encoded in the lower 8 bits of this 4-byte field. Each bit represents a particular platform type for a given CPUID. Software should use the processor flags field in conjunction with the platform Id bits in MSR (17H) to determine whether or not an update is appropriate to load on a processor. Multiple bits may be set representing support for multiple platform IDs.
Data Size	28	4	Specifies the size of the encrypted data in bytes, and must be a multiple of DWORDs. If this value is 00000000H, then the microcode update encrypted data is 2000 bytes (or 500 DWORDs).
Total Size	32	4	Specifies the total size of the microcode update in bytes. It is the summation of the header size, the encrypted data size and the size of the optional extended signature table. This value is always a multiple of 1024.
Reserved	36	12	Reserved fields for future expansion.
Update Data	48	Data Size or 2000	Update data.
Extended Signature Count	Data Size + 48	4	Specifies the number of extended signature structures (Processor Signature[n], processor flags[n] and checksum[n]) that exist in this microcode update.
Extended Checksum	Data Size + 52	4	Checksum of update extended processor signature table. Used to verify the integrity of the extended processor signature table. Checksum is correct when the summation of the DWORDs that comprise the extended processor signature table results in 00000000H.
Reserved	Data Size + 56	12	Reserved fields.

Table 9-7. Microcode Update Field Definitions (Contd.)

Field Name	Offset (bytes)	Length (bytes)	Description
Processor Signature[n]	Data Size + 68 + (n * 12)	4	*Extended family, extended model, type, family, model*, and *stepping* of processor that requires this particular update revision (e.g., 00000650H). Each microcode update is designed specifically for a given extended family, extended model, *type, family, model*, and *stepping* of the processor. Software should use the processor signature field in conjunction with the CPUID instruction to determine whether or not an update is appropriate to load on a processor. The information encoded within this field exactly corresponds to the bit representations returned by the CPUID instruction.
Processor Flags[n]	Data Size + 72 + (n * 12)	4	Platform type information is encoded in the lower 8 bits of this 4-byte field. Each bit represents a particular platform type for a given CPUID. Software should use the processor flags field in conjunction with the platform Id bits in MSR (17H) to determine whether or not an update is appropriate to load on a processor. Multiple bits may be set representing support for multiple platform IDs.
Checksum[n]	Data Size + 76 + (n * 12)	4	Used by utility software to decompose a microcode update into multiple microcode updates where each of the new updates is constructed without the optional Extended Processor Signature Table. To calculate the Checksum, substitute the Primary Processor Signature entry and the Processor Flags entry with the corresponding Extended Patch entry. Delete the Extended Processor Signature Table entries. The Checksum is correct when the summation of all DWORDs that comprise the created Extended Processor Patch results in 00000000H.

Table 9-8. Microcode Update Format

31 24	16	8	0	Bytes
Header Version				0
Update Revision				4
Month: 8	Day: 8	Year: 16		8
Processor Signature (CPUID)				12
Res: 4 / Extended Family: 8 / Extended Mode: 4 / Reserved: 2 / Type: 2 / Family: 4 / Model: 4 / Stepping: 4				
Checksum				16
Loader Revision				20
Processor Flags				24
Reserved (24 bits)			P7 P6 P5 P4 P3 P2 P1 P0	
Data Size				28
Total Size				32
Reserved (12 Bytes)				36

Table 9-8. Microcode Update Format (Contd.)

31 24 16 8 0	Bytes
Update Data (Data Size bytes, or 2000 Bytes if Data Size = 00000000H)	48
Extended Signature Count 'n'	Data Size + 48
Extended Processor Signature Table Checksum	Data Size + 52
Reserved (12 Bytes)	Data Size + 56
Processor Signature[n]	Data Size + 68 + (n * 12)
Processor Flags[n]	Data Size + 72 + (n * 12)
Checksum[n]	Data Size + 76 + (n * 12)

9.11.2 Optional Extended Signature Table

The extended signature table is a structure that may be appended to the end of the encrypted data when the encrypted data only supports a single processor signature (optional case). The extended signature table will always be present when the encrypted data supports multiple processor steppings and/or models (required case).

The extended signature table consists of a 20-byte extended signature header structure, which contains the extended signature count, the extended processor signature table checksum, and 12 reserved bytes (Table 9-9). Following the extended signature header structure, the extended signature table contains 0-to-n extended processor signature structures.

Each processor signature structure consist of the processor signature, processor flags, and a checksum (Table 9-10).

The extended signature count in the extended signature header structure indicates the number of processor signature structures that exist in the extended signature table.

The extended processor signature table checksum is a checksum of all DWORDs that comprise the extended signature table. That includes the extended signature count, extended processor signature table checksum, 12 reserved bytes and the n processor signature structures. A valid extended signature table exists when the result of a DWORD checksum is 00000000H.

Table 9-9. Extended Processor Signature Table Header Structure

Extended Signature Count 'n'	Data Size + 48
Extended Processor Signature Table Checksum	Data Size + 52
Reserved (12 Bytes)	Data Size + 56

Table 9-10. Processor Signature Structure

Processor Signature[n]	Data Size + 68 + (n * 12)
Processor Flags[n]	Data Size + 72 + (n * 12)
Checksum[n]	Data Size + 76 + (n * 12)

9.11.3 Processor Identification

Each microcode update is designed to for a specific processor or set of processors. To determine the correct microcode update to load, software must ensure that one of the processor signatures embedded in the microcode update matches the 32-bit processor signature returned by the CPUID instruction when executed by the target processor with EAX = 1. Attempting to load a microcode update that does not match a processor signature embedded in the microcode update with the processor signature returned by CPUID will cause the BIOS to reject the update.

Example 9-5 shows how to check for a valid processor signature match between the processor and microcode update.

Example 9-5. Pseudo Code to Validate the Processor Signature

```
ProcessorSignature ← CPUID(1):EAX

If (Update.HeaderVersion = 00000001h)
{
       // first check the ProcessorSignature field
   If (ProcessorSignature = Update.ProcessorSignature)
       Success

       // if extended signature is present
   Else If (Update.TotalSize > (Update.DataSize + 48))
   {

       //
       // Assume the Data Size has been used to calculate the
       // location of Update.ProcessorSignature[0].
       //

       For (N ← 0; ((N < Update.ExtendedSignatureCount) AND
           (ProcessorSignature ≠ Update.ProcessorSignature[N])); N++);

           // if the loops ended when the iteration count is
           // less than the number of processor signatures in
           // the table, we have a match
       If (N < Update.ExtendedSignatureCount)
           Success
       Else
           Fail
   }
   Else
       Fail
Else
   Fail
```

9.11.4 Platform Identification

In addition to verifying the processor signature, the intended processor platform type must be determined to properly target the microcode update. The intended processor platform type is determined by reading the IA32_PLATFORM_ID register, (MSR 17H). This 64-bit register must be read using the RDMSR instruction.

The three platform ID bits, when read as a binary coded decimal (BCD) number, indicate the bit position in the microcode update header's processor flags field associated with the installed processor. The processor flags in the 48-byte header and the processor flags field associated with the extended processor signature structures may have multiple bits set. Each set bit represents a different platform ID that the update supports.

Register Name: IA32_PLATFORM_ID
MSR Address: 017H

Access: Read Only

IA32_PLATFORM_ID is a 64-bit register accessed only when referenced as a Qword through a RDMSR instruction.

Table 9-11. Processor Flags

Bit	Descriptions
63:53	Reserved
52:50	Platform Id Bits (RO). The field gives information concerning the intended platform for the processor. See also Table 9-8. 52 51 50 0 0 0 Processor Flag 0 0 0 1 Processor Flag 1 0 1 0 Processor Flag 2 0 1 1 Processor Flag 3 1 0 0 Processor Flag 4 1 0 1 Processor Flag 5 1 1 0 Processor Flag 6 1 1 1 Processor Flag 7
49:0	Reserved

To validate the platform information, software may implement an algorithm similar to the algorithms in Example 9-6.

Example 9-6. Pseudo Code Example of Processor Flags Test

```
Flag ← 1 << IA32_PLATFORM_ID[52:50]

If (Update.HeaderVersion = 00000001h)
{
    If (Update.ProcessorFlags & Flag)
    {
        Load Update
    }
    Else
    {

        //
        // Assume the Data Size has been used to calculate the
        // location of Update.ProcessorSignature[N] and a match
        // on Update.ProcessorSignature[N] has already succeeded
        //

        If (Update.ProcessorFlags[n] & Flag)
        {
            Load Update
        }
    }
}
```

9.11.5 Microcode Update Checksum

Each microcode update contains a DWORD checksum located in the update header. It is software's responsibility to ensure that a microcode update is not corrupt. To check for a corrupt microcode update, software must perform a unsigned DWORD (32-bit) checksum of the microcode update. Even though some fields are signed, the checksum

procedure treats all DWORDs as unsigned. Microcode updates with a header version equal to 00000001H must sum all DWORDs that comprise the microcode update. A valid checksum check will yield a value of 00000000H. Any other value indicates the microcode update is corrupt and should not be loaded.

The checksum algorithm shown by the pseudo code in Example 9-7 treats the microcode update as an array of unsigned DWORDs. If the data size DWORD field at byte offset 32 equals 00000000H, the size of the encrypted data is 2000 bytes, resulting in 500 DWORDs. Otherwise the microcode update size in DWORDs = (*Total Size / 4*), where the total size is a multiple of 1024 bytes (1 KBytes).

Example 9-7. Pseudo Code Example of Checksum Test

```
N ← 512

If (Update.DataSize ≠ 00000000H)
   N ← Update.TotalSize / 4

ChkSum ← 0
For (I ← 0; I < N; I++)
{
   ChkSum ← ChkSum + MicrocodeUpdate[I]
}

If (ChkSum = 00000000H)
   Success
Else
   Fail
```

9.11.6 Microcode Update Loader

This section describes an update loader used to load an update into a P6 family or later processors. It also discusses the requirements placed on the BIOS to ensure proper loading. The update loader described contains the minimal instructions needed to load an update. The specific instruction sequence that is required to load an update is dependent upon the loader revision field contained within the update header. This revision is expected to change infrequently (potentially, only when new processor models are introduced).

Example 9-8 below represents the update loader with a loader revision of 00000001H. Note that the microcode update must be aligned on a 16-byte boundary and the size of the microcode update must be 1-KByte granular.

Example 9-8. Assembly Code Example of Simple Microcode Update Loader

```
mov   ecx,79h              ; MSR to write in ECX
xor   eax,eax              ; clear EAX
xor   ebx,ebx              ; clear EBX
mov   ax,cs                ; Segment of microcode update
shl   eax,4
mov   bx,offset Update     ; Offset of microcode update
add   eax,ebx              ; Linear Address of Update in EAX
add   eax,48d              ; Offset of the Update Data within the Update
xor   edx,edx              ; Zero in EDX
WRMSR                     ; microcode update trigger
```

The loader shown in Example 9-8 assumes that *update* is the address of a microcode update (header and data) embedded within the code segment of the BIOS. It also assumes that the processor is operating in real mode. The data may reside anywhere in memory, aligned on a 16-byte boundary, that is accessible by the processor within its current operating mode.

Before the BIOS executes the microcode update trigger (WRMSR) instruction, the following must be true:

- In 64-bit mode, EAX contains the lower 32-bits of the microcode update linear address. In protected mode, EAX contains the full 32-bit linear address of the microcode update.
- In 64-bit mode, EDX contains the upper 32-bits of the microcode update linear address. In protected mode, EDX equals zero.
- ECX contains 79H (address of IA32_BIOS_UPDT_TRIG).

Other requirements are:

- The addresses for the microcode update data must be in canonical form.
- If paging is enabled, the microcode update data must map that data as present.
- The microcode update data must start at a 16-byte aligned linear address.

9.11.6.1 Hard Resets in Update Loading

The effects of a loaded update are cleared from the processor upon a hard reset. Therefore, each time a hard reset is asserted during the BIOS POST, the update must be reloaded on all processors that observed the reset. The effects of a loaded update are, however, maintained across a processor INIT. There are no side effects caused by loading an update into a processor multiple times.

9.11.6.2 Update in a Multiprocessor System

A multiprocessor (MP) system requires loading each processor with update data appropriate for its CPUID and platform ID bits. The BIOS is responsible for ensuring that this requirement is met and that the loader is located in a module executed by all processors in the system. If a system design permits multiple steppings of Pentium 4, Intel Xeon, and P6 family processors to exist concurrently; then the BIOS must verify individual processors against the update header information to ensure appropriate loading. Given these considerations, it is most practical to load the update during MP initialization.

9.11.6.3 Update in a System Supporting Intel Hyper-Threading Technology

Intel Hyper-Threading Technology has implications on the loading of the microcode update. The update must be loaded for each core in a physical processor. Thus, for a processor supporting Intel Hyper-Threading Technology, only one logical processor per core is required to load the microcode update. Each individual logical processor can independently load the update. However, MP initialization must provide some mechanism (e.g. a software semaphore) to force serialization of microcode update loads and to prevent simultaneous load attempts to the same core.

9.11.6.4 Update in a System Supporting Dual-Core Technology

Dual-core technology has implications on the loading of the microcode update. The microcode update facility is not shared between processor cores in the same physical package. The update must be loaded for each core in a physical processor.

If processor core supports Intel Hyper-Threading Technology, the guideline described in Section 9.11.6.3 also applies.

9.11.6.5 Update Loader Enhancements

The update loader presented in Section 9.11.6, “Microcode Update Loader,” is a minimal implementation that can be enhanced to provide additional functionality. Potential enhancements are described below:

- BIOS can incorporate multiple updates to support multiple steppings of the Pentium 4, Intel Xeon, and P6 family processors. This feature provides for operating in a mixed stepping environment on an MP system and enables a user to upgrade to a later version of the processor. In this case, modify the loader to check the CPUID and platform ID bits of the processor that it is running on against the available headers before loading a particular update. The number of updates is only limited by available BIOS space.

- A loader can load the update and test the processor to determine if the update was loaded correctly. See Section 9.11.7, "Update Signature and Verification."
- A loader can verify the integrity of the update data by performing a checksum on the double words of the update summing to zero. See Section 9.11.5, "Microcode Update Checksum."
- A loader can provide power-on messages indicating successful loading of an update.

9.11.7 Update Signature and Verification

The P6 family and later processors provide capabilities to verify the authenticity of a particular update and to identify the current update revision. This section describes the model-specific extensions of processors that support this feature. The update verification method below assumes that the BIOS will only verify an update that is more recent than the revision currently loaded in the processor.

CPUID returns a value in a model specific register in addition to its usual register return values. The semantics of CPUID cause it to deposit an update ID value in the 64-bit model-specific register at address 08BH (IA32_BIOS_SIGN_ID). If no update is present in the processor, the value in the MSR remains unmodified. The BIOS must pre-load a zero into the MSR before executing CPUID. If a read of the MSR at 8BH still returns zero after executing CPUID, this indicates that no update is present.

The update ID value returned in the EDX register after RDMSR executes indicates the revision of the update loaded in the processor. This value, in combination with the CPUID value returned in the EAX register, uniquely identifies a particular update. The signature ID can be directly compared with the update revision field in a microcode update header for verification of a correct load. No consecutive updates released for a given stepping of a processor may share the same signature. The processor signature returned by CPUID differentiates updates for different steppings.

9.11.7.1 Determining the Signature

An update that is successfully loaded into the processor provides a signature that matches the update revision of the currently functioning revision. This signature is available any time after the actual update has been loaded. Requesting the signature does not have a negative impact upon a loaded update.

The procedure for determining this signature shown in Example 9-9.

Example 9-9. Assembly Code to Retrieve the Update Revision

```
MOV    ECX, 08BH          ;IA32_BIOS_SIGN_ID
XOR    EAX, EAX           ;clear EAX
XOR    EDX, EDX           ;clear EDX
WRMSR                     ;Load 0 to MSR at 8BH
MOV    EAX, 1
cpuid
MOV    ECX, 08BH           ;IA32_BIOS_SIGN_ID
rdmsr                      ;Read Model Specific Register
```

If there is an update active in the processor, its revision is returned in the EDX register after the RDMSR instruction executes.

IA32_BIOS_SIGN_ID	Microcode Update Signature Register
MSR Address:	08BH Accessed as a Qword
Default Value:	XXXX XXXX XXXX XXXXh
Access:	Read/Write

The IA32_BIOS_SIGN_ID register is used to report the microcode update signature when CPUID executes. The signature is returned in the upper DWORD (Table 9-12).

Table 9-12. Microcode Update Signature

Bit	Description
63:32	Microcode update signature. This field contains the signature of the currently loaded microcode update when read following the execution of the CPUID instruction, function 1. It is required that this register field be pre-loaded with zero prior to executing the CPUID, function 1. If the field remains equal to zero, then there is no microcode update loaded. Another non-zero value will be the signature.
31:0	Reserved.

9.11.7.2 Authenticating the Update

An update may be authenticated by the BIOS using the signature primitive, described above, and the algorithm in Example 9-10.

Example 9-10. Pseudo Code to Authenticate the Update

```
Z ← Obtain Update Revision from the Update Header to be authenticated;
X ← Obtain Current Update Signature from MSR 8BH;

If (Z > X)
{
    Load Update that is to be authenticated;
    Y ← Obtain New Signature from MSR 8BH;

    If (Z = Y)
        Success
    Else
        Fail
}
Else
    Fail
```

Example 9-10 requires that the BIOS only authenticate updates that contain a numerically larger revision than the currently loaded revision, where Current Signature (X) < New Update Revision (Z). A processor with no loaded update is considered to have a revision equal to zero.

This authentication procedure relies upon the decoding provided by the processor to verify an update from a potentially hostile source. As an example, this mechanism in conjunction with other safeguards provides security for dynamically incorporating field updates into the BIOS.

9.11.8 Optional Processor Microcode Update Specifications

This section an interface that an OEM-BIOS may provide to its client system software to manage processor microcode updates. System software may choose to build its own facility to manage microcode updates (e.g. similar to the facility described in Section 9.11.6) or rely on a facility provided by the BIOS to perform microcode updates.

Sections 9.11.8.1-9.11.8.9 describes an extension (Function 0D042H) to the real mode INT 15H service. INT 15H 0D042H function is one of several alternatives that a BIOS may choose to implement microcode update facility and offer to its client application (e.g. an OS). Other alternative microcode update facility that BIOS can choose are dependent on platform-specific capabilities, including the Capsule Update mechanism from the UEFI specification (www.uefi.org). In this discussion, the application is referred to as the calling program or caller.

The real mode INT15 call specification described here is an Intel extension to an OEM BIOS. This extension allows an application to read and modify the contents of the microcode update data in NVRAM. The update loader, which is part of the system BIOS, cannot be updated by the interface. All of the functions defined in the specification must be implemented for a system to be considered compliant with the specification. The INT15 functions are accessible only from real mode.

9.11.8.1 Responsibilities of the BIOS

If a BIOS passes the presence test (INT 15H, AX = 0D042H, BL = 0H), it must implement all of the sub-functions defined in the INT 15H, AX = 0D042H specification. There are no optional functions. BIOS must load the appropriate update for each processor during system initialization.

A Header Version of an update block containing the value 0FFFFFFFFH indicates that the update block is unused and available for storing a new update.

The BIOS is responsible for providing a region of non-volatile storage (NVRAM) for each potential processor stepping within a system. This storage unit consists of one or more update blocks. An update block is a contiguous 2048-byte block of memory. The BIOS for a single processor system need only provide update blocks to store one microcode update. If the BIOS for a multiple processor system is intended to support mixed processor steppings, then the BIOS needs to provide enough update blocks to store each unique microcode update or for each processor socket on the OEM's system board.

The BIOS is responsible for managing the NVRAM update blocks. This includes garbage collection, such as removing microcode updates that exist in NVRAM for which a corresponding processor does not exist in the system. This specification only provides the mechanism for ensuring security, the uniqueness of an entry, and that stale entries are not loaded. The actual update block management is implementation specific on a per-BIOS basis.

As an example, the BIOS may use update blocks sequentially in ascending order with CPU signatures sorted versus the first available block. In addition, garbage collection may be implemented as a setup option to clear all NVRAM slots or as BIOS code that searches and eliminates unused entries during boot.

NOTES

For IA-32 processors starting with family 0FH and model 03H and Intel 64 processors, the microcode update may be as large as 16 KBytes. Thus, BIOS must allocate 8 update blocks for each microcode update. In a MP system, a common microcode update may be sufficient for each socket in the system.

For IA-32 processors earlier than family 0FH and model 03H, the microcode update is 2 KBytes. An MP-capable BIOS that supports multiple steppings must allocate a block for each socket in the system.

A single-processor BIOS that supports variable-sized microcode update and fixed-sized microcode update must allocate one 16-KByte region and a second region of at least 2 KBytes.

The following algorithm (Example 9-11) describes the steps performed during BIOS initialization used to load the updates into the processor(s). The algorithm assumes:

- The BIOS ensures that no update contained within NVRAM has a header version or loader version that does not match one currently supported by the BIOS.
- The update contains a correct checksum.
- The BIOS ensures that (at most) one update exists for each processor stepping.
- Older update revisions are not allowed to overwrite more recent ones.

These requirements are checked by the BIOS during the execution of the write update function of this interface. The BIOS sequentially scans through all of the update blocks in NVRAM starting with index 0. The BIOS scans until it finds an update where the processor fields in the header match the processor signature (extended family, extended model, type, family, model, and stepping) as well as the platform bits of the current processor.

Example 9-11. Pseudo Code, Checks Required Prior to Loading an Update

```
For each processor in the system
{
    Determine the Processor Signature via CPUID function 1;
    Determine the Platform Bits ← 1 << IA32_PLATFORM_ID[52:50];

    For (I ← UpdateBlock 0, I < NumOfBlocks; I++)
    {
        If (Update.Header_Version = 00000001H)
        {
```

```
            If ((Update.ProcessorSignature = Processor Signature) &&
                 (Update.ProcessorFlags & Platform Bits))
            {
                Load Update.UpdateData into the Processor;
                Verify update was correctly loaded into the processor
                Go on to next processor
                    Break;
            }
            Else If (Update.TotalSize > (Update.DataSize + 48))
            {
                N ← 0
                While (N < Update.ExtendedSignatureCount)
                {
                    If ((Update.ProcessorSignature[N] =
                         Processor Signature) &&
                         (Update.ProcessorFlags[N] & Platform Bits))
                    {
                        Load Update.UpdateData into the Processor;
                        Verify update correctly loaded into the processor
                        Go on to next processor
                            Break;
                    }
                    N ← N + 1
                }
                I ← I + (Update.TotalSize / 2048)
                If ((Update.TotalSize MOD 2048) = 0)
                    I ← I + 1
            }
        }
    }
}
```

NOTES

The platform Id bits in IA32_PLATFORM_ID are encoded as a three-bit binary coded decimal field. The platform bits in the microcode update header are individually bit encoded. The algorithm must do a translation from one format to the other prior to doing a check.

When performing the INT 15H, 0D042H functions, the BIOS must assume that the caller has no knowledge of platform specific requirements. It is the responsibility of BIOS calls to manage all chipset and platform specific prerequisites for managing the NVRAM device. When writing the update data using the Write Update sub-function, the BIOS must maintain implementation specific data requirements (such as the update of NVRAM checksum). The BIOS should also attempt to verify the success of write operations on the storage device used to record the update.

9.11.8.2 Responsibilities of the Calling Program

This section of the document lists the responsibilities of a calling program using the interface specifications to load microcode update(s) into BIOS NVRAM.

- The calling program should call the INT 15H, 0D042H functions from a pure real mode program and should be executing on a system that is running in pure real mode.
- The caller should issue the presence test function (sub function 0) and verify the signature and return codes of that function.
- It is important that the calling program provides the required scratch RAM buffers for the BIOS and the proper stack size as specified in the interface definition.
- The calling program should read any update data that already exists in the BIOS in order to make decisions about the appropriateness of loading the update. The BIOS must refuse to overwrite a newer update with an

older version. The update header contains information about version and processor specifics for the calling program to make an intelligent decision about loading.

- There can be no ambiguous updates. The BIOS must refuse to allow multiple updates for the same CPU to exist at the same time; it also must refuse to load updates for processors that don't exist on the system.
- The calling application should implement a verify function that is run after the update write function successfully completes. This function reads back the update and verifies that the BIOS returned an image identical to the one that was written.

Example 9-12 represents a calling program.

Example 9-12. INT 15 DO42 Calling Program Pseudo-code

```
//
// We must be in real mode
//
If the system is not in Real mode exit
//
// Detect presence of Genuine Intel processor(s) that can be updated
// using(CPUID)
//
If no Intel processors exist that can be updated exit
//
// Detect the presence of the Intel microcode update extensions
//
If the BIOS fails the PresenceTestexit
//
// If the APIC is enabled, see if any other processors are out there
//
Read IA32_APICBASE
If APIC enabled
{
    Send Broadcast Message to all processors except self via APIC
    Have all processors execute CPUID, record the Processor Signature
    (i.e.,Extended Family, Extended Model, Type, Family, Model, Stepping)
    Have all processors read IA32_PLATFORM_ID[52:50], record Platform
     Id Bits

    If current processor cannot be updated
       exit
}
//
// Determine the number of unique update blocks needed for this system
//
NumBlocks = 0
For each processor
{
    If ((this is a unique processor stepping) AND
       (we have a unique update in the database for this processor))
    {
       Checksum the update from the database;
       If Checksum fails
          exit
       NumBlocks ← NumBlocks + size of microcode update / 2048
    }
}

//
// Do we have enough update slots for all CPUs?
//
```

```
If there are more blocks required to support the unique processor steppings than update blocks
provided by the BIOS exit
//
// Do we need any update blocks at all?  If not, we are done
//
If (NumBlocks = 0)
    exit
//
// Record updates for processors in NVRAM.
//
For (I=0; I<NumBlocks; I++)
{
    //
    // Load each Update
    //
    Issue the WriteUpdate function

    If (STORAGE_FULL) returned
    {
        Display Error -- BIOS is not managing NVRAM appropriately
        exit
    }

    If (INVALID_REVISION) returned
    {
        Display Message: More recent update already loaded in NVRAM for
         this stepping
        continue
    }

    If any other error returned
    {
        Display Diagnostic
        exit
    }

    //
    // Verify the update was loaded correctly
    //
    Issue the ReadUpdate function

    If an error occurred
    {
        Display Diagnostic
        exit
    }
    //
    // Compare the Update read to that written
    //
    If (Update read ≠ Update written)
    {
        Display Diagnostic
        exit
    }

    I ← I + (size of microcode update / 2048)
}
//
// Enable Update Loading, and inform user
```

```
//
Issue the Update Control function with Task = Enable.
```

9.11.8.3 Microcode Update Functions

Table 9-13 defines the processor microcode update functions that implementations of INT 15H 0D042H must support.

Table 9-13. Microcode Update Functions

Microcode Update Function	Function Number	Description	Required/Optional
Presence test	00H	Returns information about the supported functions.	Required
Write update data	01H	Writes one of the update data areas (slots).	Required
Update control	02H	Globally controls the loading of updates.	Required
Read update data	03H	Reads one of the update data areas (slots).	Required

9.11.8.4 INT 15H-based Interface

If an OEM-BIOS is implementing INT 15H 0D042H interface and offer to its client, the BIOS should allow additional microcode updates to be added to system flash.

The program that calls this interface is responsible for providing three 64-kilobyte RAM areas for BIOS use during calls to the read and write functions. These RAM scratch pads can be used by the BIOS for any purpose, but only for the duration of the function call. The calling routine places real mode segments pointing to the RAM blocks in the CX, DX and SI registers. Calls to functions in this interface must be made with a minimum of 32 kilobytes of stack available to the BIOS.

In general, each function returns with CF cleared and AH contains the returned status. The general return codes and other constant definitions are listed in Section 9.11.8.9, "Return Codes."

The OEM error field (AL) is provided for the OEM to return additional error information specific to the platform. If the BIOS provides no additional information about the error, OEM error must be set to SUCCESS. The OEM error field is undefined if AH contains either SUCCESS (00H) or NOT_IMPLEMENTED (86H). In all other cases, it must be set with either SUCCESS or a value meaningful to the OEM.

The following sections describe functions provided by the INT15H-based interface.

9.11.8.5 Function 00H—Presence Test

This function verifies that the BIOS has implemented required microcode update functions. Table 9-14 lists the parameters and return codes for the function.

Table 9-14. Parameters for the Presence Test

Input		
AX	Function Code	0D042H
BL	Sub-function	00H - Presence test
Output		
CF	Carry Flag	Carry Set - Failure - AH contains status Carry Clear - All return values valid
AH	Return Code	
AL	OEM Error	Additional OEM information.
EBX	Signature Part 1	'INTE' - Part one of the signature
ECX	Signature Part 2	'LPEP'- Part two of the signature
EDX	Loader Version	Version number of the microcode update loader

Table 9-14. Parameters for the Presence Test (Contd.)

Input		
SI	Update Count	Number of 2048 update blocks in NVRAM the BIOS allocated to storing microcode updates
Return Codes (see Table 9-19 for code definitions		
SUCCESS		The function completed successfully.
NOT_IMPLEMENTED		The function is not implemented.

In order to assure that the BIOS function is present, the caller must verify the carry flag, the return code, and the 64-bit signature. The update count reflects the number of 2048-byte blocks available for storage within one non-volatile RAM.

The loader version number refers to the revision of the update loader program that is included in the system BIOS image.

9.11.8.6 Function 01H—Write Microcode Update Data

This function integrates a new microcode update into the BIOS storage device. Table 9-15 lists the parameters and return codes for the function.

Table 9-15. Parameters for the Write Update Data Function

Input		
AX	Function Code	0D042H
BL	Sub-function	01H - Write update
ES:DI	Update Address	Real Mode pointer to the Intel Update structure. This buffer is 2048 bytes in length if the processor supports only fixed-size microcode update or... Real Mode pointer to the Intel Update structure. This buffer is 64 KBytes in length if the processor supports a variable-size microcode update.
CX	Scratch Pad1	Real mode segment address of 64 KBytes of RAM block
DX	Scratch Pad2	Real mode segment address of 64 KBytes of RAM block
SI	Scratch Pad3	Real mode segment address of 64 KBytes of RAM block
SS:SP	Stack pointer	32 KBytes of stack minimum
Output		
CF	Carry Flag	Carry Set - Failure - AH Contains status Carry Clear - All return values valid
AH	Return Code	Status of the call
AL	OEM Error	Additional OEM information
Return Codes (see Table 9-19 for code definitions		
SUCCESS		The function completed successfully.
NOT_IMPLEMENTED		The function is not implemented.
WRITE_FAILURE		A failure occurred because of the inability to write the storage device.
ERASE_FAILURE		A failure occurred because of the inability to erase the storage device.
READ_FAILURE		A failure occurred because of the inability to read the storage device.

Table 9-15. Parameters for the Write Update Data Function (Contd.)

Input	
STORAGE_FULL	The BIOS non-volatile storage area is unable to accommodate the update because all available update blocks are filled with updates that are needed for processors in the system.
CPU_NOT_PRESENT	The processor stepping does not currently exist in the system.
INVALID_HEADER	The update header contains a header or loader version that is not recognized by the BIOS.
INVALID_HEADER_CS	The update does not checksum correctly.
SECURITY_FAILURE	The processor rejected the update.
INVALID_REVISION	The same or more recent revision of the update exists in the storage device.

Description

The BIOS is responsible for selecting an appropriate update block in the non-volatile storage for storing the new update. This BIOS is also responsible for ensuring the integrity of the information provided by the caller, including authenticating the proposed update before incorporating it into storage.

Before writing the update block into NVRAM, the BIOS should ensure that the update structure meets the following criteria in the following order:

1. The update header version should be equal to an update header version recognized by the BIOS.
2. The update loader version in the update header should be equal to the update loader version contained within the BIOS image.
3. The update block must checksum. This checksum is computed as a 32-bit summation of all double words in the structure, including the header, data, and processor signature table.

The BIOS selects update block(s) in non-volatile storage for storing the candidate update. The BIOS can select any available update block as long as it guarantees that only a single update exists for any given processor stepping in non-volatile storage. If the update block selected already contains an update, the following additional criteria apply to overwrite it:

- The processor signature in the proposed update must be equal to the processor signature in the header of the current update in NVRAM (Processor Signature + platform ID bits).
- The update revision in the proposed update should be greater than the update revision in the header of the current update in NVRAM.

If no unused update blocks are available and the above criteria are not met, the BIOS can overwrite update block(s) for a processor stepping that is no longer present in the system. This can be done by scanning the update blocks and comparing the processor steppings, identified in the MP Specification table, to the processor steppings that currently exist in the system.

Finally, before storing the proposed update in NVRAM, the BIOS must verify the authenticity of the update via the mechanism described in Section 9.11.6, "Microcode Update Loader." This includes loading the update into the current processor, executing the CPUID instruction, reading MSR 08Bh, and comparing a calculated value with the update revision in the proposed update header for equality.

When performing the write update function, the BIOS must record the entire update, including the header, the update data, and the extended processor signature table (if applicable). When writing an update, the original contents may be overwritten, assuming the above criteria have been met. It is the responsibility of the BIOS to ensure that more recent updates are not overwritten through the use of this BIOS call, and that only a single update exists within the NVRAM for any processor stepping and platform ID.

Figure 9-8 and Figure 9-9 show the process the BIOS follows to choose an update block and ensure the integrity of the data when it stores the new microcode update.

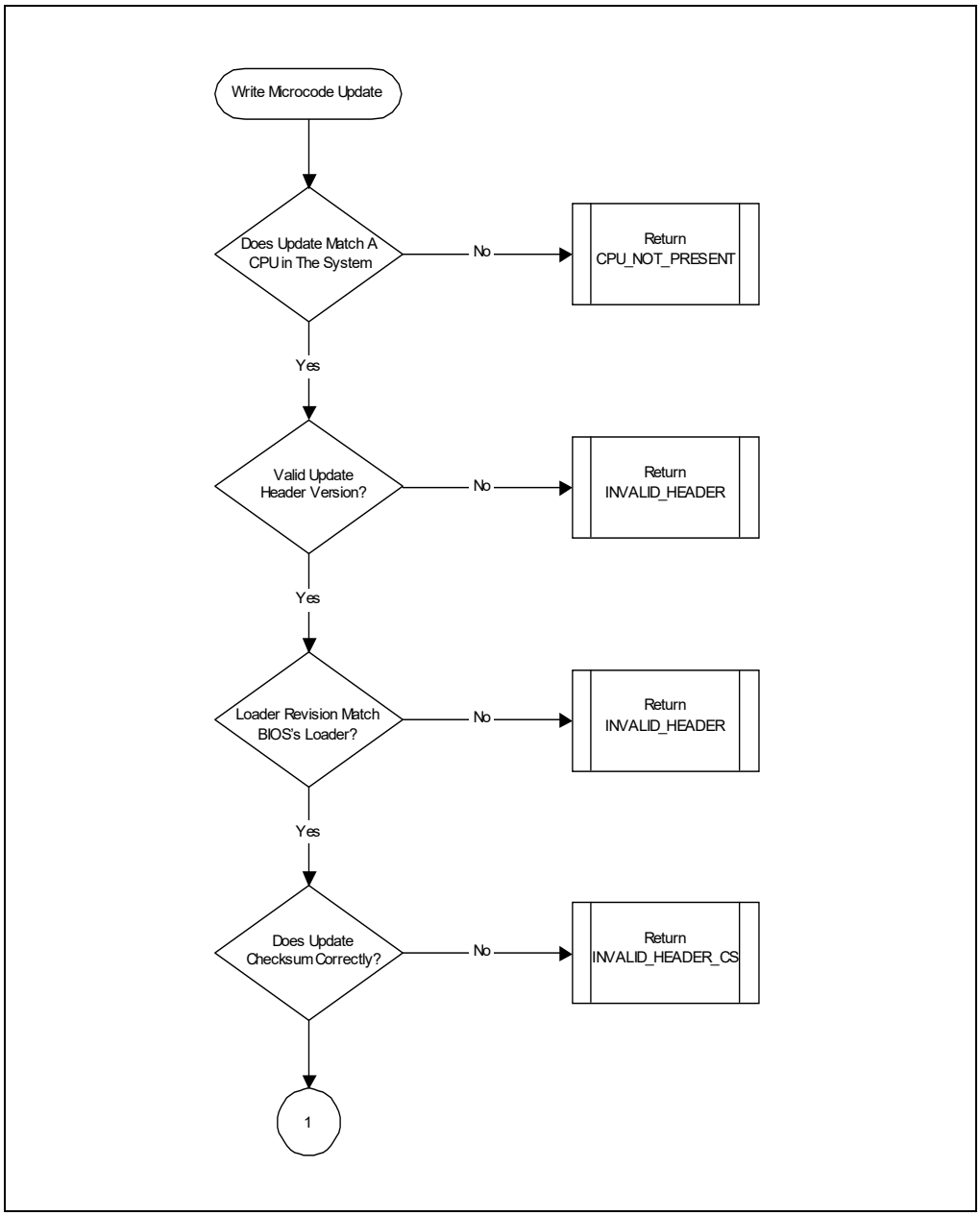


Figure 9-8. Microcode Update Write Operation Flow [1]

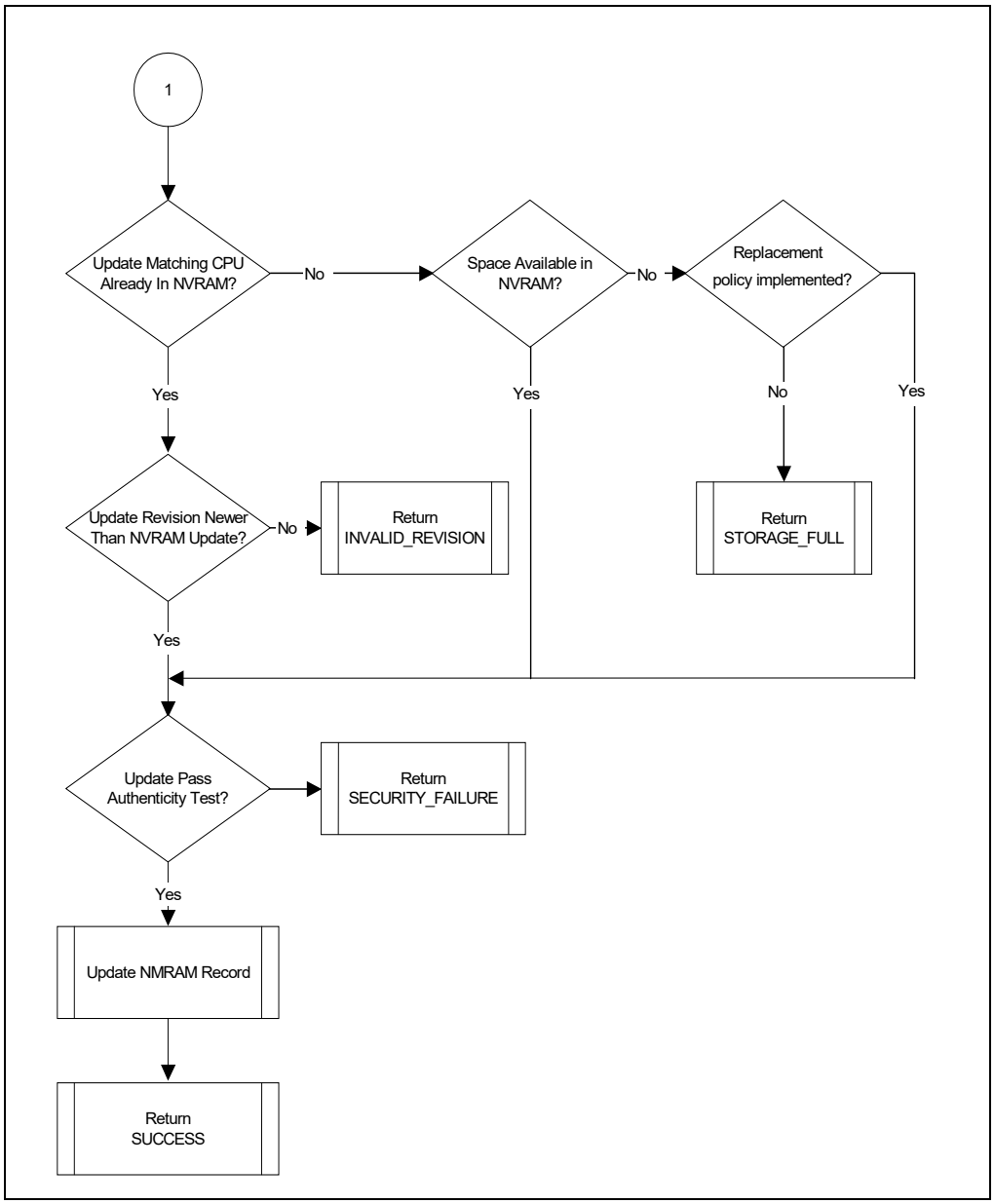


Figure 9-9. Microcode Update Write Operation Flow [2]

9.11.8.7 Function 02H—Microcode Update Control

This function enables loading of binary updates into the processor. Table 9-16 lists the parameters and return codes for the function.

Table 9-16. Parameters for the Control Update Sub-function

Input		
AX	Function Code	0D042H
BL	Sub-function	02H - Control update
BH	Task	See the description below.
CX	Scratch Pad1	Real mode segment of 64 KBytes of RAM block
DX	Scratch Pad2	Real mode segment of 64 KBytes of RAM block
SI	Scratch Pad3	Real mode segment of 64 KBytes of RAM block
SS:SP	Stack pointer	32 kilobytes of stack minimum
Output		
CF	Carry Flag	Carry Set - Failure - AH contains status Carry Clear - All return values valid.
AH	Return Code	Status of the call
AL	OEM Error	Additional OEM Information.
BL	Update Status	Either enable or disable indicator
Return Codes (see Table 9-19 for code definitions)		
SUCCESS		Function completed successfully.
READ_FAILURE		A failure occurred because of the inability to read the storage device.

This control is provided on a global basis for all updates and processors. The caller can determine the current status of update loading (enabled or disabled) without changing the state. The function does not allow the caller to disable loading of binary updates, as this poses a security risk.

The caller specifies the requested operation by placing one of the values from Table 9-17 in the BH register. After successfully completing this function, the BL register contains either the enable or the disable designator. Note that if the function fails, the update status return value is undefined.

Table 9-17. Mnemonic Values

Mnemonic	**Value**	**Meaning**
Enable	1	Enable the Update loading at initialization time.
Query	2	Determine the current state of the update control without changing its status.

The READ_FAILURE error code returned by this function has meaning only if the control function is implemented in the BIOS NVRAM. The state of this feature (enabled/disabled) can also be implemented using CMOS RAM bits where READ failure errors cannot occur.

9.11.8.8 Function 03H—Read Microcode Update Data

This function reads a currently installed microcode update from the BIOS storage into a caller-provided RAM buffer. Table 9-18 lists the parameters and return codes.

Table 9-18. Parameters for the Read Microcode Update Data Function

Input		
AX	Function Code	0D042H
BL	Sub-function	03H - Read Update
ES:DI	Buffer Address	Real Mode pointer to the Intel Update structure that will be written with the binary data

Table 9-18. Parameters for the Read Microcode Update Data Function (Contd.)

ECX	Scratch Pad1	Real Mode Segment address of 64 KBytes of RAM Block (lower 16 bits)
ECX	Scratch Pad2	Real Mode Segment address of 64 KBytes of RAM Block (upper 16 bits)
DX	Scratch Pad3	Real Mode Segment address of 64 KBytes of RAM Block
SS:SP	Stack pointer	32 KBytes of Stack Minimum
SI	Update Number	This is the index number of the update block to be read. This value is zero based and must be less than the update count returned from the presence test function.
Output		
CF	Carry Flag	Carry Set - Failure - AH contains Status
Carry Clear - All return values are valid.		
AH	Return Code	Status of the Call
AL	OEM Error	Additional OEM Information
Return Codes (see Table 9-19 for code definitions)		
SUCCESS		The function completed successfully.
READ_FAILURE		There was a failure because of the inability to read the storage device.
UPDATE_NUM_INVALID		Update number exceeds the maximum number of update blocks implemented by the BIOS.
NOT_EMPTY		The specified update block is a subsequent block in use to store a valid microcode update that spans multiple blocks. The specified block is not a header block and is not empty.

The read function enables the caller to read any microcode update data that already exists in a BIOS and make decisions about the addition of new updates. As a result of a successful call, the BIOS copies the microcode update into the location pointed to by ES:DI, with the contents of all Update block(s) that are used to store the specified microcode update.

If the specified block is not a header block, but does contain valid data from a microcode update that spans multiple update blocks, then the BIOS must return Failure with the NOT_EMPTY error code in AH.

An update block is considered unused and available for storing a new update if its Header Version contains the value 0FFFFFFFFH after return from this function call. The actual implementation of NVRAM storage management is not specified here and is BIOS dependent. As an example, the actual data value used to represent an empty block by the BIOS may be zero, rather than 0FFFFFFFFH. The BIOS is responsible for translating this information into the header provided by this function.

9.11.8.9 Return Codes

After the call has been made, the return codes listed in Table 9-19 are available in the AH register.

Table 9-19. Return Code Definitions

Return Code	Value	Description
SUCCESS	00H	The function completed successfully.
NOT_IMPLEMENTED	86H	The function is not implemented.
ERASE_FAILURE	90H	A failure because of the inability to erase the storage device.
WRITE_FAILURE	91H	A failure because of the inability to write the storage device.
READ_FAILURE	92H	A failure because of the inability to read the storage device.
STORAGE_FULL	93H	The BIOS non-volatile storage area is unable to accommodate the update because all available update blocks are filled with updates that are needed for processors in the system.
CPU_NOT_PRESENT	94H	The processor stepping does not currently exist in the system.
INVALID_HEADER	95H	The update header contains a header or loader version that is not recognized by the BIOS.
INVALID_HEADER_CS	96H	The update does not checksum correctly.
SECURITY_FAILURE	97H	The update was rejected by the processor.
INVALID_REVISION	98H	The same or more recent revision of the update exists in the storage device.
UPDATE_NUM_INVALID	99H	The update number exceeds the maximum number of update blocks implemented by the BIOS.
NOT_EMPTY	9AH	The specified update block is a subsequent block in use to store a valid microcode update that spans multiple blocks. The specified block is not a header block and is not empty.

13.Updates to Chapter 17, Volume 3B

Change bars show changes to Chapter 17 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3B:* System Programming Guide, Part 2.

Changes to this chapter: Updates to Sections 17.4.8.1 “LBR Stack and Intel® 64 Processors” and 17.6 “Last Branch, Call Stack, Interrupt, and Exception Recording for Processors based on Goldmont Microarchitecture”. Update to Section 17.4.9 “BTS and DS Save Area”. Addition of new Section 17.7 “Last Branch, Call Stack, Interrupt, and Exception Recording for Processors based on Goldmont Plus Microarchitecture”.

CHAPTER 17
DEBUG, BRANCH PROFILE, TSC, AND RESOURCE MONITORING FEATURES

Intel 64 and IA-32 architectures provide debug facilities for use in debugging code and monitoring performance. These facilities are valuable for debugging application software, system software, and multitasking operating systems. Debug support is accessed using debug registers (DR0 through DR7) and model-specific registers (MSRs):

- Debug registers hold the addresses of memory and I/O locations called breakpoints. Breakpoints are user-selected locations in a program, a data-storage area in memory, or specific I/O ports. They are set where a programmer or system designer wishes to halt execution of a program and examine the state of the processor by invoking debugger software. A debug exception (#DB) is generated when a memory or I/O access is made to a breakpoint address.
- MSRs monitor branches, interrupts, and exceptions; they record addresses of the last branch, interrupt or exception taken and the last branch taken before an interrupt or exception.
- Time stamp counter is described in Section 17.17, "Time-Stamp Counter".
- Features which allow monitoring of shared platform resources such as the L3 cache are described in Section 17.18, "Intel® Resource Director Technology (Intel® RDT) Monitoring Features".
- Features which enable control over shared platform resources are described in Section 17.19, "Intel® Resource Director Technology (Intel® RDT) Allocation Features".

17.1 OVERVIEW OF DEBUG SUPPORT FACILITIES

The following processor facilities support debugging and performance monitoring:

- **Debug exception (#DB) —** Transfers program control to a debug procedure or task when a debug event occurs.
- **Breakpoint exception (#BP) —** See breakpoint instruction (INT 3) below.
- **Breakpoint-address registers (DR0 through DR3) —** Specifies the addresses of up to 4 breakpoints.
- **Debug status register (DR6) —** Reports the conditions that were in effect when a debug or breakpoint exception was generated.
- **Debug control register (DR7) —** Specifies the forms of memory or I/O access that cause breakpoints to be generated.
- **T (trap) flag, TSS —** Generates a debug exception (#DB) when an attempt is made to switch to a task with the T flag set in its TSS.
- **RF (resume) flag, EFLAGS register —** Suppresses multiple exceptions to the same instruction.
- **TF (trap) flag, EFLAGS register —** Generates a debug exception (#DB) after every execution of an instruction.
- **Breakpoint instruction (INT 3) —** Generates a breakpoint exception (#BP) that transfers program control to the debugger procedure or task. This instruction is an alternative way to set code breakpoints. It is especially useful when more than four breakpoints are desired, or when breakpoints are being placed in the source code.
- **Last branch recording facilities** — Store branch records in the last branch record (LBR) stack MSRs for the most recent taken branches, interrupts, and/or exceptions in MSRs. A branch record consist of a branch-from and a branch-to instruction address. Send branch records out on the system bus as branch trace messages (BTMs).

These facilities allow a debugger to be called as a separate task or as a procedure in the context of the current program or task. The following conditions can be used to invoke the debugger:

- Task switch to a specific task.
- Execution of the breakpoint instruction.

- Execution of any instruction.
- Execution of an instruction at a specified address.
- Read or write to a specified memory address/range.
- Write to a specified memory address/range.
- Input from a specified I/O address/range.
- Output to a specified I/O address/range.
- Attempt to change the contents of a debug register.

17.2 DEBUG REGISTERS

Eight debug registers (see Figure 17-1 for 32-bit operation and Figure 17-2 for 64-bit operation) control the debug operation of the processor. These registers can be written to and read using the move to/from debug register form of the MOV instruction. A debug register may be the source or destination operand for one of these instructions.

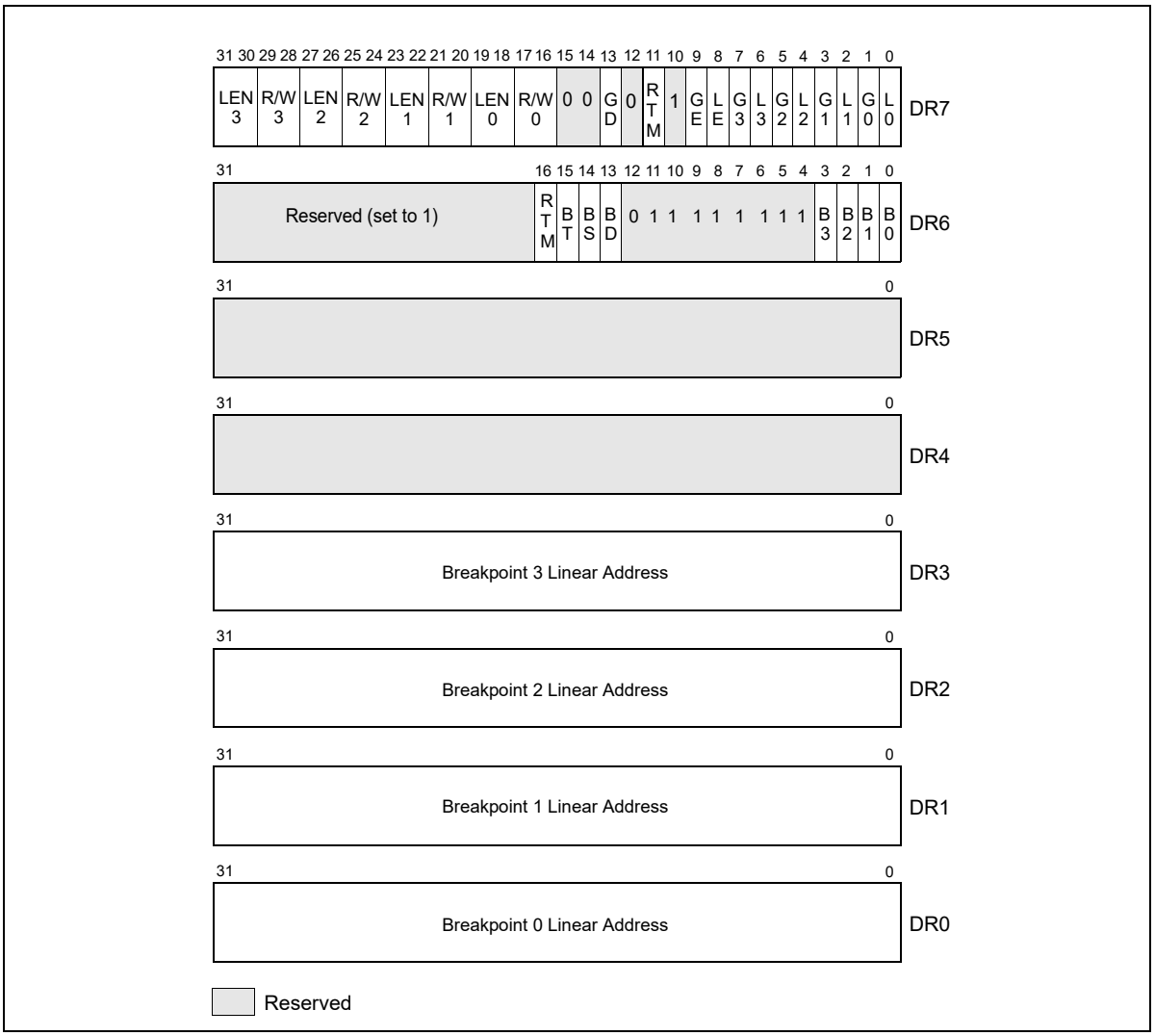


Figure 17-1. Debug Registers

Debug registers are privileged resources; a MOV instruction that accesses these registers can only be executed in real-address mode, in SMM or in protected mode at a CPL of 0. An attempt to read or write the debug registers from any other privilege level generates a general-protection exception (#GP).

The primary function of the debug registers is to set up and monitor from 1 to 4 breakpoints, numbered 0 though 3. For each breakpoint, the following information can be specified:

- The linear address where the breakpoint is to occur.
- The length of the breakpoint location: 1, 2, 4, or 8 bytes (refer to the notes in Section 17.2.4).
- The operation that must be performed at the address for a debug exception to be generated.
- Whether the breakpoint is enabled.
- Whether the breakpoint condition was present when the debug exception was generated.

The following paragraphs describe the functions of flags and fields in the debug registers.

17.2.1 Debug Address Registers (DR0-DR3)

Each of the debug-address registers (DR0 through DR3) holds the 32-bit linear address of a breakpoint (see Figure 17-1). Breakpoint comparisons are made before physical address translation occurs. The contents of debug register DR7 further specifies breakpoint conditions.

17.2.2 Debug Registers DR4 and DR5

Debug registers DR4 and DR5 are reserved when debug extensions are enabled (when the DE flag in control register CR4 is set) and attempts to reference the DR4 and DR5 registers cause invalid-opcode exceptions (#UD). When debug extensions are not enabled (when the DE flag is clear), these registers are aliased to debug registers DR6 and DR7.

17.2.3 Debug Status Register (DR6)

The debug status register (DR6) reports debug conditions that were sampled at the time the last debug exception was generated (see Figure 17-1). Updates to this register only occur when an exception is generated. The flags in this register show the following information:

- **B0 through B3 (breakpoint condition detected) flags (bits 0 through 3) —** Indicates (when set) that its associated breakpoint condition was met when a debug exception was generated. These flags are set if the condition described for each breakpoint by the LEN*n*, and R/W*n* flags in debug control register DR7 is true. They may or may not be set if the breakpoint is not enabled by the Ln or the Gn flags in register DR7. Therefore on a #DB, a debug handler should check only those B0-B3 bits which correspond to an enabled breakpoint.
- **BD (debug register access detected) flag (bit 13) —** Indicates that the next instruction in the instruction stream accesses one of the debug registers (DR0 through DR7). This flag is enabled when the GD (general detect) flag in debug control register DR7 is set. See Section 17.2.4, "Debug Control Register (DR7)," for further explanation of the purpose of this flag.
- **BS (single step) flag (bit 14) —** Indicates (when set) that the debug exception was triggered by the single-step execution mode (enabled with the TF flag in the EFLAGS register). The single-step mode is the highest-priority debug exception. When the BS flag is set, any of the other debug status bits also may be set.
- **BT (task switch) flag (bit 15) —** Indicates (when set) that the debug exception resulted from a task switch where the T flag (debug trap flag) in the TSS of the target task was set. See Section 7.2.1, "Task-State Segment (TSS)," for the format of a TSS. There is no flag in debug control register DR7 to enable or disable this exception; the T flag of the TSS is the only enabling flag.
- **RTM (restricted transactional memory) flag (bit 16)** — Indicates (when **clear**) that a debug exception (#DB) or breakpoint exception (#BP) occurred inside an RTM region while advanced debugging of RTM transactional regions was enabled (see Section 17.3.3). This bit is set for any other debug exception (including all those that occur when advanced debugging of RTM transactional regions is not enabled). This bit is always 1 if the processor does not support RTM.

Certain debug exceptions may clear bits 0-3. The remaining contents of the DR6 register are never cleared by the processor. To avoid confusion in identifying debug exceptions, debug handlers should clear the register (except bit 16, which they should set) before returning to the interrupted task.

17.2.4 Debug Control Register (DR7)

The debug control register (DR7) enables or disables breakpoints and sets breakpoint conditions (see Figure 17-1). The flags and fields in this register control the following things:

- **L0 through L3 (local breakpoint enable) flags (bits 0, 2, 4, and 6) —** Enables (when set) the breakpoint condition for the associated breakpoint for the current task. When a breakpoint condition is detected and its associated L*n* flag is set, a debug exception is generated. The processor automatically clears these flags on every task switch to avoid unwanted breakpoint conditions in the new task.
- **G0 through G3 (global breakpoint enable) flags (bits 1, 3, 5, and 7) —** Enables (when set) the breakpoint condition for the associated breakpoint for all tasks. When a breakpoint condition is detected and its associated G*n* flag is set, a debug exception is generated. The processor does not clear these flags on a task switch, allowing a breakpoint to be enabled for all tasks.
- **LE and GE (local and global exact breakpoint enable) flags (bits 8, 9) —** This feature is not supported in the P6 family processors, later IA-32 processors, and Intel 64 processors. When set, these flags cause the processor to detect the exact instruction that caused a data breakpoint condition. For backward and forward compatibility with other Intel processors, we recommend that the LE and GE flags be set to 1 if exact breakpoints are required.
- **RTM (restricted transactional memory) flag (bit 11)** — Enables (when set) advanced debugging of RTM transactional regions (see Section 17.3.3). This advanced debugging is enabled only if IA32_DEBUGCTL.RTM is also set.
- **GD (general detect enable) flag (bit 13) —** Enables (when set) debug-register protection, which causes a debug exception to be generated prior to any MOV instruction that accesses a debug register. When such a condition is detected, the BD flag in debug status register DR6 is set prior to generating the exception. This condition is provided to support in-circuit emulators.

  When the emulator needs to access the debug registers, emulator software can set the GD flag to prevent interference from the program currently executing on the processor.

  The processor clears the GD flag upon entering to the debug exception handler, to allow the handler access to the debug registers.
- **R/W0 through R/W3 (read/write) fields (bits 16, 17, 20, 21, 24, 25, 28, and 29) —** Specifies the breakpoint condition for the corresponding breakpoint. The DE (debug extensions) flag in control register CR4 determines how the bits in the R/W*n* fields are interpreted. When the DE flag is set, the processor interprets bits as follows:

  00 — Break on instruction execution only.
  01 — Break on data writes only.
  10 — Break on I/O reads or writes.
  11 — Break on data reads or writes but not instruction fetches.

  When the DE flag is clear, the processor interprets the R/W*n* bits the same as for the Intel386™ and Intel486™ processors, which is as follows:

  00 — Break on instruction execution only.
  01 — Break on data writes only.
  10 — Undefined.
  11 — Break on data reads or writes but not instruction fetches.
- **LEN0 through LEN3 (Length) fields (bits 18, 19, 22, 23, 26, 27, 30, and 31) —** Specify the size of the memory location at the address specified in the corresponding breakpoint address register (DR0 through DR3). These fields are interpreted as follows:

  00 — 1-byte length.
  01 — 2-byte length.
  10 — Undefined (or 8 byte length, see note below).
  11 — 4-byte length.

If the corresponding RW*n* field in register DR7 is 00 (instruction execution), then the LEN*n* field should also be 00. The effect of using other lengths is undefined. See Section 17.2.5, "Breakpoint Field Recognition," below.

NOTES

For Pentium® 4 and Intel® Xeon® processors with a CPUID signature corresponding to family 15 (model 3, 4, and 6), break point conditions permit specifying 8-byte length on data read/write with an of encoding 10B in the LEN*n* field.

Encoding 10B is also supported in processors based on Intel Core microarchitecture or enhanced Intel Core microarchitecture, the respective CPUID signatures corresponding to family 6, model 15, and family 6, DisplayModel value 23 (see CPUID instruction in Chapter 3, "Instruction Set Reference, A-L" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A*). The Encoding 10B is supported in processors based on Intel® Atom™ microarchitecture, with CPUID signature of family 6, DisplayModel value 1CH. The encoding 10B is undefined for other processors.

17.2.5 Breakpoint Field Recognition

Breakpoint address registers (debug registers DR0 through DR3) and the LEN*n* fields for each breakpoint define a range of sequential byte addresses for a data or I/O breakpoint. The LEN*n* fields permit specification of a 1-, 2-, 4- or 8-byte range, beginning at the linear address specified in the corresponding debug register (DR*n*). Two-byte ranges must be aligned on word boundaries; 4-byte ranges must be aligned on doubleword boundaries, 8-byte ranges must be aligned on quadword boundaries. I/O addresses are zero-extended (from 16 to 32 bits, for comparison with the breakpoint address in the selected debug register). These requirements are enforced by the processor; it uses LEN*n* field bits to mask the lower address bits in the debug registers. Unaligned data or I/O breakpoint addresses do not yield valid results.

A data breakpoint for reading or writing data is triggered if any of the bytes participating in an access is within the range defined by a breakpoint address register and its LEN*n* field. Table 17-1 provides an example setup of debug registers and data accesses that would subsequently trap or not trap on the breakpoints.

A data breakpoint for an unaligned operand can be constructed using two breakpoints, where each breakpoint is byte-aligned and the two breakpoints together cover the operand. The breakpoints generate exceptions only for the operand, not for neighboring bytes.

Instruction breakpoint addresses must have a length specification of 1 byte (the LEN*n* field is set to 00). Code breakpoints for other operand sizes are undefined. The processor recognizes an instruction breakpoint address only when it points to the first byte of an instruction. If the instruction has prefixes, the breakpoint address must point to the first prefix.

Table 17-1. Breakpoint Examples

Debug Register Setup			
Debug Register	**R/W*n***	**Breakpoint Address**	**LEN*n***
DR0 DR1 DR2 DR3	R/W0 = 11 (Read/Write) R/W1 = 01 (Write) R/W2 = 11 (Read/Write) R/W3 = 01 (Write)	A0001H A0002H B0002H C0000H	LEN0 = 00 (1 byte) LEN1 = 00 (1 byte) LEN2 = 01) (2 bytes) LEN3 = 11 (4 bytes)
Data Accesses			
Operation		**Address**	**Access Length (In Bytes)**
Data operations that trap - Read or write - Read or write - Write - Write - Read or write - Read or write - Read or write - Write - Write - Write		 A0001H A0001H A0002H A0002H B0001H B0002H B0002H C0000H C0001H C0003H	 1 2 1 2 4 1 2 4 2 1
Data operations that do not trap - Read or write - Read - Read or write - Read or write - Read - Read or write		 A0000H A0002H A0003H B0000H C0000H C0004H	 1 1 4 2 2 4

17.2.6 Debug Registers and Intel® 64 Processors

For Intel 64 architecture processors, debug registers DR0–DR7 are 64 bits. In 16-bit or 32-bit modes (protected mode and compatibility mode), writes to a debug register fill the upper 32 bits with zeros. Reads from a debug register return the lower 32 bits. In 64-bit mode, MOV DRn instructions read or write all 64 bits. Operand-size prefixes are ignored.

In 64-bit mode, the upper 32 bits of DR6 and DR7 are reserved and must be written with zeros. Writing 1 to any of the upper 32 bits results in a #GP(0) exception (see Figure 17-2). All 64 bits of DR0–DR3 are writable by software. However, MOV DRn instructions do not check that addresses written to DR0–DR3 are in the linear-address limits of the processor implementation (address matching is supported only on valid addresses generated by the processor implementation). Break point conditions for 8-byte memory read/writes are supported in all modes.

17.3 DEBUG EXCEPTIONS

The Intel 64 and IA-32 architectures dedicate two interrupt vectors to handling debug exceptions: vector 1 (debug exception, #DB) and vector 3 (breakpoint exception, #BP). The following sections describe how these exceptions are generated and typical exception handler operations.

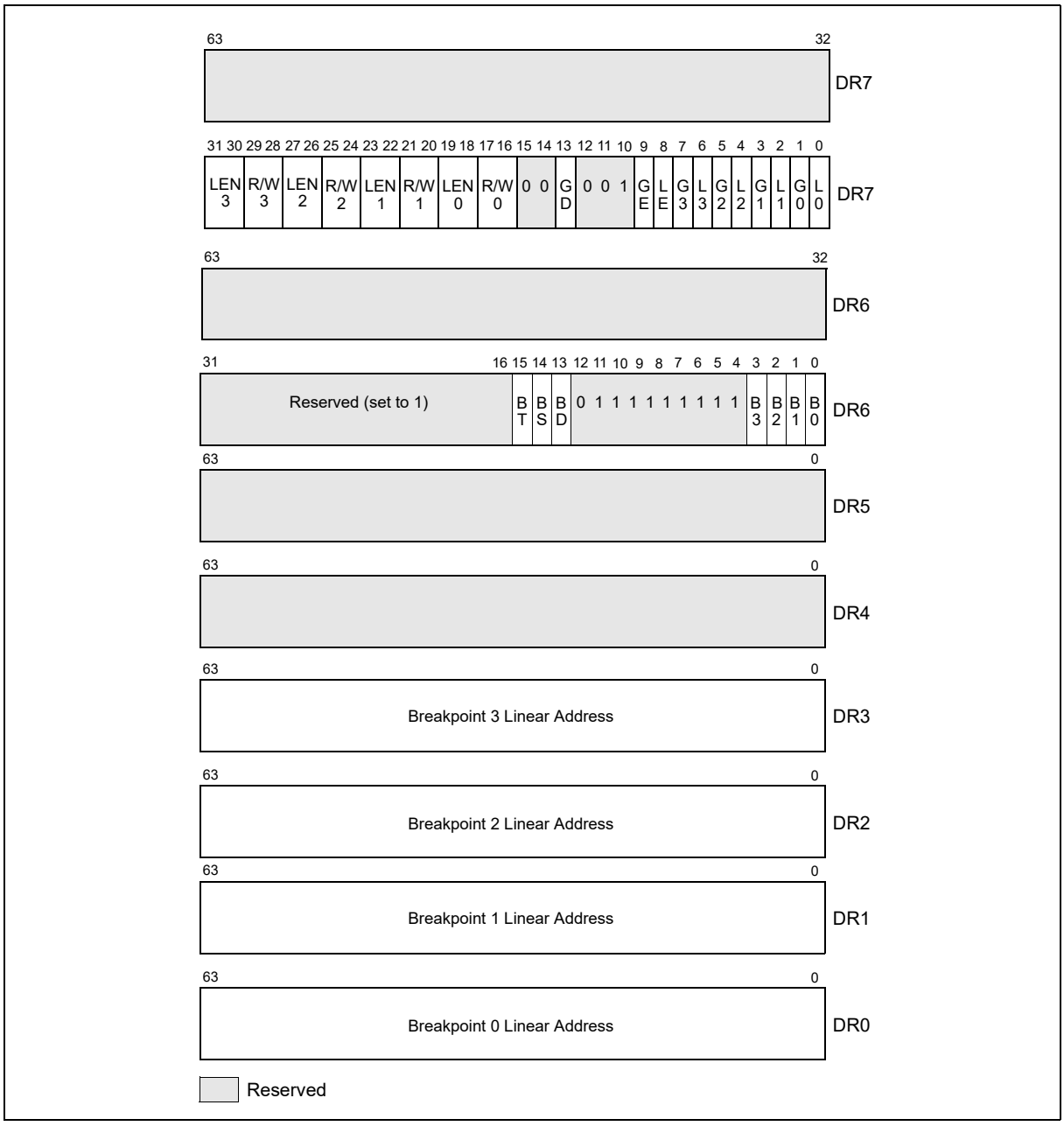


Figure 17-2. DR6/DR7 Layout on Processors Supporting Intel® 64 Architecture

17.3.1 Debug Exception (#DB)—Interrupt Vector 1

The debug-exception handler is usually a debugger program or part of a larger software system. The processor generates a debug exception for any of several conditions. The debugger checks flags in the DR6 and DR7 registers to determine which condition caused the exception and which other conditions might apply. Table 17-2 shows the states of these flags following the generation of each kind of breakpoint condition.

Instruction-breakpoint and general-detect condition (see Section 17.3.1.3, "General-Detect Exception Condition") result in faults; other debug-exception conditions result in traps. The debug exception may report one or both at one time. The following sections describe each class of debug exception.

See also: Chapter 6, "Interrupt 1—Debug Exception (#DB)," in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*.

Table 17-2. Debug Exception Conditions

Debug or Breakpoint Condition	DR6 Flags Tested	DR7 Flags Tested	Exception Class
Single-step trap	BS = 1		Trap
Instruction breakpoint, at addresses defined by DR*n* and LEN*n*	B*n* = 1 and (G*n* or L*n* = 1)	R/Wn = 0	Fault
Data write breakpoint, at addresses defined by DR*n* and LEN*n*	B*n* = 1 and (G*n* or L*n* = 1)	R/Wn = 1	Trap
I/O read or write breakpoint, at addresses defined by DR*n* and LEN*n*	B*n* = 1 and (G*n* or L*n* = 1)	R/Wn = 2	Trap
Data read or write (but not instruction fetches), at addresses defined by DR*n* and LEN*n*	B*n* = 1 and (G*n* or L*n* = 1)	R/Wn = 3	Trap
General detect fault, resulting from an attempt to modify debug registers (usually in conjunction with in-circuit emulation)	BD = 1		Fault
Task switch	BT = 1		Trap

17.3.1.1 Instruction-Breakpoint Exception Condition

The processor reports an instruction breakpoint when it attempts to execute an instruction at an address specified in a breakpoint-address register (DR0 through DR3) that has been set up to detect instruction execution (R/W flag is set to 0). Upon reporting the instruction breakpoint, the processor generates a fault-class, debug exception (#DB) before it executes the target instruction for the breakpoint.

Instruction breakpoints are the highest priority debug exceptions. They are serviced before any other exceptions detected during the decoding or execution of an instruction. However, if a code instruction breakpoint is placed on an instruction located immediately after a POP SS/MOV SS instruction, the breakpoint may not be triggered. In most situations, POP SS/MOV SS will inhibit such interrupts (see "MOV—Move" and "POP—Pop a Value from the Stack" in Chapter 4 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2B*).

Because the debug exception for an instruction breakpoint is generated before the instruction is executed, if the instruction breakpoint is not removed by the exception handler; the processor will detect the instruction breakpoint again when the instruction is restarted and generate another debug exception. To prevent looping on an instruction breakpoint, the Intel 64 and IA-32 architectures provide the RF flag (resume flag) in the EFLAGS register (see Section 2.3, "System Flags and Fields in the EFLAGS Register," in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*). When the RF flag is set, the processor ignores instruction breakpoints.

All Intel 64 and IA-32 processors manage the RF flag as follows. The RF Flag is cleared at the start of the instruction after the check for code breakpoint, CS limit violation and FP exceptions. Task Switches and IRETD/IRETQ instructions transfer the RF image from the TSS/stack to the EFLAGS register.

When calling an event handler, Intel 64 and IA-32 processors establish the value of the RF flag in the EFLAGS image pushed on the stack:

- For any fault-class exception except a debug exception generated in response to an instruction breakpoint, the value pushed for RF is 1.
- For any interrupt arriving after any iteration of a repeated string instruction but the last iteration, the value pushed for RF is 1.
- For any trap-class exception generated by any iteration of a repeated string instruction but the last iteration, the value pushed for RF is 1.
- For other cases, the value pushed for RF is the value that was in EFLAG.RF at the time the event handler was called. This includes:
  - — Debug exceptions generated in response to instruction breakpoints

— Hardware-generated interrupts arriving between instructions (including those arriving after the last iteration of a repeated string instruction)

— Trap-class exceptions generated after an instruction completes (including those generated after the last iteration of a repeated string instruction)

— Software-generated interrupts (RF is pushed as 0, since it was cleared at the start of the software interrupt)

As noted above, the processor does not set the RF flag prior to calling the debug exception handler for debug exceptions resulting from instruction breakpoints. The debug exception handler can prevent recurrence of the instruction breakpoint by setting the RF flag in the EFLAGS image on the stack. If the RF flag in the EFLAGS image is set when the processor returns from the exception handler, it is copied into the RF flag in the EFLAGS register by IRETD/IRETQ or a task switch that causes the return. The processor then ignores instruction breakpoints for the duration of the next instruction. (Note that the POPF, POPFD, and IRET instructions do not transfer the RF image into the EFLAGS register.) Setting the RF flag does not prevent other types of debug-exception conditions (such as, I/O or data breakpoints) from being detected, nor does it prevent non-debug exceptions from being generated.

For the Pentium processor, when an instruction breakpoint coincides with another fault-type exception (such as a page fault), the processor may generate one spurious debug exception after the second exception has been handled, even though the debug exception handler set the RF flag in the EFLAGS image. To prevent a spurious exception with Pentium processors, all fault-class exception handlers should set the RF flag in the EFLAGS image.

17.3.1.2 Data Memory and I/O Breakpoint Exception Conditions

Data memory and I/O breakpoints are reported when the processor attempts to access a memory or I/O address specified in a breakpoint-address register (DR0 through DR3) that has been set up to detect data or I/O accesses (R/W flag is set to 1, 2, or 3). The processor generates the exception after it executes the instruction that made the access, so these breakpoint condition causes a trap-class exception to be generated.

Because data breakpoints are traps, an instruction that writes memory overwrites the original data before the debug exception generated by a data breakpoint is generated. If a debugger needs to save the contents of a write breakpoint location, it should save the original contents before setting the breakpoint. The handler can report the saved value after the breakpoint is triggered. The address in the debug registers can be used to locate the new value stored by the instruction that triggered the breakpoint.

If a data breakpoint is detected during an iteration of a string instruction executed with fast-string operation (see Section 7.3.9.3 of *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*), delivery of the resulting debug exception may be delayed until completion of the corresponding group of iterations.

Intel486 and later processors ignore the GE and LE flags in DR7. In Intel386 processors, exact data breakpoint matching does not occur unless it is enabled by setting the LE and/or the GE flags.

For repeated INS and OUTS instructions that generate an I/O-breakpoint debug exception, the processor generates the exception after the completion of the first iteration. Repeated INS and OUTS instructions generate a data-breakpoint debug exception after the iteration in which the memory address breakpoint location is accessed.

17.3.1.3 General-Detect Exception Condition

When the GD flag in DR7 is set, the general-detect debug exception occurs when a program attempts to access any of the debug registers (DR0 through DR7) at the same time they are being used by another application, such as an emulator or debugger. This protection feature guarantees full control over the debug registers when required. The debug exception handler can detect this condition by checking the state of the BD flag in the DR6 register. The processor generates the exception before it executes the MOV instruction that accesses a debug register, which causes a fault-class exception to be generated.

17.3.1.4 Single-Step Exception Condition

The processor generates a single-step debug exception if (while an instruction is being executed) it detects that the TF flag in the EFLAGS register is set. The exception is a trap-class exception, because the exception is generated after the instruction is executed. The processor will not generate this exception after the instruction that sets the TF flag. For example, if the POPF instruction is used to set the TF flag, a single-step trap does not occur until after the instruction that follows the POPF instruction.

The processor clears the TF flag before calling the exception handler. If the TF flag was set in a TSS at the time of a task switch, the exception occurs after the first instruction is executed in the new task.

The TF flag normally is not cleared by privilege changes inside a task. The INT *n* and INTO instructions, however, do clear this flag. Therefore, software debuggers that single-step code must recognize and emulate INT *n* or INTO instructions rather than executing them directly. To maintain protection, the operating system should check the CPL after any single-step trap to see if single stepping should continue at the current privilege level.

The interrupt priorities guarantee that, if an external interrupt occurs, single stepping stops. When both an external interrupt and a single-step interrupt occur together, the single-step interrupt is processed first. This operation clears the TF flag. After saving the return address or switching tasks, the external interrupt input is examined before the first instruction of the single-step handler executes. If the external interrupt is still pending, then it is serviced. The external interrupt handler does not run in single-step mode. To single step an interrupt handler, single step an INT *n* instruction that calls the interrupt handler.

17.3.1.5 Task-Switch Exception Condition

The processor generates a debug exception after a task switch if the T flag of the new task's TSS is set. This exception is generated after program control has passed to the new task, and prior to the execution of the first instruction of that task. The exception handler can detect this condition by examining the BT flag of the DR6 register.

If entry 1 (#DB) in the IDT is a task gate, the T bit of the corresponding TSS should not be set. Failure to observe this rule will put the processor in a loop.

17.3.2 Breakpoint Exception (#BP)—Interrupt Vector 3

The breakpoint exception (interrupt 3) is caused by execution of an INT 3 instruction. See Chapter 6, "Interrupt 3—Breakpoint Exception (#BP)." Debuggers use break exceptions in the same way that they use the breakpoint registers; that is, as a mechanism for suspending program execution to examine registers and memory locations. With earlier IA-32 processors, breakpoint exceptions are used extensively for setting instruction breakpoints.

With the Intel386 and later IA-32 processors, it is more convenient to set breakpoints with the breakpoint-address registers (DR0 through DR3). However, the breakpoint exception still is useful for breakpointing debuggers, because a breakpoint exception can call a separate exception handler. The breakpoint exception is also useful when it is necessary to set more breakpoints than there are debug registers or when breakpoints are being placed in the source code of a program under development.

17.3.3 Debug Exceptions, Breakpoint Exceptions, and Restricted Transactional Memory (RTM)

Chapter 16, "Programming with Intel® Transactional Synchronization Extensions," of *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1* describes Restricted Transactional Memory (RTM). This is an instruction-set interface that allows software to identify **transactional regions** (or critical sections) using the XBEGIN and XEND instructions.

Execution of an RTM transactional region begins with an XBEGIN instruction. If execution of the region successfully reaches an XEND instruction, the processor ensures that all memory operations performed within the region appear to have occurred instantaneously when viewed from other logical processors. Execution of an RTM transaction region does not succeed if the processor cannot commit the updates atomically. When this happens, the processor rolls back the execution, a process referred to as a **transactional abort**. In this case, the processor discards all updates performed in the region, restores architectural state to appear as if the execution had not occurred, and resumes execution at a fallback instruction address that was specified with the XBEGIN instruction.

If debug exception (#DB) or breakpoint exception (#BP) occurs within an RTM transaction region, a transactional abort occurs, the processor sets EAX[4], and no exception is delivered.

Software can enable **advanced debugging of RTM transactional regions** by setting DR7.RTM[bit 11] and IA32_DEBUGCTL.RTM[bit 15]. If these bits are both set, the transactional abort caused by a #DB or #BP within an RTM transaction region does **not** resume execution at the fallback instruction address specified with the XBEGIN instruction that begin the region. Instead, execution is resumed at that XBEGIN instruction, and a #DB is delivered.

(A #DB is delivered even if the transactional abort was caused by a #BP.) Such a #DB will clear DR6.RTM[bit 16] (all other debug exceptions set DR6[16]).

17.4 LAST BRANCH, INTERRUPT, AND EXCEPTION RECORDING OVERVIEW

P6 family processors introduced the ability to set breakpoints on taken branches, interrupts, and exceptions, and to single-step from one branch to the next. This capability has been modified and extended in the Pentium 4, Intel Xeon, Pentium M, Intel® Core™ Solo, Intel® Core™ Duo, Intel® Core™2 Duo, Intel® Core™ i7 and Intel® Atom™ processors to allow logging of branch trace messages in a branch trace store (BTS) buffer in memory.

See the following sections for processor specific implementation of last branch, interrupt and exception recording:

- Section 17.5, "Last Branch, Interrupt, and Exception Recording (Intel® Core™ 2 Duo and Intel® Atom™ Processors)"
- Section 17.6, "Last Branch, Call Stack, Interrupt, and Exception Recording for Processors based on Goldmont Microarchitecture"
- Section 17.9, "Last Branch, Interrupt, and Exception Recording for Processors based on Intel® Microarchitecture code name Nehalem"
- Section 17.10, "Last Branch, Interrupt, and Exception Recording for Processors based on Intel® Microarchitecture code name Sandy Bridge"
- Section 17.11, "Last Branch, Call Stack, Interrupt, and Exception Recording for Processors based on Haswell Microarchitecture"
- Section 17.12, "Last Branch, Call Stack, Interrupt, and Exception Recording for Processors based on Skylake Microarchitecture"
- Section 17.14, "Last Branch, Interrupt, and Exception Recording (Intel® Core™ Solo and Intel® Core™ Duo Processors)"
- Section 17.15, "Last Branch, Interrupt, and Exception Recording (Pentium M Processors)"
- Section 17.16, "Last Branch, Interrupt, and Exception Recording (P6 Family Processors)"

The following subsections of Section 17.4 describe common features of profiling branches. These features are generally enabled using the IA32_DEBUGCTL MSR (older processor may have implemented a subset or model-specific features, see definitions of MSR_DEBUGCTLA, MSR_DEBUGCTLB, MSR_DEBUGCTL).

17.4.1 IA32_DEBUGCTL MSR

The **IA32_DEBUGCTL** MSR provides bit field controls to enable debug trace interrupts, debug trace stores, trace messages enable, single stepping on branches, last branch record recording, and to control freezing of LBR stack or performance counters on a PMI request. IA32_DEBUGCTL MSR is located at register address 01D9H.

See Figure 17-3 for the MSR layout and the bullets below for a description of the flags:

- **LBR (last branch/interrupt/exception) flag (bit 0) —** When set, the processor records a running trace of the most recent branches, interrupts, and/or exceptions taken by the processor (prior to a debug exception being generated) in the last branch record (LBR) stack. For more information, see the Section 17.5.1, "LBR Stack" (Intel® Core™2 Duo and Intel® Atom™ Processor Family) and Section 17.9.1, "LBR Stack" (processors based on Intel® Microarchitecture code name Nehalem).
- **BTF (single-step on branches) flag (bit 1) —** When set, the processor treats the TF flag in the EFLAGS register as a "single-step on branches" flag rather than a "single-step on instructions" flag. This mechanism allows single-stepping the processor on taken branches. See Section 17.4.3, "Single-Stepping on Branches," for more information about the BTF flag.
- **TR (trace message enable) flag (bit 6) —** When set, branch trace messages are enabled. When the processor detects a taken branch, interrupt, or exception; it sends the branch record out on the system bus as a branch trace message (BTM). See Section 17.4.4, "Branch Trace Messages," for more information about the TR flag.

- **BTS (branch trace store) flag (bit 7) —** When set, the flag enables BTS facilities to log BTMs to a memory-resident BTS buffer that is part of the DS save area. See Section 17.4.9, "BTS and DS Save Area."
- **BTINT (branch trace interrupt) flag (bit 8) —** When set, the BTS facilities generate an interrupt when the BTS buffer is full. When clear, BTMs are logged to the BTS buffer in a circular fashion. See Section 17.4.5, "Branch Trace Store (BTS)," for a description of this mechanism.

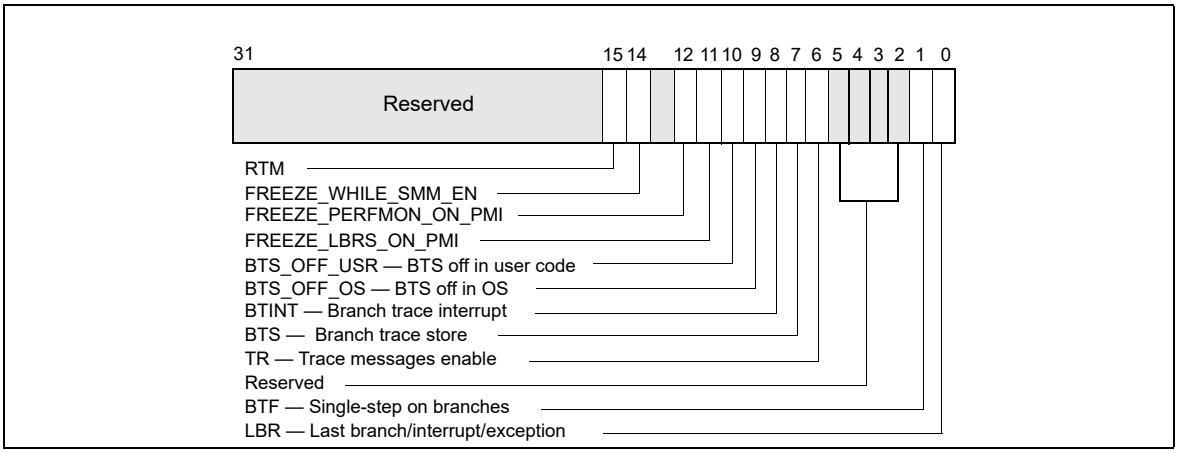


Figure 17-3. IA32_DEBUGCTL MSR for Processors based on Intel Core microarchitecture

- **BTS_OFF_OS (branch trace off in privileged code) flag (bit 9) —** When set, BTS or BTM is skipped if CPL is 0. See Section 17.13.2.
- **BTS_OFF_USR (branch trace off in user code) flag (bit 10) —** When set, BTS or BTM is skipped if CPL is greater than 0. See Section 17.13.2.
- **FREEZE_LBRS_ON_PMI flag (bit 11) —** When set, the LBR stack is frozen on a hardware PMI request (e.g. when a counter overflows and is configured to trigger PMI). See Section 17.4.7 for details.
- **FREEZE_PERFMON_ON_PMI flag (bit 12) —** When set, the performance counters (IA32_PMCx and IA32_FIXED_CTRx) are frozen on a PMI request. See Section 17.4.7 for details.
- **FREEZE_WHILE_SMM_EN (bit 14) —** If this bit is set, upon the delivery of an SMI, the processor will clear all the enable bits of IA32_PERF_GLOBAL_CTRL, save a copy of the content of IA32_DEBUGCTL and disable LBR, BTF, TR, and BTS fields of IA32_DEBUGCTL before transferring control to the SMI handler. Subsequently, the enable bits of IA32_PERF_GLOBAL_CTRL will be set to 1, the saved copy of IA32_DEBUGCTL prior to SMI delivery will be restored, after the SMI handler issues RSM to complete its service. Note that system software must check if the processor supports the IA32_DEBUGCTL.FREEZE_WHILE_SMM_EN control bit. IA32_DEBUGCTL.FREEZE_WHILE_SMM_EN is supported if IA32_PERF_CAPABILITIES.FREEZE_WHILE_SMM[Bit 12] is reporting 1. See Section 18.8 for details of detecting the presence of IA32_PERF_CAPABILITIES MSR.
- **RTM (bit 15)** — If this bit is set, advanced debugging of RTM transactional regions is enabled if DR7.RTM is also set. See Section 17.3.3.

17.4.2 Monitoring Branches, Exceptions, and Interrupts

When the LBR flag (bit 0) in the IA32_DEBUGCTL MSR is set, the processor automatically begins recording branch records for taken branches, interrupts, and exceptions (except for debug exceptions) in the LBR stack MSRs.

When the processor generates a debug exception (#DB), it automatically clears the LBR flag before executing the exception handler. This action does not clear previously stored LBR stack MSRs.

A debugger can use the linear addresses in the LBR stack to re-set breakpoints in the breakpoint address registers (DR0 through DR3). This allows a backward trace from the manifestation of a particular bug toward its source.

On some processors, if the LBR flag is cleared and TR flag in the IA32_DEBUGCTL MSR remains set, the processor will continue to update LBR stack MSRs. This is because those processors use the entries in the LBR stack in the process of generating BTM/BTS records. A #DB does not automatically clear the TR flag.

17.4.3 Single-Stepping on Branches

When software sets both the BTF flag (bit 1) in the IA32_DEBUGCTL MSR and the TF flag in the EFLAGS register, the processor generates a single-step debug exception only after instructions that cause a branch.[1] This mechanism allows a debugger to single-step on control transfers caused by branches. This "branch single stepping" helps isolate a bug to a particular block of code before instruction single-stepping further narrows the search. The processor clears the BTF flag when it generates a debug exception. The debugger must set the BTF flag before resuming program execution to continue single-stepping on branches.

17.4.4 Branch Trace Messages

Setting the TR flag (bit 6) in the IA32_DEBUGCTL MSR enables branch trace messages (BTMs). Thereafter, when the processor detects a branch, exception, or interrupt, it sends a branch record out on the system bus as a BTM. A debugging device that is monitoring the system bus can read these messages and synchronize operations with taken branch, interrupt, and exception events.

When interrupts or exceptions occur in conjunction with a taken branch, additional BTMs are sent out on the bus, as described in Section 17.4.2, "Monitoring Branches, Exceptions, and Interrupts."

For P6 processor family, Pentium M processor family, processors based on Intel Core microarchitecture, TR and LBR bits can not be set at the same time due to hardware limitation. The content of LBR stack is undefined when TR is set.

For processors with Intel NetBurst microarchitecture, Intel Atom processors, and Intel Core and related Intel Xeon processors both starting with the Nehalem microarchitecture, the processor can collect branch records in the LBR stack and at the same time send/store BTMs when both the TR and LBR flags are set in the IA32_DEBUGCTL MSR (or the equivalent MSR_DEBUGCTLA, MSR_DEBUGCTLB).

The following exception applies:

- BTM may not be observable on Intel Atom processor families that do not provide an externally visible system bus (i.e., processors based on the Silvermont microarchitecture or later).

17.4.4.1 Branch Trace Message Visibility

Branch trace message (BTM) visibility is implementation specific and limited to systems with a front side bus (FSB). BTMs may not be visible to newer system link interfaces or a system bus that deviates from a traditional FSB.

17.4.5 Branch Trace Store (BTS)

A trace of taken branches, interrupts, and exceptions is useful for debugging code by providing a method of determining the decision path taken to reach a particular code location. The LBR flag (bit 0) of IA32_DEBUGCTL provides a mechanism for capturing records of taken branches, interrupts, and exceptions and saving them in the last branch record (LBR) stack MSRs, setting the TR flag for sending them out onto the system bus as BTMs. The branch trace store (BTS) mechanism provides the additional capability of saving the branch records in a memory-resident BTS buffer, which is part of the DS save area. The BTS buffer can be configured to be circular so that the most recent branch records are always available or it can be configured to generate an interrupt when the buffer is nearly full so that all the branch records can be saved. The BTINT flag (bit 8) can be used to enable the generation of interrupt when the BTS buffer is full. See Section 17.4.9.2, "Setting Up the DS Save Area." for additional details.

1. Executions of CALL, IRET, and JMP that cause task switches never cause single-step debug exceptions (regardless of the value of the BTF flag). A debugger desiring debug exceptions on switches to a task should set the T flag (debug trap flag) in the TSS of that task. See Section 7.2.1, "Task-State Segment (TSS)."

Setting this flag (BTS) alone can greatly reduce the performance of the processor. CPL-qualified branch trace storing mechanism can help mitigate the performance impact of sending/logging branch trace messages.

17.4.6 CPL-Qualified Branch Trace Mechanism

CPL-qualified branch trace mechanism is available to a subset of Intel 64 and IA-32 processors that support the branch trace storing mechanism. The processor supports the CPL-qualified branch trace mechanism if CPUID.01H:ECX[bit 4] = 1.

The CPL-qualified branch trace mechanism is described in Section 17.4.9.4. System software can selectively specify CPL qualification to not send/store Branch Trace Messages associated with a specified privilege level. Two bit fields, BTS_OFF_USR (bit 10) and BTS_OFF_OS (bit 9), are provided in the debug control register to specify the CPL of BTMs that will not be logged in the BTS buffer or sent on the bus.

17.4.7 Freezing LBR and Performance Counters on PMI

Many issues may generate a performance monitoring interrupt (PMI); a PMI service handler will need to determine cause to handle the situation. Two capabilities that allow a PMI service routine to improve branch tracing and performance monitoring are available for processors supporting architectural performance monitoring version 2 or greater (i.e. CPUID.0AH:EAX[7:0] > 1). These capabilities provides the following interface in IA32_DEBUGCTL to reduce runtime overhead of PMI servicing, profiler-contributed skew effects on analysis or counter metrics:

- **Freezing LBRs on PMI (bit 11)**— Allows the PMI service routine to ensure the content in the LBR stack are associated with the target workload and not polluted by the branch flows of handling the PMI. Depending on the version ID enumerated by CPUID.0AH:EAX.ArchPerfMonVerID[bits 7:0], two flavors are supported:
  - Legacy Freeze_LBR_on_PMI is supported for ArchPerfMonVerID <= 3 and ArchPerfMonVerID >1. If IA32_DEBUGCTL.Freeze_LBR_On_PMI = 1, the LBR is frozen on the overflowed condition of the buffer area, the processor clears the LBR bit (bit 0) in IA32_DEBUGCTL. Software must then re-enable IA32_DEBUGCTL.LBR to resume recording branches. When using this feature, software should be careful about writes to IA32_DEBUGCTL to avoid re-enabling LBRs by accident if they were just disabled.
  - Streamlined Freeze_LBR_on_PMI is supported for ArchPerfMonVerID >= 4. If IA32_DEBUGCTL.Freeze_LBR_On_PMI = 1, the processor behaves as follows:
    - sets IA32_PERF_GLOBAL_STATUS.LBR_Frz =1 to disable recording, but does not change the LBR bit (bit 0) in IA32_DEBUGCTL. The LBRs are frozen on the overflowed condition of the buffer area.
- **Freezing PMCs on PMI** (bit 12) — Allows the PMI service routine to ensure the content in the performance counters are associated with the target workload and not polluted by the PMI and activities within the PMI service routine. Depending on the version ID enumerated by CPUID.0AH:EAX.ArchPerfMonVerID[bits 7:0], two flavors are supported:
  - Legacy Freeze_Perfmon_on_PMI is supported for ArchPerfMonVerID <= 3 and ArchPerfMonVerID >1. If IA32_DEBUGCTL.Freeze_Perfmon_On_PMI = 1, the performance counters are frozen on the counter overflowed condition when the processor clears the IA32_PERF_GLOBAL_CTRL MSR (see Figure 18-3). The PMCs affected include both general-purpose counters and fixed-function counters (see Section 18.6.2.1, "Fixed-function Performance Counters"). Software must re-enable counts by writing 1s to the corresponding enable bits in IA32_PERF_GLOBAL_CTRL before leaving a PMI service routine to continue counter operation.
  - Streamlined Freeze_Perfmon_on_PMI is supported for ArchPerfMonVerID >= 4. The processor behaves as follows:
    - sets IA32_PERF_GLOBAL_STATUS.CTR_Frz =1 to disable counting on a counter overflow condition, but does not change the IA32_PERF_GLOBAL_CTRL MSR.

Freezing LBRs and PMCs on PMIs (both legacy and streamlined operation) occur when one of the following applies:

- A performance counter had an overflow and was programmed to signal a PMI in case of an overflow.
  - For the general-purpose counters; enabling PMI is done by setting bit 20 of the IA32_PERFEVTSELx register.

— For the fixed-function counters; enabling PMI is done by setting the 3rd bit in the corresponding 4-bit control field of the MSR_PERF_FIXED_CTR_CTRL register (see Figure 18-1) or IA32_FIXED_CTR_CTRL MSR (see Figure 18-2).

- The PEBS buffer is almost full and reaches the interrupt threshold.
- The BTS buffer is almost full and reaches the interrupt threshold.

Table 17-3 compares the interaction of the processor with the PMI handler using the legacy versus streamlined Freeza_Perfmon_On_PMI interface.

Table 17-3. Legacy and Streamlined Operation with Freeze_Perfmon_On_PMI = 1, Counter Overflowed

Legacy Freeze_Perfmon_On_PMI	Streamlined Freeze_Perfmon_On_PMI	Comment
Processor freezes the counters on overflow	Processor freezes the counters on overflow	Unchanged
Processor clears IA32_PERF_GLOBAL_CTRL	Processor set IA32_PERF_GLOBAL_STATUS.CTR_FTZ	
Handler reads IA32_PERF_GLOBAL_STATUS (0x38E) to examine which counter(s) overflowed	**mask** = RDMSR(0x38E)	Similar
Handler services the PMI	Handler services the PMI	Unchanged
Handler writes 1s to IA32_PERF_GLOBAL_OVF_CTL (0x390)	Handler writes **mask** into IA32_PERF_GLOBAL_OVF_RESET (0x390)	
Processor clears IA32_PERF_GLOBAL_STATUS	Processor clears IA32_PERF_GLOBAL_STATUS	Unchanged
Handler re-enables IA32_PERF_GLOBAL_CTRL	None	Reduced software overhead

17.4.8 LBR Stack

The last branch record stack and top-of-stack (TOS) pointer MSRs are supported across Intel 64 and IA-32 processor families. However, the number of MSRs in the LBR stack and the valid range of TOS pointer value can vary between different processor families. Table 17-4 lists the LBR stack size and TOS pointer range for several processor families according to the CPUID signatures of DisplayFamily_DisplayModel encoding (see CPUID instruction in Chapter 3 of *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A*).

Table 17-4. LBR Stack Size and TOS Pointer Range

DisplayFamily_DisplayModel	Size of LBR Stack	Component of an LBR Entry	Range of TOS Pointer
06_5CH, 06_5FH	32	FROM_IP, TO_IP	0 to 31
06_4EH, 06_5EH, 06_8EH, 06_9EH, 06_55H, 06_66H, 06_7AH	32	FROM_IP, TO_IP, LBR_INFO[1]	0 to 31
06_3DH, 06_47H, 06_4FH, 06_56H, 06_3CH, 06_45H, 06_46H, 06_3FH, 06_2AH, 06_2DH, 06_3AH, 06_3EH, 06_1AH, 06_1EH, 06_1FH, 06_2EH, 06_25H, 06_2CH, 06_2FH	16	FROM_IP, TO_IP	0 to 15
06_17H, 06_1DH, 06_0FH	4	FROM_IP, TO_IP	0 to 3
06_37H, 06_4AH, 06_4CH, 06_4DH, 06_5AH, 06_5DH, 06_1CH, 06_26H, 06_27H, 06_35H, 06_36H	8	FROM_IP, TO_IP	0 to 7

NOTES:

1. See Section 17.12.

The last branch recording mechanism tracks not only branch instructions (like JMP, Jcc, LOOP and CALL instructions), but also other operations that cause a change in the instruction pointer (like external interrupts, traps and faults). The branch recording mechanisms generally employs a set of MSRs, referred to as last branch record (LBR) stack. The size and exact locations of the LBR stack are generally model-specific (see Chapter 2, "Model-Specific Registers (MSRs)" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 4* for model-specific MSR addresses).

- **Last Branch Record (LBR) Stack —** The LBR consists of N pairs of MSRs (N is listed in the LBR stack size column of Table 17-4) that store source and destination address of recent branches (see Figure 17-3):
  - MSR_LASTBRANCH_0_FROM_IP (address is model specific) through the next consecutive (N-1) MSR address store source addresses.
  - MSR_LASTBRANCH_0_TO_IP (address is model specific) through the next consecutive (N-1) MSR address store destination addresses.
- **Last Branch Record Top-of-Stack (TOS) Pointer** — The lowest significant M bits of the TOS Pointer MSR (MSR_LASTBRANCH_TOS, address is model specific) contains an M-bit pointer to the MSR in the LBR stack that contains the most recent branch, interrupt, or exception recorded. The valid range of the M-bit POS pointer is given in Table 17-4.

17.4.8.1 LBR Stack and Intel® 64 Processors

LBR MSRs are 64-bits. In 64-bit mode, last branch records store the full address. Outside of 64-bit mode, the upper 32-bits of branch addresses will be stored as 0.

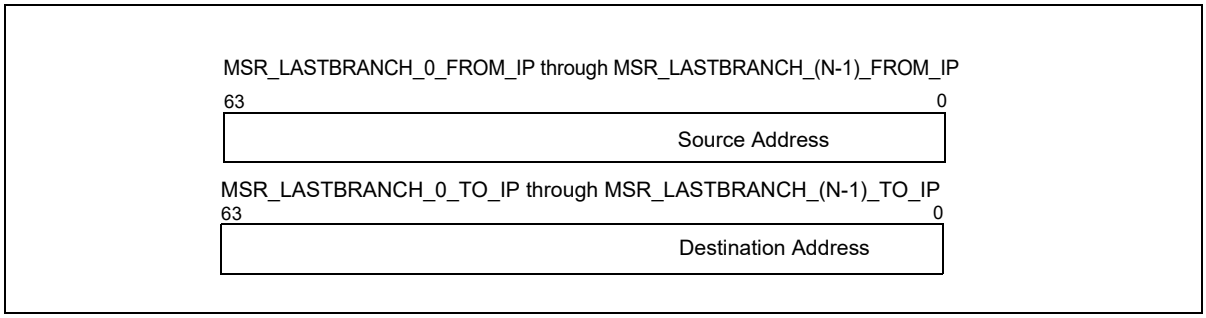


Figure 17-4. 64-bit Address Layout of LBR MSR

Software should query an architectural MSR IA32_PERF_CAPABILITIES[5:0] about the format of the address that is stored in the LBR stack. Four formats are defined by the following encoding:

- **000000B (32-bit record format) —** Stores 32-bit offset in current CS of respective source/destination,
- **000001B (64-bit LIP record format) —** Stores 64-bit linear address of respective source/destination,
- **000010B (64-bit EIP record format) —** Stores 64-bit offset (effective address) of respective source/destination.
- **000011B (64-bit EIP record format) and Flags** — Stores 64-bit offset (effective address) of respective source/destination. Misprediction info is reported in the upper bit of 'FROM' registers in the LBR stack. See LBR stack details below for flag support and definition.
- **000100B (64-bit EIP record format), Flags and TSX** — Stores 64-bit offset (effective address) of respective source/destination. Misprediction and TSX info are reported in the upper bits of 'FROM' registers in the LBR stack.
- **000101B (64-bit EIP record format), Flags, TSX, LBR_INFO** — Stores 64-bit offset (effective address) of respective source/destination. Misprediction, TSX, and elapsed cycles since the last LBR update are reported in the LBR_INFO MSR stack.
- **000110B (64-bit LIP record format), Flags, Cycles** — Stores 64-bit linear address (CS.Base + effective address) of respective source/destination. Misprediction info is reported in the upper bits of

'FROM' registers in the LBR stack. Elapsed cycles since the last LBR update are reported in the upper 16 bits of the 'TO' registers in the LBR stack (see Section 17.6).

— **000111B (64-bit LIP record format), Flags, TSX, LBR_INFO** — Stores 64-bit linear address (CS.Base + effective address) of respective source/destination. Misprediction, TSX, and elapsed cycles since the last LBR update are reported in the LBR_INFO MSR stack.

Processor's support for the architectural MSR IA32_PERF_CAPABILITIES is provided by CPUID.01H:ECX[PERF_CAPAB_MSR] (bit 15).

17.4.8.2 LBR Stack and IA-32 Processors

The LBR MSRs in IA-32 processors introduced prior to Intel 64 architecture store the 32-bit "To Linear Address" and "From Linear Address" using the high and low half of each 64-bit MSR.

17.4.8.3 Last Exception Records and Intel 64 Architecture

Intel 64 and IA-32 processors also provide MSRs that store the branch record for the last branch taken prior to an exception or an interrupt. The location of the last exception record (LER) MSRs are model specific. The MSRs that store last exception records are 64-bits. If IA-32e mode is disabled, only the lower 32-bits of the address is recorded. If IA-32e mode is enabled, the processor writes 64-bit values into the MSR. In 64-bit mode, last exception records store 64-bit addresses; in compatibility mode, the upper 32-bits of last exception records are cleared.

17.4.9 BTS and DS Save Area

The **Debug store (DS)** feature flag (bit 21), returned by CPUID.1:EDX[21] indicates that the processor provides the debug store (DS) mechanism. The DS mechanism allows:

- BTMs to be stored in a memory-resident BTS buffer. See Section 17.4.5, "Branch Trace Store (BTS)."
- Processor event-based sampling (PEBS) also uses the DS save area provided by debug store mechanism. The capability of PEBS varies across different microarchitectures. See Section 18.6.2.4, "Processor Event Based Sampling (PEBS)," and the relevant PEBS sub-sections across the core PMU sections in Chapter 18, "Performance Monitoring."

When CPUID.1:EDX[21] is set:

- The BTS_UNAVAILABLE and PEBS_UNAVAILABLE flags in the IA32_MISC_ENABLE MSR indicate (when clear) the availability of the BTS and PEBS facilities, including the ability to set the BTS and BTINT bits in the appropriate DEBUGCTL MSR.
- The IA32_DS_AREA MSR exists and points to the DS save area.

The debug store (DS) save area is a software-designated area of memory that is used to collect the following two types of information:

- **Branch records —** When the BTS flag in the IA32_DEBUGCTL MSR is set, a branch record is stored in the BTS buffer in the DS save area whenever a taken branch, interrupt, or exception is detected.
- **PEBS records —** When a performance counter is configured for PEBS, a PEBS record is stored in the PEBS buffer in the DS save area after the counter overflow occurs. This record contains the architectural state of the processor (state of the 8 general purpose registers, EIP register, and EFLAGS register) at the next occurrence of the PEBS event that caused the counter to overflow. When the state information has been logged, the counter is automatically reset to a specified value, and event counting begins again. The content layout of a PEBS record varies across different implementations that support PEBS. See Section 18.6.2.4.2 for details of enumerating PEBS record format.

NOTES

Prior to processors based on the Goldmont microarchitecture, PEBS facility only supports a subset of implementation-specific precise events. See Section 18.5.3.1 for a PEBS enhancement that can generate records for both precise and non-precise events.

The DS save area and recording mechanism are disabled on INIT, processor Reset or transition to system-management mode (SMM) or IA-32e mode. It is similarly disabled on the generation of a machine-check exception on 45nm and 32nm Intel Atom processors and on processors with Netburst or Intel Core microarchitecture.

The BTS and PEBS facilities may not be available on all processors. The availability of these facilities is indicated by the BTS_UNAVAILABLE and PEBS_UNAVAILABLE flags, respectively, in the IA32_MISC_ENABLE MSR (see Chapter 2, "Model-Specific Registers (MSRs)" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 4*).

The DS save area is divided into three parts: buffer management area, branch trace store (BTS) buffer, and PEBS buffer (see Figure 17-5). The buffer management area is used to define the location and size of the BTS and PEBS buffers. The processor then uses the buffer management area to keep track of the branch and/or PEBS records in their respective buffers and to record the performance counter reset value. The linear address of the first byte of the DS buffer management area is specified with the IA32_DS_AREA MSR.

The fields in the buffer management area are as follows:

- **BTS buffer base —** Linear address of the first byte of the BTS buffer. This address should point to a natural doubleword boundary.
- **BTS index —** Linear address of the first byte of the next BTS record to be written to. Initially, this address should be the same as the address in the BTS buffer base field.
- **BTS absolute maximum —** Linear address of the next byte past the end of the BTS buffer. This address should be a multiple of the BTS record size (12 bytes) plus 1.
- **BTS interrupt threshold —** Linear address of the BTS record on which an interrupt is to be generated. This address must point to an offset from the BTS buffer base that is a multiple of the BTS record size. Also, it must be several records short of the BTS absolute maximum address to allow a pending interrupt to be handled prior to processor writing the BTS absolute maximum record.
- **PEBS buffer base —** Linear address of the first byte of the PEBS buffer. This address should point to a natural doubleword boundary.
- **PEBS index —** Linear address of the first byte of the next PEBS record to be written to. Initially, this address should be the same as the address in the PEBS buffer base field.
- **PEBS absolute maximum —** Linear address of the next byte past the end of the PEBS buffer. This address should be a multiple of the PEBS record size (40 bytes) plus 1.
- **PEBS interrupt threshold —** Linear address of the PEBS record on which an interrupt is to be generated. This address must point to an offset from the PEBS buffer base that is a multiple of the PEBS record size. Also, it must be several records short of the PEBS absolute maximum address to allow a pending interrupt to be handled prior to processor writing the PEBS absolute maximum record.
- **PEBS counter reset value —** A 64-bit value that the counter is to be set to when a PEBS record is written. Bits beyond the size of the counter are ignored. This value allows state information to be collected regularly every time the specified number of events occur.

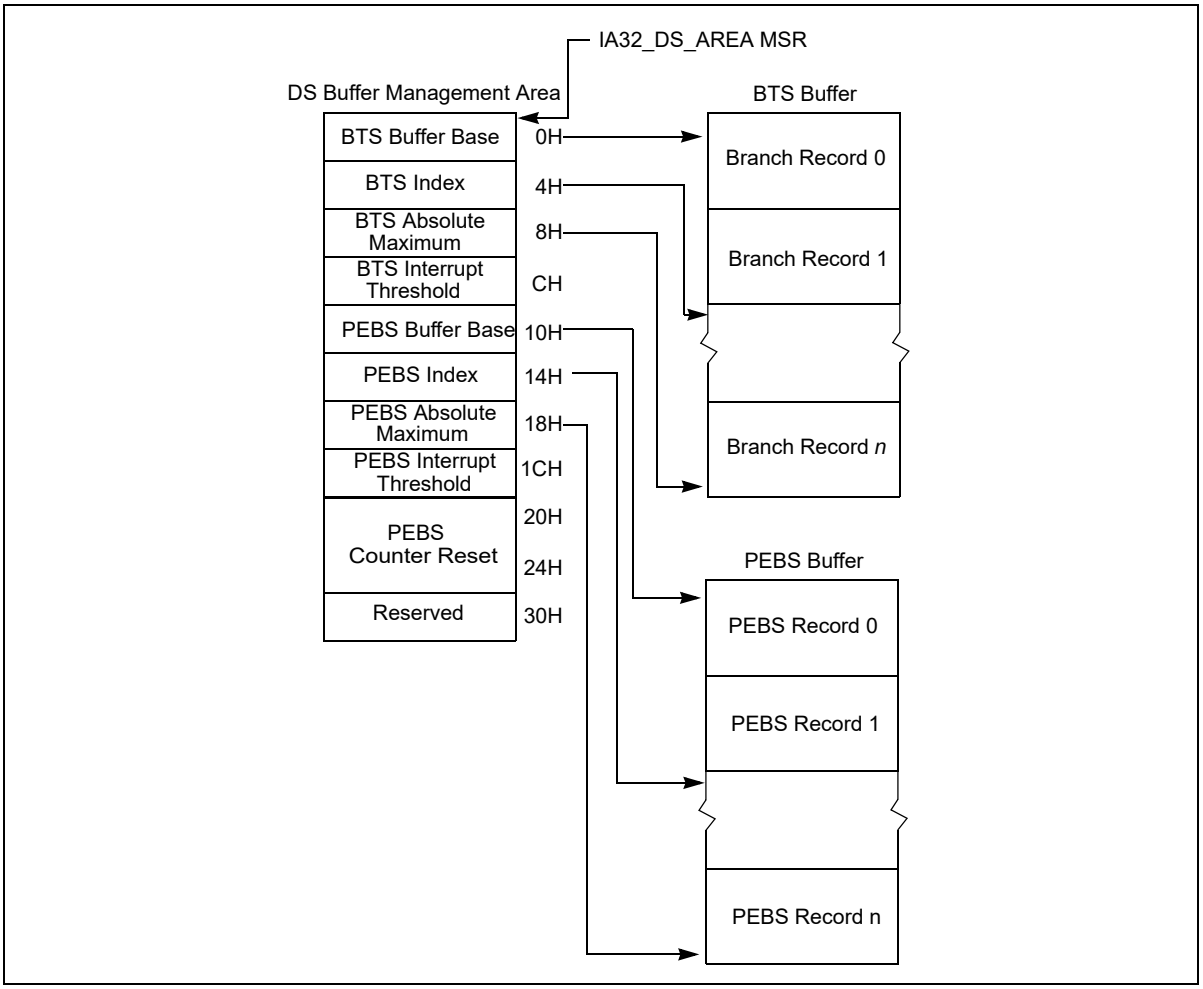


Figure 17-5. DS Save Area Example[1]

NOTES:

1. This example represents the format for a system that supports PEBS on only one counter.

Figure 17-6 shows the structure of a 12-byte branch record in the BTS buffer. The fields in each record are as follows:

- **Last branch from —** Linear address of the instruction from which the branch, interrupt, or exception was taken.
- **Last branch to —** Linear address of the branch target or the first instruction in the interrupt or exception service routine.
- **Branch predicted —** Bit 4 of field indicates whether the branch that was taken was predicted (set) or not predicted (clear).

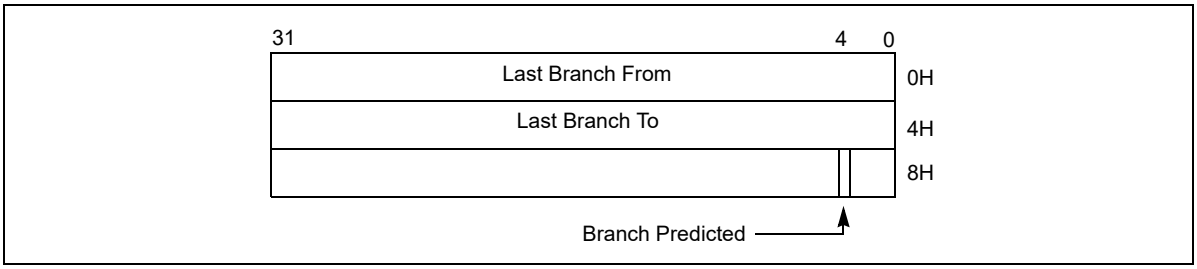


Figure 17-6. 32-bit Branch Trace Record Format

Figure 17-7 shows the structure of the 40-byte PEBS records. Nominally the register values are those at the beginning of the instruction that caused the event. However, there are cases where the registers may be logged in a partially modified state. The linear IP field shows the value in the EIP register translated from an offset into the current code segment to a linear address.

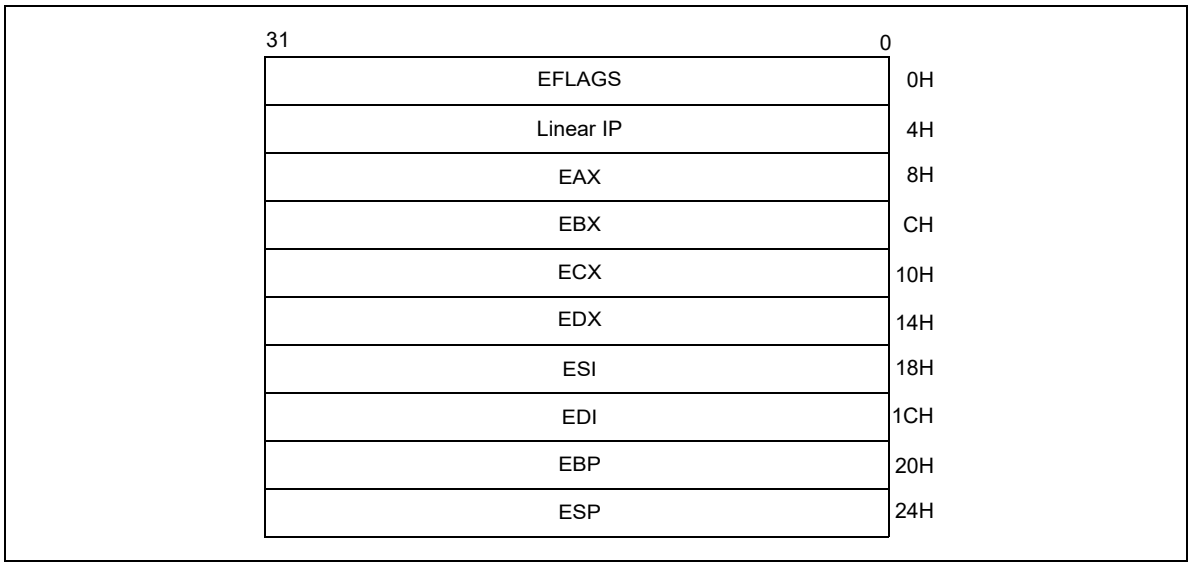


Figure 17-7. PEBS Record Format

17.4.9.1 64 Bit Format of the DS Save Area

When DTES64 = 1 (CPUID.1.ECX[2] = 1), the structure of the DS save area is shown in Figure 17-8.

When DTES64 = 0 (CPUID.1.ECX[2] = 0) and IA-32e mode is active, the structure of the DS save area is shown in Figure 17-8. If IA-32e mode is not active the structure of the DS save area is as shown in Figure 17-5.

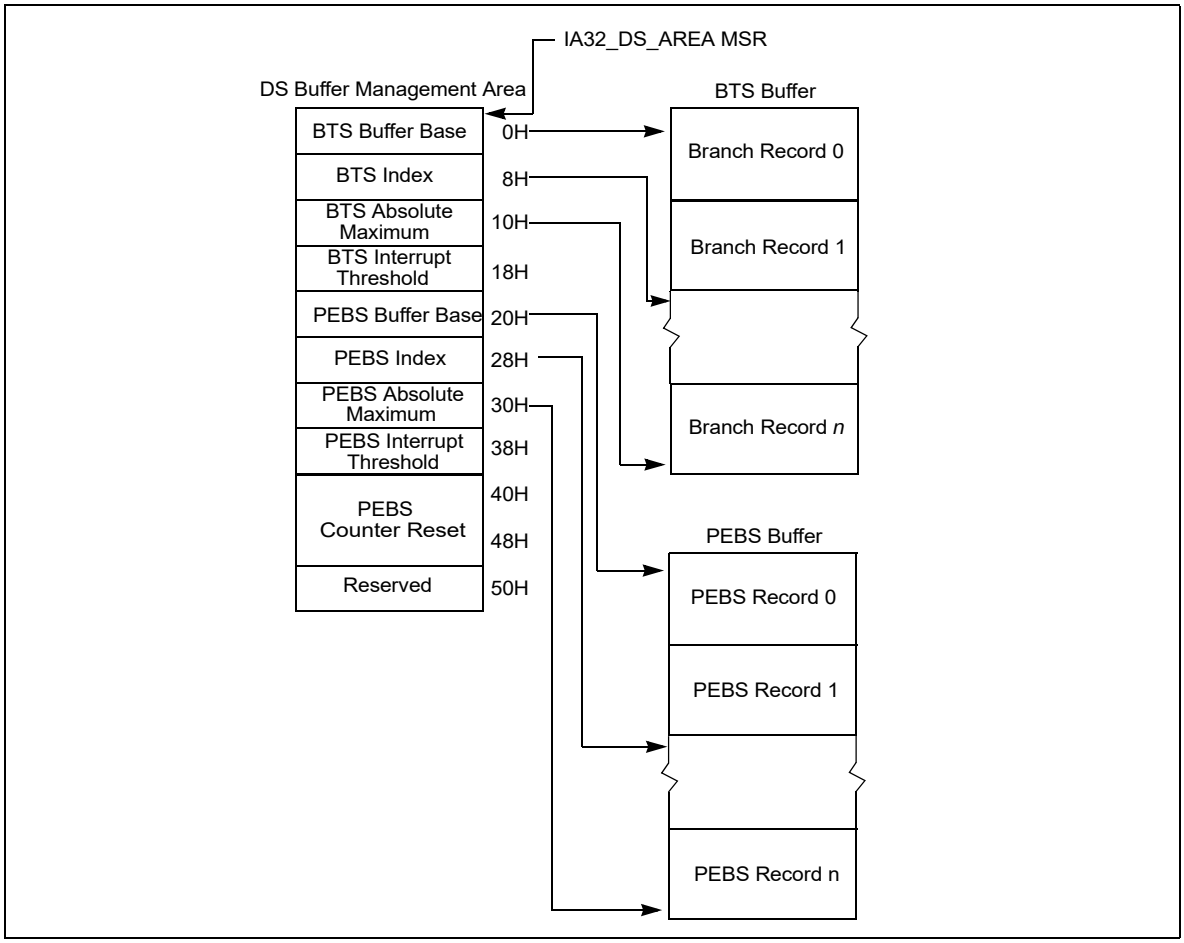


Figure 17-8. IA-32e Mode DS Save Area Example[1]

NOTES:

1. This example represents the format for a system that supports PEBS on only one counter.

The IA32_DS_AREA MSR holds the 64-bit linear address of the first byte of the DS buffer management area. The structure of a branch trace record is similar to that shown in Figure 17-6, but each field is 8 bytes in length. This makes each BTS record 24 bytes (see Figure 17-9). The structure of a PEBS record is similar to that shown in Figure 17-7, but each field is 8 bytes in length and architectural states include register R8 through R15. This makes the size of a PEBS record in 64-bit mode 144 bytes (see Figure 17-10).

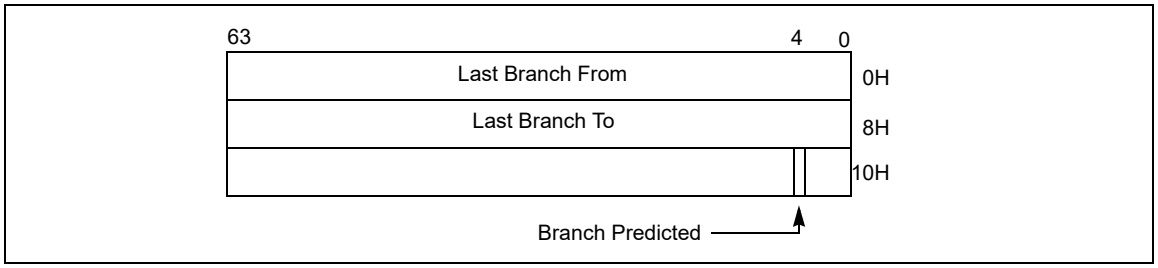


Figure 17-9. 64-bit Branch Trace Record Format

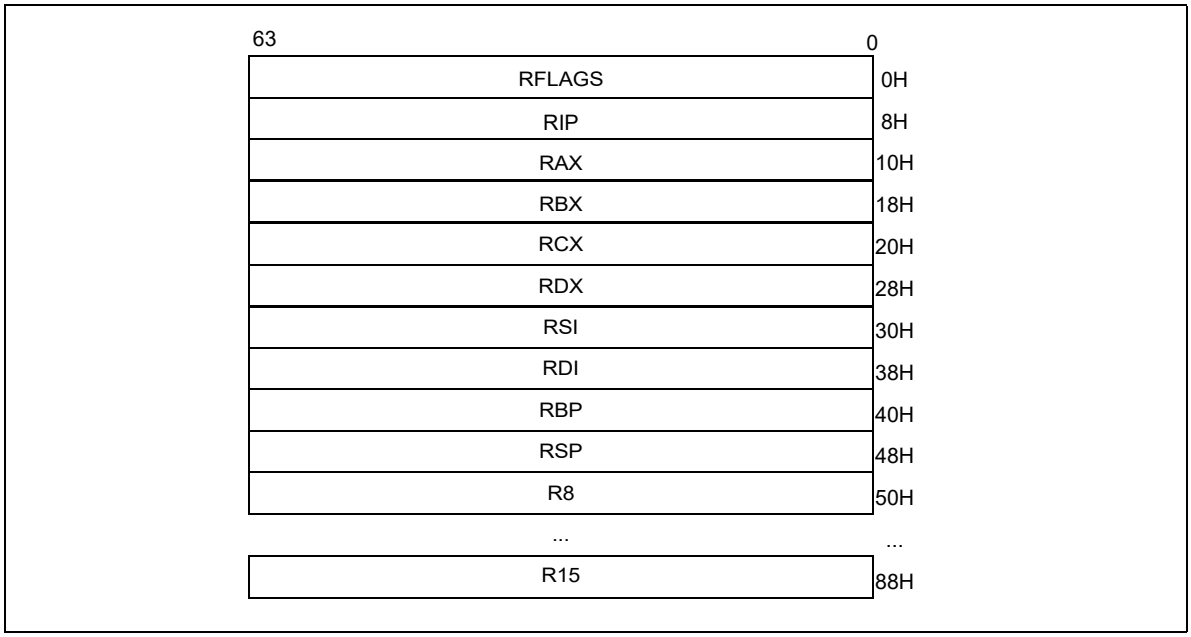


Figure 17-10. 64-bit PEBS Record Format

Fields in the buffer management area of a DS save area are described in Section 17.4.9.

The format of a branch trace record and a PEBS record are the same as the 64-bit record formats shown in Figures 17-9 and Figures 17-10, with the exception that the branch predicted bit is not supported by Intel Core microarchitecture or Intel Atom microarchitecture. The 64-bit record formats for BTS and PEBS apply to DS save area for all operating modes.

The procedures used to program IA32_DEBUGCTL MSR to set up a BTS buffer or a CPL-qualified BTS are described in Section 17.4.9.3 and Section 17.4.9.4.

Required elements for writing a DS interrupt service routine are largely the same on processors that support using DS Save area for BTS or PEBS records. However, on processors based on Intel NetBurst® microarchitecture, re-enabling counting requires writing to CCCRs. But a DS interrupt service routine on processors supporting architectural performance monitoring should:

- Re-enable the enable bits in IA32_PERF_GLOBAL_CTRL MSR if it is servicing an overflow PMI due to PEBS.
- Clear overflow indications by writing to IA32_PERF_GLOBAL_OVF_CTRL when a counting configuration is changed. This includes bit 62 (ClrOvfBuffer) and the overflow indication of counters used in either PEBS or general-purpose counting (specifically: bits 0 or 1; see Figures 18-3).

17.4.9.2 Setting Up the DS Save Area

To save branch records with the BTS buffer, the DS save area must first be set up in memory as described in the following procedure (See Section 18.6.2.4.1, "Setting up the PEBS Buffer," for instructions for setting up a PEBS buffer, respectively, in the DS save area):

1. Create the DS buffer management information area in memory (see Section 17.4.9, "BTS and DS Save Area," and Section 17.4.9.1, "64 Bit Format of the DS Save Area"). Also see the additional notes in this section.
2. Write the base linear address of the DS buffer management area into the IA32_DS_AREA MSR.
3. Set up the performance counter entry in the xAPIC LVT for fixed delivery and edge sensitive. See Section 10.5.1, "Local Vector Table."
4. Establish an interrupt handler in the IDT for the vector associated with the performance counter entry in the xAPIC LVT.

5. Write an interrupt service routine to handle the interrupt. See Section 17.4.9.5, "Writing the DS Interrupt Service Routine."

The following restrictions should be applied to the DS save area.

- The three DS save area sections should be allocated from a non-paged pool, and marked accessed and dirty. It is the responsibility of the operating system to keep the pages that contain the buffer present and to mark them accessed and dirty. The implication is that the operating system cannot do "lazy" page-table entry propagation for these pages.
- The DS save area can be larger than a page, but the pages must be mapped to contiguous linear addresses. The buffer may share a page, so it need not be aligned on a 4-KByte boundary. For performance reasons, the base of the buffer must be aligned on a doubleword boundary and should be aligned on a cache line boundary.
- It is recommended that the buffer size for the BTS buffer and the PEBS buffer be an integer multiple of the corresponding record sizes.
- The precise event records buffer should be large enough to hold the number of precise event records that can occur while waiting for the interrupt to be serviced.
- The DS save area should be in kernel space. It must not be on the same page as code, to avoid triggering self-modifying code actions.
- There are no memory type restrictions on the buffers, although it is recommended that the buffers be designated as WB memory type for performance considerations.
- Either the system must be prevented from entering A20M mode while DS save area is active, or bit 20 of all addresses within buffer bounds must be 0.
- Pages that contain buffers must be mapped to the same physical addresses for all processes, such that any change to control register CR3 will not change the DS addresses.
- The DS save area is expected to used only on systems with an enabled APIC. The LVT Performance Counter entry in the APCI must be initialized to use an interrupt gate instead of the trap gate.

17.4.9.3 Setting Up the BTS Buffer

Three flags in the MSR_DEBUGCTLA MSR (see Table 17-5), IA32_DEBUGCTL (see Figure 17-3), or MSR_DEBUGCTLB (see Figure 17-16) control the generation of branch records and storing of them in the BTS buffer; these are TR, BTS, and BTINT. The TR flag enables the generation of BTMs. The BTS flag determines whether the BTMs are sent out on the system bus (clear) or stored in the BTS buffer (set). BTMs cannot be simultaneously sent to the system bus and logged in the BTS buffer. The BTINT flag enables the generation of an interrupt when the BTS buffer is full. When this flag is clear, the BTS buffer is a circular buffer.

Table 17-5. IA32_DEBUGCTL Flag Encodings

TR	BTS	BTINT	Description
0	X	X	Branch trace messages (BTMs) off
1	0	X	Generate BTMs
1	1	0	Store BTMs in the BTS buffer, used here as a circular buffer
1	1	1	Store BTMs in the BTS buffer, and generate an interrupt when the buffer is nearly full

The following procedure describes how to set up a DS Save area to collect branch records in the BTS buffer:

1. Place values in the BTS buffer base, BTS index, BTS absolute maximum, and BTS interrupt threshold fields of the DS buffer management area to set up the BTS buffer in memory.
2. Set the TR and BTS flags in the IA32_DEBUGCTL for Intel Core Solo and Intel Core Duo processors or later processors (or MSR_DEBUGCTLA MSR for processors based on Intel NetBurst Microarchitecture; or MSR_DEBUGCTLB for Pentium M processors).
3. Clear the BTINT flag in the corresponding IA32_DEBUGCTL (or MSR_DEBUGCTLA MSR; or MSR_DEBUGCTLB) if a circular BTS buffer is desired.

NOTES

If the buffer size is set to less than the minimum allowable value (i.e. BTS absolute maximum < 1 + size of BTS record), the results of BTS is undefined.

In order to prevent generating an interrupt, when working with circular BTS buffer, SW need to set BTS interrupt threshold to a value greater than BTS absolute maximum (fields of the DS buffer management area). It's not enough to clear the BTINT flag itself only.

17.4.9.4 Setting Up CPL-Qualified BTS

If the processor supports CPL-qualified last branch recording mechanism, the generation of branch records and storing of them in the BTS buffer are determined by: TR, BTS, BTS_OFF_OS, BTS_OFF_USR, and BTINT. The encoding of these five bits are shown in Table 17-6.

Table 17-6. CPL-Qualified Branch Trace Store Encodings

TR	BTS	BTS_OFF_OS	BTS_OFF_USR	BTINT	Description
0	X	X	X	X	Branch trace messages (BTMs) off
1	0	X	X	X	Generates BTMs but do not store BTMs
1	1	0	0	0	Store all BTMs in the BTS buffer, used here as a circular buffer
1	1	1	0	0	Store BTMs with CPL > 0 in the BTS buffer
1	1	0	1	0	Store BTMs with CPL = 0 in the BTS buffer
1	1	1	1	X	Generate BTMs but do not store BTMs
1	1	0	0	1	Store all BTMs in the BTS buffer; generate an interrupt when the buffer is nearly full
1	1	1	0	1	Store BTMs with CPL > 0 in the BTS buffer; generate an interrupt when the buffer is nearly full
1	1	0	1	1	Store BTMs with CPL = 0 in the BTS buffer; generate an interrupt when the buffer is nearly full

17.4.9.5 Writing the DS Interrupt Service Routine

The BTS, non-precise event-based sampling, and PEBS facilities share the same interrupt vector and interrupt service routine (called the debug store interrupt service routine or DS ISR). To handle BTS, non-precise event-based sampling, and PEBS interrupts: separate handler routines must be included in the DS ISR. Use the following guidelines when writing a DS ISR to handle BTS, non-precise event-based sampling, and/or PEBS interrupts.

- The DS interrupt service routine (ISR) must be part of a kernel driver and operate at a current privilege level of 0 to secure the buffer storage area.
- Because the BTS, non-precise event-based sampling, and PEBS facilities share the same interrupt vector, the DS ISR must check for all the possible causes of interrupts from these facilities and pass control on to the appropriate handler.

  BTS and PEBS buffer overflow would be the sources of the interrupt if the buffer index matches/exceeds the interrupt threshold specified. Detection of non-precise event-based sampling as the source of the interrupt is accomplished by checking for counter overflow.
- There must be separate save areas, buffers, and state for each processor in an MP system.
- Upon entering the ISR, branch trace messages and PEBS should be disabled to prevent race conditions during access to the DS save area. This is done by clearing TR flag in the IA32_DEBUGCTL (or MSR_DEBUGCTLA MSR) and by clearing the precise event enable flag in the MSR_PEBS_ENABLE MSR. These settings should be restored to their original values when exiting the ISR.
- The processor will not disable the DS save area when the buffer is full and the circular mode has not been selected. The current DS setting must be retained and restored by the ISR on exit.

- After reading the data in the appropriate buffer, up to but not including the current index into the buffer, the ISR must reset the buffer index to the beginning of the buffer. Otherwise, everything up to the index will look like new entries upon the next invocation of the ISR.
- The ISR must clear the mask bit in the performance counter LVT entry.
- The ISR must re-enable the counters to count via IA32_PERF_GLOBAL_CTRL/IA32_PERF_GLOBAL_OVF_CTRL if it is servicing an overflow PMI due to PEBS (or via CCCR's ENABLE bit on processor based on Intel NetBurst microarchitecture).
- The Pentium 4 Processor and Intel Xeon Processor mask PMIs upon receiving an interrupt. Clear this condition before leaving the interrupt handler.

17.5 LAST BRANCH, INTERRUPT, AND EXCEPTION RECORDING (INTEL® CORE™ 2 DUO AND INTEL® ATOM™ PROCESSORS)

The Intel Core 2 Duo processor family and Intel Xeon processors based on Intel Core microarchitecture or enhanced Intel Core microarchitecture provide last branch interrupt and exception recording. The facilities described in this section also apply to 45 nm and 32 nm Intel Atom processors. These capabilities are similar to those found in Pentium 4 processors, including support for the following facilities:

- **Debug Trace and Branch Recording Control —** The IA32_DEBUGCTL MSR provide bit fields for software to configure mechanisms related to debug trace, branch recording, branch trace store, and performance counter operations. See Section 17.4.1 for a description of the flags. See Figure 17-3 for the MSR layout.
- **Last branch record (LBR) stack —** There are a collection of MSR pairs that store the source and destination addresses related to recently executed branches. See Section 17.5.1.
- **Monitoring and single-stepping of branches, exceptions, and interrupts**
  - See Section 17.4.2 and Section 17.4.3. In addition, the ability to freeze the LBR stack on a PMI request is available.
  - 45 nm and 32 nm Intel Atom processors clear the TR flag when the FREEZE_LBRS_ON_PMI flag is set.
- **Branch trace messages —** See Section 17.4.4.
- **Last exception records —** See Section 17.13.3.
- **Branch trace store and CPL-qualified BTS —** See Section 17.4.5.
- **FREEZE_LBRS_ON_PMI flag (bit 11) —** see Section 17.4.7 for legacy Freeze_LBRs_On_PMI operation.
- **FREEZE_PERFMON_ON_PMI flag (bit 12) —** see Section 17.4.7 for legacy Freeze_Perfmon_On_PMI operation.
- **FREEZE_WHILE_SMM_EN (bit 14) —** FREEZE_WHILE_SMM_EN is supported if IA32_PERF_CAPABILITIES.FREEZE_WHILE_SMM[Bit 12] is reporting 1. See Section 17.4.1.

17.5.1 LBR Stack

The last branch record stack and top-of-stack (TOS) pointer MSRs are supported across Intel Core 2, Intel Atom processor families, and Intel processors based on Intel NetBurst microarchitecture.

Four pairs of MSRs are supported in the LBR stack for Intel Core 2 processors families and Intel processors based on Intel NetBurst microarchitecture:

- **Last Branch Record (LBR) Stack**
  - MSR_LASTBRANCH_0_FROM_IP (address 40H) through MSR_LASTBRANCH_3_FROM_IP (address 43H) store source addresses
  - MSR_LASTBRANCH_0_TO_IP (address 60H) through MSR_LASTBRANCH_3_TO_IP (address 63H) store destination addresses

- **Last Branch Record Top-of-Stack (TOS) Pointer** — The lowest significant 2 bits of the TOS Pointer MSR (MSR_LASTBRANCH_TOS, address 1C9H) contains a pointer to the MSR in the LBR stack that contains the most recent branch, interrupt, or exception recorded.

Eight pairs of MSRs are supported in the LBR stack for 45 nm and 32 nm Intel Atom processors:

- **Last Branch Record (LBR) Stack**
  - MSR_LASTBRANCH_0_FROM_IP (address 40H) through MSR_LASTBRANCH_7_FROM_IP (address 47H) store source addresses
  - MSR_LASTBRANCH_0_TO_IP (address 60H) through MSR_LASTBRANCH_7_TO_IP (address 67H) store destination addresses
- **Last Branch Record Top-of-Stack (TOS) Pointer** — The lowest significant 3 bits of the TOS Pointer MSR (MSR_LASTBRANCH_TOS, address 1C9H) contains a pointer to the MSR in the LBR stack that contains the most recent branch, interrupt, or exception recorded.

The address format written in the FROM_IP/TO_IP MSRS may differ between processors. Software should query IA32_PERF_CAPABILITIES[5:0] and consult Section 17.4.8.1. The behavior of the MSR_LER_TO_LIP and the MSR_LER_FROM_LIP MSRs corresponds to that of the LastExceptionToIP and LastExceptionFromIP MSRs found in P6 family processors.

17.5.2 LBR Stack in Intel Atom Processors based on the Silvermont Microarchitecture

The last branch record stack and top-of-stack (TOS) pointer MSRs are supported in Intel Atom processors based on the Silvermont and Airmont microarchitectures. Eight pairs of MSRs are supported in the LBR stack.

LBR filtering is supported. Filtering of LBRs based on a combination of CPL and branch type conditions is supported. When LBR filtering is enabled, the LBR stack only captures the subset of branches that are specified by MSR_LBR_SELECT. The layout of MSR_LBR_SELECT is described in Table 17-11.

17.6 LAST BRANCH, CALL STACK, INTERRUPT, AND EXCEPTION RECORDING FOR PROCESSORS BASED ON GOLDMONT MICROARCHITECTURE

Processors based on the Goldmont microarchitecture extend the capabilities described in Section 17.5.2 with the following enhancements:

- Supports new LBR format encoding 00110b in IA32_PERF_CAPABILITIES[5:0].
- Size of LBR stack increased to 32. Each entry includes MSR_LASTBRANCH_x_FROM_IP (address 0x680..0x69f) and MSR_LASTBRANCH_x_TO_IP (address 0x6c0..0x6df).
- LBR call stack filtering supported. The layout of MSR_LBR_SELECT is described in Table 17-13.
- Elapsed cycle information is added to MSR_LASTBRANCH_x_TO_IP. Format is shown in Table 17-7.
- Misprediction info is reported in the upper bits of MSR_LASTBRANCH_x_FROM_IP. MISPRED bit format is shown in Table 17-8.
- Streamlined Freeze_LBRs_On_PMI operation; see Section 17.12.2.
- LBR MSRs may be cleared when MWAIT is used to request a C-state that is numerically higher than C1; see Section 17.12.3.

Table 17-7. MSR_LASTBRANCH_x_TO_IP for the Goldmont Microarchitecture

Bit Field	Bit Offset	Access	Description
Data	47:0	R/W	This is the "branch to" address. See Section 17.4.8.1 for address format.
Cycle Count (Saturating)	63:48	R/W	Elapsed core clocks since last update to the LBR stack.

17.7 LAST BRANCH, CALL STACK, INTERRUPT, AND EXCEPTION RECORDING FOR PROCESSORS BASED ON GOLDMONT PLUS MICROARCHITECTURE

Next generation Intel Atom processors are based on the Goldmont Plus microarchitecture. Processors based on the Goldmont Plus microarchitecture extend the capabilities described in Section 17.6 with the following changes:

- Enumeration of new LBR format: encoding 00111b in IA32_PERF_CAPABILITIES[5:0] is supported, see Section 17.4.8.1.
- Each LBR stack entry consists of three MSRs:
  - MSR_LASTBRANCH_x_FROM_IP, the layout is simplified, see Table 17-9.
  - MSR_LASTBRANCH_x_TO_IP, the layout is the same as Table 17-9.
  - MSR_LBR_INFO_x, stores branch prediction flag, TSX info, and elapsed cycle data. Layout is the same as Table 17-16.

17.8 LAST BRANCH, INTERRUPT AND EXCEPTION RECORDING FOR INTEL® XEON PHI™ PROCESSOR 7200/5200/3200

The last branch record stack and top-of-stack (TOS) pointer MSRs are supported in the Intel® Xeon Phi™ processor 7200/5200/3200 series based on the Knights Landing microarchitecture. Eight pairs of MSRs are supported in the LBR stack, per thread:

- **Last Branch Record (LBR) Stack**
  - MSR_LASTBRANCH_0_FROM_IP (address 680H) through MSR_LASTBRANCH_7_FROM_IP (address 687H) store source addresses.
  - MSR_LASTBRANCH_0_TO_IP (address 6C0H) through MSR_LASTBRANCH_7_TO_IP (address 6C7H) store destination addresses.
- **Last Branch Record Top-of-Stack (TOS) Pointer —** The lowest significant 3 bits of the TOS Pointer MSR (MSR_LASTBRANCH_TOS, address 1C9H) contains a pointer to the MSR in the LBR stack that contains the most recent branch, interrupt, or exception recorded.

LBR filtering is supported. Filtering of LBRs based on a combination of CPL and branch type conditions is supported. When LBR filtering is enabled, the LBR stack only captures the subset of branches that are specified by MSR_LBR_SELECT. The layout of MSR_LBR_SELECT is described in Table 17-11.

The address format written in the FROM_IP/TO_IP MSRS may differ between processors. Software should query IA32_PERF_CAPABILITIES[5:0] and consult Section 17.4.8.1.The behavior of the MSR_LER_TO_LIP and the MSR_LER_FROM_LIP MSRs corresponds to that of the LastExceptionToIP and LastExceptionFromIP MSRs found in the P6 family processors.

17.9 LAST BRANCH, INTERRUPT, AND EXCEPTION RECORDING FOR PROCESSORS BASED ON INTEL® MICROARCHITECTURE CODE NAME NEHALEM

The processors based on Intel® microarchitecture code name Nehalem and Intel® microarchitecture code name Westmere support last branch interrupt and exception recording. These capabilities are similar to those found in Intel Core 2 processors and adds additional capabilities:

- **Debug Trace and Branch Recording Control —** The IA32_DEBUGCTL MSR provides bit fields for software to configure mechanisms related to debug trace, branch recording, branch trace store, and performance counter operations. See Section 17.4.1 for a description of the flags. See Figure 17-11 for the MSR layout.
- **Last branch record (LBR) stack —** There are 16 MSR pairs that store the source and destination addresses related to recently executed branches. See Section 17.9.1.

- **Monitoring and single-stepping of branches, exceptions, and interrupts —** See Section 17.4.2 and Section 17.4.3. In addition, the ability to freeze the LBR stack on a PMI request is available.
- **Branch trace messages —** The IA32_DEBUGCTL MSR provides bit fields for software to enable each logical processor to generate branch trace messages. See Section 17.4.4. However, not all BTM messages are observable using the Intel® QPI link.
- **Last exception records —** See Section 17.13.3.
- **Branch trace store and CPL-qualified BTS —** See Section 17.4.6 and Section 17.4.5.
- **FREEZE_LBRS_ON_PMI flag (bit 11) —** see Section 17.4.7 for legacy Freeze_LBRs_On_PMI operation.
- **FREEZE_PERFMON_ON_PMI flag (bit 12) —** see Section 17.4.7 for legacy Freeze_Perfmon_On_PMI operation.
- **UNCORE_PMI_EN (bit 13) —** When set. this logical processor is enabled to receive an counter overflow interrupt form the uncore.
- **FREEZE_WHILE_SMM_EN (bit 14) —** FREEZE_WHILE_SMM_EN is supported if IA32_PERF_CAPABILITIES.FREEZE_WHILE_SMM[Bit 12] is reporting 1. See Section 17.4.1.

Processors based on Intel microarchitecture code name Nehalem provide additional capabilities:

- **Independent control of uncore PMI —** The IA32_DEBUGCTL MSR provides a bit field (see Figure 17-11) for software to enable each logical processor to receive an uncore counter overflow interrupt.
- **LBR filtering —** Processors based on Intel microarchitecture code name Nehalem support filtering of LBR based on combination of CPL and branch type conditions. When LBR filtering is enabled, the LBR stack only captures the subset of branches that are specified by MSR_LBR_SELECT.

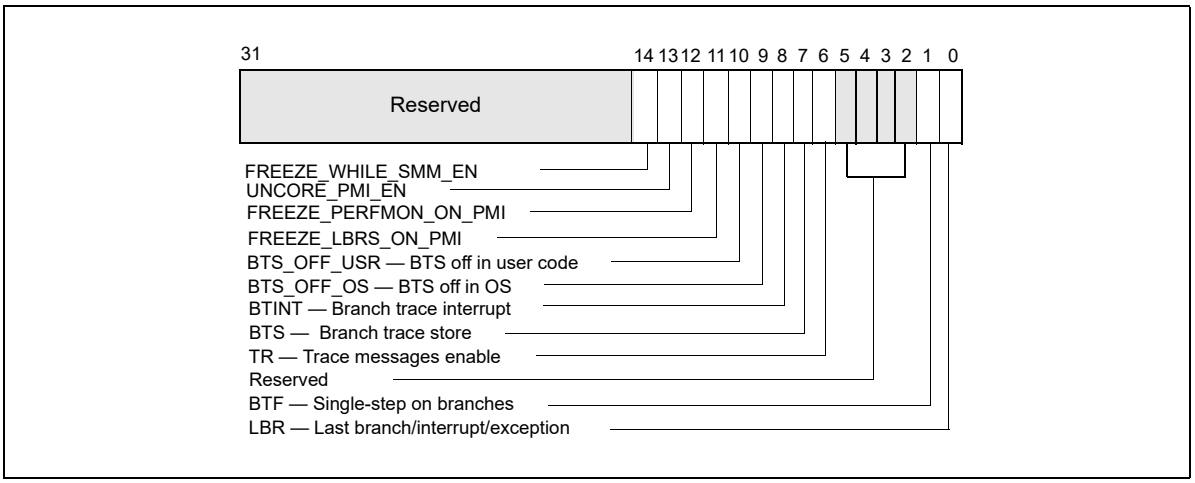


Figure 17-11. IA32_DEBUGCTL MSR for Processors based on Intel microarchitecture code name Nehalem

17.9.1 LBR Stack

Processors based on Intel microarchitecture code name Nehalem provide 16 pairs of MSR to record last branch record information. The layout of each MSR pair is shown in Table 17-8 and Table 17-9.

Table 17-8. MSR_LASTBRANCH_x_FROM_IP

Bit Field	Bit Offset	Access	Description
Data	47:0	R/W	This is the "branch from" address. See Section 17.4.8.1 for address format.
SIGN_EXt	62:48	R/W	Signed extension of bit 47 of this register.
MISPRED	63	R/W	When set, indicates either the target of the branch was mispredicted and/or the direction (taken/non-taken) was mispredicted; otherwise, the target branch was predicted.

Table 17-9. MSR_LASTBRANCH_x_TO_IP

Bit Field	Bit Offset	Access	Description
Data	47:0	R/W	This is the "branch to" address. See Section 17.4.8.1 for address format
SIGN_EXt	63:48	R/W	Signed extension of bit 47 of this register.

Processors based on Intel microarchitecture code name Nehalem have an LBR MSR Stack as shown in Table 17-10.

Table 17-10. LBR Stack Size and TOS Pointer Range

DisplayFamily_DisplayModel	Size of LBR Stack	Range of TOS Pointer
06_1AH	16	0 to 15

17.9.2 Filtering of Last Branch Records

MSR_LBR_SELECT is cleared to zero at RESET, and LBR filtering is disabled, i.e. all branches will be captured. MSR_LBR_SELECT provides bit fields to specify the conditions of subsets of branches that will not be captured in the LBR. The layout of MSR_LBR_SELECT is shown in Table 17-11.

Table 17-11. MSR_LBR_SELECT for Intel microarchitecture code name Nehalem

Bit Field	Bit Offset	Access	Description
CPL_EQ_0	0	R/W	When set, do not capture branches occurring in ring 0
CPL_NEQ_0	1	R/W	When set, do not capture branches occurring in ring >0
JCC	2	R/W	When set, do not capture conditional branches
NEAR_REL_CALL	3	R/W	When set, do not capture near relative calls
NEAR_IND_CALL	4	R/W	When set, do not capture near indirect calls
NEAR_RET	5	R/W	When set, do not capture near returns
NEAR_IND_JMP	6	R/W	When set, do not capture near indirect jumps
NEAR_REL_JMP	7	R/W	When set, do not capture near relative jumps
FAR_BRANCH	8	R/W	When set, do not capture far branches
Reserved	63:9		Must be zero

17.10 LAST BRANCH, INTERRUPT, AND EXCEPTION RECORDING FOR PROCESSORS BASED ON INTEL® MICROARCHITECTURE CODE NAME SANDY BRIDGE

Generally, all of the last branch record, interrupt and exception recording facility described in Section 17.9, "Last Branch, Interrupt, and Exception Recording for Processors based on Intel® Microarchitecture code name Nehalem", apply to processors based on Intel microarchitecture code name Sandy Bridge. For processors based on Intel microarchitecture code name Ivy Bridge, the same holds true.

One difference of note is that MSR_LBR_SELECT is shared between two logical processors in the same core. In Intel microarchitecture code name Sandy Bridge, each logical processor has its own MSR_LBR_SELECT. The filtering semantics for "Near_ind_jmp" and "Near_rel_jmp" has been enhanced, see Table 17-12.

Table 17-12. MSR_LBR_SELECT for Intel® microarchitecture code name Sandy Bridge

Bit Field	Bit Offset	Access	Description
CPL_EQ_0	0	R/W	When set, do not capture branches occurring in ring 0
CPL_NEQ_0	1	R/W	When set, do not capture branches occurring in ring >0
JCC	2	R/W	When set, do not capture conditional branches
NEAR_REL_CALL	3	R/W	When set, do not capture near relative calls
NEAR_IND_CALL	4	R/W	When set, do not capture near indirect calls
NEAR_RET	5	R/W	When set, do not capture near returns
NEAR_IND_JMP	6	R/W	When set, do not capture near indirect jumps except near indirect calls and near returns
NEAR_REL_JMP	7	R/W	When set, do not capture near relative jumps except near relative calls.
FAR_BRANCH	8	R/W	When set, do not capture far branches
Reserved	63:9		Must be zero

17.11 LAST BRANCH, CALL STACK, INTERRUPT, AND EXCEPTION RECORDING FOR PROCESSORS BASED ON HASWELL MICROARCHITECTURE

Generally, all of the last branch record, interrupt and exception recording facility described in Section 17.10, "Last Branch, Interrupt, and Exception Recording for Processors based on Intel® Microarchitecture code name Sandy Bridge", apply to next generation processors based on Intel microarchitecture code name Haswell.

The LBR facility also supports an alternate capability to profile call stack profiles. Configuring the LBR facility to conduct call stack profiling is by writing 1 to the MSR_LBR_SELECT.EN_CALLSTACK[bit 9]; see Table 17-13. If MSR_LBR_SELECT.EN_CALLSTACK is clear, the LBR facility will capture branches normally as described in Section 17.10.

Table 17-13. MSR_LBR_SELECT for Intel® microarchitecture code name Haswell

Bit Field	Bit Offset	Access	Description
CPL_EQ_0	0	R/W	When set, do not capture branches occurring in ring 0
CPL_NEQ_0	1	R/W	When set, do not capture branches occurring in ring >0
JCC	2	R/W	When set, do not capture conditional branches
NEAR_REL_CALL	3	R/W	When set, do not capture near relative calls
NEAR_IND_CALL	4	R/W	When set, do not capture near indirect calls
NEAR_RET	5	R/W	When set, do not capture near returns
NEAR_IND_JMP	6	R/W	When set, do not capture near indirect jumps except near indirect calls and near returns
NEAR_REL_JMP	7	R/W	When set, do not capture near relative jumps except near relative calls.

Table 17-13. MSR_LBR_SELECT for Intel® microarchitecture code name Haswell

Bit Field	Bit Offset	Access	Description
FAR_BRANCH	8	R/W	When set, do not capture far branches
EN_CALLSTACK[1]	9		Enable LBR stack to use LIFO filtering to capture Call stack profile
Reserved	63:10		Must be zero

NOTES:

1. Must set valid combination of bits 0-8 in conjunction with bit 9 (as described below), otherwise the contents of the LBR MSRs are undefined.

The call stack profiling capability is an enhancement of the LBR facility. The LBR stack is a ring buffer typically used to profile control flow transitions resulting from branches. However, the finite depth of the LBR stack often become less effective when profiling certain high-level languages (e.g. C++), where a transition of the execution flow is accompanied by a large number of leaf function calls, each of which returns an individual parameter to form the list of parameters for the main execution function call. A long list of such parameters returned by the leaf functions would serve to flush the data captured in the LBR stack, often losing the main execution context.

When the call stack feature is enabled, the LBR stack will capture unfiltered call data normally, but as return instructions are executed the last captured branch record is flushed from the on-chip registers in a last-in first-out (LIFO) manner. Thus, branch information relative to leaf functions will not be captured, while preserving the call stack information of the main line execution path.

The configuration of the call stack facility is summarized below:

- Set IA32_DEBUGCTL.LBR (bit 0) to enable the LBR stack to capture branch records. The source and target addresses of the call branches will be captured in the 16 pairs of From/To LBR MSRs that form the LBR stack.
- Program the Top of Stack (TOS) MSR that points to the last valid from/to pair. This register is incremented by 1, modulo 16, before recording the next pair of addresses.
- Program the branch filtering bits of MSR_LBR_SELECT (bits 0:8) as desired.
- Program the MSR_LBR_SELECT to enable LIFO filtering of return instructions with:
  - The following bits in MSR_LBR_SELECT must be set to '1': JCC, NEAR_IND_JMP, NEAR_REL_JMP, FAR_BRANCH, EN_CALLSTACK;
  - The following bits in MSR_LBR_SELECT must be cleared: NEAR_REL_CALL, NEAR-IND_CALL, NEAR_RET;
  - At most one of CPL_EQ_0, CPL_NEQ_0 is set.

Note that when call stack profiling is enabled, "zero length calls" are excluded from writing into the LBRs. (A "zero length call" uses the attribute of the call instruction to push the immediate instruction pointer on to the stack and then pops off that address into a register. This is accomplished without any matching return on the call.)

17.11.1 LBR Stack Enhancement

Processors based on Intel microarchitecture code name Haswell provide 16 pairs of MSR to record last branch record information. The layout of each MSR pair is enumerated by IA32_PERF_CAPABILITIES[5:0] = 04H, and is shown in Table 17-14 and Table 17-9.

Table 17-14. MSR_LASTBRANCH_x_FROM_IP with TSX Information

Bit Field	Bit Offset	Access	Description
Data	47:0	R/W	This is the "branch from" address. See Section 17.4.8.1 for address format.
SIGN_EXT	60:48	R/W	Signed extension of bit 47 of this register.
TSX_ABORT	61	R/W	When set, indicates a TSX Abort entry LBR_FROM: EIP at the time of the TSX Abort LBR_TO: EIP of the start of HLE region, or EIP of the RTM Abort Handler
IN_TSX	62	R/W	When set, indicates the entry occurred in a TSX region

Table 17-14. MSR_LASTBRANCH_x_FROM_IP with TSX Information (Contd.)

Bit Field	Bit Offset	Access	Description
MISPRED	63	R/W	When set, indicates either the target of the branch was mispredicted and/or the direction (taken/non-taken) was mispredicted; otherwise, the target branch was predicted.

17.12 LAST BRANCH, CALL STACK, INTERRUPT, AND EXCEPTION RECORDING FOR PROCESSORS BASED ON SKYLAKE MICROARCHITECTURE

Processors based on the Skylake microarchitecture provide a number of enhancement with storing last branch records:

- enumeration of new LBR format: encoding 00101b in IA32_PERF_CAPABILITIES[5:0] is supported, see Section 17.4.8.1.
- Each LBR stack entry consists of a triplets of MSRs:
  - MSR_LASTBRANCH_x_FROM_IP, the layout is simplified, see Table 17-9.
  - MSR_LASTBRANCH_x_TO_IP, the layout is the same as Table 17-9.
  - MSR_LBR_INFO_x, stores branch prediction flag, TSX info, and elapsed cycle data.
- Size of LBR stack increased to 32.

Processors based on the Skylake microarchitecture supports the same LBR filtering capabilities as described in Table 17-13.

Table 17-15. LBR Stack Size and TOS Pointer Range

DisplayFamily_DisplayModel	Size of LBR Stack	Range of TOS Pointer
06_4EH, 06_5EH	32	0 to 31

17.12.1 MSR_LBR_INFO_x MSR

The layout of each MSR_LBR_INFO_x MSR is shown in Table 17-16.

Table 17-16. MSR_LBR_INFO_x

Bit Field	Bit Offset	Access	Description
Cycle Count (saturating)	15:0	R/W	Elapsed core clocks since last update to the LBR stack.
Reserved	60:16	R/W	Reserved
TSX_ABORT	61	R/W	When set, indicates a TSX Abort entry LBR_FROM: EIP at the time of the TSX Abort LBR_TO: EIP of the start of HLE region OR EIP of the RTM Abort Handler
IN_TSX	62	R/W	When set, indicates the entry occurred in a TSX region.
MISPRED	63	R/W	When set, indicates either the target of the branch was mispredicted and/or the direction (taken/non-taken) was mispredicted; otherwise, the target branch was predicted.

17.12.2 Streamlined Freeze_LBRs_On_PMI Operation

The FREEZE_LBRS_ON_PMI feature causes the LBRs to be frozen on a hardware request for a PMI. This prevents the LBRs from being overwritten by new branches, allowing the PMI handler to examine the control flow that preceded the PMI generation. Architectural performance monitoring version 4 and above supports a streamlined FREEZE_LBRs_ON_PMI operation for PMI service routine that replaces the legacy FREEZE_LBRs_ON_PMI operation (see Section 17.4.7).

While the legacy FREEZE_LBRS_ON_PMI clear the LBR bit in the IA32_DEBUGCTL MSR on a PMI request, the streamlined FREEZE_LBRS_ON_PMI will set the LBR_FRZ bit in IA32_PERF_GLOBAL_STATUS. Branches will not cause the LBRs to be updated when LBR_FRZ is set. Software can clear LBR_FRZ at the same time as it clears overflow bits by setting the LBR_FRZ bit as well as the needed overflow bit when writing to IA32_PERF_GLOBAL_STATUS_RESET MSR.

This streamlined behavior avoids race conditions between software and processor writes to IA32_DEBUGCTL that are possible with FREEZE_LBRS_ON_PMI clearing of the LBR enable.

17.12.3 LBR Behavior and Deep C-State

When MWAIT is used to request a C-state that is numerically higher than C1, then LBR state may be initialized to zero depending on optimized "waiting" state that is selected by the processor The affected LBR states include the FROM, TO, INFO, LAST_BRANCH, LER and LBR_TOS registers. The LBR enable bit and LBR_FROZEN bit are not affected. The LBR-time of the first LBR record inserted after an exit from such a C-state request will be zero.

17.13 LAST BRANCH, INTERRUPT, AND EXCEPTION RECORDING (PROCESSORS BASED ON INTEL NETBURST® MICROARCHITECTURE)

Pentium 4 and Intel Xeon processors based on Intel NetBurst microarchitecture provide the following methods for recording taken branches, interrupts and exceptions:

- Store branch records in the last branch record (LBR) stack MSRs for the most recent taken branches, interrupts, and/or exceptions in MSRs. A branch record consist of a branch-from and a branch-to instruction address.
- Send the branch records out on the system bus as branch trace messages (BTMs).
- Log BTMs in a memory-resident branch trace store (BTS) buffer.

To support these functions, the processor provides the following MSRs and related facilities:

- **MSR_DEBUGCTLA MSR** — Enables last branch, interrupt, and exception recording; single-stepping on taken branches; branch trace messages (BTMs); and branch trace store (BTS). This register is named DebugCtlMSR in the P6 family processors.
- **Debug store (DS) feature flag (CPUID.1:EDX.DS[bit 21])** — Indicates that the processor provides the debug store (DS) mechanism, which allows BTMs to be stored in a memory-resident BTS buffer.
- **CPL-qualified debug store (DS) feature flag (CPUID.1:ECX.DS-CPL[bit 4])** — Indicates that the processor provides a CPL-qualified debug store (DS) mechanism, which allows software to selectively skip sending and storing BTMs, according to specified current privilege level settings, into a memory-resident BTS buffer.
- **IA32_MISC_ENABLE MSR** — Indicates that the processor provides the BTS facilities.
- **Last branch record (LBR) stack** — The LBR stack is a circular stack that consists of four MSRs (MSR_LASTBRANCH_0 through MSR_LASTBRANCH_3) for the Pentium 4 and Intel Xeon processor family [CPUID family 0FH, models 0H-02H]. The LBR stack consists of 16 MSR pairs (MSR_LASTBRANCH_0_FROM_IP through MSR_LASTBRANCH_15_FROM_IP and MSR_LASTBRANCH_0_TO_IP through MSR_LASTBRANCH_15_TO_IP) for the Pentium 4 and Intel Xeon processor family [CPUID family 0FH, model 03H].
- **Last branch record top-of-stack (TOS) pointer** — The TOS Pointer MSR contains a 2-bit pointer (0-3) to the MSR in the LBR stack that contains the most recent branch, interrupt, or exception recorded for the

Pentium 4 and Intel Xeon processor family [CPUID family 0FH, models 0H-02H]. This pointer becomes a 4-bit pointer (0-15) for the Pentium 4 and Intel Xeon processor family [CPUID family 0FH, model 03H]. See also: Table 17-17, Figure 17-12, and Section 17.13.2, "LBR Stack for Processors Based on Intel NetBurst® Microarchitecture."

- **Last exception record** — See Section 17.13.3, "Last Exception Records."

17.13.1 MSR_DEBUGCTLA MSR

The MSR_DEBUGCTLA MSR enables and disables the various last branch recording mechanisms described in the previous section. This register can be written to using the WRMSR instruction, when operating at privilege level 0 or when in real-address mode. A protected-mode operating system procedure is required to provide user access to this register. Figure 17-12 shows the flags in the MSR_DEBUGCTLA MSR. The functions of these flags are as follows:

- **LBR (last branch/interrupt/exception) flag (bit 0) —** When set, the processor records a running trace of the most recent branches, interrupts, and/or exceptions taken by the processor (prior to a debug exception being generated) in the last branch record (LBR) stack. Each branch, interrupt, or exception is recorded as a 64-bit branch record. The processor clears this flag whenever a debug exception is generated (for example, when an instruction or data breakpoint or a single-step trap occurs). See Section 17.13.2, "LBR Stack for Processors Based on Intel NetBurst® Microarchitecture."
- **BTF (single-step on branches) flag (bit 1) —** When set, the processor treats the TF flag in the EFLAGS register as a "single-step on branches" flag rather than a "single-step on instructions" flag. This mechanism allows single-stepping the processor on taken branches. See Section 17.4.3, "Single-Stepping on Branches."
- **TR (trace message enable) flag (bit 2) —** When set, branch trace messages are enabled. Thereafter, when the processor detects a taken branch, interrupt, or exception, it sends the branch record out on the system bus as a branch trace message (BTM). See Section 17.4.4, "Branch Trace Messages."

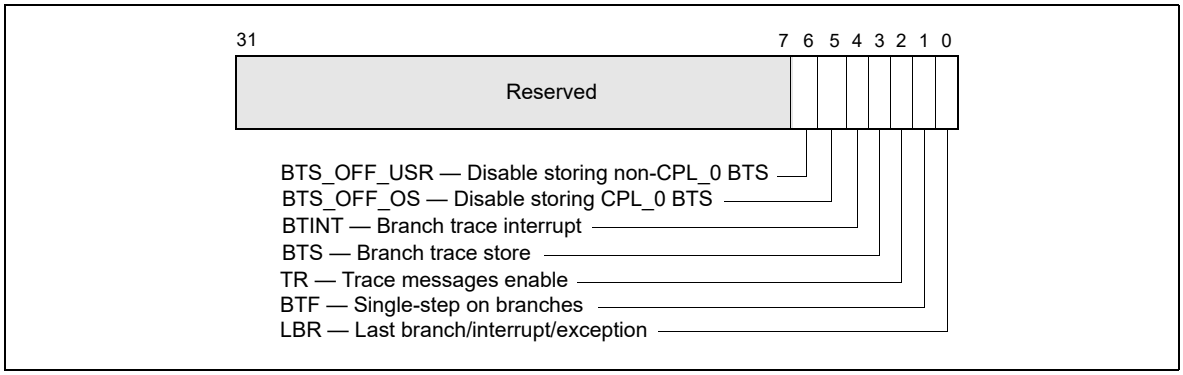


Figure 17-12. MSR_DEBUGCTLA MSR for Pentium 4 and Intel Xeon Processors

- **BTS (branch trace store) flag (bit 3) —** When set, enables the BTS facilities to log BTMs to a memory-resident BTS buffer that is part of the DS save area. See Section 17.4.9, "BTS and DS Save Area."
- **BTINT (branch trace interrupt) flag (bits 4) —** When set, the BTS facilities generate an interrupt when the BTS buffer is full. When clear, BTMs are logged to the BTS buffer in a circular fashion. See Section 17.4.5, "Branch Trace Store (BTS)."
- **BTS_OFF_OS (disable ring 0 branch trace store) flag (bit 5) —** When set, enables the BTS facilities to skip sending/logging CPL_0 BTMs to the memory-resident BTS buffer. See Section 17.13.2, "LBR Stack for Processors Based on Intel NetBurst® Microarchitecture."
- **BTS_OFF_USR (disable ring 0 branch trace store) flag (bit 6) —** When set, enables the BTS facilities to skip sending/logging non-CPL_0 BTMs to the memory-resident BTS buffer. See Section 17.13.2, "LBR Stack for Processors Based on Intel NetBurst® Microarchitecture."

NOTE

The initial implementation of BTS_OFF_USR and BTS_OFF_OS in MSR_DEBUGCTLA is shown in Figure 17-12. The BTS_OFF_USR and BTS_OFF_OS fields may be implemented on other model-specific debug control register at different locations.

See Chapter 2, "Model-Specific Registers (MSRs)" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 4* for a detailed description of each of the last branch recording MSRs.

17.13.2 LBR Stack for Processors Based on Intel NetBurst® Microarchitecture

The LBR stack is made up of LBR MSRs that are treated by the processor as a circular stack. The TOS pointer (MSR_LASTBRANCH_TOS MSR) points to the LBR MSR (or LBR MSR pair) that contains the most recent (last) branch record placed on the stack. Prior to placing a new branch record on the stack, the TOS is incremented by 1. When the TOS pointer reaches it maximum value, it wraps around to 0. See Table 17-17 and Figure 17-12.

Table 17-17. LBR MSR Stack Size and TOS Pointer Range for the Pentium® 4 and the Intel® Xeon® Processor Family

DisplayFamily_DisplayModel	Size of LBR Stack	Range of TOS Pointer
Family 0FH, Models 0H-02H; MSRs at locations 1DBH-1DEH.	4	0 to 3
Family 0FH, Models; MSRs at locations 680H-68FH.	16	0 to 15
Family 0FH, Model 03H; MSRs at locations 6C0H-6CFH.	16	0 to 15

The registers in the LBR MSR stack and the MSR_LASTBRANCH_TOS MSR are read-only and can be read using the RDMSR instruction.

Figure 17-13 shows the layout of a branch record in an LBR MSR (or MSR pair). Each branch record consists of two linear addresses, which represent the "from" and "to" instruction pointers for a branch, interrupt, or exception. The contents of the from and to addresses differ, depending on the source of the branch:

- **Taken branch** — If the record is for a taken branch, the "from" address is the address of the branch instruction and the "to" address is the target instruction of the branch.
- **Interrupt** — If the record is for an interrupt, the "from" address the return instruction pointer (RIP) saved for the interrupt and the "to" address is the address of the first instruction in the interrupt handler routine. The RIP is the linear address of the next instruction to be executed upon returning from the interrupt handler.
- **Exception** — If the record is for an exception, the "from" address is the linear address of the instruction that caused the exception to be generated and the "to" address is the address of the first instruction in the exception handler routine.

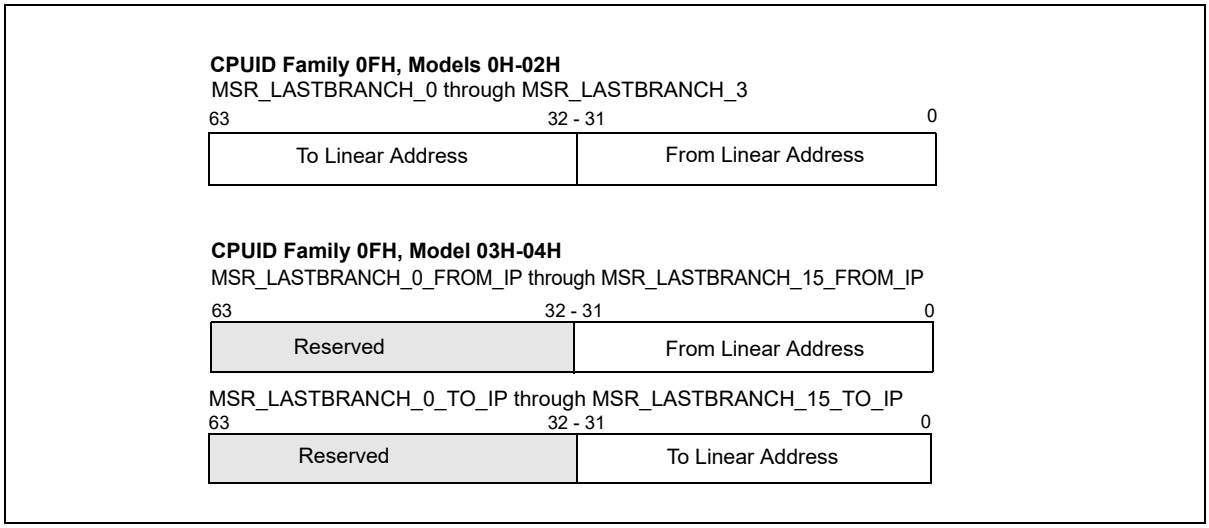


Figure 17-13. LBR MSR Branch Record Layout for the Pentium 4 and Intel Xeon Processor Family

Additional information is saved if an exception or interrupt occurs in conjunction with a branch instruction. If a branch instruction generates a trap type exception, two branch records are stored in the LBR stack: a branch record for the branch instruction followed by a branch record for the exception.

If a branch instruction is immediately followed by an interrupt, a branch record is stored in the LBR stack for the branch instruction followed by a record for the interrupt.

17.13.3 Last Exception Records

The Pentium 4, Intel Xeon, Pentium M, Intel® Core™ Solo, Intel® Core™ Duo, Intel® Core™2 Duo, Intel® Core™ i7 and Intel® Atom™ processors provide two MSRs (the MSR_LER_TO_LIP and the MSR_LER_FROM_LIP MSRs) that duplicate the functions of the LastExceptionToIP and LastExceptionFromIP MSRs found in the P6 family processors. The MSR_LER_TO_LIP and MSR_LER_FROM_LIP MSRs contain a branch record for the last branch that the processor took prior to an exception or interrupt being generated.

17.14 LAST BRANCH, INTERRUPT, AND EXCEPTION RECORDING (INTEL® CORE™ SOLO AND INTEL® CORE™ DUO PROCESSORS)

Intel Core Solo and Intel Core Duo processors provide last branch interrupt and exception recording. This capability is almost identical to that found in Pentium 4 and Intel Xeon processors. There are differences in the stack and in some MSR names and locations.

Note the following:

- **IA32_DEBUGCTL MSR —** Enables debug trace interrupt, debug trace store, trace messages enable, performance monitoring breakpoint flags, single stepping on branches, and last branch. IA32_DEBUGCTL MSR is located at register address 01D9H.

  See Figure 17-14 for the layout and the entries below for a description of the flags:

  — **LBR (last branch / interrupt / exception) flag (bit 0) —** When set, the processor records a running trace of the most recent branches, interrupts, and/or exceptions taken by the processor (prior to a debug exception being generated) in the last branch record (LBR) stack. For more information, see the "Last Branch Record (LBR) Stack" below.

  — **BTF (single-step on branches) flag (bit 1) —** When set, the processor treats the TF flag in the EFLAGS register as a "single-step on branches" flag rather than a "single-step on instructions" flag. This mechanism

allows single-stepping the processor on taken branches. See Section 17.4.3, "Single-Stepping on Branches," for more information about the BTF flag.

— **TR (trace message enable) flag (bit 6) —** When set, branch trace messages are enabled. When the processor detects a taken branch, interrupt, or exception; it sends the branch record out on the system bus as a branch trace message (BTM). See Section 17.4.4, "Branch Trace Messages," for more information about the TR flag.

— **BTS (branch trace store) flag (bit 7) —** When set, the flag enables BTS facilities to log BTMs to a memory-resident BTS buffer that is part of the DS save area. See Section 17.4.9, "BTS and DS Save Area."

— **BTINT (branch trace interrupt) flag (bits 8) —** When set, the BTS facilities generate an interrupt when the BTS buffer is full. When clear, BTMs are logged to the BTS buffer in a circular fashion. See Section 17.4.5, "Branch Trace Store (BTS)," for a description of this mechanism.

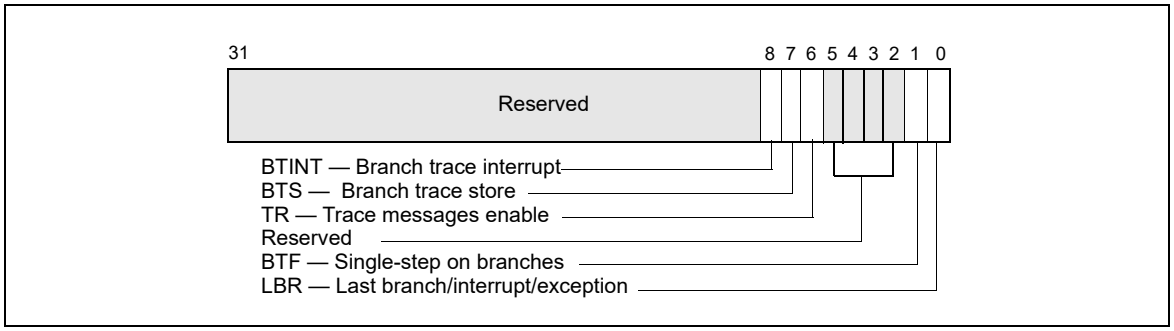


Figure 17-14. IA32_DEBUGCTL MSR for Intel Core Solo and Intel Core Duo Processors

- **Debug store (DS) feature flag (bit 21), returned by the CPUID instruction** — Indicates that the processor provides the debug store (DS) mechanism, which allows BTMs to be stored in a memory-resident BTS buffer. See Section 17.4.5, "Branch Trace Store (BTS)."
- **Last Branch Record (LBR) Stack** — The LBR stack consists of 8 MSRs (MSR_LASTBRANCH_0 through MSR_LASTBRANCH_7); bits 31-0 hold the 'from' address, bits 63-32 hold the 'to' address (MSR addresses start at 40H). See Figure 17-15.
- **Last Branch Record Top-of-Stack (TOS) Pointer** — The TOS Pointer MSR contains a 3-bit pointer (bits 2-0) to the MSR in the LBR stack that contains the most recent branch, interrupt, or exception recorded. For Intel Core Solo and Intel Core Duo processors, this MSR is located at register address 01C9H.

For compatibility, the Intel Core Solo and Intel Core Duo processors provide two 32-bit MSRs (the MSR_LER_TO_LIP and the MSR_LER_FROM_LIP MSRs) that duplicate functions of the LastExceptionToIP and LastExceptionFromIP MSRs found in P6 family processors.

For details, see Section 17.12, "Last Branch, Call Stack, Interrupt, and Exception Recording for Processors based on Skylake Microarchitecture," and Section 2.19, "MSRs In Intel® Core™ Solo and Intel® Core™ Duo Processors" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 4*.

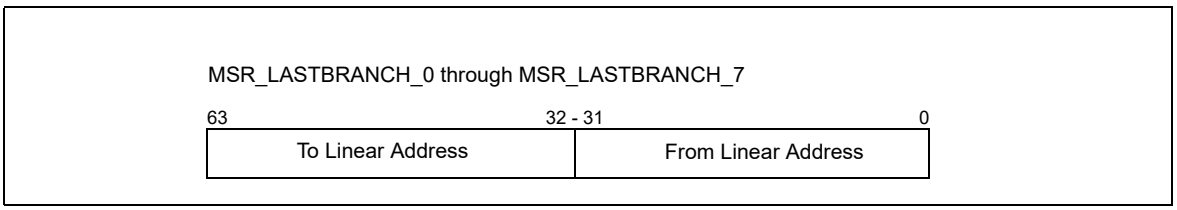


Figure 17-15. LBR Branch Record Layout for the Intel Core Solo and Intel Core Duo Processor

17.15 LAST BRANCH, INTERRUPT, AND EXCEPTION RECORDING (PENTIUM M PROCESSORS)

Like the Pentium 4 and Intel Xeon processor family, Pentium M processors provide last branch interrupt and exception recording. The capability operates almost identically to that found in Pentium 4 and Intel Xeon processors. There are differences in the shape of the stack and in some MSR names and locations. Note the following:

- **MSR_DEBUGCTLB MSR —** Enables debug trace interrupt, debug trace store, trace messages enable, performance monitoring breakpoint flags, single stepping on branches, and last branch. For Pentium M processors, this MSR is located at register address 01D9H. See Figure 17-16 and the entries below for a description of the flags.
  - **LBR (last branch / interrupt / exception) flag (bit 0) —** When set, the processor records a running trace of the most recent branches, interrupts, and/or exceptions taken by the processor (prior to a debug exception being generated) in the last branch record (LBR) stack. For more information, see the "Last Branch Record (LBR) Stack" bullet below.
  - **BTF (single-step on branches) flag (bit 1) —** When set, the processor treats the TF flag in the EFLAGS register as a "single-step on branches" flag rather than a "single-step on instructions" flag. This mechanism allows single-stepping the processor on taken branches. See Section 17.4.3, "Single-Stepping on Branches," for more information about the BTF flag.
  - **PB*i* (performance monitoring / breakpoint pins) flags (bits 5-2) —** When these flags are set, the performance monitoring/breakpoint pins on the processor (BP0#, BP1#, BP2#, and BP3#) report breakpoint matches in the corresponding breakpoint-address registers (DR0 through DR3). The processor asserts then deasserts the corresponding BP*i*# pin when a breakpoint match occurs. When a PB*i* flag is clear, the performance monitoring/breakpoint pins report performance events. Processor execution is not affected by reporting performance events.
  - **TR (trace message enable) flag (bit 6) —** When set, branch trace messages are enabled. When the processor detects a taken branch, interrupt, or exception, it sends the branch record out on the system bus as a branch trace message (BTM). See Section 17.4.4, "Branch Trace Messages," for more information about the TR flag.
  - **BTS (branch trace store) flag (bit 7) —** When set, enables the BTS facilities to log BTMs to a memory-resident BTS buffer that is part of the DS save area. See Section 17.4.9, "BTS and DS Save Area."
  - **BTINT (branch trace interrupt) flag (bits 8) —** When set, the BTS facilities generate an interrupt when the BTS buffer is full. When clear, BTMs are logged to the BTS buffer in a circular fashion. See Section 17.4.5, "Branch Trace Store (BTS)," for a description of this mechanism.

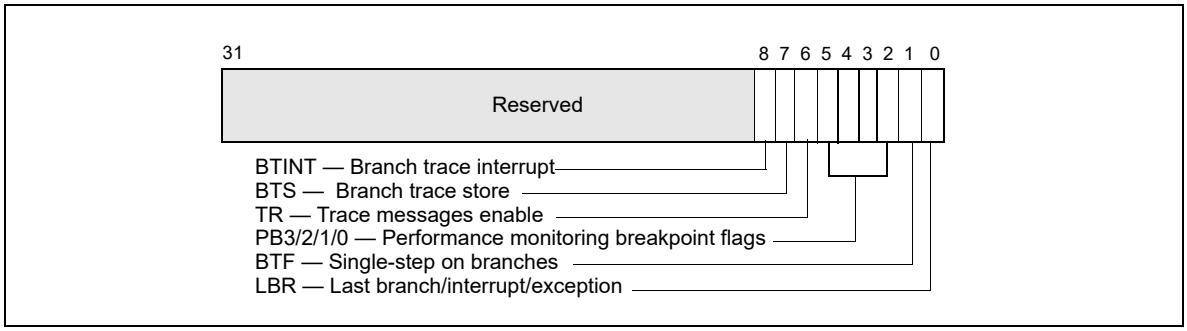


Figure 17-16. MSR_DEBUGCTLB MSR for Pentium M Processors

- **Debug store (DS) feature flag (bit 21), returned by the CPUID instruction** — Indicates that the processor provides the debug store (DS) mechanism, which allows BTMs to be stored in a memory-resident BTS buffer. See Section 17.4.5, "Branch Trace Store (BTS)."

- **Last Branch Record (LBR) Stack** — The LBR stack consists of 8 MSRs (MSR_LASTBRANCH_0 through MSR_LASTBRANCH_7); bits 31-0 hold the 'from' address, bits 63-32 hold the 'to' address. For Pentium M Processors, these pairs are located at register addresses 040H-047H. See Figure 17-17.
- **Last Branch Record Top-of-Stack (TOS) Pointer** — The TOS Pointer MSR contains a 3-bit pointer (bits 2-0) to the MSR in the LBR stack that contains the most recent branch, interrupt, or exception recorded. For Pentium M Processors, this MSR is located at register address 01C9H.

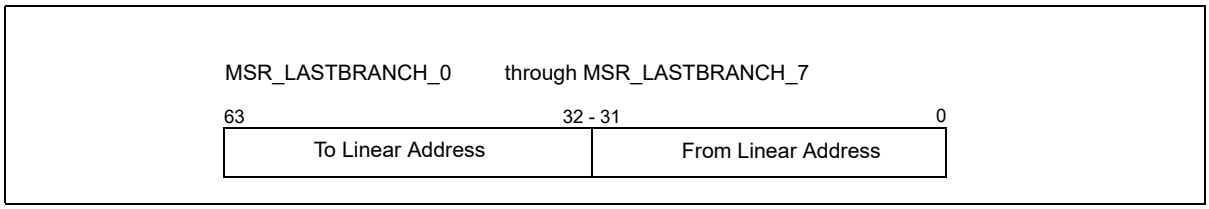


Figure 17-17. LBR Branch Record Layout for the Pentium M Processor

For more detail on these capabilities, see Section 17.13.3, "Last Exception Records," and Section 2.20, "MSRs In the Pentium M Processor" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 4*.

17.16 LAST BRANCH, INTERRUPT, AND EXCEPTION RECORDING (P6 FAMILY PROCESSORS)

The P6 family processors provide five MSRs for recording the last branch, interrupt, or exception taken by the processor: DEBUGCTLMSR, LastBranchToIP, LastBranchFromIP, LastExceptionToIP, and LastExceptionFromIP. These registers can be used to collect last branch records, to set breakpoints on branches, interrupts, and exceptions, and to single-step from one branch to the next.

See Chapter 2, "Model-Specific Registers (MSRs)" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 4* for a detailed description of each of the last branch recording MSRs.

17.16.1 DEBUGCTLMSR Register

The version of the DEBUGCTLMSR register found in the P6 family processors enables last branch, interrupt, and exception recording; taken branch breakpoints; the breakpoint reporting pins; and trace messages. This register can be written to using the WRMSR instruction, when operating at privilege level 0 or when in real-address mode. A protected-mode operating system procedure is required to provide user access to this register. Figure 17-18 shows the flags in the DEBUGCTLMSR register for the P6 family processors. The functions of these flags are as follows:

- **LBR (last branch/interrupt/exception) flag (bit 0) —** When set, the processor records the source and target addresses (in the LastBranchToIP, LastBranchFromIP, LastExceptionToIP, and LastExceptionFromIP MSRs) for the last branch and the last exception or interrupt taken by the processor prior to a debug exception being generated. The processor clears this flag whenever a debug exception, such as an instruction or data breakpoint or single-step trap occurs.

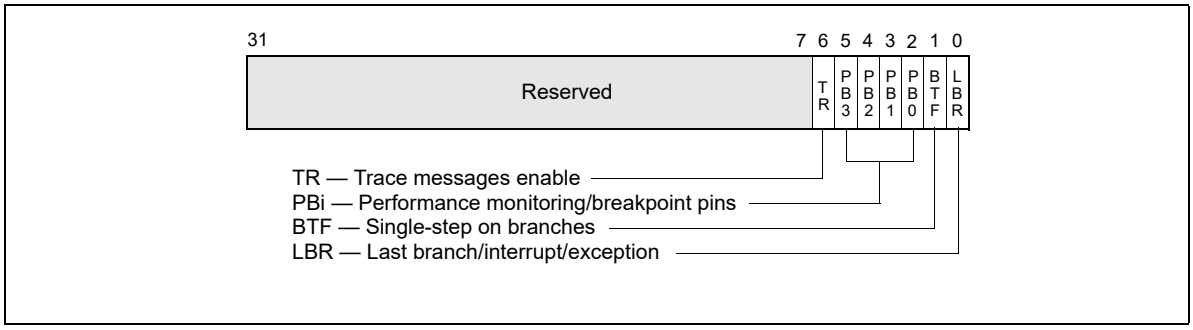


Figure 17-18. DEBUGCTLMSR Register (P6 Family Processors)

- **BTF (single-step on branches) flag (bit 1) —** When set, the processor treats the TF flag in the EFLAGS register as a "single-step on branches" flag. See Section 17.4.3, "Single-Stepping on Branches."
- **PB*i* (performance monitoring/breakpoint pins) flags (bits 2 through 5) —** When these flags are set, the performance monitoring/breakpoint pins on the processor (BP0#, BP1#, BP2#, and BP3#) report breakpoint matches in the corresponding breakpoint-address registers (DR0 through DR3). The processor asserts then deasserts the corresponding BP*i*# pin when a breakpoint match occurs. When a PB*i* flag is clear, the performance monitoring/breakpoint pins report performance events. Processor execution is not affected by reporting performance events.
- **TR (trace message enable) flag (bit 6) —** When set, trace messages are enabled as described in Section 17.4.4, "Branch Trace Messages." Setting this flag greatly reduces the performance of the processor. When trace messages are enabled, the values stored in the LastBranchToIP, LastBranchFromIP, LastExceptionToIP, and LastExceptionFromIP MSRs are undefined.

17.16.2 Last Branch and Last Exception MSRs

The LastBranchToIP and LastBranchFromIP MSRs are 32-bit registers for recording the instruction pointers for the last branch, interrupt, or exception that the processor took prior to a debug exception being generated. When a branch occurs, the processor loads the address of the branch instruction into the LastBranchFromIP MSR and loads the target address for the branch into the LastBranchToIP MSR.

When an interrupt or exception occurs (other than a debug exception), the address of the instruction that was interrupted by the exception or interrupt is loaded into the LastBranchFromIP MSR and the address of the exception or interrupt handler that is called is loaded into the LastBranchToIP MSR.

The LastExceptionToIP and LastExceptionFromIP MSRs (also 32-bit registers) record the instruction pointers for the last branch that the processor took prior to an exception or interrupt being generated. When an exception or interrupt occurs, the contents of the LastBranchToIP and LastBranchFromIP MSRs are copied into these registers before the to and from addresses of the exception or interrupt are recorded in the LastBranchToIP and LastBranchFromIP MSRs.

These registers can be read using the RDMSR instruction.

Note that the values stored in the LastBranchToIP, LastBranchFromIP, LastExceptionToIP, and LastExceptionFromIP MSRs are offsets into the current code segment, as opposed to linear addresses, which are saved in last branch records for the Pentium 4 and Intel Xeon processors.

17.16.3 Monitoring Branches, Exceptions, and Interrupts

When the LBR flag in the DEBUGCTLMSR register is set, the processor automatically begins recording branches that it takes, exceptions that are generated (except for debug exceptions), and interrupts that are serviced. Each time a branch, exception, or interrupt occurs, the processor records the to and from instruction pointers in the LastBranchToIP and LastBranchFromIP MSRs. In addition, for interrupts and exceptions, the processor copies the contents of the LastBranchToIP and LastBranchFromIP MSRs into the LastExceptionToIP and LastExceptionFromIP MSRs prior to recording the to and from addresses of the interrupt or exception.

When the processor generates a debug exception (#DB), it automatically clears the LBR flag before executing the exception handler, but does not touch the last branch and last exception MSRs. The addresses for the last branch, interrupt, or exception taken are thus retained in the LastBranchToIP and LastBranchFromIP MSRs and the addresses of the last branch prior to an interrupt or exception are retained in the LastExceptionToIP, and LastExceptionFromIP MSRs.

The debugger can use the last branch, interrupt, and/or exception addresses in combination with code-segment selectors retrieved from the stack to reset breakpoints in the breakpoint-address registers (DR0 through DR3), allowing a backward trace from the manifestation of a particular bug toward its source. Because the instruction pointers recorded in the LastBranchToIP, LastBranchFromIP, LastExceptionToIP, and LastExceptionFromIP MSRs are offsets into a code segment, software must determine the segment base address of the code segment associated with the control transfer to calculate the linear address to be placed in the breakpoint-address registers. The segment base address can be determined by reading the segment selector for the code segment from the stack and using it to locate the segment descriptor for the segment in the GDT or LDT. The segment base address can then be read from the segment descriptor.

Before resuming program execution from a debug-exception handler, the handler must set the LBR flag again to re-enable last branch and last exception/interrupt recording.

17.17 TIME-STAMP COUNTER

The Intel 64 and IA-32 architectures (beginning with the Pentium processor) define a time-stamp counter mechanism that can be used to monitor and identify the relative time occurrence of processor events. The counter's architecture includes the following components:

- **TSC flag —** A feature bit that indicates the availability of the time-stamp counter. The counter is available in an if the function CPUID.1:EDX.TSC[bit 4] = 1.
- **IA32_TIME_STAMP_COUNTER MSR** (called TSC MSR in P6 family and Pentium processors) **—** The MSR used as the counter.
- **RDTSC instruction —** An instruction used to read the time-stamp counter.
- **TSD flag —** A control register flag is used to enable or disable the time-stamp counter (enabled if CR4.TSD[bit 2] = 1).

The time-stamp counter (as implemented in the P6 family, Pentium, Pentium M, Pentium 4, Intel Xeon, Intel Core Solo and Intel Core Duo processors and later processors) is a 64-bit counter that is set to 0 following a RESET of the processor. Following a RESET, the counter increments even when the processor is halted by the HLT instruction or the external STPCLK# pin. Note that the assertion of the external DPSLP# pin may cause the time-stamp counter to stop.

Processor families increment the time-stamp counter differently:

- For Pentium M processors (family [06H], models [09H, 0DH]); for Pentium 4 processors, Intel Xeon processors (family [0FH], models [00H, 01H, or 02H]); and for P6 family processors: the time-stamp counter increments with every internal processor clock cycle.

  The internal processor clock cycle is determined by the current core-clock to bus-clock ratio. Intel® SpeedStep® technology transitions may also impact the processor clock.
- For Pentium 4 processors, Intel Xeon processors (family [0FH], models [03H and higher]); for Intel Core Solo and Intel Core Duo processors (family [06H], model [0EH]); for the Intel Xeon processor 5100 series and Intel Core 2 Duo processors (family [06H], model [0FH]); for Intel Core 2 and Intel Xeon processors (family [06H], DisplayModel [17H]); for Intel Atom processors (family [06H], DisplayModel [1CH]): the time-stamp counter increments at a constant rate. That rate may be set by the maximum core-clock to bus-clock ratio of the processor or may be set by the maximum resolved frequency at which the processor is booted. The maximum resolved frequency may differ from the processor base frequency, see Section 18.7.2 for more detail. On certain processors, the TSC frequency may not be the same as the frequency in the brand string.

  The specific processor configuration determines the behavior. Constant TSC behavior ensures that the duration of each clock tick is uniform and supports the use of the TSC as a wall clock timer even if the processor core changes frequency. This is the architectural behavior moving forward.

NOTE

To determine average processor clock frequency, Intel recommends the use of performance monitoring logic to count processor core clocks over the period of time for which the average is required. See Section 18.6.4.5, "Counting Clocks on systems with Intel Hyper-Threading Technology in Processors Based on Intel NetBurst® Microarchitecture," and Chapter 19, "Performance-Monitoring Events," for more information.

The RDTSC instruction reads the time-stamp counter and is guaranteed to return a monotonically increasing unique value whenever executed, except for a 64-bit counter wraparound. Intel guarantees that the time-stamp counter will not wraparound within 10 years after being reset. The period for counter wrap is longer for Pentium 4, Intel Xeon, P6 family, and Pentium processors.

Normally, the RDTSC instruction can be executed by programs and procedures running at any privilege level and in virtual-8086 mode. The TSD flag allows use of this instruction to be restricted to programs and procedures running at privilege level 0. A secure operating system would set the TSD flag during system initialization to disable user access to the time-stamp counter. An operating system that disables user access to the time-stamp counter should emulate the instruction through a user-accessible programming interface.

The RDTSC instruction is not serializing or ordered with other instructions. It does not necessarily wait until all previous instructions have been executed before reading the counter. Similarly, subsequent instructions may begin execution before the RDTSC instruction operation is performed.

The RDMSR and WRMSR instructions read and write the time-stamp counter, treating the time-stamp counter as an ordinary MSR (address 10H). In the Pentium 4, Intel Xeon, and P6 family processors, all 64-bits of the time-stamp counter are read using RDMSR (just as with RDTSC). When WRMSR is used to write the time-stamp counter on processors before family [0FH], models [03H, 04H]: only the low-order 32-bits of the time-stamp counter can be written (the high-order 32 bits are cleared to 0). For family [0FH], models [03H, 04H, 06H]; for family [06H]], model [0EH, 0FH]; for family [06H]], DisplayModel [17H, 1AH, 1CH, 1DH]: all 64 bits are writable.

17.17.1 Invariant TSC

The time stamp counter in newer processors may support an enhancement, referred to as invariant TSC. Processor's support for invariant TSC is indicated by CPUID.80000007H:EDX[8].

The invariant TSC will run at a constant rate in all ACPI P-, C-. and T-states. This is the architectural behavior moving forward. On processors with invariant TSC support, the OS may use the TSC for wall clock timer services (instead of ACPI or HPET timers). TSC reads are much more efficient and do not incur the overhead associated with a ring transition or access to a platform resource.

17.17.2 IA32_TSC_AUX Register and RDTSCP Support

Processors based on Intel microarchitecture code name Nehalem provide an auxiliary TSC register, IA32_TSC_AUX that is designed to be used in conjunction with IA32_TSC. IA32_TSC_AUX provides a 32-bit field that is initialized by privileged software with a signature value (for example, a logical processor ID).

The primary usage of IA32_TSC_AUX in conjunction with IA32_TSC is to allow software to read the 64-bit time stamp in IA32_TSC and signature value in IA32_TSC_AUX with the instruction RDTSCP in an atomic operation. RDTSCP returns the 64-bit time stamp in EDX:EAX and the 32-bit TSC_AUX signature value in ECX. The atomicity of RDTSCP ensures that no context switch can occur between the reads of the TSC and TSC_AUX values.

Support for RDTSCP is indicated by CPUID.80000001H:EDX[27]. As with RDTSC instruction, non-ring 0 access is controlled by CR4.TSD (Time Stamp Disable flag).

User mode software can use RDTSCP to detect if CPU migration has occurred between successive reads of the TSC. It can also be used to adjust for per-CPU differences in TSC values in a NUMA system.

17.17.3 Time-Stamp Counter Adjustment

Software can modify the value of the time-stamp counter (TSC) of a logical processor by using the WRMSR instruction to write to the IA32_TIME_STAMP_COUNTER MSR (address 10H). Because such a write applies only to that logical processor, software seeking to synchronize the TSC values of multiple logical processors must perform these writes on each logical processor. It may be difficult for software to do this in a way than ensures that all logical processors will have the same value for the TSC at a given point in time.

The synchronization of TSC adjustment can be simplified by using the 64-bit IA32_TSC_ADJUST MSR (address 3BH). Like the IA32_TIME_STAMP_COUNTER MSR, the IA32_TSC_ADJUST MSR is maintained separately for each logical processor. A logical processor maintains and uses the IA32_TSC_ADJUST MSR as follows:

- On RESET, the value of the IA32_TSC_ADJUST MSR is 0.
- If an execution of WRMSR to the IA32_TIME_STAMP_COUNTER MSR adds (or subtracts) value X from the TSC, the logical processor also adds (or subtracts) value X from the IA32_TSC_ADJUST MSR.
- If an execution of WRMSR to the IA32_TSC_ADJUST MSR adds (or subtracts) value X from that MSR, the logical processor also adds (or subtracts) value X from the TSC.

Unlike the TSC, the value of the IA32_TSC_ADJUST MSR changes only in response to WRMSR (either to the MSR itself, or to the IA32_TIME_STAMP_COUNTER MSR). Its value does not otherwise change as time elapses. Software seeking to adjust the TSC can do so by using WRMSR to write the same value to the IA32_TSC_ADJUST MSR on each logical processor.

Processor support for the IA32_TSC_ADJUST MSR is indicated by CPUID.(EAX=07H, ECX=0H):EBX.TSC_ADJUST (bit 1).

17.17.4 Invariant Time-Keeping

The invariant TSC is based on the invariant timekeeping hardware (called Always Running Timer or ART), that runs at the core crystal clock frequency. The ratio defined by CPUID leaf 15H expresses the frequency relationship between the ART hardware and TSC.

If CPUID.15H:EBX[31:0] != 0 and CPUID.80000007H:EDX[InvariantTSC] = 1, the following linearity relationship holds between TSC and the ART hardware:

TSC_Value = (ART_Value * CPUID.15H:EBX[31:0])/ CPUID.15H:EAX[31:0] + K

Where 'K' is an offset that can be adjusted by a privileged agent[2].

When ART hardware is reset, both invariant TSC and K are also reset.

17.18 INTEL® RESOURCE DIRECTOR TECHNOLOGY (INTEL® RDT) MONITORING FEATURES

The Intel Resource Director Technology (Intel RDT) feature set provides a set of monitoring capabilities including Cache Monitoring Technology (CMT) and Memory Bandwidth Monitoring (MBM). The Intel® Xeon® processor E5 v3 family introduced resource monitoring capability in each logical processor to measure specific platform shared resource metrics, for example, L3 cache occupancy. The programming interface for these monitoring features is described in this section. Two features within the monitoring feature set provided are described - Cache Monitoring Technology (CMT) and Memory Bandwidth Monitoring.

Cache Monitoring Technology (CMT) allows an Operating System, Hypervisor or similar system management agent to determine the usage of cache by applications running on the platform. The initial implementation is directed at L3 cache monitoring (currently the last level cache in most server platforms).

Memory Bandwidth Monitoring (MBM), introduced in the Intel® Xeon® processor E5 v4 family, builds on the CMT infrastructure to allow monitoring of bandwidth from one level of the cache hierarchy to the next - in this case

2. IA32_TSC_ADJUST MSR and the TSC-offset field in the VM execution controls of VMCS are some of the common interfaces that privileged software can use to manage the time stamp counter for keeping time

focusing on the L3 cache, which is typically backed directly by system memory. As a result of this implementation, memory bandwidth can be monitored.

The monitoring mechanisms described provide the following key shared infrastructure features:

- A mechanism to enumerate the presence of the monitoring capabilities within the platform (via a CPUID feature bit).
- A framework to enumerate the details of each sub-feature (including CMT and MBM, as discussed later, via CPUID leaves and sub-leaves).
- A mechanism for the OS or Hypervisor to indicate a software-defined ID for each of the software threads (applications, virtual machines, etc.) that are scheduled to run on a logical processor. These identifiers are known as Resource Monitoring IDs (RMIDs).
- Mechanisms in hardware to monitor cache occupancy and bandwidth statistics as applicable to a given product generation on a per software-id basis.
- Mechanisms for the OS or Hypervisor to read back the collected metrics such as L3 occupancy or Memory Bandwidth for a given software ID at any point during runtime.

17.18.1 Overview of Cache Monitoring Technology and Memory Bandwidth Monitoring

The shared resource monitoring features described in this chapter provide a layer of abstraction between applications and logical processors through the use of **Resource Monitoring ID**s (RMIDs). Each logical processor in the system can be assigned an RMID independently, or multiple logical processors can be assigned to the same RMID value (e.g., to track an application with multiple threads). For each logical processor, only one RMID value is active at a time. This is enforced by the IA32_PQR_ASSOC MSR, which specifies the active RMID of a logical processor. Writing to this MSR by software changes the active RMID of the logical processor from an old value to a new value.

The underlying platform shared resource monitoring hardware tracks cache metrics such as cache utilization and misses as a result of memory accesses according to the RMIDs and reports monitored data via a counter register (IA32_QM_CTR). The specific event types supported vary by generation and can be enumerated via CPUID. Before reading back monitored data software must configure an event selection MSR (IA32_QM_EVTSEL) to specify which metric is to be reported, and the specific RMID for which the data should be returned.

Processor support of the monitoring framework and sub-features such as CMT is reported via the CPUID instruction. The resource type available to the monitoring framework is enumerated via a new leaf function in CPUID. Reading and writing to the monitoring MSRs requires the RDMSR and WRMSR instructions.

The Cache Monitoring Technology feature set provides the following unique mechanisms:

- A mechanism to enumerate the presence and details of the CMT feature as applicable to a given level of the cache hierarchy, independent of other monitoring features.
- CMT-specific event codes to read occupancy for a given level of the cache hierarchy.

The Memory Bandwidth Monitoring feature provides the following unique mechanisms:

- A mechanism to enumerate the presence and details of the MBM feature as applicable to a given level of the cache hierarchy, independent of other monitoring features.
- MBM-specific event codes to read bandwidth out to the next level of the hierarchy and various sub-event codes to read more specific metrics as discussed later (e.g., total bandwidth vs. bandwidth only from local memory controllers on the same package).

17.18.2 Enabling Monitoring: Usage Flow

Figure 17-19 illustrates the key steps for OS/VMM to detect support of shared resource monitoring features such as CMT and enable resource monitoring for available resource types and monitoring events.

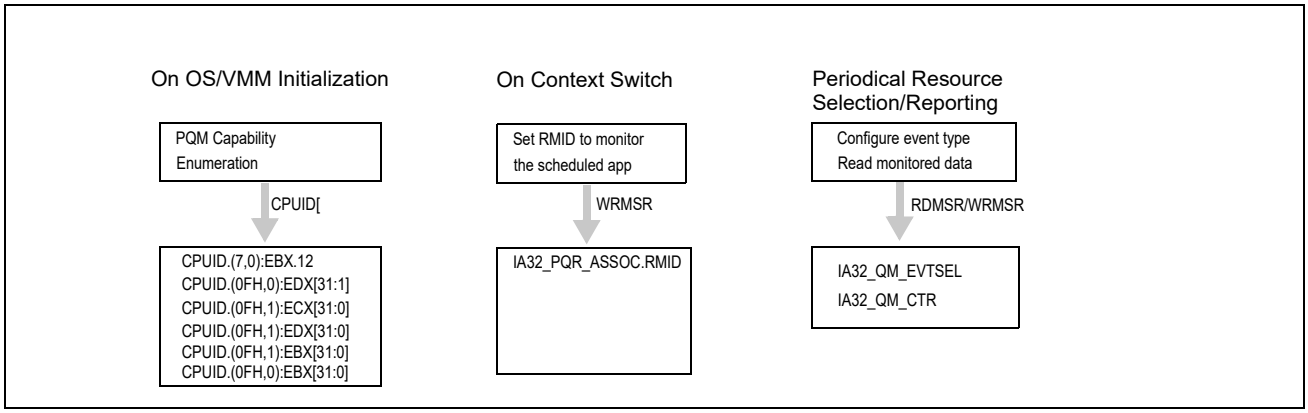


Figure 17-19. Platform Shared Resource Monitoring Usage Flow

17.18.3 Enumeration and Detecting Support of Cache Monitoring Technology and Memory Bandwidth Monitoring

Software can query processor support of shared resource monitoring features capabilities by executing CPUID instruction with EAX = 07H, ECX = 0H as input. If CPUID.(EAX=07H, ECX=0):EBX.PQM[bit 12] reports 1, the processor provides the following programming interfaces for shared resource monitoring, including Cache Monitoring Technology:

- CPUID leaf function 0FH (Shared Resource Monitoring Enumeration leaf) provides information on available resource types (see Section 17.18.4), and monitoring capabilities for each resource type (see Section 17.18.5). Note CMT and MBM capabilities are enumerated as separate event vectors using shared enumeration infrastructure under a given resource type.
- IA32_PQR_ASSOC.RMID: The per-logical-processor MSR, IA32_PQR_ASSOC, that OS/VMM can use to assign an RMID to each logical processor, see Section 17.18.6.
- IA32_QM_EVTSEL: This MSR specifies an Event ID (EvtID) and an RMID which the platform uses to look up and provide monitoring data in the monitoring counter, IA32_QM_CTR, see Section 17.18.7.
- IA32_QM_CTR: This MSR reports monitored resource data when available along with bits to allow software to check for error conditions and verify data validity.

Software must follow the following sequence of enumeration to discover Cache Monitoring Technology capabilities:

1. Execute CPUID with EAX=0 to discover the "cpuid_maxLeaf" supported in the processor;
2. If cpuid_maxLeaf >= 7, then execute CPUID with EAX=7, ECX= 0 to verify CPUID.(EAX=07H, ECX=0):EBX.PQM[bit 12] is set;
3. If CPUID.(EAX=07H, ECX=0):EBX.PQM[bit 12] = 1, then execute CPUID with EAX=0FH, ECX= 0 to query available resource types that support monitoring;
4. If CPUID.(EAX=0FH, ECX=0):EDX.L3[bit 1] = 1, then execute CPUID with EAX=0FH, ECX= 1 to query the specific capabilities of L3 Cache Monitoring Technology (CMT) and Memory Bandwidth Monitoring.
5. If CPUID.(EAX=0FH, ECX=0):EDX reports additional resource types supporting monitoring, then execute CPUID with EAX=0FH, ECX set to a corresponding resource type ID (ResID) as enumerated by the bit position of CPUID.(EAX=0FH, ECX=0):EDX.

17.18.4 Monitoring Resource Type and Capability Enumeration

CPUID leaf function 0FH (Shared Resource Monitoring Enumeration leaf) provides one sub-leaf (sub-function 0) that reports shared enumeration infrastructure, and one or more sub-functions that report feature-specific enumeration data:

- Monitoring leaf sub-function 0 enumerates available resources that support monitoring, i.e. executing CPUID with EAX=0FH and ECX=0H. In the initial implementation, L3 cache is the only resource type available. Each

supported resource type is represented by a bit in CPUID.(EAX=0FH, ECX=0):EDX[31:1]. The bit position corresponds to the sub-leaf index (ResID) that software must use to query details of the monitoring capability of that resource type (see Figure 17-21 and Figure 17-22). Reserved bits of CPUID.(EAX=0FH, ECX=0):EDX[31:2] correspond to unsupported sub-leaves of the CPUID.0FH leaf. Additionally, CPUID.(EAX=0FH, ECX=0H):EBX reports the highest RMID value of any resource type that supports monitoring in the processor.

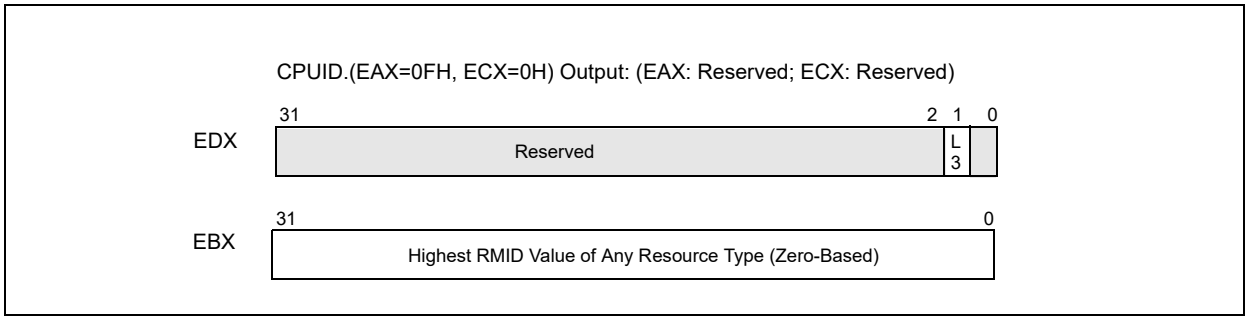


Figure 17-20. CPUID.(EAX=0FH, ECX=0H) Monitoring Resource Type Enumeration

17.18.5 Feature-Specific Enumeration

Each additional sub-leaf of CPUID.(EAX=0FH, ECX=ResID) enumerates the specific details for software to program Monitoring MSRs using the resource type associated with the given ResID.

Note that in future Monitoring implementations the meanings of the returned registers may vary in other sub-leaves that are not yet defined. The registers will be specified and defined on a per-ResID basis.

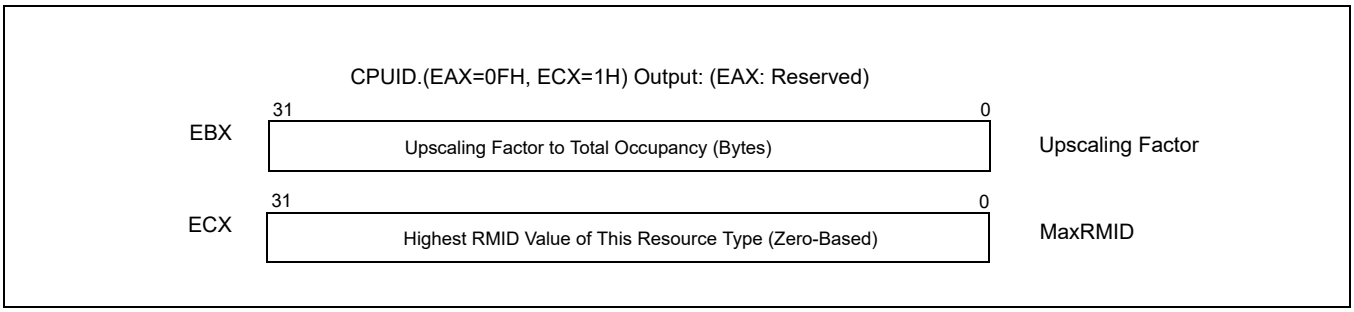


Figure 17-21. L3 Cache Monitoring Capability Enumeration Data (CPUID.(EAX=0FH, ECX=1H))

For each supported Cache Monitoring resource type, hardware supports only a finite number of RMIDs. CPUID.(EAX=0FH, ECX=1H).ECX enumerates the highest RMID value that can be monitored with this resource type, see Figure 17-21.

CPUID.(EAX=0FH, ECX=1H).EDX specifies a bit vector that is used to look up the EventID (See Figure 17-22 and Table 17-18) that software must program with IA32_QM_EVTSEL in order to retrieve event data. After software configures IA32_QMEVTSEL with the desired RMID and EventID, it can read the resulting data from IA32_QM_CTR. The raw numerical value reported from IA32_QM_CTR can be converted to the final value (occupancy in bytes or bandwidth in bytes per sampled time period) by multiplying the counter value by the value from CPUID.(EAX=0FH, ECX=1H).EBX, see Figure 17-21.

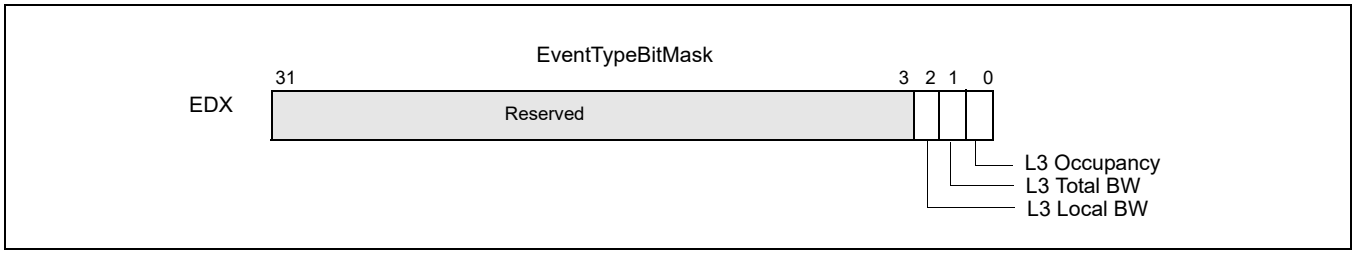


Figure 17-22. L3 Cache Monitoring Capability Enumeration Event Type Bit Vector (CPUID.(EAX=0FH, ECX=1H))

17.18.5.1 Cache Monitoring Technology

On processors for which Cache Monitoring Technology supports the L3 cache occupancy event, CPUID.(EAX=0FH, ECX=1H).EDX would return with only bit 0 set. The corresponding event ID can be looked up from Table 17-18. The L3 occupancy data accumulated in IA32_QM_CTR can be converted to total occupancy (in bytes) by multiplying with CPUID.(EAX=0FH, ECX=1H).EBX.

Event codes for Cache Monitoring Technology are discussed in the next section.

17.18.5.2 Memory Bandwidth Monitoring

On processors that monitoring supports Memory Bandwidth Monitoring using ResID=1 (L3), two additional bits will be set in the vector at CPUID.(EAX=0FH, ECX=1H).EDX:

- CPUID.(EAX=0FH, ECX=1H).EDX[bit 1]: indicates the L3 total external bandwidth monitoring event is supported if set. This event monitors the L3 total external bandwidth to the next level of the cache hierarchy, including all demand and prefetch misses from the L3 to the next hierarchy of the memory system. In most platforms, this represents memory bandwidth.
- CPUID.(EAX=0FH, ECX=1H).EDX[bit 2]: indicates L3 local memory bandwidth monitoring event is supported if set. This event monitors the L3 external bandwidth satisfied by the local memory. In most platforms that support this event, L3 requests are likely serviced by a memory system with non-uniform memory architecture. This allows bandwidth to off-package memory resources to be tracked by subtracting local from total bandwidth (for instance, bandwidth over QPI to a memory controller on another physical processor could be tracked by subtraction).

The corresponding Event ID can be looked up from Table 17-18. The L3 bandwidth data accumulated in IA32_QM_CTR can be converted to total bandwidth (in bytes) using CPUID.(EAX=0FH, ECX=1H).EBX.

Table 17-18. Monitoring Supported Event IDs

Event Type	Event ID	Context
L3 Cache Occupancy	01H	Cache Monitoring Technology
L3 Total External Bandwidth	02H	MBM
L3 Local External Bandwidth	03H	MBM
Reserved	All other event codes	N/A

17.18.6 Monitoring Resource RMID Association

After Monitoring and sub-features has been enumerated, software can begin using the monitoring features. The first step is to associate a given software thread (or multiple threads as part of an application, VM, group of applications or other abstraction) with an RMID.

Note that the process of associating an RMID with a given software thread is the same for all shared resource monitoring features (CMT, MBM), and a given RMID number has the same meaning from the viewpoint of any logical processors in a package. Stated another way, a thread may be associated in a 1:1 mapping with an RMID, and that

RMID may allow cache occupancy, memory bandwidth information or other monitoring data to be read back later with monitoring event codes (retrieving data is discussed in a previous section).

The association of an application thread with an RMID requires an OS to program the per-logical-processor MSR IA32_PQR_ASSOC at context swap time (updates may also be made at any other arbitrary points during program execution such as application phase changes). The IA32_PQR_ASSOC MSR specifies the active RMID that monitoring hardware will use to tag internal operations, such as L3 cache requests. The layout of the MSR is shown in Figure 17-23. Software specifies the active RMID to monitor in the IA32_PQR_ASSOC.RMID field. The width of the RMID field can vary from one implementation to another, and is derived from Ceil (LOG_2 (1 + CPUID.(EAX=0FH, ECX=0):EBX[31:0])). The value of IA32_PQR_ASSOC after power-on is 0.

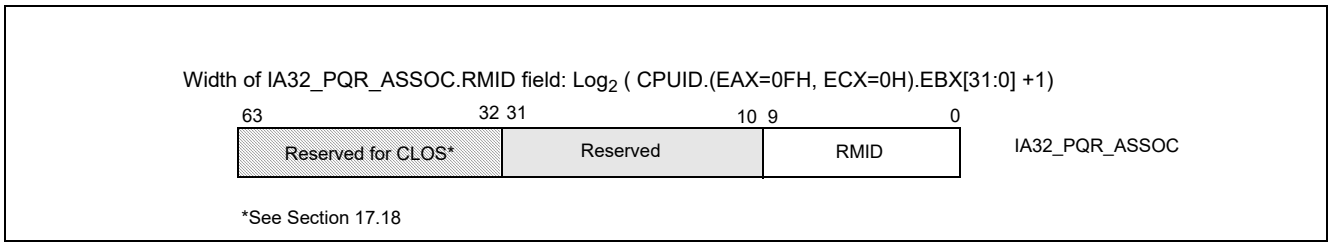


Figure 17-23. IA32_PQR_ASSOC MSR

In the initial implementation, the width of the RMID field is up to 10 bits wide, zero-referenced and fully encoded. However, software must use CPUID to query the maximum RMID supported by the processor. If a value larger than the maximum RMID is written to IA32_PQR_ASSOC.RMID, a #GP(0) fault will be generated.

RMIDs have a global scope within the physical package- if an RMID is assigned to one logical processor then the same RMID can be used to read multiple thread attributes later (for example, L3 cache occupancy or external bandwidth from the L3 to the next level of the cache hierarchy). In a multiple LLC platform the RMIDs are to be reassigned by the OS or VMM scheduler when an application is migrated across LLCs.

Note that in a situation where Monitoring supports multiple resource types, some upper range of RMIDs (e.g. RMID 31) may only be supported by one resource type but not by another resource type.

17.18.7 Monitoring Resource Selection and Reporting Infrastructure

The reporting mechanism for Cache Monitoring Technology and other related features is architecturally exposed as an MSR pair that can be programmed and read to measure various metrics such as the L3 cache occupancy (CMT) and bandwidths (MBM) depending on the level of Monitoring support provided by the platform. Data is reported back on a per-RMID basis. These events do not trigger based on event counts or trigger APIC interrupts (e.g. no Performance Monitoring Interrupt occurs based on counts). Rather, they are used to sample counts explicitly.

The MSR pair for the shared resource monitoring features (CMT, MBM) is separate from and not shared with architectural Perfmon counters, meaning software can use these monitoring features simultaneously with the Perfmon counters.

Access to the aggregated monitoring information is accomplished through the following programmable monitoring MSRs:

- IA32_QM_EVTSEL: This MSR provides a role similar to the event select MSRs for programmable performance monitoring described in Chapter 18. The simplified layout of the MSR is shown in Figure 17-24. Bits IA32_QM_EVTSEL.EvtID (bits 7:0) specify an event code of a supported resource type for hardware to report monitored data associated with IA32_QM_EVTSEL.RMID (bits 41:32). Software can configure IA32_QM_EVTSEL.RMID with any RMID that is active within the physical processor. The width of IA32_QM_EVTSEL.RMID matches that of IA32_PQR_ASSOC.RMID. Supported event codes for the IA32_QM_EVTSEL register are shown in Table 17-18. Note that valid event codes may not necessarily map directly to the bit position used to enumerate support for the resource via CPUID.

  Software can program an RMID / Event ID pair into the IA32_QM_EVTSEL MSR bit field to select an RMID to read a particular counter for a given resource. The currently supported list of Monitoring Event IDs is discussed in Section 17.18.5, which covers feature-specific details.

Thread access to the IA32_QM_EVTSEL and IA32_QM_CTR MSR pair should be serialized to avoid situations where one thread changes the RMID/EvtID just before another thread reads monitoring data from IA32_QM_CTR.

- IA32_QM_CTR: This MSR reports monitored data when available. It contains three bit fields. If software configures an unsupported RMID or event type in IA32_QM_EVTSEL, then IA32_QM_CTR.Error (bit 63) will be set, indicating there is no valid data to report. If IA32_QM_CTR.Unavailable (bit 62) is set, it indicates monitored data for the RMID is not available, and IA32_QM_CTR.data (bits 61:0) should be ignored. Therefore, IA32_QM_CTR.data (bits 61:0) is valid only if bit 63 and 62 are both clear. For Cache Monitoring Technology, software can convert IA32_QM_CTR.data into cache occupancy or bandwidth metrics expressed in bytes by multiplying with the conversion factor from CPUID.(EAX=0FH, ECX=1H).EBX.

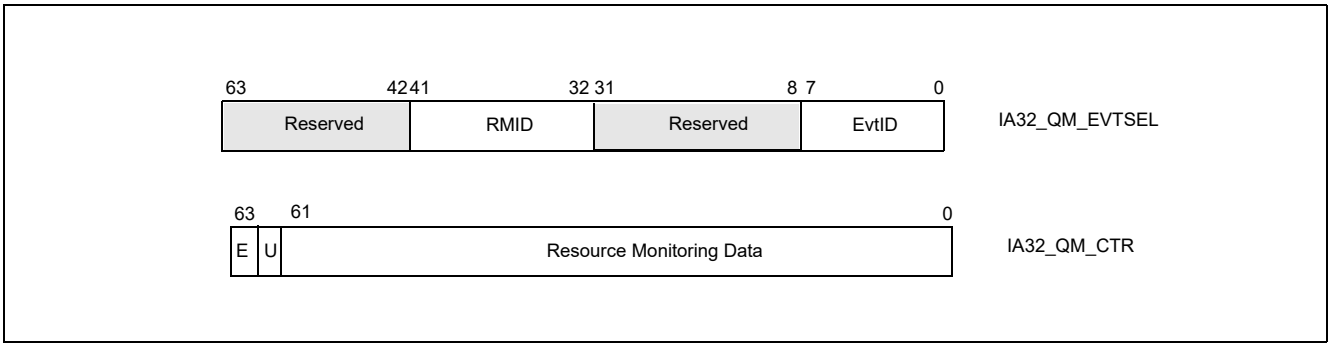


Figure 17-24. IA32_QM_EVTSEL and IA32_QM_CTR MSRs

17.18.8 Monitoring Programming Considerations

Figure 17-23 illustrates how system software can program IA32_QOSEVTSEL and IA32_QM_CTR to perform resource monitoring.

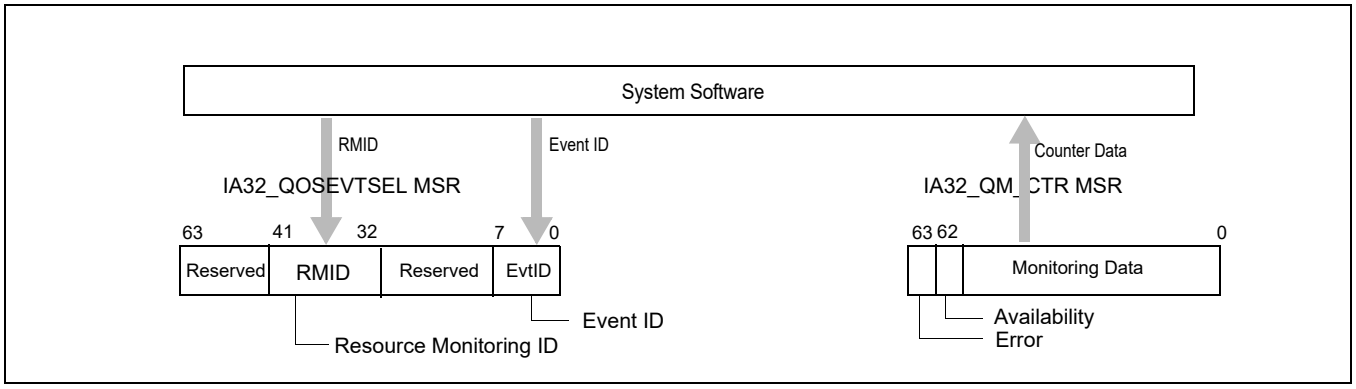


Figure 17-25. Software Usage of Cache Monitoring Resources

Though the field provided in IA32_QM_CTR allows for up to 62 bits of data to be returned, often a subset of bits are used. With Cache Monitoring Technology for instance, the number of bits used will be proportional to the base-two logarithm of the total cache size divided by the Upscaling Factor from CPUID.

In Memory Bandwidth Monitoring the initial counter size is 24 bits, and retrieving the value at 1Hz or faster is sufficient to ensure at most one rollover per sampling period. Any future changes to counter width will be enumerated to software.

17.18.8.1 Monitoring Dynamic Configuration

Both the IA32_QM_EVTSEL and IA32_PQR_ASSOC registers are accessible and modifiable at any time during execution using RDMSR/WRMSR unless otherwise noted. When writing to these MSRs a #GP(0) will be generated if any of the following conditions occur:

- A reserved bit is modified,
- An RMID exceeding the maxRMID is used.

17.18.8.2 Monitoring Operation With Power Saving Features

Note that some advanced power management features such as deep package C-states may shrink the L3 cache and cause CMT occupancy count to be reduced. MBM bandwidth counts may increase due to flushing cached data out of L3.

17.18.8.3 Monitoring Operation with Other Operating Modes

The states in IA32_PQR_ASSOC and monitoring counter are unmodified across an SMI delivery. Thus, the execution of SMM handler code and SMM handler's data can manifest as spurious contribution in the monitored data.

It is possible for an SMM handler to minimize the impact on of spurious contribution in the QOS monitoring counters by reserving a dedicated RMID for monitoring the SMM handler. Such an SMM handler can save the previously configured QOS Monitoring state immediately upon entering SMM, and restoring the QOS monitoring state back to the prev-SMM RMID upon exit.

17.18.8.4 Monitoring Operation with RAS Features

In general the Reliability, Availability and Serviceability (RAS) features present in Intel Platforms are not expected to significantly affect shared resource monitoring counts. In cases where software RAS features cause memory copies or cache accesses these may be tracked and may influence the shared resource monitoring counter values.

17.19 INTEL® RESOURCE DIRECTOR TECHNOLOGY (INTEL® RDT) ALLOCATION FEATURES

The Intel Resource Director Technology (Intel RDT) feature set provides a set of allocation (resource control) capabilities including Cache Allocation Technology (CAT) and Code and Data Prioritization (CDP). The Intel Xeon processor E5 v4 family (and subset of communication-focused Intel Xeon processors E5 v3 family) introduce capabilities to configure and make use of the Cache Allocation Technology (CAT) mechanisms on the L3 cache. Some future Intel platforms may also provide support for control over the L2 cache, with capabilities as described below. The programming interface for Cache Allocation Technology and for the more general allocation capabilities are described in the rest of this chapter.

Future Intel processors introduce the Memory Bandwidth Allocation (MBA) feature which provides indirect control over the memory bandwidth available to CPU cores, and is discussed later in this chapter.

17.19.1 Introduction to Cache Allocation Technology (CAT)

Cache Allocation Technology enables an Operating System (OS), Hypervisor /Virtual Machine Manager (VMM) or similar system service management agent to specify the amount of cache space into which an application can fill (as a hint to hardware - certain features such as power management may override CAT settings). Specialized user-level implementations with minimal OS support are also possible, though not necessarily recommended (see notes below for OS/Hypervisor with respect to ring 3 software and virtual guests). Depending on the processor family, L2 or L3 cache allocation capability may be provided, and the technology is designed to scale across multiple cache levels and technology generations.

Software can determine which levels are supported in a give platform programmatically using CPUID as described in the following sections.

The CAT mechanisms defined in this document provide the following key features:

- A mechanism to enumerate platform Cache Allocation Technology capabilities and available resource types that provides CAT control capabilities. For implementations that support Cache Allocation Technology, CPUID provides enumeration support to query which levels of the cache hierarchy are supported and specific CAT capabilities, such as the max allocation bitmask size,
- A mechanism for the OS or Hypervisor to configure the amount of a resource available to a particular Class of Service via a list of allocation bitmasks,
- Mechanisms for the OS or Hypervisor to signal the Class of Service to which an application belongs, and
- Hardware mechanisms to guide the LLC fill policy when an application has been designated to belong to a specific Class of Service.

Note that for many usages, an OS or Hypervisor may not want to expose Cache Allocation Technology mechanisms to Ring3 software or virtualized guests.

The Cache Allocation Technology feature enables more cache resources (i.e. cache space) to be made available for high priority applications based on guidance from the execution environment as shown in Figure 17-26. The architecture also allows dynamic resource reassignment during runtime to further optimize the performance of the high priority application with minimal degradation to the low priority app. Additionally, resources can be rebalanced for system throughput benefit across uses cases of OSes, VMMs, containers and other scenarios by managing the CPUID and MSR interfaces. This section describes the hardware and software support required in the platform including what is required of the execution environment (i.e. OS/VMM) to support such resource control. Note that in Figure 17-26 the L3 Cache is shown as an example resource.

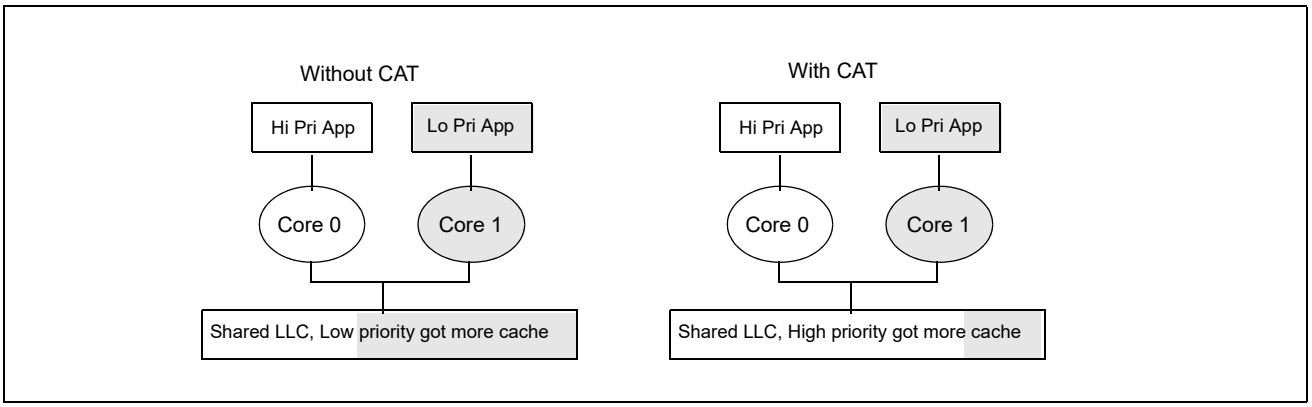


Figure 17-26. Cache Allocation Technology Allocates More Resource to High Priority Applications

17.19.2 Cache Allocation Technology Architecture

The fundamental goal of Cache Allocation Technology is to enable resource allocation based on application priority or Class of Service (COS or CLOS). The processor exposes a set of Classes of Service into which applications (or individual threads) can be assigned. Cache allocation for the respective applications or threads is then restricted based on the class with which they are associated. Each Class of Service can be configured using capacity bitmasks (CBMs) which represent capacity and indicate the degree of overlap and isolation between classes. For each logical processor there is a register exposed (referred to here as the IA32_PQR_ASSOC MSR or PQR) to allow the OS/VMM to specify a COS when an application, thread or VM is scheduled.

The usage of Classes of Service (COS) are consistent across resources - and a COS may have multiple re-source control attributes attached, which reduces software overhead at context swap time. Rather than adding new types of COS tags per resource for instance, the COS management overhead is constant. Cache allocation for the indicated application/thread/VM is then controlled automatically by the hardware based on the class and the bitmask associated with that class. Bitmasks are configured via the IA32_resourceType_MASK_n MSRs, where resourceType indicates a resource type (e.g. "L3" for the L3 cache) and n indicates a COS number.

The basic ingredients of Cache Allocation Technology are as follows:

- An architecturally exposed mechanism using CPUID to indicate whether CAT is supported, and what resource types are available which can be controlled,

- For each available resourceType, CPUID also enumerates the total number of Classes of Services and the length of the capacity bitmasks that can be used to enforce cache allocation to applications on the platform,
- An architecturally exposed mechanism to allow the execution environment (OS/VMM) to configure the behavior of different classes of service using the bitmasks available,
- An architecturally exposed mechanism to allow the execution environment (OS/VMM) to assign a COS to an executing software thread (i.e. associating the active CR3 of a logical processor with the COS in IA32_PQR_ASSOC),
- Implementation-dependent mechanisms to indicate which COS is associated with a memory access and to enforce the cache allocation on a per COS basis.

A capacity bitmask (CBM) provides a hint to the hardware indicating the cache space an application should be limited to as well as providing an indication of overlap and isolation in the CAT-capable cache from other applications contending for the cache. The bitlength of the capacity mask available generally depends on the configuration of the cache and is specified in the enumeration process for CAT in CPUID (this may vary between models in a processor family as well). Similarly, other parameters such as the number of supported COS may vary for each resource type, and these details can be enumerated via CPUID.

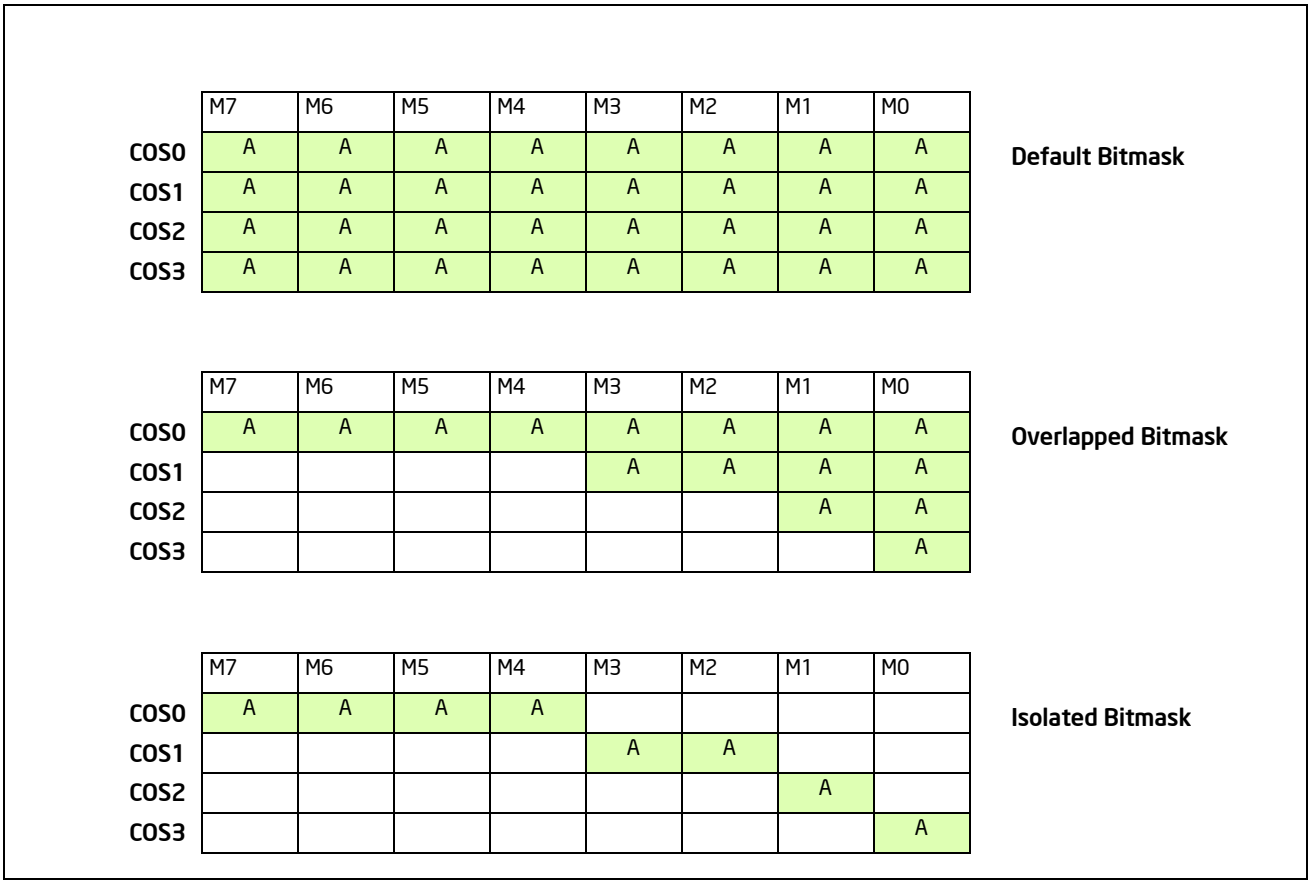


Figure 17-27. Examples of Cache Capacity Bitmasks

Sample cache capacity bitmasks for a bitlength of 8 are shown in Figure 17-27. Please note that all (and only) contiguous '1' combinations are allowed (e.g. FFFFH, 0FF0H, 003CH, etc.). Attempts to program a value without contiguous '1's (including zero) will result in a general protection fault (#GP(0)). It is generally expected that in way-based implementations, one capacity mask bit corresponds to some number of ways in cache, but the specific mapping is implementation-dependent. In all cases, a mask bit set to '1' specifies that a particular Class of Service can allocate into the cache subset represented by that bit. A value of '0' in a mask bit specifies that a Class of Service cannot allocate into the given cache subset. In general, allocating more cache to a given application is usually beneficial to its performance.

Figure 17-27 also shows three examples of sets of Cache Capacity Bitmasks. For simplicity these are represented as 8-bit vectors, though this may vary depending on the implementation and how the mask is mapped to the available cache capacity. The first example shows the default case where all 4 Classes of Service (the total number of COS are implementation-dependent) have full access to the cache. The second case shows an overlapped case, which would allow some lower-priority threads share cache space with the highest priority threads. The third case shows various non-overlapped partitioning schemes. As a matter of software policy for extensibility COS0 should typically be considered and configured as the highest priority COS, followed by COS1, and so on, though there is no hardware restriction enforcing this mapping. When the system boots all threads are initialized to COS0, which has full access to the cache by default.

Though the representation of the CBMs looks similar to a way-based mapping they are independent of any specific enforcement implementation (e.g. way partitioning.) Rather, this is a convenient manner to represent capacity, overlap and isolation of cache space. For example, executing a POPCNT instruction (population count of set bits) on the capacity bitmask can provide the fraction of cache space that a class of service can allocate into. In addition to the fraction, the exact location of the bits also shows whether the class of service overlaps with other classes of service or is entirely isolated in terms of cache space used.

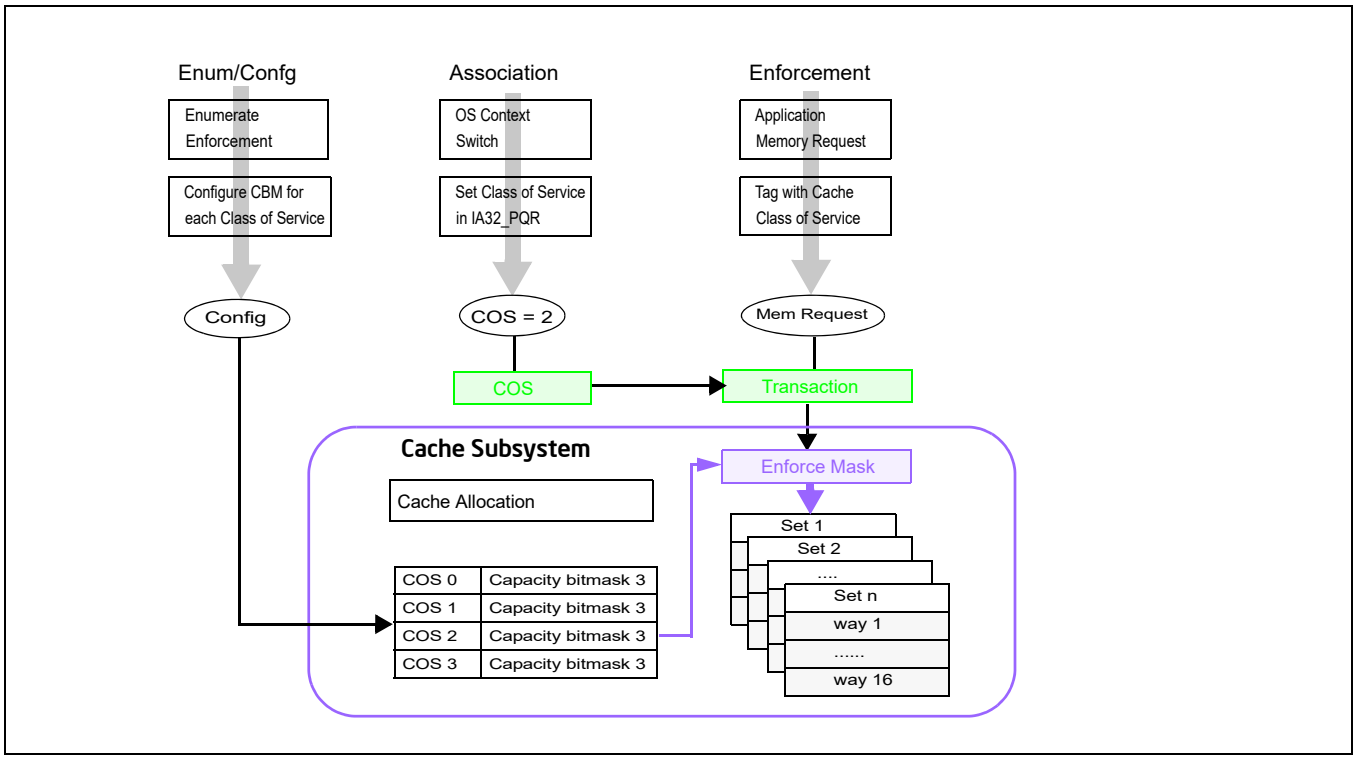


Figure 17-28. Class of Service and Cache Capacity Bitmasks

Figure 17-28 shows how the Cache Capacity Bitmasks and the per-logical-processor Class of Service are logically used to enable Cache Allocation Technology. All (and only) contiguous 1's in the CBM are permitted. The length of CBM may vary from resource to resource or between processor generations and can be enumerated using CPUID. From the available mask set and based on the goals of the OS/VMM (shared or isolated cache, etc.) bitmasks are selected and associated with different classes of service. For the available Classes of Service the associated CBMs can be programmed via the global set of CAT configuration registers (in the case of L3 CAT, via the IA32_L3_MASK_n MSRs, where "n" is the Class of Service, starting from zero). In all architectural implementations supporting CPUID it is possible to change the CBMs dynamically, during program execution, unless stated otherwise by Intel.

The currently running application's Class of Service is communicated to the hardware through the per-logical-processor PQR MSR (IA32_PQR_ASSOC MSR). When the OS schedules an application thread on a logical processor, the application thread is associated with a specific COS (i.e. the corresponding COS in the PQR) and all requests to the CAT-capable resource from that logical processor are tagged with that COS (in other words, the application

thread is configured to belong to a specific COS). The cache subsystem uses this tagged request information to enforce QoS. The capacity bitmask may be mapped into a way bitmask (or a similar enforcement entity based on the implementation) at the cache before it is applied to the allocation policy. For example, the capacity bitmask can be an 8-bit mask and the enforcement may be accomplished using a 16-way bitmask for a cache enforcement implementation based on way partitioning.

The following sections describe extensions of CAT such as Code and Data Prioritization (CDP), followed by details on specific features such as L3 CAT, L3 CDP, and L2 CAT. Depending on the specific processor a mix of features may be supported, and CPUID provides enumeration capabilities to enable software to detect the set of supported features.

17.19.3 Code and Data Prioritization (CDP) Technology

Code and Data Prioritization Technology is an extension of CAT. CDP enables isolation and separate prioritization of code and data fetches to the L3 cache in a software configurable manner, which can enable workload prioritization and tuning of cache capacity to the characteristics of the workload. CDP extends Cache Allocation Technology (CAT) by providing separate code and data masks per Class of Service (COS).

By default, CDP is disabled on the processor. If the CAT MSRs are used without enabling CDP, the processor operates in a traditional CAT-only mode. When CDP is enabled,

- the CAT mask MSRs are re-mapped into interleaved pairs of mask MSRs for data or code fetches (see Figure 17-29),
- the range of COS for CAT is re-indexed, with the lower-half of the COS range available for CDP.

Using the CDP feature, virtual isolation between code and data can be configured on the L3 cache if desired, similar to how some processor cache levels provide separate L1 data and L1 instruction caches.

Like the CAT feature, CDP may be dynamically configured by privileged software at any point during normal system operation, including dynamically enabling or disabling the feature provided that certain software configuration requirements are met (see Section 17.19.5).

An example of the operating mode of CDP is shown in Figure 17-29. Shown at the top are traditional CAT usage models where capacity masks map 1:1 with a COS number to enable control over the cache space which a given COS (and thus applications, threads or VMs) may occupy. Shown at the bottom are example mask configurations where CDP is enabled, and each COS number maps 1:2 to two masks, one for code and one for data. This enables code and data to be either overlapped or isolated to varying degrees either globally or on a per-COS basis, depending on application and system needs.

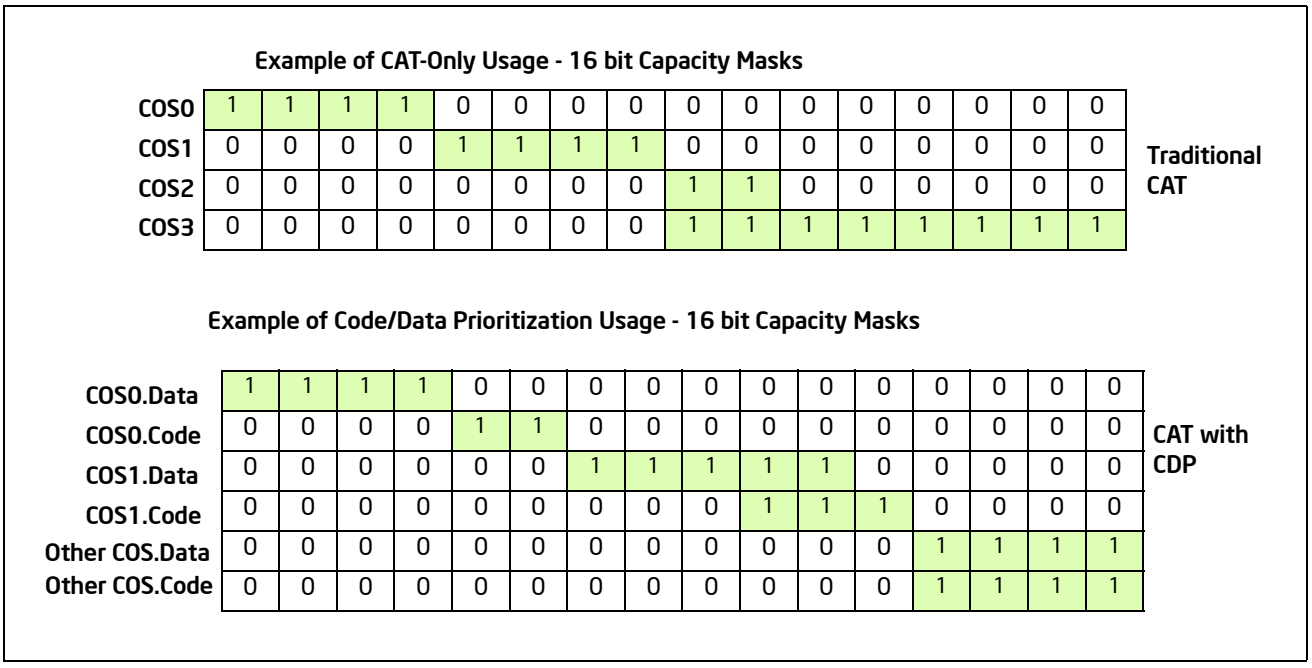


Figure 17-29. Code and Data Capacity Bitmasks of CDP

When CDP is enabled, the existing mask space for CAT-only operation is split. As an example if the system supports 16 CAT-only COS, when CDP is enabled the same MSR interfaces are used, however half of the masks correspond to code, half correspond to data, and the effective number of COS is reduced by half. Code/Data masks are defined per-COS and interleaved in the MSR space as described in subsequent sections.

In cases where CPUID exposes a non-even number of supported Classes of Service for the CAT or CDP features, software using CDP should use the lower matched pairs of code/data masks, and any upper unpaired masks should not be used. As an example, if CPUID exposes 5 CLOS, when CDP is enabled then two code/data pairs are available (masks 0/1 for CLOS[0] data/code and masks 2/3 for CLOS[1] data/code), however the upper un-paired mask should not be used (mask 4 in this case).

17.19.4 Enabling Cache Allocation Technology Usage Flow

Figure 17-30 illustrates the key steps for OS/VMM to detect support of Cache Allocation Technology and enable priority-based resource allocation for a CAT-capable resource.

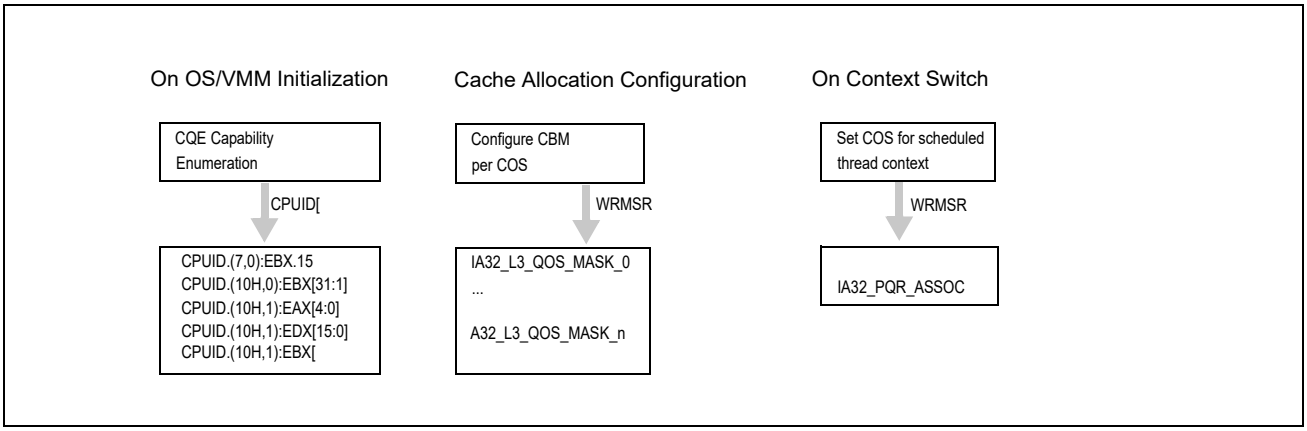


Figure 17-30. Cache Allocation Technology Usage Flow

Enumeration and configuration of L2 CAT is similar to L3 CAT, however CPUID details and MSR addresses differ. Common CLOS are used across the features.

17.19.4.1 Enumeration and Detection Support of Cache Allocation Technology

Software can query processor support of CAT capabilities by executing CPUID instruction with EAX = 07H, ECX = 0H as input. If CPUID.(EAX=07H, ECX=0):EBX.PQE[bit 15] reports 1, the processor supports software control over shared processor resources. Software must use CPUID leaf 10H to enumerate additional details of available resource types, classes of services and capability bitmasks. The programming interfaces provided by Cache Allocation Technology include:

- CPUID leaf function 10H (Cache Allocation Technology Enumeration leaf) and its sub-functions provide information on available resource types, and CAT capability for each resource type (see Section 17.19.4.2).
- IA32_L3_MASK_n: A range of MSRs is provided for each resource type, each MSR within that range specifying a software-configured capacity bitmask for each class of service. For L3 with Cache Allocation support, the CBM is specified using one of the IA32_L3_QOS_MASK_n MSR, where 'n' corresponds to a number within the supported range of COS, i.e. the range between 0 and CPUID.(EAX=10H, ECX=ResID):EDX[15:0], inclusive. See Section 17.19.4.3 for details.
- IA32_L2_MASK_n: A range of MSRs is provided for L2 Cache Allocation Technology, enabling software control over the amount of L2 cache available for each CLOS. Similar to L3 CAT, a CBM is specified for each CLOS using the set of registers, IA32_L2_QOS_MASK_n MSR, where 'n' ranges from zero to the maximum CLOS number reported for L2 CAT in CPUID. See Section 17.19.4.3 for details.

  The L2 mask MSRs are scoped at the same level as the L2 cache (similarly, the L3 mask MSRs are scoped at the same level as the L3 cache). Software may determine which logical processors share an MSR (for instance local to a core, or shared across multiple cores) by performing a write to one of these MSRs and noting which logical threads observe the change. Example flows for a similar method to determine register scope are described in Section 15.5.2, “System Software Recommendation for Managing CMCI and Machine Check Resources”. Software may also use CPUID leaf 4 to determine the maximum number of logical processor IDs that may share a given level of the cache.
- IA32_PQR_ASSOC.CLOS: The IA32_PQR_ASSOC MSR provides a COS field that OS/VMM can use to assign a logical processor to an available COS. The set of COS are common across all allocation features, meaning that multiple features may be supported in the same processor without additional software COS management overhead at context swap time. See Section 17.19.4.4 for details.

17.19.4.2 Cache Allocation Technology: Resource Type and Capability Enumeration

CPUID leaf function 10H (Cache Allocation Technology Enumeration leaf) provides two or more sub-functions:

- CAT Enumeration leaf sub-function 0 enumerates available resource types that support allocation control, i.e. by executing CPUID with EAX=10H and ECX=0H. Each supported resource type is represented by a bit field in

CPUID.(EAX=10H, ECX=0):EBX[31:1]. The bit position of each set bit corresponds to a Resource ID (ResID), for instance ResID=1 is used to indicate L3 CAT support, and ResID=2 indicates L2 CAT support. The ResID is also the sub-leaf index that software must use to query details of the CAT capability of that resource type (see Figure 17-31).

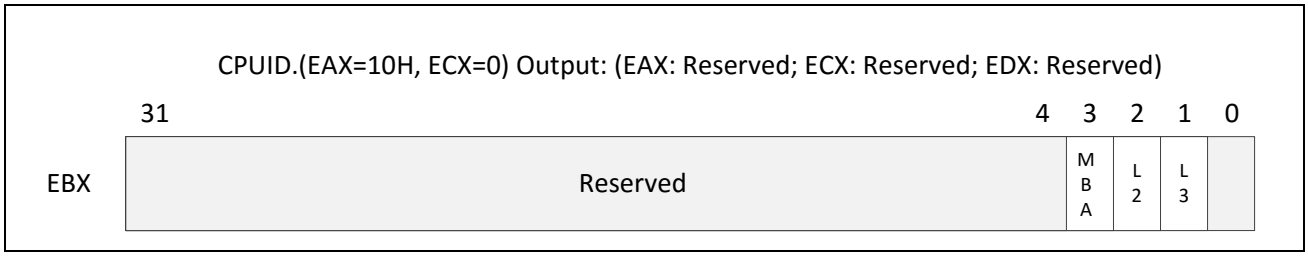


Figure 17-31. CPUID.(EAX=10H, ECX=0H) Available Resource Type Identification

— For ECX>0, EAX[4:0] reports the length of the capacity bitmask length (ECX=1 or 2 for L2 CAT or L3 CAT respectively) using minus-one notation, e.g., a value of 15 corresponds to the capacity bitmask having length of 16 bits. Bits 31:5 of EAX are reserved.

- Sub-functions of CPUID.EAX=10H with a non-zero ECX input matching a supported ResID enumerate the specific enforcement details of the corresponding ResID. The capabilities enumerated include the length of the capacity bitmasks and the number of Classes of Service for a given ResID. Software should query the capability of each available ResID that supports CAT from a sub-leaf of leaf 10H using the sub-leaf index reported by the corresponding non-zero bit in CPUID.(EAX=10H, ECX=0):EBX[31:1] in order to obtain additional feature details.
- CAT capability for L3 is enumerated by CPUID.(EAX=10H, ECX=1H), see Figure 17-32. The specific CAT capabilities reported by CPUID.(EAX=10H, ECX=1) are:

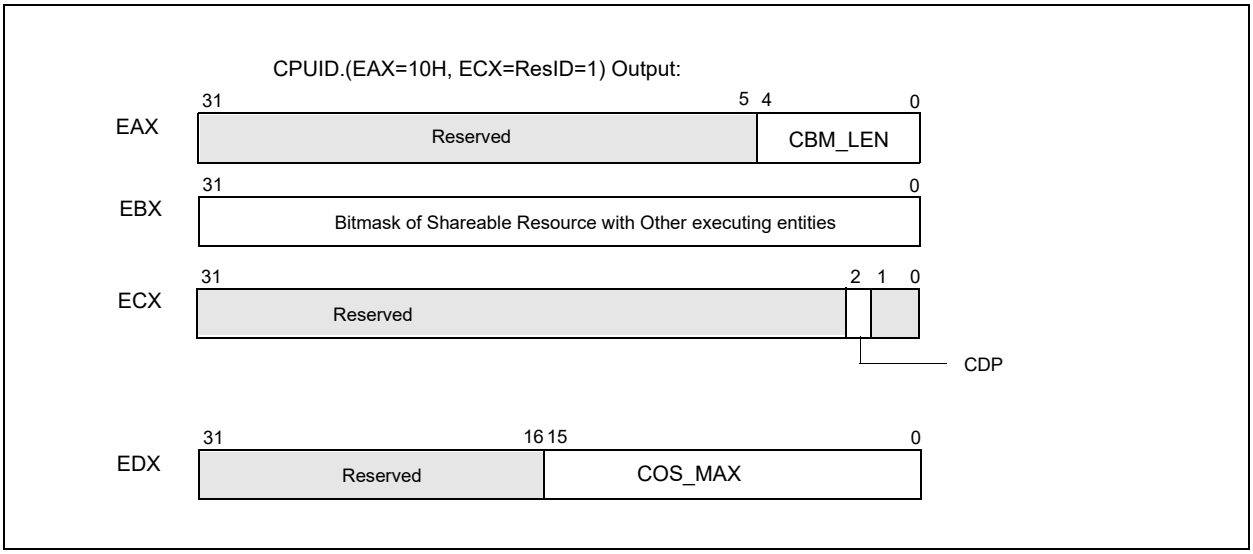


Figure 17-32. L3 Cache Allocation Technology and CDP Enumeration

— CPUID.(EAX=10H, ECX=ResID=1):EAX[4:0] reports the length of the capacity bitmask length using minus-one notation, i.e. a value of 15 corresponds to the capability bitmask having length of 16 bits. Bits 31:5 of EAX are reserved.

— CPUID.(EAX=10H, ECX=1):EBX[31:0] reports a bit mask. Each set bit within the length of the CBM indicates the corresponding unit of the L3 allocation may be used by other entities in the platform (e.g. an

integrated graphics engine or hardware units outside the processor core and have direct access to L3). Each cleared bit within the length of the CBM indicates the corresponding allocation unit can be configured to implement a priority-based allocation scheme chosen by an OS/VMM without interference with other hardware agents in the system. Bits outside the length of the CBM are reserved.

— CPUID.(EAX=10H, ECX=1):ECX.CDP[bit 2]: If 1, indicates Code and Data Prioritization Technology is supported (see Section 17.19.5). Other bits of CPUID.(EAX=10H, ECX=1):ECX are reserved.

— CPUID.(EAX=10H, ECX=1):EDX[15:0] reports the maximum COS supported for the resource (COS are zero-referenced, meaning a reported value of '15' would indicate 16 total supported COS). Bits 31:16 are reserved.

- CAT capability for L2 is enumerated by CPUID.(EAX=10H, ECX=2H), see Figure 17-33. The specific CAT capabilities reported by CPUID.(EAX=10H, ECX=2) are:

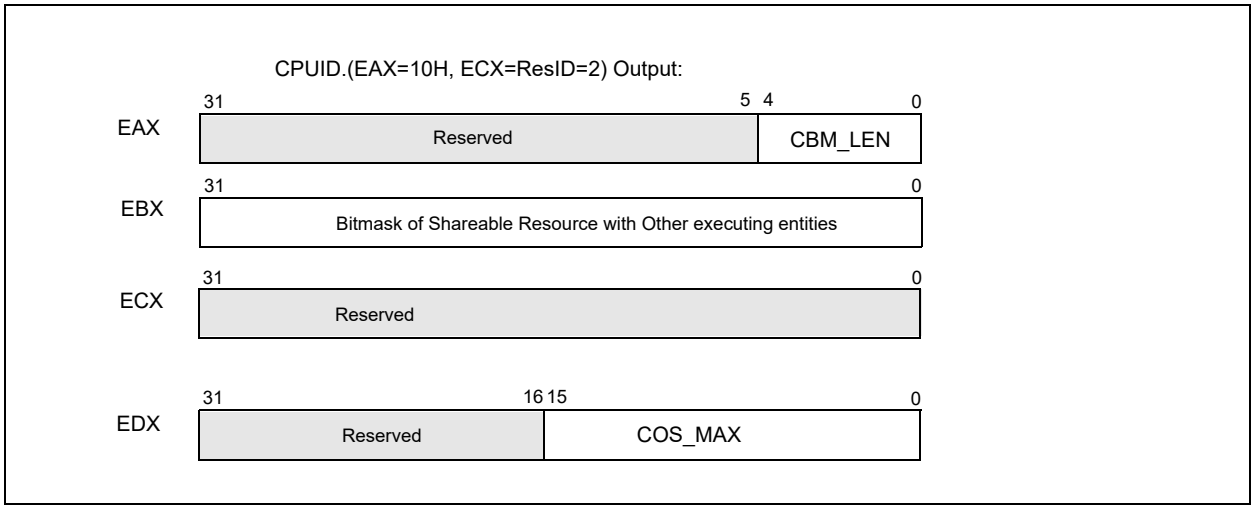


Figure 17-33. L2 Cache Allocation Technology

— CPUID.(EAX=10H, ECX=ResID=2):EAX[4:0] reports the length of the capacity bitmask length using minus-one notation, i.e. a value of 15 corresponds to the capability bitmask having length of 16 bits. Bits 31:5 of EAX are reserved.

— CPUID.(EAX=10H, ECX=2):EBX[31:0] reports a bit mask. Each set bit within the length of the CBM indicates the corresponding unit of the L2 allocation may be used by other entities in the platform. Each cleared bit within the length of the CBM indicates the corresponding allocation unit can be configured to implement a priority-based allocation scheme chosen by an OS/VMM without interference with other hardware agents in the system. Bits outside the length of the CBM are reserved.

— CPUID.(EAX=10H, ECX=2):ECX: reserved.

— CPUID.(EAX=10H, ECX=2):EDX[15:0] reports the maximum COS supported for the resource (COS are zero-referenced, meaning a reported value of '15' would indicate 16 total supported COS). Bits 31:16 are reserved.

A note on migration of Classes of Service (COS): Software should minimize migrations of COS across logical processors (across threads or cores), as a reduction in the performance of the Cache Allocation Technology feature may result if COS are migrated frequently. This is aligned with the industry-standard practice of minimizing unnecessary thread migrations across processor cores in order to avoid excessive time spent warming up processor caches after a migration. In general, for best performance, minimize thread migration and COS migration across processor logical threads and processor cores.

17.19.4.3 Cache Allocation Technology: Cache Mask Configuration

After determining the length of the capacity bitmasks (CBM) and number of COS supported using CPUID (see Section 17.19.4.2), each COS needs to be programmed with a CBM to dictate its available cache via a write to the corresponding IA32_resourceType_MASK_n register, where 'n' corresponds to a number within the supported range of COS, i.e. the range between 0 and CPUID.(EAX=10H, ECX=ResID):EDX[15:0], inclusive, and 'resourceType' corresponds to a specific resource as enumerated by the set bits of CPUID.(EAX=10H, ECX=0):EAX[31:1], for instance, ‘L2’ or ‘L3’ cache.

A hierarchy of MSRs is reserved for Cache Allocation Technology registers of the form IA32_resourceType_MASK_n:

- From 0C90H through 0D8FH (inclusive), providing support for multiple sub-ranges to support varying resource types. The first supported resourceType is 'L3', corresponding to the L3 cache in a platform. The MSRs range from 0C90H through 0D0FH (inclusive), enables support for up to 128 L3 CAT Classes of Service.

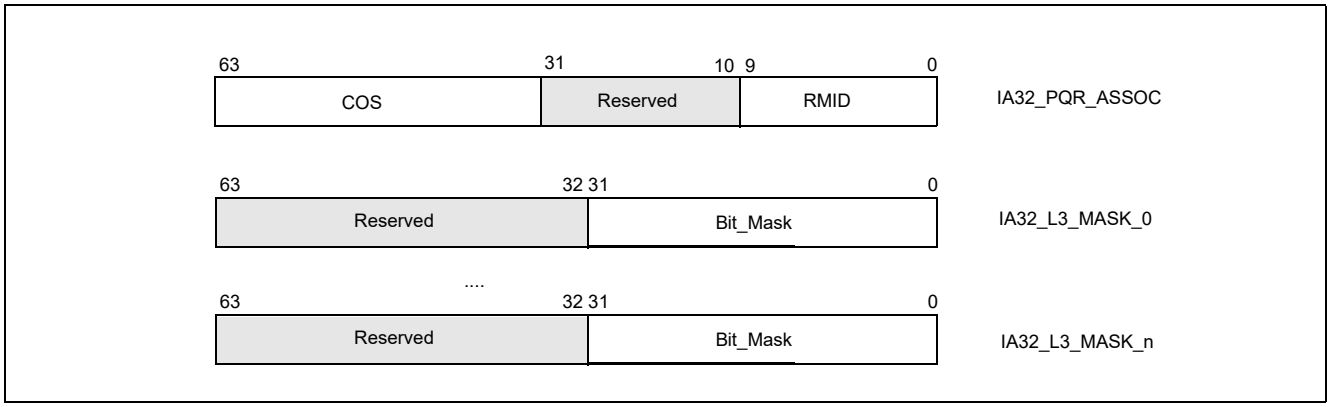


Figure 17-34. IA32_PQR_ASSOC, IA32_L3_MASK_n MSRs

- Within the same CAT range hierarchy, another set of registers is defined for resourceType 'L2', corresponding to the L2 cache in a platform, and MSRs IA32_L2_MASK_n are defined for n=[0,63] at addresses 0D10H through 0D4FH (inclusive).

Figure 17-34 and Figure 17-35 provide an overview of the relevant registers.

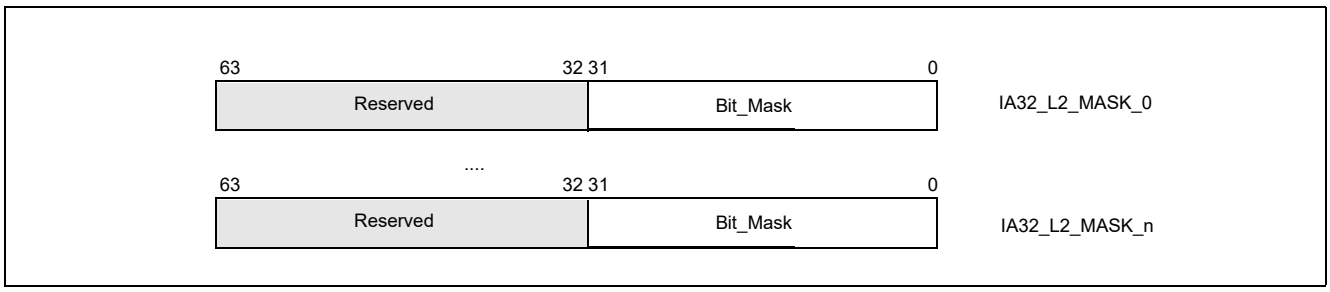


Figure 17-35. IA32_L2_MASK_n MSRs

All CAT configuration registers can be accessed using the standard RDMSR / WRMSR instructions.

Note that once L3 or L2 CAT masks are configured, threads can be grouped into Classes of Service (COS) using the IA32_PQR_ASSOC MSR as described in Chapter 17, “Class of Service to Cache Mask Association: Common Across Allocation Features”.

17.19.4.4 Class of Service to Cache Mask Association: Common Across Allocation Features

After configuring the available classes of service with the preferred set of capacity bitmasks, the OS/VMM can set the IA32_PQR_ASSOC.COS of a logical processor to the class of service with the desired CBM when a thread

context switch occurs. This allows the OS/VMM to indicate which class of service an executing thread/VM belongs within. Each logical processor contains an instance of the IA32_PQR_ASSOC register at MSR location 0C8FH, and Figure 17-34 shows the bit field layout for this register. Bits[63:32] contain the COS field for each logical processor.

Note that placing the RMID field within the same PQR register enables both RMID and CLOS to be swapped at context swap time for simultaneous use of monitoring and allocation features with a single register write for efficiency.

When CDP is enabled, Specifying a COS value in IA32_PQR_ASSOC.COS greater than MAX_COS_CDP =(CPUID.(EAX=10H, ECX=1):EDX[15:0] >> 1) will cause undefined performance impact to code and data fetches.

Note that if the IA32_PQR_ASSOC.COS is never written then the CAT capability defaults to using COS 0, which in turn is set to the default mask in IA32_L3_MASK_0 - which is all "1"s (on reset). This essentially disables the enforcement feature by default or for legacy operating systems and software.

See Section 17.19.6, "Cache Allocation Technology Programming Considerations" for important COS programming considerations including maximum values when using CAT and CDP.

17.19.5 Code and Data Prioritization (CDP): Enumerating and Enabling L3 CDP Technology

CDP is an extension of CAT. The presence of the CDP feature is enumerated via CPUID.(EAX=10H, ECX=1):ECX.CDP[bit 2] (see Figure 17-32). Most of the CPUID.(EAX=10H, ECX=1) sub-leaf data that applies to CAT also apply to CDP. However, CPUID.(EAX=10H, ECX=1):EDX.COS_MAX_CAT specifies the maximum COS applicable to CAT-only operation. For CDP operations, COS_MAX_CDP is equal to (CPUID.(EAX=10H, ECX=1):EDX.COS_MAX_CAT >>1).

If CPUID.(EAX=10H, ECX=1):ECX.CDP[bit 2] =1, the processor supports CDP and provides a new MSR IA32_L3_QOS_CFG at address 0C81H. The layout of IA32_L3_QOS_CFG is shown in Figure 17-36. The bit field definition of IA32_L3_QOS_CFG are:

- Bit 0: L3 CDP Enable. If set, enables CDP, maps CAT mask MSRs into pairs of Data Mask and Code Mask MSRs. The maximum allowed value to write into IA32_PQR_ASSOC.COS is COS_MAX_CDP.
- Bits 63:1: Reserved. Attempts to write to reserved bits result in a #GP(0).

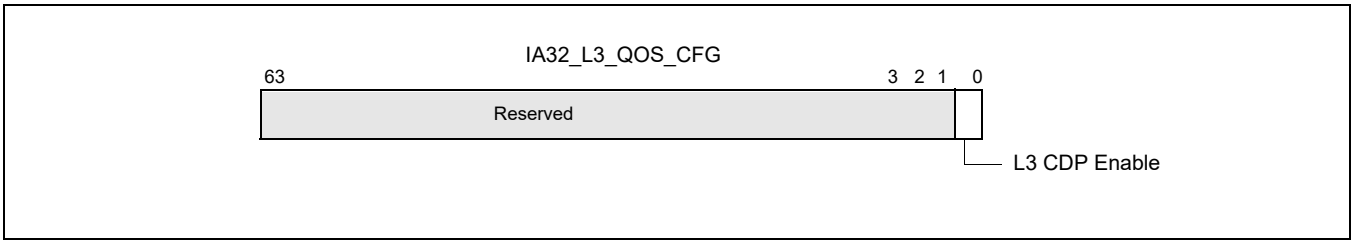


Figure 17-36. Layout of IA32_L3_QOS_CFG

IA32_L3_QOS_CFG default values are all 0s at RESET, the mask MSRs are all 1s. Hence. all logical processors are initialized in COS0 allocated with the entire L3 with CDP disabled, until software programs CAT and CDP.

Before enabling or disabling CDP, software should write all 1's to all of the CAT/CDP masks to ensure proper behavior (e.g., the IA32_L3_QOS_Mask_n set of MSRs). When enabling CDP, software should also ensure that only COS number which are valid in CDP operation is used, otherwise undefined behavior may result. For instance in a case with 16 CAT COS, since COS are reduced by half when CDP is enabled, software should ensure that only COS 0-7 are in use before enabling CDP (along with writing 1's to all mask bits before enabling or disabling CDP).

Software should also account for the fact that mask interpretations change when CDP is enabled or disabled, meaning for instance that a CAT mask for a given COS may become a code mask for a different Class of Service when CDP is enabled. In order to simplify this behavior and prevent unintended remapping software should consider resetting all threads to COS[0] before enabling or disabling CDP.

17.19.5.1 Mapping Between L3 CDP Masks and CAT Masks

When CDP is enabled, the existing CAT mask MSR space is re-mapped to provide a code mask and a data mask per COS. The re-mapping is shown in Table 17-19.

Table 17-19. Re-indexing of COS Numbers and Mapping to CAT/CDP Mask MSRs

Mask MSR	CAT-only Operation	CDP Operation
IA32_L3_QOS_Mask_0	COS0	COS0.Data
IA32_L3_QOS_Mask_1	COS1	COS0.Code
IA32_L3_QOS_Mask_2	COS2	COS1.Data
IA32_L3_QOS_Mask_3	COS3	COS1.Code
IA32_L3_QOS_Mask_4	COS4	COS2.Data
IA32_L3_QOS_Mask_5	COS5	COS2.Code
….	….	….
IA32_L3_QOS_Mask_'2n'	COS'2n'	COS'n'.Data
IA32_L3_QOS_Mask_'2n+1'	COS'2n+1'	COS'n'.Code

One can derive the MSR address for the data mask or code mask for a given COS number 'n' by:

- data_mask_address (n) = base + (n <<1), where base is the address of IA32_L3_QOS_MASK_0.
- code_mask_address (n) = base + (n <<1) +1.

When CDP is enabled, each COS is mapped 1:2 with mask MSRs, with one mask enabling programmatic control over data fill location and one mask enabling control over data placement. A variety of overlapped and isolated mask configurations are possible (see the example in Figure 17-29).

Mask MSR field definitions remain the same. Capacity masks must be formed of contiguous set bits, with a length of 1 bit or longer and should not exceed the maximum mask length specified in CPUID. As examples, valid masks on a cache with max bitmask length of 16b (from CPUID) include 0xFFFF, 0xFF00, 0x00FF, 0x00F0, 0x0001, 0x0003 and so on. Maximum valid mask lengths are unchanged whether CDP is enabled or disabled, and writes of invalid mask values may lead to undefined behavior. Writes to reserved bits will generate #GP(0).

17.19.5.2 L3 CAT: Disabling CDP

Before enabling or disabling CDP, software should write all 1's to all of the CAT/CDP masks to ensure proper behavior (e.g., the IA32_L3_QOS_Mask_n set of MSRs).

Software should also account for the fact that mask interpretations change when CDP is enabled or disabled, meaning for instance that a CAT mask for a given COS may become a code mask for a different Class of Service when CDP is enabled. In order to simplify this behavior and prevent unintended remapping software should consider resetting all threads to COS[0] before enabling or disabling CDP.

17.19.6 Cache Allocation Technology Programming Considerations

17.19.6.1 Cache Allocation Technology Dynamic Configuration

Both the CAT masks and CQM registers are accessible and modifiable at any time during execution using RDMSR/WRMSR unless otherwise noted. When writing to these MSRs a #GP(0) will be generated if any of the following conditions occur:

- A reserved bit is modified,
- Accessing a QOS mask register outside the supported COS (the max COS number is specified in CPUID.(EAX=10H, ECX=ResID):EDX[15:0]), or

- Writing a COS greater than the supported maximum (specified as the maximum value of CPUID.(EAX=10H, ECX=ResID):EDX[15:0] for all valid ResID values) is written to the IA32_PQR_ASSOC.CLOS field.

When CDP is enabled, specifying a COS value in IA32_PQR_ASSOC.COS outside of the lower half of the COS space will cause undefined performance impact to code and data fetches due to MSR space re-indexing into code/data masks when CDP is enabled.

When reading the IA32_PQR_ASSOC register the currently programmed COS on the core will be returned.

When reading an IA32_*resourceType*_MASK_*n* register the current capacity bit mask for COS 'n' will be returned.

As noted previously, software should minimize migrations of COS across logical processors (across threads or cores), as a reduction in the accuracy of the Cache Allocation feature may result if COS are migrated frequently. This is aligned with the industry standard practice of minimizing unnecessary thread migrations across processor cores in order to avoid excessive time spent warming up processor caches after a migration. In general, for best performance, minimize thread migration and COS migration across processor logical threads and processor cores.

17.19.6.2 Cache Allocation Technology Operation With Power Saving Features

Note that the Cache Allocation Technology feature cannot be used to enforce cache coherency, and that some advanced power management features such as C-states which may shrink or power off various caches within the system may interfere with CAT hints - in such cases the CAT bitmasks are ignored and the other features take precedence. If the highest possible level of CAT differentiation or determinism is required, disable any power-saving features which shrink the caches or power off caches. The details of the power management interfaces are typically implementation-specific, but can be found at *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3C*.

If software requires differentiation between threads but not absolute determinism then in many cases it is possible to leave power-saving cache shrink features enabled, which can provide substantial power savings and increase battery life in mobile platforms. In such cases when the caches are powered off (e.g., package C-states) the entire cache of a portion thereof may be powered off. Upon resuming an active state any new incoming data to the cache will be filled subject to the cache capacity bitmasks. Any data in the cache prior to the cache shrink or power off may have been flushed to memory during the process of entering the idle state, however, and is not guaranteed to remain in the cache. If differentiation between threads is the goal of system software then this model allows substantial power savings while continuing to deliver performance differentiation. If system software needs optimal determinism then power saving modes which flush portions of the caches and power them off should be disabled.

NOTE

IA32_PQR_ASSOC is saved and restored across C6 entry/exit. Similarly, the mask register contents are saved across package C-state entry/exit and are not lost.

17.19.6.3 Cache Allocation Technology Operation with Other Operating Modes

The states in IA32_PQR_ASSOC and mask registers are unmodified across an SMI delivery. Thus, the execution of SMM handler code can interact with the Cache Allocation Technology resource and manifest some degree of non-determinism to the non-SMM software stack. An SMM handler may also perform certain system-level or power management practices that affect CAT operation.

It is possible for an SMM handler to minimize the impact on data determinism in the cache by reserving a COS with a dedicated partition in the cache. Such an SMM handler can switch to the dedicated COS immediately upon entering SMM, and switching back to the previously running COS upon exit.

17.19.6.4 Associating Threads with CAT/CDP Classes of Service

Threads are associated with Classes of Service (CLOS) via the per-logical-processor IA32_PQR_ASSOC MSR. The same COS concept applies to both CAT and CDP (for instance, COS[5] means the same thing whether CAT or CDP is in use, and the COS has associated resource usage constraint attributes including cache capacity masks). The mapping of COS to mask MSRs does change when CDP is enabled, according to the following guidelines:

- In CAT-only Mode - one set of bitmasks in one mask MSR control both code and data.
  - Each COS number map 1:1 with a capacity mask on the applicable resource (e.g., L3 cache).
- When CDP is enabled,
  - Two mask sets exist for each COS number, one for code, one for data.
  - Masks for code/data are interleaved in the MSR address space (see Table 17-19).

17.19.7 Introduction to Memory Bandwidth Allocation

The Memory Bandwidth Allocation (MBA) feature provides indirect and approximate control over memory bandwidth available per-core, and is introduced on future Intel processors. This feature provides a method to control applications which may be over-utilizing bandwidth relative to their priority in environments such as the data-center.

The MBA feature uses existing constructs from the Resource Director Technology (RDT) feature set including Classes of Service (CLOS). A given CLOS used for L3 CAT for instance means the same thing as a CLOS used for MBA. Infrastructure such as the MSR used to associate a thread with a CLOS (the IA32_PQR_ASSOC_MSR) and some elements of the CPUID enumeration (such as CPUID leaf 10H) are shared.

- The high-level implementation of Memory Bandwidth Allocation is shown in Figure 17-37.

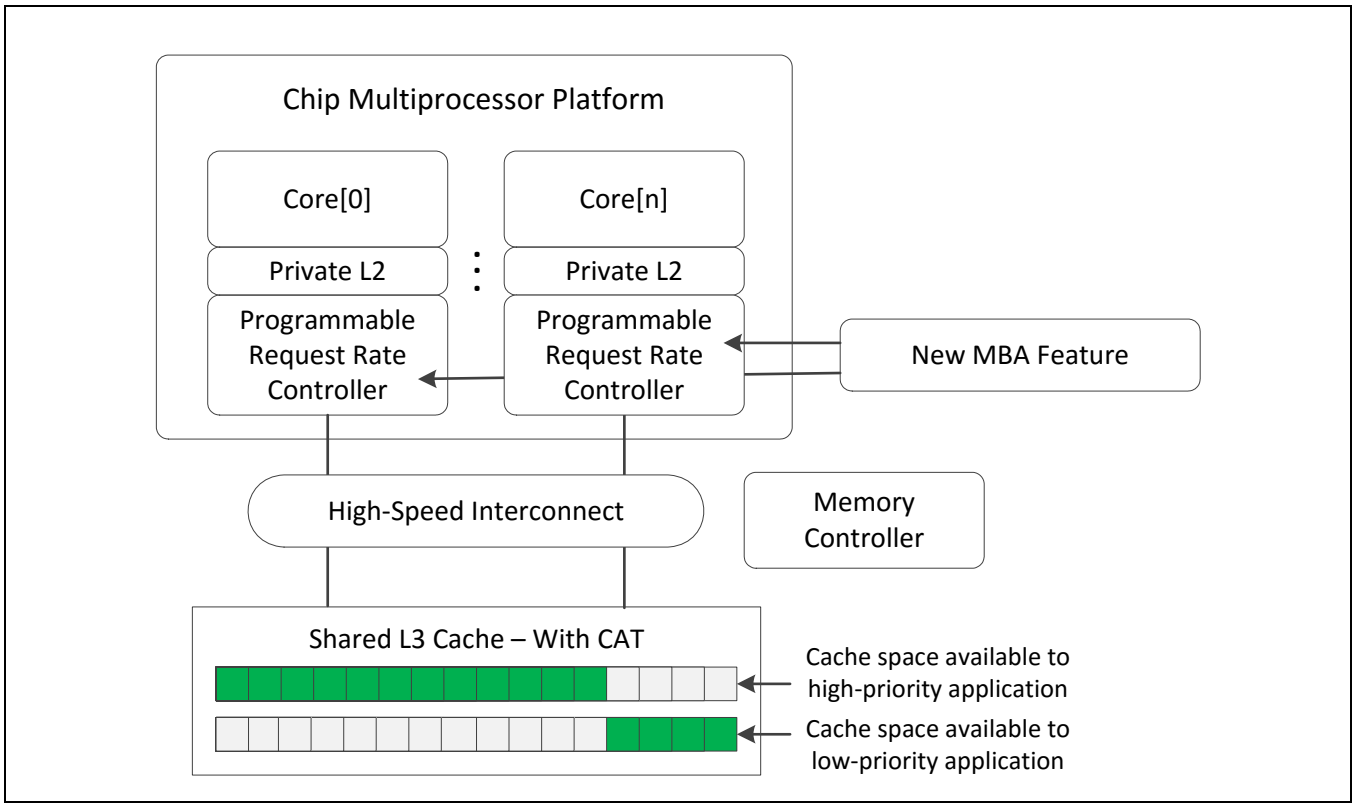


Figure 17-37. A High-Level Overview of the MBA Feature

As shown in Figure 17-37 the MBA feature introduces a programmable request rate controller between the cores and the high-speed interconnect, enabling indirect control over memory bandwidth for cores over-utilizing bandwidth relative to their priority. For instance, high-priority cores may be run un-throttled, but lower priority cores generating an excessive amount of traffic may be throttled to enable more bandwidth availability for the high-priority cores.

Since MBA uses a programmable rate controller between the cores and the interconnect, higher-level shared caches and memory controller, bandwidth to these caches may also be reduced, so care should be taken to throttle only bandwidth-intense applications which do not use the off-core caches effectively.

The throttling values exposed by MBA are approximate, and are calibrated to specific traffic patterns. As work-load characteristics vary, the throttling values provided may affect each workload differently. In cases where precise control is needed, the Memory Bandwidth Monitoring (MBM) feature can be used as input to a software controller which makes decisions about the MBA throttling level to apply.

Enumeration and configuration details are discussed below followed by usage model considerations.

17.19.7.1 Memory Bandwidth Allocation Enumeration

Similar to other RDT features, enumeration of the presence and details of the MBA feature is provided via a sub-leaf of the CPUID instruction.

Key components of the enumeration are as follows.

- Support for the MBA feature on the processor, and if MBA is supported, the following details:
  - Number of supported Classes of Service (CLOS) for the processor.
  - The maximum MBA delay value supported (which also implicitly provides a definition of the granularity).
  - An indication of whether the delay values which can be programmed are linearly spaced or not.

The presence of any of the RDT features which enable control over shared platform resources is enumerated by executing CPUID instruction with EAX = 07H, ECX = 0H as input. If CPUID.(EAX=07H, ECX=0):EBX.PQE[bit 15] reports 1, the processor supports software control over shared processor resources. Software may then use CPUID leaf 10H to enumerate additional details on the specific controls provided.

Through CPUID leaf 10H software may determine whether MBA is supported on the platform. Specifically, as shown in Figure 17-31, bit 3 of the EBX register indicates whether MBA is supported on the processor, and the bit position (3) constitutes a Resource ID (ResID) which allows enumeration of MBA details. For instance, if bit 3 is supported this implies the presence of CPUID.10H.[ResID=3] as shown in Figure 17-38 which provides the following details.

- CPUID.(EAX=10H, ECX=ResID=3):EAX[11:0] reports the maximum MBA throttling value supported, minus one. For instance, a value of 89 indicates that a maximum throttling value of 90 is supported. Additionally, in cases where a linear interface (see below) is supported then one hundred minus the maximum throttling value indicates the granularity, 10% in this example.
- CPUID.(EAX=10H, ECX=ResID=3):EBX is reserved.
- CPUID.(EAX=10H, ECX=ResID=3):ECX[2] reports whether the response of the delay values is linear (see text).
- CPUID.(EAX=10H, ECX=ResID=3):EDX[15:0] reports the number of Classes of Service (CLOS) supported for the feature (minus one). For instance, a reported value of 15 implies a maximum of 16 supported MBA CLOS.

The number of CLOS supported for the MBA feature may or may not align with other resources such as L3 CAT. In cases where the RDT features support different numbers of CLOS the lowest numerical CLOS support the common set of features, while higher CLOS may support a subset. For instance, if L3 CAT supports 8 CLOS while MBA supports 4 CLOS, all 8 CLOS would have L3 CAT masks available for cache control, but the upper 4 CLOS would not offer MBA support. In this case the upper 4 CLOS would not be subject to any throttling control. Software can manage supported resources / CLOS in order to either have consistent capabilities across CLOS by using the common subset or enable more flexibility by selectively applying resource control where needed based on careful CLOS and thread mapping. In all cases, CLOS[0] supports all RDT resource control features present on the platform.

Discussion on the interpretation and usage of the MBA delay values is provided in Section 17.19.7.2 on MBA configuration.

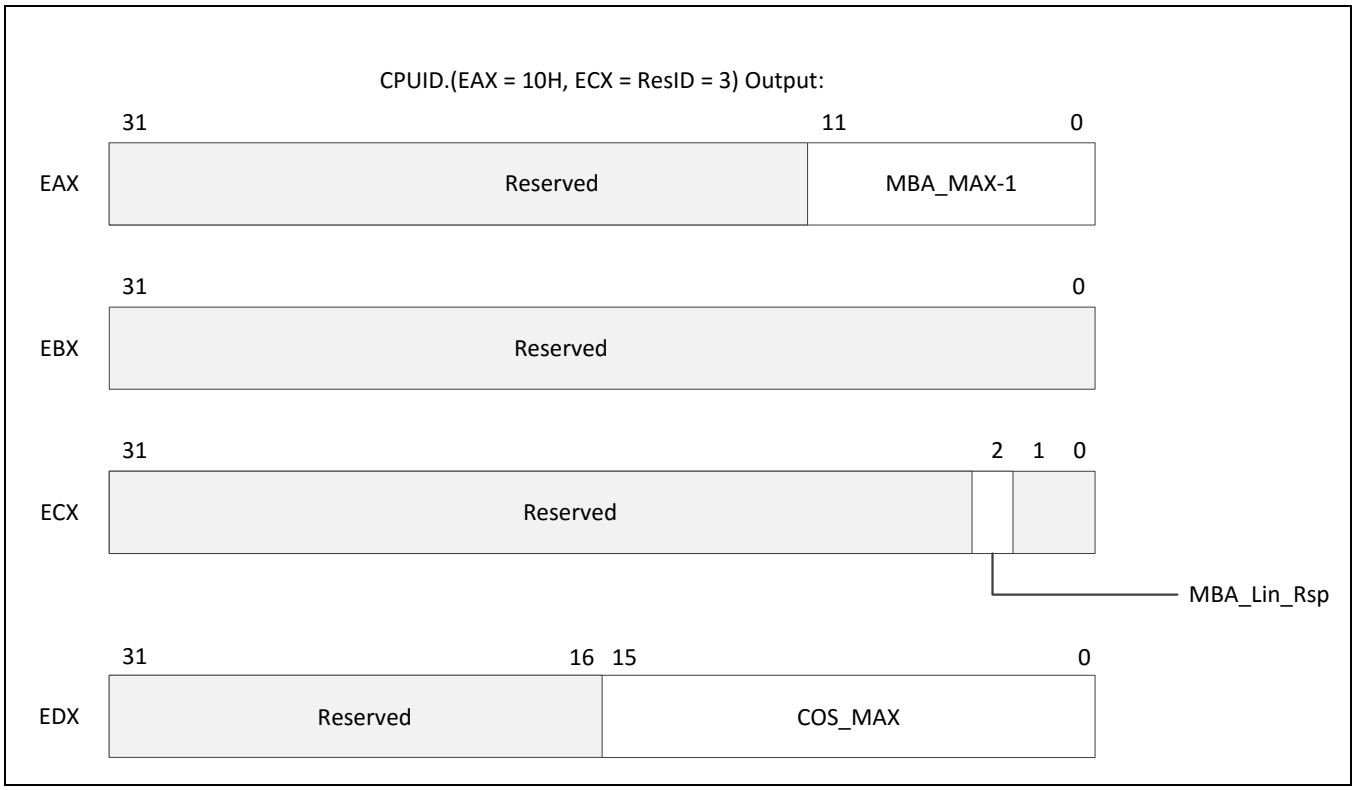


Figure 17-38. CPUID.(EAX=10H, ECX=3H) MBA Feature Details Identification

17.19.7.2 Memory Bandwidth Allocation Configuration

The configuration of MBA takes consists of two processes once enumeration is complete.

- Association of threads to Classes of Service (CLOS) - accomplished in a common fashion across RDT features as described in Section 17.19.7.1 via the IA32_PQR_ASSOC MSR. As with features such as L3 CAT, software may update the CLOS field of the PQR MSR at context swap time in order to maintain the proper association of software threads to Classes of Service on the hardware. While logical processors may each be associated with independent CLOS, see Section 17.19.7.3 for important usage model considerations (initial versions of the MBA feature select the maximum delay value across threads).
- Configuration of the per-CLOS delay values, accomplished via the IA32_L2_QoS_Ext_BW_Thrtl_n MSR set shown in Table 17-20.

The MBA delay values which may be programmed range from zero (implying zero delay, and full bandwidth available) to the maximum (MBA_MAX) specified in CPUID as discussed in Section 17.19.7.1.

Software may select an MBA delay value then write the value into one or more of the IA32_L2_QoS_Ext_BW_Thrtl_n MSRs to update the delay values applied for a specific CLOS. As shown in Table 17.20 the base address of the MSRs is at D50H, and the range corresponds to the maximum supported CLOS from CPUID.(EAX=10H, ECX=ResID=1):EDX[15:0] as described in Section 17.19.7.1. For instance, if 16 CLOS are supported then the valid MSR range will extend from D50H through D5F inclusive.

Table 17-20. MBA Delay Value MSRs

Delay Value MSR	Address
IA32_L2_QoS_Ext_BW_Thrtl_0	D50H
IA32_L2_QoS_Ext_BW_Thrtl_1	D51H
IA32_L2_QoS_Ext_BW_Thrtl_2	D52H
….	….
IA32_L2_QoS_Ext_BW_Thrtl_'COS_MAX'	D50H + COS_MAX from CPUID.10H.3

The definition for the MBA delay value MSRs is provided in Figure 17.39. The lower 16 bits are used for MBA delay values, and values from zero to the maximum from the CPUID MBA_MAX-1 value are supported. Values outside this range will generate #GP(0).

If linear input throttling values are indicated by CPUID.(EAX=10H, ECX=ResID=3):ECX[bit 2] then values from zero through the MBA_MAX field from CPUID.(EAX=10H, ECX=ResID=3):EAX[11:0] are supported as inputs. In the linear mode the input precision is defined as 100-(MBA_MAX). For instance, if the MBA_MAX value is 90, the input precision is 10%. Values not an even multiple of the precision (e.g., 12%) will be rounded down (e.g., to 10% delay applied).

- If linear values are not supported (CPUID.(EAX=10H, ECX=ResID=3):ECX[bit 2] = 0) then input delay values are powers-of-two from zero to the MBA_MAX value from CPUID. In this case any values not a power of two will be rounded down the next nearest power of two.

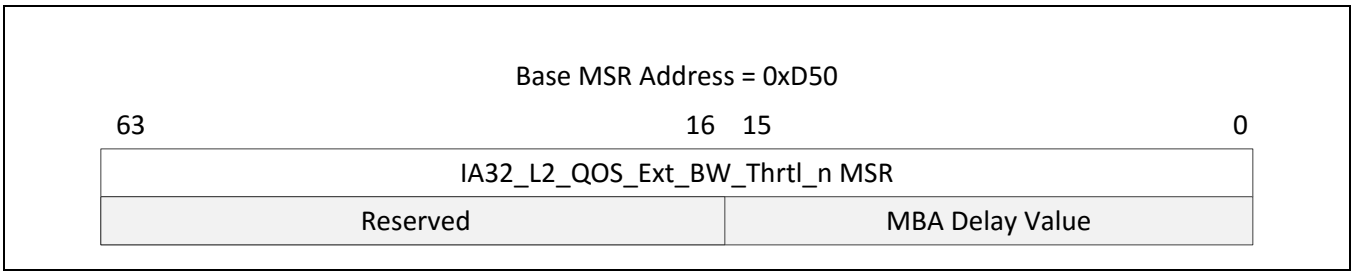


Figure 17-39. IA32_L2_QoS_Ext_BW_Thrtl_n MSR Definition

Note that the throttling values provided to software are calibrated through specific traffic patterns, however as workload characteristics may vary the response precision and linearity of the delay values will vary across products, and should be treated as approximate values only.

17.19.7.3 Memory Bandwidth Allocation Usage Considerations

As the memory bandwidth control that MBA provides is indirect, using the feature with a closed-loop controller to also monitor memory bandwidth and how effectively the applications use the cache (via the Cache Monitoring Technology feature) may provide additional value. This approach also allows administrators to provide a band-width target or set-point which a controller could use to guide MBA throttling values applied, and this allows bandwidth control independent of the execution characteristics of the application.

As control is provided per processor core (the max of the delay values of the per-thread CLOS applied to the core) care should be taking in scheduling threads so as to not inadvertently place a high-priority thread (with zero intended MBA throttling) next to a low-priority thread (with MBA throttling intended), which would lead to inadvertent throttling of the high-priority thread.

14.Updates to Chapter 18, Volume 3B

Change bars show changes to Chapter 18 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3B:* System Programming Guide, Part 2.

Changes to this chapter: Reorganized chapter. Updates to Sections 18.2.4.3 "IA32_PERF_GLOBAL_INUSE MSR", 18.5.3 "Performance Monitoring for Goldmont Microarchitecture", 18.5.3.1 "Processor Event Based Sampling (PEBS)", and 18.5.3.1.1 "PEBS Data Linear Address Profiling". New Sections 18.5.4 "Performance Monitoring for Goldmont Plus Microarchitecture".

CHAPTER 18
PERFORMANCE MONITORING

Intel 64 and IA-32 architectures provide facilities for monitoring performance via a PMU (Performance Monitoring Unit).

18.1 PERFORMANCE MONITORING OVERVIEW

Performance monitoring was introduced in the Pentium processor with a set of model-specific performance-monitoring counter MSRs. These counters permit selection of processor performance parameters to be monitored and measured. The information obtained from these counters can be used for tuning system and compiler performance.

In Intel P6 family of processors, the performance monitoring mechanism was enhanced to permit a wider selection of events to be monitored and to allow greater control events to be monitored. Next, Intel processors based on Intel NetBurst microarchitecture introduced a distributed style of performance monitoring mechanism and performance events.

The performance monitoring mechanisms and performance events defined for the Pentium, P6 family, and Intel processors based on Intel NetBurst microarchitecture are not architectural. They are all model specific (not compatible among processor families). Intel Core Solo and Intel Core Duo processors support a set of architectural performance events and a set of non-architectural performance events. Newer Intel processor generations support enhanced architectural performance events and non-architectural performance events.

Starting with Intel Core Solo and Intel Core Duo processors, there are two classes of performance monitoring capabilities. The first class supports events for monitoring performance using counting or interrupt-based event sampling usage. These events are non-architectural and vary from one processor model to another. They are similar to those available in Pentium M processors. These non-architectural performance monitoring events are specific to the microarchitecture and may change with enhancements. They are discussed in Section 18.6.3, "Performance Monitoring (Processors Based on Intel NetBurst® Microarchitecture)." Non-architectural events for a given microarchitecture cannot be enumerated using CPUID; and they are listed in Chapter 19, "Performance-Monitoring Events."

The second class of performance monitoring capabilities is referred to as architectural performance monitoring. This class supports the same counting and Interrupt-based event sampling usages, with a smaller set of available events. The visible behavior of architectural performance events is consistent across processor implementations. Availability of architectural performance monitoring capabilities is enumerated using the CPUID.0AH. These events are discussed in Section 18.2.

See also:

- Section 18.2, "Architectural Performance Monitoring"
- Section 18.3, "Performance Monitoring (Intel® Core™ Processors and Intel® Xeon® Processors)"
  - Section 18.3.1, "Performance Monitoring for Processors Based on Intel® Microarchitecture Code Name Nehalem"
  - Section 18.3.2, "Performance Monitoring for Processors Based on Intel® Microarchitecture Code Name Westmere"
  - Section 18.3.3, "Intel® Xeon® Processor E7 Family Performance Monitoring Facility"
  - Section 18.3.4, "Performance Monitoring for Processors Based on Intel® Microarchitecture Code Name Sandy Bridge"
  - Section 18.3.5, "3rd Generation Intel® Core™ Processor Performance Monitoring Facility"
  - Section 18.3.6, "4th Generation Intel® Core™ Processor Performance Monitoring Facility"
  - Section 18.3.7, "5th Generation Intel® Core™ Processor and Intel® Core™ M Processor Performance Monitoring Facility"

18.2 ARCHITECTURAL PERFORMANCE MONITORING

Performance monitoring events are architectural when they behave consistently across microarchitectures. Intel Core Solo and Intel Core Duo processors introduced architectural performance monitoring. The feature provides a mechanism for software to enumerate performance events and provides configuration and counting facilities for events.

Architectural performance monitoring does allow for enhancement across processor implementations. The CPUID.0AH leaf provides version ID for each enhancement. Intel Core Solo and Intel Core Duo processors support base level functionality identified by version ID of 1. Processors based on Intel Core microarchitecture support, at a minimum, the base level functionality of architectural performance monitoring. Intel Core 2 Duo processor T 7700 and newer processors based on Intel Core microarchitecture support both the base level functionality and enhanced architectural performance monitoring identified by version ID of 2.

45 nm and 32 nm Intel Atom processors and Intel Atom processors based on the Silvermont microarchitecture support the functionality provided by versionID 1, 2, and 3; CPUID.0AH:EAX[7:0] reports versionID = 3 to indicate the aggregate of architectural performance monitoring capabilities. Intel Atom processors based on the Airmont microarchitecture support the same performance monitoring capabilities as those based on the Silvermont micro-architecture.

Intel Core processors and related Intel Xeon processor families based on the Nehalem through Broadwell microar-chitectures support version ID 1, 2, and 3. Intel processors based on the Skylake and Kaby Lake microarchitectures support versionID 4.

Next generation Intel Atom processors are based on the Goldmont microarchitecture. Intel processors based on the Goldmont microarchitecture support versionID 4.

18.2.1 Architectural Performance Monitoring Version 1

Configuring an architectural performance monitoring event involves programming performance event select registers. There are a finite number of performance event select MSRs (IA32_PERFEVTSELx MSRs). The result of a performance monitoring event is reported in a performance monitoring counter (IA32_PMCx MSR). Performance monitoring counters are paired with performance monitoring select registers.

Performance monitoring select registers and counters are architectural in the following respects:

- Bit field layout of IA32_PERFEVTSELx is consistent across microarchitectures.
- Addresses of IA32_PERFEVTSELx MSRs remain the same across microarchitectures.
- Addresses of IA32_PMC MSRs remain the same across microarchitectures.
- Each logical processor has its own set of IA32_PERFEVTSELx and IA32_PMCx MSRs. Configuration facilities and counters are not shared between logical processors sharing a processor core.

Architectural performance monitoring provides a CPUID mechanism for enumerating the following information:

- Number of performance monitoring counters available in a logical processor (each IA32_PERFEVTSELx MSR is paired to the corresponding IA32_PMCx MSR)
- Number of bits supported in each IA32_PMCx
- Number of architectural performance monitoring events supported in a logical processor

Software can use CPUID to discover architectural performance monitoring availability (CPUID.0AH). The architectural performance monitoring leaf provides an identifier corresponding to the version number of architectural performance monitoring available in the processor.

The version identifier is retrieved by querying CPUID.0AH:EAX[bits 7:0] (see Chapter 3, "Instruction Set Reference, A-L," in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A*). If the version identifier is greater than zero, architectural performance monitoring capability is supported. Software queries the CPUID.0AH for the version identifier first; it then analyzes the value returned in CPUID.0AH.EAX, CPUID.0AH.EBX to determine the facilities available.

In the initial implementation of architectural performance monitoring; software can determine how many IA32_PERFEVTSELx/ IA32_PMCx MSR pairs are supported per core, the bit-width of PMC, and the number of architectural performance monitoring events available.

18.2.1.1 Architectural Performance Monitoring Version 1 Facilities

Architectural performance monitoring facilities include a set of performance monitoring counters and performance event select registers. These MSRs have the following properties:

- IA32_PMCx MSRs start at address 0C1H and occupy a contiguous block of MSR address space; the number of MSRs per logical processor is reported using CPUID.0AH:EAX[15:8].
- IA32_PERFEVTSELx MSRs start at address 186H and occupy a contiguous block of MSR address space. Each performance event select register is paired with a corresponding performance counter in the 0C1H address block.
- The bit width of an IA32_PMCx MSR is reported using the CPUID.0AH:EAX[23:16]. This the number of valid bits for read operation. On write operations, the lower-order 32 bits of the MSR may be written with any value, and the high-order bits are sign-extended from the value of bit 31.
- Bit field layout of IA32_PERFEVTSELx MSRs is defined architecturally.

See Figure 18-1 for the bit field layout of IA32_PERFEVTSELx MSRs. The bit fields are:

- **Event select field (bits 0 through 7) —** Selects the event logic unit used to detect microarchitectural conditions (see Table 18-1, for a list of architectural events and their 8-bit codes). The set of values for this field is defined architecturally; each value corresponds to an event logic unit for use with an architectural performance event. The number of architectural events is queried using CPUID.0AH:EAX. A processor may support only a subset of pre-defined values.

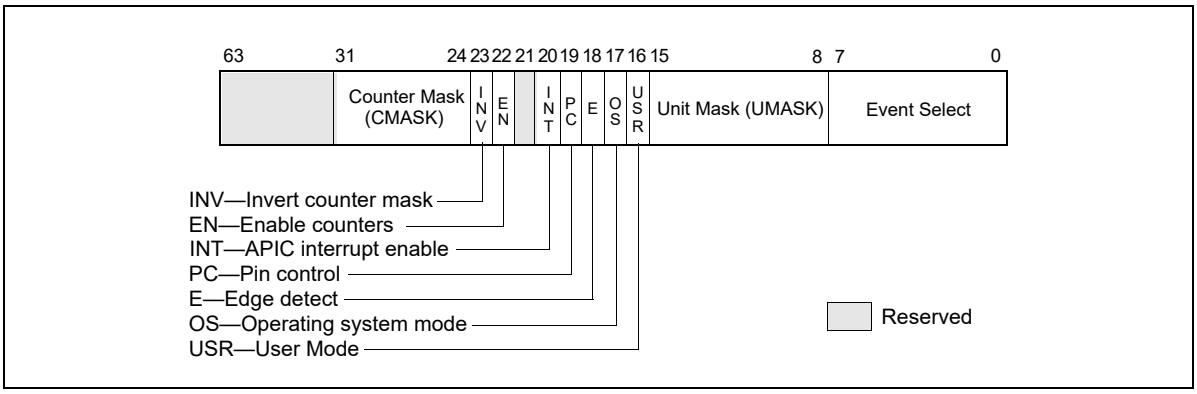


Figure 18-1. Layout of IA32_PERFEVTSELx MSRs

- **Unit mask (UMASK) field (bits 8 through 15)** — These bits qualify the condition that the selected event logic unit detects. Valid UMASK values for each event logic unit are specific to the unit. For each architectural performance event, its corresponding UMASK value defines a specific microarchitectural condition.

  A pre-defined microarchitectural condition associated with an architectural event may not be applicable to a given processor. The processor then reports only a subset of pre-defined architectural events. Pre-defined architectural events are listed in Table 18-1; support for pre-defined architectural events is enumerated using CPUID.0AH:EBX. Architectural performance events available in the initial implementation are listed in Table 19-1.
- **USR (user mode) flag (bit 16) —** Specifies that the selected microarchitectural condition is counted when the logical processor is operating at privilege levels 1, 2 or 3. This flag can be used with the OS flag.
- **OS (operating system mode) flag (bit 17) —** Specifies that the selected microarchitectural condition is counted when the logical processor is operating at privilege level 0. This flag can be used with the USR flag.
- **E (edge detect) flag (bit 18) —** Enables (when set) edge detection of the selected microarchitectural condition. The logical processor counts the number of deasserted to asserted transitions for any condition that can be expressed by the other fields. The mechanism does not permit back-to-back assertions to be distinguished.

  This mechanism allows software to measure not only the fraction of time spent in a particular state, but also the average length of time spent in such a state (for example, the time spent waiting for an interrupt to be serviced).
- **PC (pin control) flag (bit 19) —** When set, the logical processor toggles the PM*i* pins and increments the counter when performance-monitoring events occur; when clear, the processor toggles the PM*i* pins when the counter overflows. The toggling of a pin is defined as assertion of the pin for a single bus clock followed by deassertion.
- **INT (APIC interrupt enable) flag (bit 20) —** When set, the logical processor generates an exception through its local APIC on counter overflow.
- **EN (Enable Counters) Flag (bit 22) —** When set, performance counting is enabled in the corresponding performance-monitoring counter; when clear, the corresponding counter is disabled. The event logic unit for a UMASK must be disabled by setting IA32_PERFEVTSELx[bit 22] = 0, before writing to IA32_PMCx.
- **INV (invert) flag (bit 23) —** When set, inverts the counter-mask (CMASK) comparison, so that both greater than or equal to and less than comparisons can be made (0: greater than or equal; 1: less than). Note if counter-mask is programmed to zero, INV flag is ignored.

- **Counter mask (CMASK) field (bits 24 through 31) —** When this field is not zero, a logical processor compares this mask to the events count of the detected microarchitectural condition during a single cycle. If the event count is greater than or equal to this mask, the counter is incremented by one. Otherwise the counter is not incremented.

  This mask is intended for software to characterize microarchitectural conditions that can count multiple occurrences per cycle (for example, two or more instructions retired per clock; or bus queue occupations). If the counter-mask field is 0, then the counter is incremented each cycle by the event count associated with multiple occurrences.

18.2.1.2 Pre-defined Architectural Performance Events

Table 18-1 lists architecturally defined events.

Table 18-1. UMask and Event Select Encodings for Pre-Defined Architectural Performance Events

Bit Position CPUID.AH.EBX	Event Name	UMask	Event Select
0	UnHalted Core Cycles	00H	3CH
1	Instruction Retired	00H	C0H
2	UnHalted Reference Cycles	01H	3CH
3	LLC Reference	4FH	2EH
4	LLC Misses	41H	2EH
5	Branch Instruction Retired	00H	C4H
6	Branch Misses Retired	00H	C5H

A processor that supports architectural performance monitoring may not support all the predefined architectural performance events (Table 18-1). The non-zero bits in CPUID.0AH:EBX indicate the events that are not available.

The behavior of each architectural performance event is expected to be consistent on all processors that support that event. Minor variations between microarchitectures are noted below:

- **UnHalted Core Cycles —** Event select 3CH, Umask 00H

  This event counts core clock cycles when the clock signal on a specific core is running (not halted). The counter does not advance in the following conditions:

  — an ACPI C-state other than C0 for normal operation

  — HLT

  — STPCLK# pin asserted

  — being throttled by TM1

  — during the frequency switching phase of a performance state transition (see Chapter 14, “Power and Thermal Management”)

  The performance counter for this event counts across performance state transitions using different core clock frequencies

- **Instructions Retired —** Event select C0H, Umask 00H

  This event counts the number of instructions at retirement. For instructions that consist of multiple micro-ops, this event counts the retirement of the last micro-op of the instruction. An instruction with a REP prefix counts as one instruction (not per iteration). Faults before the retirement of the last micro-op of a multi-ops instruction are not counted.

  This event does not increment under VM-exit conditions. Counters continue counting during hardware interrupts, traps, and inside interrupt handlers.

- **UnHalted Reference Cycles —** Event select 3CH, Umask 01H

  This event counts reference clock cycles at a fixed frequency while the clock signal on the core is running. The event counts at a fixed frequency, irrespective of core frequency changes due to performance state transitions. Processors may implement this behavior differently. Current implementations use the core crystal clock, TSC or the bus clock. Because the rate may differ between implementations, software should calibrate it to a time source with known frequency.

- **Last Level Cache References —** Event select 2EH, Umask 4FH

  This event counts requests originating from the core that reference a cache line in the last level on-die cache. The event count includes speculation and cache line fills due to the first-level cache hardware prefetcher, but may exclude cache line fills due to other hardware-prefetchers.

  Because cache hierarchy, cache sizes and other implementation-specific characteristics; value comparison to estimate performance differences is not recommended.

- **Last Level Cache Misses —** Event select 2EH, Umask 41H

  This event counts each cache miss condition for references to the last level on-die cache. The event count may include speculation and cache line fills due to the first-level cache hardware prefetcher, but may exclude cache line fills due to other hardware-prefetchers.

  Because cache hierarchy, cache sizes and other implementation-specific characteristics; value comparison to estimate performance differences is not recommended.

- **Branch Instructions Retired —** Event select C4H, Umask 00H

  This event counts branch instructions at retirement. It counts the retirement of the last micro-op of a branch instruction.

- **All Branch Mispredict Retired —** Event select C5H, Umask 00H

  This event counts mispredicted branch instructions at retirement. It counts the retirement of the last micro-op of a branch instruction in the architectural path of execution and experienced misprediction in the branch prediction hardware.

  Branch prediction hardware is implementation-specific across microarchitectures; value comparison to estimate performance differences is not recommended.

NOTE

Programming decisions or software precisians on functionality should not be based on the event values or dependent on the existence of performance monitoring events.

18.2.2 Architectural Performance Monitoring Version 2

The enhanced features provided by architectural performance monitoring version 2 include the following:

- **Fixed-function performance counter register and associated control register** — Three of the architectural performance events are counted using three fixed-function MSRs (IA32_FIXED_CTR0 through IA32_FIXED_CTR2). Each of the fixed-function PMC can count only one architectural performance event.

  Configuring the fixed-function PMCs is done by writing to bit fields in the MSR (IA32_FIXED_CTR_CTRL) located at address 38DH. Unlike configuring performance events for general-purpose PMCs (IA32_PMCx) via UMASK field in (IA32_PERFEVTSELx), configuring, programming IA32_FIXED_CTR_CTRL for fixed-function PMCs do not require any UMASK.

- **Simplified event programming** — Most frequent operation in programming performance events are enabling/disabling event counting and checking the status of counter overflows. Architectural performance event version 2 provides three architectural MSRs:
  - — IA32_PERF_GLOBAL_CTRL allows software to enable/disable event counting of all or any combination of fixed-function PMCs (IA32_FIXED_CTRx) or any general-purpose PMCs via a single WRMSR.
  - — IA32_PERF_GLOBAL_STATUS allows software to query counter overflow conditions on any combination of fixed-function PMCs or general-purpose PMCs via a single RDMSR.
  - — IA32_PERF_GLOBAL_OVF_CTRL allows software to clear counter overflow conditions on any combination of fixed-function PMCs or general-purpose PMCs via a single WRMSR.
- **PMI Overhead Mitigation** — Architectural performance monitoring version 2 introduces two bit field interface in IA32_DEBUGCTL for PMI service routine to accumulate performance monitoring data and LBR records with reduced perturbation from servicing the PMI. The two bit fields are:
  - — IA32_DEBUGCTL.Freeze_LBR_On_PMI(bit 11). In architectural performance monitoring version 2, only the legacy semantic behavior is supported. See Section 17.4.7 for details of the legacy Freeze LBRs on PMI control.
  - — IA32_DEBUGCTL.Freeze_PerfMon_On_PMI(bit 12). In architectural performance monitoring version 2, only the legacy semantic behavior is supported. See Section 17.4.7 for details of the legacy Freeze LBRs on PMI control.

The facilities provided by architectural performance monitoring version 2 can be queried from CPUID leaf 0AH by examining the content of register EDX:

- Bits 0 through 4 of CPUID.0AH.EDX indicates the number of fixed-function performance counters available per core,
- Bits 5 through 12 of CPUID.0AH.EDX indicates the bit-width of fixed-function performance counters. Bits beyond the width of the fixed-function counter are reserved and must be written as zeros.

NOTE

Early generation of processors based on Intel Core microarchitecture may report in CPUID.0AH:EDX of support for version 2 but indicating incorrect information of version 2 facilities.

The IA32_FIXED_CTR_CTRL MSR include multiple sets of 4-bit field, each 4 bit field controls the operation of a fixed-function performance counter. Figure 18-2 shows the layout of 4-bit controls for each fixed-function PMC. Two sub-fields are currently defined within each control. The definitions of the bit fields are:

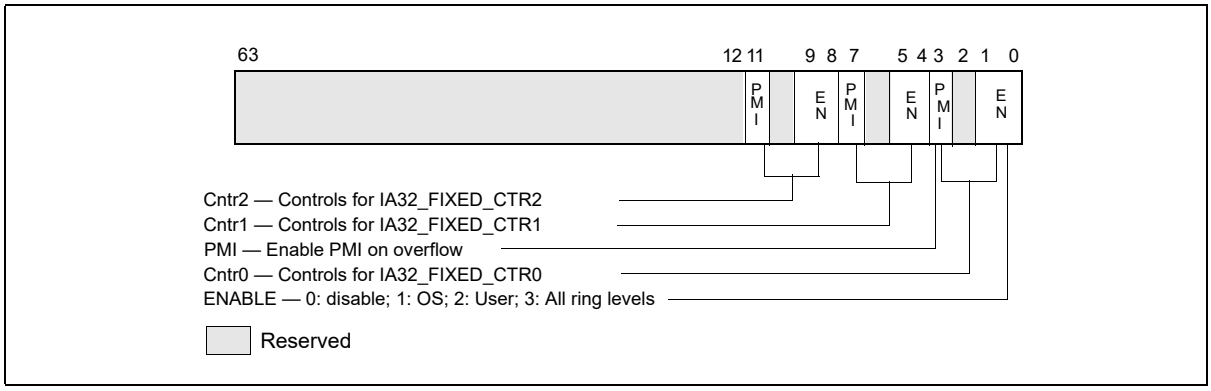


Figure 18-2. Layout of IA32_FIXED_CTR_CTRL MSR

- **Enable field (lowest 2 bits within each 4-bit control)** — When bit 0 is set, performance counting is enabled in the corresponding fixed-function performance counter to increment while the target condition associated with the architecture performance event occurred at ring 0. When bit 1 is set, performance counting is enabled in the corresponding fixed-function performance counter to increment while the target condition associated with the architecture performance event occurred at ring greater than 0. Writing 0 to both bits stops the performance counter. Writing a value of 11B enables the counter to increment irrespective of privilege levels.
- **PMI field (the fourth bit within each 4-bit control)** — When set, the logical processor generates an exception through its local APIC on overflow condition of the respective fixed-function counter.

IA32_PERF_GLOBAL_CTRL MSR provides single-bit controls to enable counting of each performance counter. Figure 18-3 shows the layout of IA32_PERF_GLOBAL_CTRL. Each enable bit in IA32_PERF_GLOBAL_CTRL is AND'ed with the enable bits for all privilege levels in the respective IA32_PERFEVTSELx or IA32_PERF_FIXED_CTR_CTRL MSRs to start/stop the counting of respective counters. Counting is enabled if the AND'ed results is true; counting is disabled when the result is false.

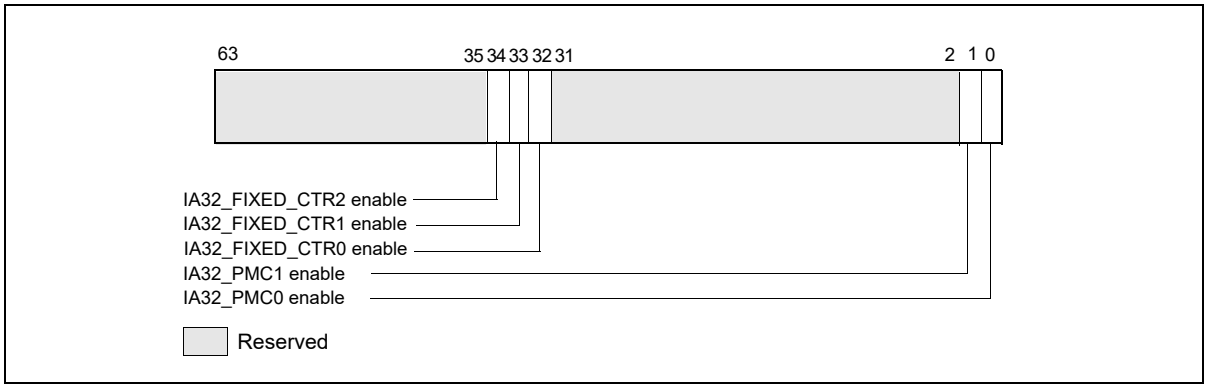


Figure 18-3. Layout of IA32_PERF_GLOBAL_CTRL MSR

The behavior of the fixed function performance counters supported by architectural performance version 2 is expected to be consistent on all processors that support those counters, and is defined as follows.

Table 18-2. Association of Fixed-Function Performance Counters with Architectural Performance Events

Fixed-Function Performance Counter	Address	Event Mask Mnemonic	Description
MSR_PERF_FIXED_CTR0/IA32_FIXED_CTR0	309H	INST_RETIRED.ANY	This event counts the number of instructions that retire execution. For instructions that consist of multiple uops, this event counts the retirement of the last uop of the instruction. The counter continues counting during hardware interrupts, traps, and in-side interrupt handlers.
MSR_PERF_FIXED_CTR1//IA32_FIXED_CTR1	30AH	CPU_CLK_UNHALTED.THREAD CPU_CLK_UNHALTED.CORE	The CPU_CLK_UNHALTED.THREAD event counts the number of core cycles while the logical processor is not in a halt state. If there is only one logical processor in a processor core, CPU_CLK_UNHALTED.CORE counts the unhalted cycles of the processor core. The core frequency may change from time to time due to transitions associated with Enhanced Intel SpeedStep Technology or TM2. For this reason this event may have a changing ratio with regards to time.
MSR_PERF_FIXED_CTR2//IA32_FIXED_CTR2	30BH	CPU_CLK_UNHALTED.REF_TSC	This event counts the number of reference cycles at the TSC rate when the core is not in a halt state and not in a TM stop-clock state. The core enters the halt state when it is running the HLT instruction or the MWAIT instruction. This event is not affected by core frequency changes (e.g., P states) but counts at the same frequency as the time stamp counter. This event can approximate elapsed time while the core was not in a halt state and not in a TM stopclock state.

IA32_PERF_GLOBAL_STATUS MSR provides single-bit status for software to query the overflow condition of each performance counter. IA32_PERF_GLOBAL_STATUS[bit 62] indicates overflow conditions of the DS area data buffer. IA32_PERF_GLOBAL_STATUS[bit 63] provides a CondChgd bit to indicate changes to the state of performance monitoring hardware. Figure 18-4 shows the layout of IA32_PERF_GLOBAL_STATUS. A value of 1 in bits 0, 1, 32 through 34 indicates a counter overflow condition has occurred in the associated counter.

When a performance counter is configured for PEBS, overflow condition in the counter generates a performance-monitoring interrupt signaling a PEBS event. On a PEBS event, the processor stores data records into the buffer area (see Section 18.15.5), clears the counter overflow status., and sets the "OvfBuffer" bit in IA32_PERF_GLOBAL_STATUS.

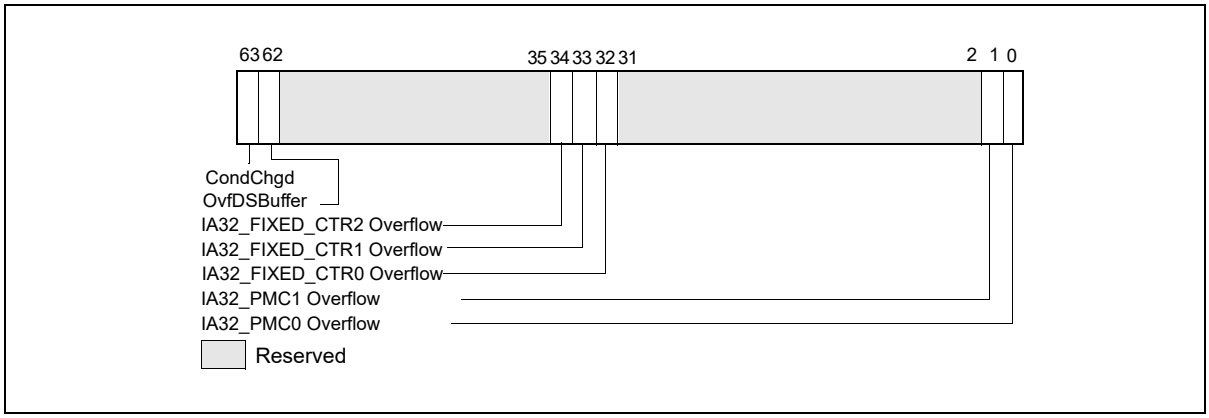


Figure 18-4. Layout of IA32_PERF_GLOBAL_STATUS MSR

IA32_PERF_GLOBAL_OVF_CTL MSR allows software to clear overflow indicator(s) of any general-purpose or fixed-function counters via a single WRMSR. Software should clear overflow indications when

- Setting up new values in the event select and/or UMASK field for counting or interrupt-based event sampling.
- Reloading counter values to continue collecting next sample.
- Disabling event counting or interrupt-based event sampling.

The layout of IA32_PERF_GLOBAL_OVF_CTL is shown in Figure 18-5.

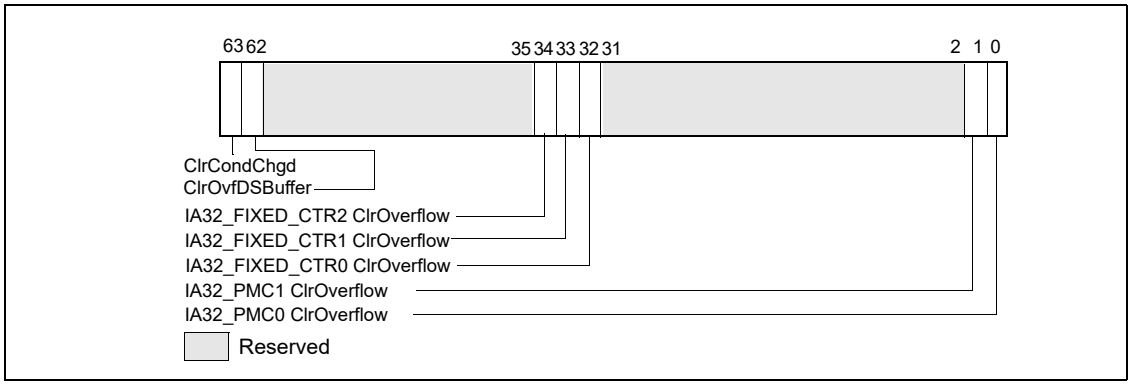


Figure 18-5. Layout of IA32_PERF_GLOBAL_OVF_CTRL MSR

18.2.3 Architectural Performance Monitoring Version 3

Processors supporting architectural performance monitoring version 3 also supports version 1 and 2, as well as capability enumerated by CPUID leaf 0AH. Specifically, version 3 provides the following enhancement in performance monitoring facilities if a processor core comprising of more than one logical processor, i.e. a processor core supporting Intel Hyper-Threading Technology or simultaneous multi-threading capability:

- AnyThread counting for processor core supporting two or more logical processors. The interface that supports AnyThread counting include:
  - Each IA32_PERFEVTSELx MSR (starting at MSR address 186H) support the bit field layout defined in Figure 18-6.

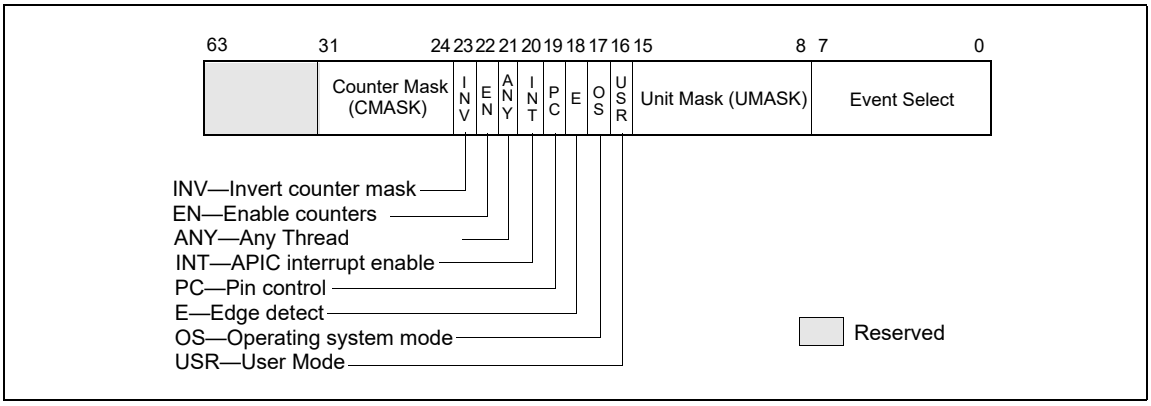


Figure 18-6. Layout of IA32_PERFEVTSELx MSRs Supporting Architectural Performance Monitoring Version 3

Bit 21 (AnyThread) of IA32_PERFEVTSELx is supported in architectural performance monitoring version 3 for processor core comprising of two or more logical processors. When set to 1, it enables counting the associated event conditions (including matching the thread's CPL with the OS/USR setting of IA32_PERFEVTSELx) occurring across all logical processors sharing a processor core. When bit 21 is 0, the counter only increments the associated event conditions (including matching the thread's CPL with the OS/USR setting of IA32_PERFEVTSELx) occurring in the logical processor which programmed the IA32_PERFEVTSELx MSR.

— Each fixed-function performance counter IA32_FIXED_CTRx (starting at MSR address 309H) is configured by a 4-bit control block in the IA32_PERF_FIXED_CTR_CTRL MSR. The control block also allow thread-specificity configuration using an AnyThread bit. The layout of IA32_PERF_FIXED_CTR_CTRL MSR is shown.

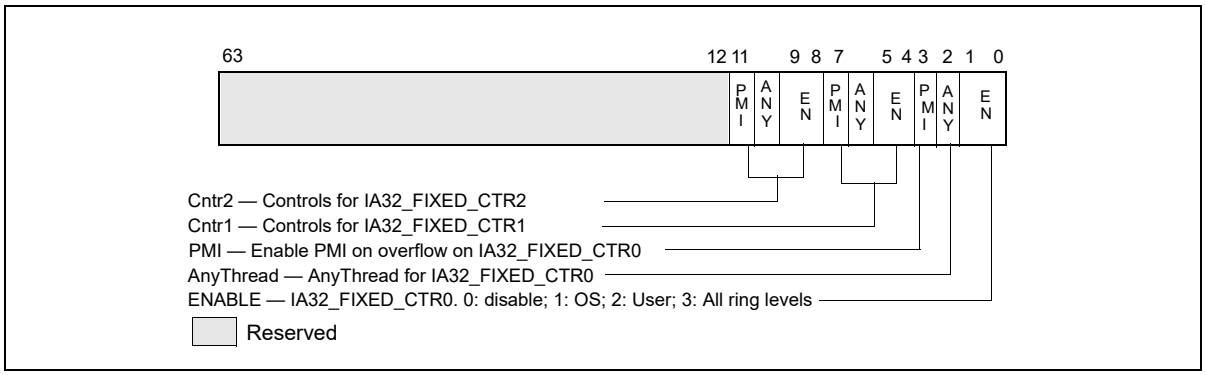


Figure 18-7. IA32_FIXED_CTR_CTRL MSR Supporting Architectural Performance Monitoring Version 3

Each control block for a fixed-function performance counter provides a **AnyThread** (bit position 2 + 4*N, N= 0, 1, etc.) bit. When set to 1, it enables counting the associated event conditions (including matching the thread's CPL with the ENABLE setting of the corresponding control block of IA32_PERF_FIXED_CTR_CTRL) occurring across all logical processors sharing a processor core. When an **AnyThread** bit is 0 in IA32_PERF_FIXED_CTR_CTRL, the corresponding fixed counter only increments the associated event conditions occurring in the logical processor which programmed the IA32_PERF_FIXED_CTR_CTRL MSR.

- The IA32_PERF_GLOBAL_CTRL, IA32_PERF_GLOBAL_STATUS, IA32_PERF_GLOBAL_OVF_CTRL MSRs provide single-bit controls/status for each general-purpose and fixed-function performance counter. Figure 18-8 and Figure 18-9 show the layout of these MSRs for N general-purpose performance counters (where N is reported by CPUID.0AH:EAX[15:8]) and three fixed-function counters.

Note: The number of general-purpose performance monitoring counters (i.e. N in Figure 18-9) can vary across processor generations within a processor family, across processor families, or could be different depending on the configuration chosen at boot time in the BIOS regarding Intel Hyper Threading Technology, (e.g. N=2 for 45 nm Intel Atom processors; N =4 for processors based on the Nehalem microarchitecture; for processors based on the Sandy Bridge microarchitecture, N = 4 if Intel Hyper Threading Technology is active and N=8 if not active).

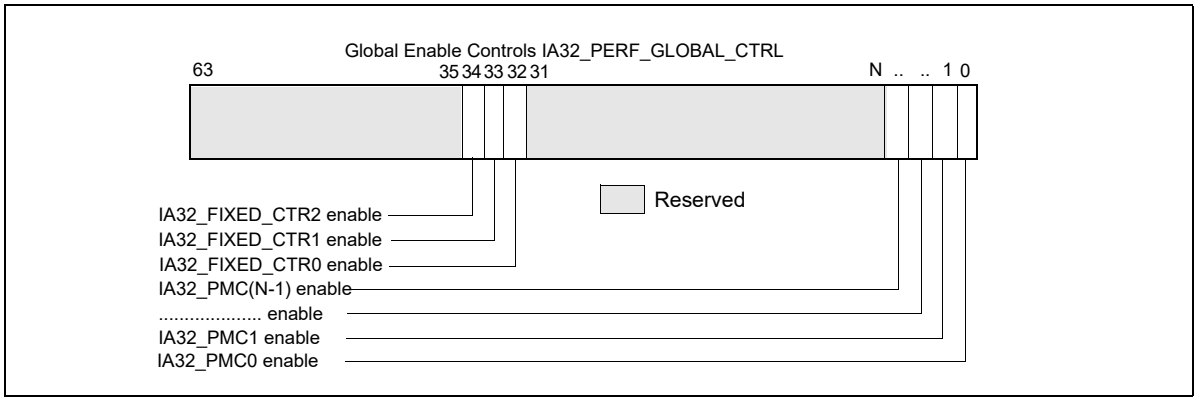


Figure 18-8. Layout of Global Performance Monitoring Control MSR

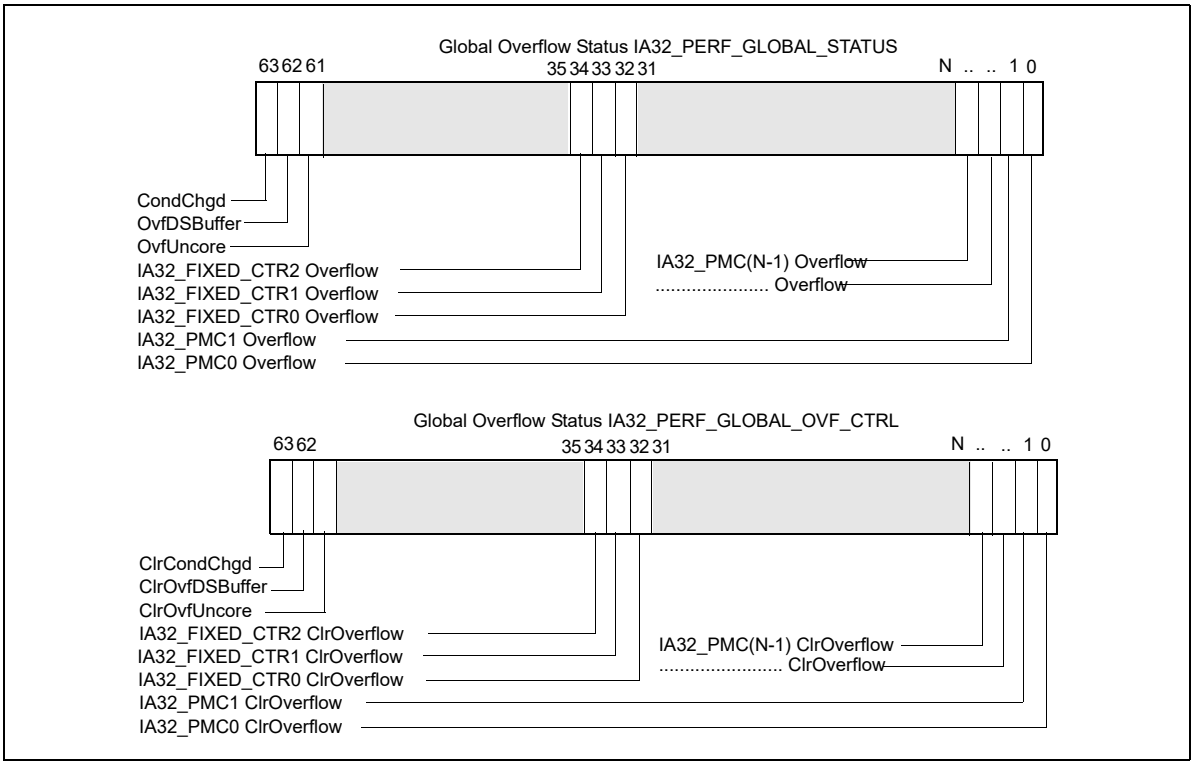


Figure 18-9. Global Performance Monitoring Overflow Status and Control MSRs

18.2.3.1 AnyThread Counting and Software Evolution

The motivation for characterizing software workload over multiple software threads running on multiple logical processors of the same processor core originates from a time earlier than the introduction of the AnyThread interface in IA32_PERFEVTSELx and IA32_FIXED_CTR_CTRL. While AnyThread counting provides some benefits in simple software environments of an earlier era, the evolution contemporary software environments introduce certain concepts and pre-requisites that AnyThread counting does not comply with.

One example is the proliferation of software environments that support multiple virtual machines (VM) under VMX (see Chapter 23, "Introduction to Virtual-Machine Extensions") where each VM represents a domain separated from one another.

A Virtual Machine Monitor (VMM) that manages the VMs may allow individual VM to employ performance monitoring facilities to profiles the performance characteristics of a workload. The use of the Anythread interface in IA32_PERFEVTSELx and IA32_FIXED_CTR_CTRL is discouraged with software environments supporting virtualization or requiring domain separation.

Specifically, Intel recommends VMM:

- configure the MSR bitmap to cause VM-exits for WRMSR to IA32_PERFEVTSELx and IA32_FIXED_CTR_CTRL in VMX non-Root operation (see CHAPTER 24 for additional information),
- clear the AnyThread bit of IA32_PERFEVTSELx and IA32_FIXED_CTR_CTRL in the MSR-load lists for VM exits and VM entries (see CHAPTER 24, CHAPTER 26, and CHAPTER 27).

Even when operating in simpler legacy software environments which might not emphasize the pre-requisites of a virtualized software environment, the use of the AnyThread interface should be moderated and follow any event-specific guidance where explicitly noted (see relevant sections of Chapter 19, "Performance Monitoring Events").

18.2.4 Architectural Performance Monitoring Version 4

Processors supporting architectural performance monitoring version 4 also supports version 1, 2, and 3, as well as capability enumerated by CPUID leaf 0AH. Version 4 introduced a streamlined PMI overhead mitigation interface that replaces the legacy semantic behavior but retains the same control interface in IA32_DEBUGCTL.Freeze_LBRs_On_PMI and Freeze_PerfMon_On_PMI. Specifically version 4 provides the following enhancement:

- New indicators (LBR_FRZ, CTR_FRZ) in IA32_PERF_GLOBAL_STATUS, see Section 18.2.4.1.
- Streamlined Freeze/PMI Overhead management interfaces to use IA32_DEBUGCTL.Freeze_LBRs_On_PMI and IA32_DEBUGCTL.Freeze_PerfMon_On_PMI: see Section 18.2.4.1. Legacy semantics of Freeze_LBRs_On_PMI and Freeze_PerfMon_On_PMI (applicable to version 2 and 3) are not supported with version 4 or higher.
- Fine-grain separation of control interface to manage overflow/status of IA32_PERF_GLOBAL_STATUS and read-only performance counter enabling interface in IA32_PERF_GLOBAL_STATUS: see Section 18.2.4.2.
- Performance monitoring resource in-use MSR to facilitate cooperative sharing protocol between perfmon-managing privilege agents.

18.2.4.1 Enhancement in IA32_PERF_GLOBAL_STATUS

The IA32_PERF_GLOBAL_STATUS MSR provides the following indicators with architectural performance monitoring version 4:

- IA32_PERF_GLOBAL_STATUS.LBR_FRZ[bit 58]: This bit is set due to the following conditions:
  - — IA32_DEBUGCTL.FREEZE_LBR_ON_PMI has been set by the profiling agent, and
  - — A performance counter, configured to generate PMI, has overflowed to signal a PMI. Consequently the LBR stack is frozen.

  Effectively, the IA32_PERF_GLOBAL_STATUS.LBR_FRZ bit also serve as an read-only control to enable capturing data in the LBR stack. To enable capturing LBR records, the following expression must hold with architectural perfmon version 4 or higher:
  - — (IA32_DEBUGCTL.LBR & (!IA32_PERF_GLOBAL_STATUS.LBR_FRZ)) =1

- IA32_PERF_GLOBAL_STATUS.CTR_FRZ[bit 59]: This bit is set due to the following conditions:
  - — IA32_DEBUGCTL.FREEZE_PERFMON_ON_PMI has been set by the profiling agent, and
  - — A performance counter, configured to generate PMI, has overflowed to signal a PMI. Consequently, all the performance counters are frozen.

  Effectively, the IA32_PERF_GLOBAL_STATUS.CTR_FRZ bit also serve as an read-only control to enable programmable performance counters and fixed counters in the core PMU. To enable counting with the performance counters, the following expression must hold with architectural perfmon version 4 or higher:
    - (IA32_PERFEVTSELn.EN & IA32_PERF_GLOBAL_CTRL.PMCn & (!IA32_PERF_GLOBAL_STATUS.CTR_FRZ)) = 1 for programmable counter ‘n’, or
    - (IA32_PERF_FIXED_CRTL.ENi & IA32_PERF_GLOBAL_CTRL.FCi & (!IA32_PERF_GLOBAL_STATUS.CTR_FRZ)) = 1 for fixed counter ‘i’

The read-only enable interface IA32_PERF_GLOBAL_STATUS.CTR_FRZ provides a more efficient flow for a PMI handler to use IA32_DEBUGCTL.Freeza_Perfmon_On_PMI to filter out data that may distort target workload analysis, see Table 17-3. It should be noted the IA32_PERF_GLOBAL_CTRL register continue to serve as the primary interface to control all performance counters of the logical processor.

For example, when the Freeze-On-PMI mode is not being used, a PMI handler would be setting IA32_PERF_GLOBAL_CTRL as the very last step to commence the overall operation after configuring the individual counter registers, controls and PEBS facility. This does not only assure atomic monitoring but also avoids unnecessary complications (e.g. race conditions) when software attempts to change the core PMU configuration while some counters are kept enabled.

Additionally, IA32_PERF_GLOBAL_STATUS.TraceToPAPMI[bit 55]: On processors that support Intel Processor Trace and configured to store trace output packets to physical memory using the ToPA scheme, bit 55 is set when a PMI occurred due to a ToPA entry memory buffer was completely filled.

IA32_PERF_GLOBAL_STATUS also provides an indicator to distinguish interaction of performance monitoring operations with other side-band activities, which apply Intel SGX on processors that support SGX (For additional information about Intel SGX, see “Intel® Software Guard Extensions Programming Reference”.):

- IA32_PERF_GLOBAL_STATUS.ASCI[bit 60]: This bit is set when data accumulated in any of the configured performance counters (i.e. IA32_PMCx or IA32_FIXED_CTRx) may include contributions from direct or indirect operation of Intel SGX to protect an enclave (since the last time IA32_PERF_GLOBAL_STATUS.ASCI was cleared).

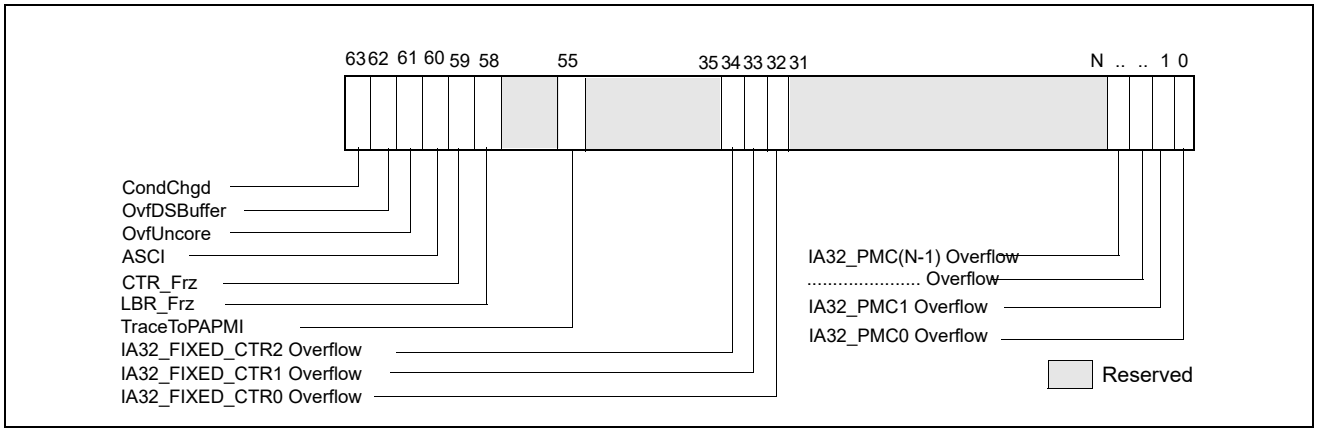


Figure 18-10. IA32_PERF_GLOBAL_STATUS MSR and Architectural Perfmon Version 4

Note, a processor’s support for IA32_PERF_GLOBAL_STATUS.TraceToPAPMI[bit 55] is enumerated as a result of CPUID enumerated capability of Intel Processor Trace and the use of the ToPA buffer scheme. Support of IA32_PERF_GLOBAL_STATUS.ASCI[bit 60] is enumerated by the CPUID enumeration of Intel SGX.

18.2.4.2 IA32_PERF_GLOBAL_STATUS_RESET and IA32_PERF_GLOBAL_STATUS_SET MSRS

With architectural performance monitoring version 3 and lower, clearing of the set bits in IA32_PERF_GLOBAL_STATUS MSR by software is done via IA32_PERF_GLOBAL_OVF_CTRL MSR. Starting with architectural performance monitoring version 4, software can manage the overflow and other indicators in IA32_PERF_GLOBAL_STATUS using separate interfaces to set or clear individual bits.

The address and the architecturally-defined bits of IA32_PERF_GLOBAL_OVF_CTRL is inherited by IA32_PERF_GLOBAL_STATUS_RESET (see Figure 18-11). Further, IA32_PERF_GLOBAL_STATUS_RESET provides additional bit fields to clear the new indicators in IA32_PERF_GLOBAL_STATUS described in Section 18.2.4.1.

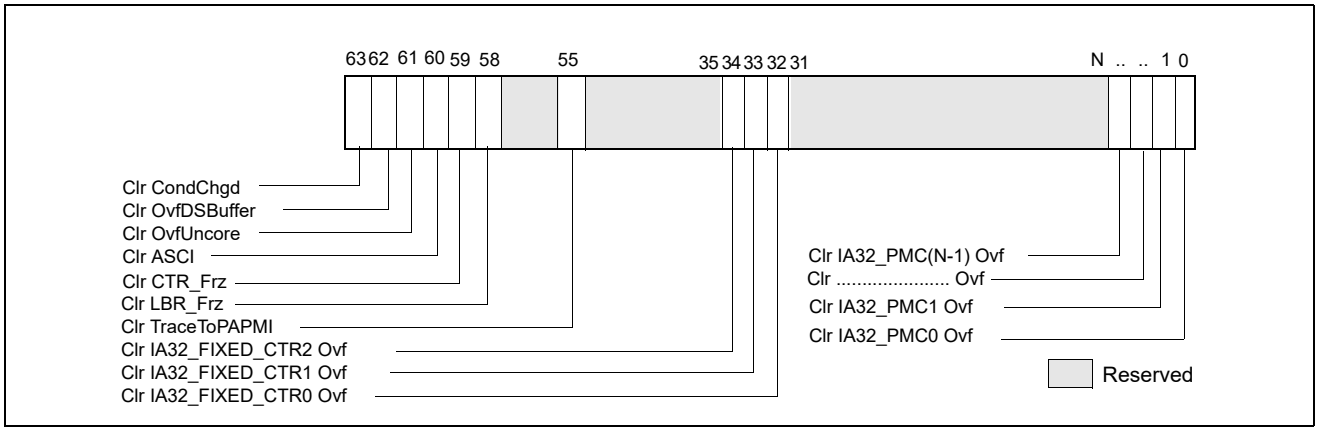


Figure 18-11. IA32_PERF_GLOBAL_STATUS_RESET MSR and Architectural Perfmon Version 4

The IA32_PERF_GLOBAL_STATUS_SET MSR is introduced with architectural performance monitoring version 4. It allows software to set individual bits in IA32_PERF_GLOBAL_STATUS. The IA32_PERF_GLOBAL_STATUS_SET interface can be used by a VMM to virtualize the state of IA32_PERF_GLOBAL_STATUS across VMs.

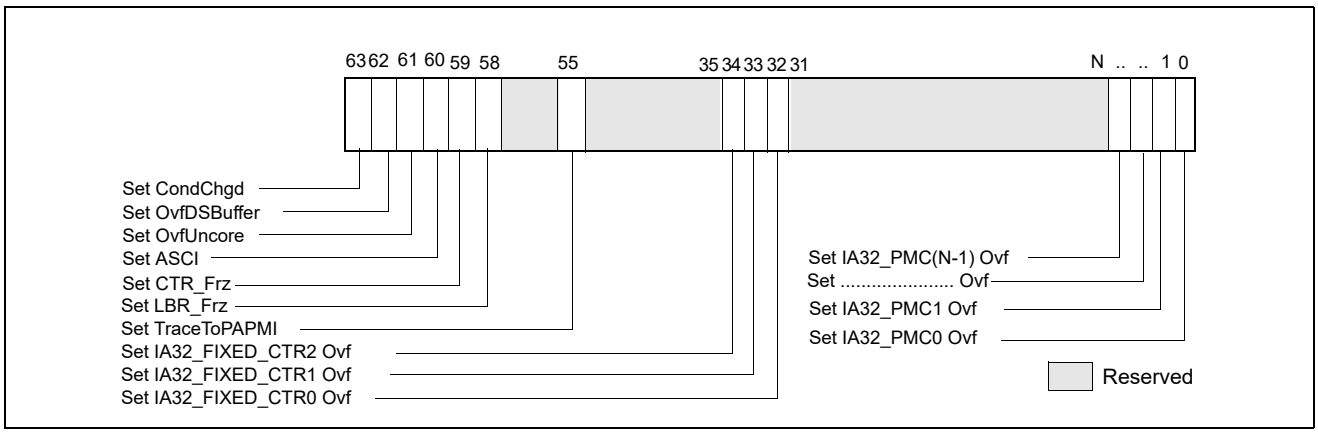


Figure 18-12. IA32_PERF_GLOBAL_STATUS_SET MSR and Architectural Perfmon Version 4

18.2.4.3 IA32_PERF_GLOBAL_INUSE MSR

In a contemporary software environment, multiple privileged service agents may wish to employ the processor's performance monitoring facilities. The IA32_MISC_ENABLE.PERFMON_AVAILABLE[bit 7] interface could not serve

the need of multiple agent adequately. A white paper, "Performance Monitoring Unit Sharing Guideline"[1], proposed a cooperative sharing protocol that is voluntary for participating software agents.

Architectural performance monitoring version 4 introduces a new MSR, IA32_PERF_GLOBAL_INUSE, that simplifies the task of multiple cooperating agents to implement the sharing protocol.

The layout of IA32_PERF_GLOBAL_INUSE is shown in Figure 18-13.

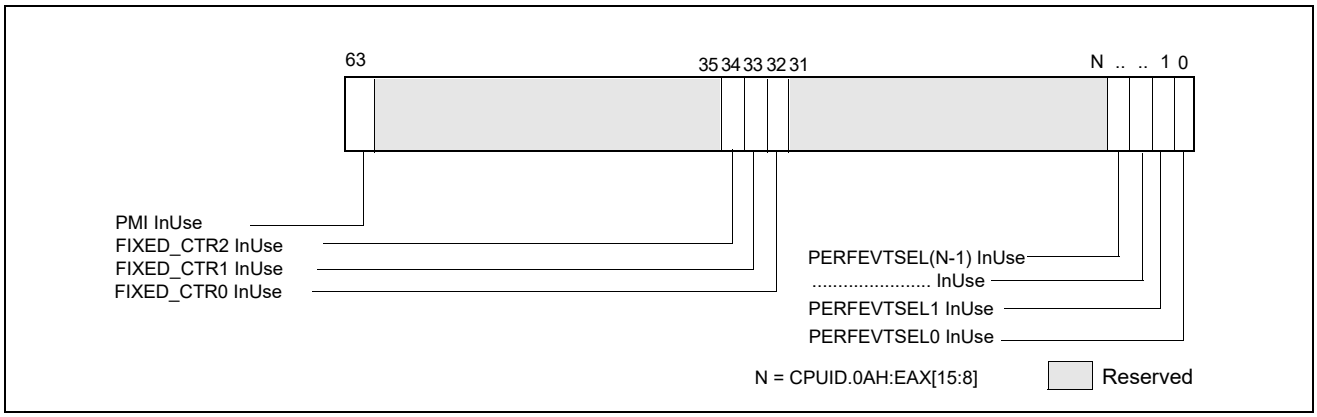


Figure 18-13. IA32_PERF_GLOBAL_INUSE MSR and Architectural Perfmon Version 4

The IA32_PERF_GLOBAL_INUSE MSR provides an "InUse" bit for each programmable performance counter and fixed counter in the processor. Additionally, it includes an indicator if the PMI mechanism has been configured by a profiling agent.

- IA32_PERF_GLOBAL_INUSE.PERFEVTSEL0_InUse[bit 0]: This bit reflects the logical state of (IA32_PERFEVTSEL0[7:0] != 0).
- IA32_PERF_GLOBAL_INUSE.PERFEVTSEL1_InUse[bit 1]: This bit reflects the logical state of (IA32_PERFEVTSEL1[7:0] != 0).
- IA32_PERF_GLOBAL_INUSE.PERFEVTSEL2_InUse[bit 2]: This bit reflects the logical state of (IA32_PERFEVTSEL2[7:0] != 0).
- IA32_PERF_GLOBAL_INUSE.PERFEVTSELn_InUse[bit n]: This bit reflects the logical state of (IA32_PERFEVTSELn[7:0] != 0), n < CPUID.0AH:EAX[15:8].
- IA32_PERF_GLOBAL_INUSE.FC0_InUse[bit 32]: This bit reflects the logical state of (IA32_FIXED_CTR_CTRL[1:0] != 0).
- IA32_PERF_GLOBAL_INUSE.FC1_InUse[bit 33]: This bit reflects the logical state of (IA32_FIXED_CTR_CTRL[5:4] != 0).
- IA32_PERF_GLOBAL_INUSE.FC2_InUse[bit 34]: This bit reflects the logical state of (IA32_FIXED_CTR_CTRL[9:8] != 0).
- IA32_PERF_GLOBAL_INUSE.PMI_InUse[bit 63]: This bit is set if any one of the following bit is set:
  - IA32_PERFEVTSELn.INT[bit 20], n < CPUID.0AH:EAX[15:8].
  - IA32_FIXED_CTR_CTRL.ENi_PMI, i = 0, 1, 2.
  - Any IA32_PEBS_ENABLES bit which enables PEBS for a general-purpose or fixed-function performance counter.

1. Available at http://www.intel.com/sdm

18.2.5 Full-Width Writes to Performance Counter Registers

The general-purpose performance counter registers IA32_PMCx are writable via WRMSR instruction. However, the value written into IA32_PMCx by WRMSR is the signed extended 64-bit value of the EAX[31:0] input of WRMSR.

A processor that supports full-width writes to the general-purpose performance counters enumerated by CPUID.0AH:EAX[15:8] will set IA32_PERF_CAPABILITIES[13] to enumerate its full-width-write capability See Figure 18-63.

If IA32_PERF_CAPABILITIES.FW_WRITE[bit 13] =1, each IA32_PMCi is accompanied by a corresponding alias address starting at 4C1H for IA32_A_PMC0.

The bit width of the performance monitoring counters is specified in CPUID.0AH:EAX[23:16].

If IA32_A_PMCi is present, the 64-bit input value (EDX:EAX) of WRMSR to IA32_A_PMCi will cause IA32_PMCi to be updated by:

COUNTERWIDTH = CPUID.0AH:EAX[23:16] bit width of the performance monitoring counter
IA32_PMCi[COUNTERWIDTH-1:32] ← EDX[COUNTERWIDTH-33:0]);
IA32_PMCi[31:0] ← EAX[31:0];
EDX[63:COUNTERWIDTH] are reserved

18.3 PERFORMANCE MONITORING (INTEL® CORE™ PROCESSORS AND INTEL® XEON® PROCESSORS)

18.3.1 Performance Monitoring for Processors Based on Intel® Microarchitecture Code Name Nehalem

Intel Core i7 processor family[1] supports architectural performance monitoring capability with version ID 3 (see Section 18.2.3) and a host of non-architectural monitoring capabilities. The Intel Core i7 processor family is based on Intel® microarchitecture code name Nehalem, and provides four general-purpose performance counters (IA32_PMC0, IA32_PMC1, IA32_PMC2, IA32_PMC3) and three fixed-function performance counters (IA32_FIXED_CTR0, IA32_FIXED_CTR1, IA32_FIXED_CTR2) in the processor core.

Non-architectural performance monitoring in Intel Core i7 processor family uses the IA32_PERFEVTSELx MSR to configure a set of non-architecture performance monitoring events to be counted by the corresponding general-purpose performance counter. The list of non-architectural performance monitoring events is listed in Table 19-28. Non-architectural performance monitoring events fall into two broad categories:

- Performance monitoring events in the processor core: These include many events that are similar to performance monitoring events available to processor based on Intel Core microarchitecture. Additionally, there are several enhancements in the performance monitoring capability for detecting microarchitectural conditions in the processor core or in the interaction of the processor core to the off-core sub-systems in the physical processor package. The off-core sub-systems in the physical processor package is loosely referred to as "uncore".
- Performance monitoring events in the uncore: The uncore sub-system is shared by more than one processor cores in the physical processor package. It provides additional performance monitoring facility outside of IA32_PMCx and performance monitoring events that are specific to the uncore sub-system.

Architectural and non-architectural performance monitoring events in Intel Core i7 processor family support thread qualification using bit 21 of IA32_PERFEVTSELx MSR.

The bit fields within each IA32_PERFEVTSELx MSR are defined in Figure 18-6 and described in Section 18.2.1.1 and Section 18.2.3.

1. Intel Xeon processor 5500 series and 3400 series are also based on Intel microarchitecture code name Nehalem; the performance monitoring facilities described in this section generally also apply.

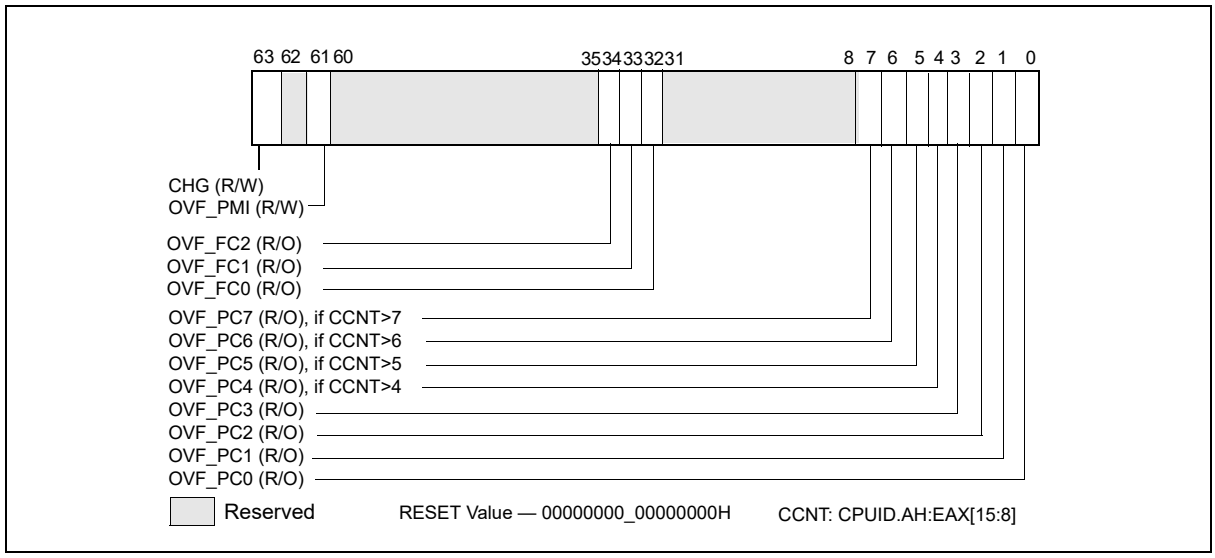


Figure 18-14. IA32_PERF_GLOBAL_STATUS MSR

18.3.1.1 Enhancements of Performance Monitoring in the Processor Core

The notable enhancements in the monitoring of performance events in the processor core include:

- Four general purpose performance counters, IA32_PMCx, associated counter configuration MSRs, IA32_PERFEVTSELx, and global counter control MSR supporting simplified control of four counters. Each of the four performance counter can support processor event based sampling (PEBS) and thread-qualification of architectural and non-architectural performance events. Width of IA32_PMCx supported by hardware has been increased. The width of counter reported by CPUID.0AH:EAX[23:16] is 48 bits. The PEBS facility in Intel micro-architecture code name Nehalem has been enhanced to include new data format to capture additional information, such as load latency.
- Load latency sampling facility. Average latency of memory load operation can be sampled using load-latency facility in processors based on Intel microarchitecture code name Nehalem. This field measures the load latency from load's first dispatch of till final data writeback from the memory subsystem. The latency is reported for retired demand load operations and in core cycles (it accounts for re-dispatches). This facility is used in conjunction with the PEBS facility.
- Off-core response counting facility. This facility in the processor core allows software to count certain transaction responses between the processor core to sub-systems outside the processor core (uncore). Counting off-core response requires additional event qualification configuration facility in conjunction with IA32_PERFEVTSELx. Two off-core response MSRs are provided to use in conjunction with specific event codes that must be specified with IA32_PERFEVTSELx.

18.3.1.1.1 Processor Event Based Sampling (PEBS)

All four general-purpose performance counters, IA32_PMCx, can be used for PEBS if the performance event supports PEBS. Software uses IA32_MISC_ENABLE[7] and IA32_MISC_ENABLE[12] to detect whether the performance monitoring facility and PEBS functionality are supported in the processor. The MSR IA32_PEBS_ENABLE provides 4 bits that software must use to enable which IA32_PMCx overflow condition will cause the PEBS record to be captured.

Additionally, the PEBS record is expanded to allow latency information to be captured. The MSR IA32_PEBS_ENABLE provides 4 additional bits that software must use to enable latency data recording in the PEBS record upon the respective IA32_PMCx overflow condition. The layout of IA32_PEBS_ENABLE for processors based on Intel microarchitecture code name Nehalem is shown in Figure 18-15.

When a counter is enabled to capture machine state (PEBS_EN_PMCx = 1), the processor will write machine state information to a memory buffer specified by software as detailed below. When the counter IA32_PMCx overflows from maximum count to zero, the PEBS hardware is armed.

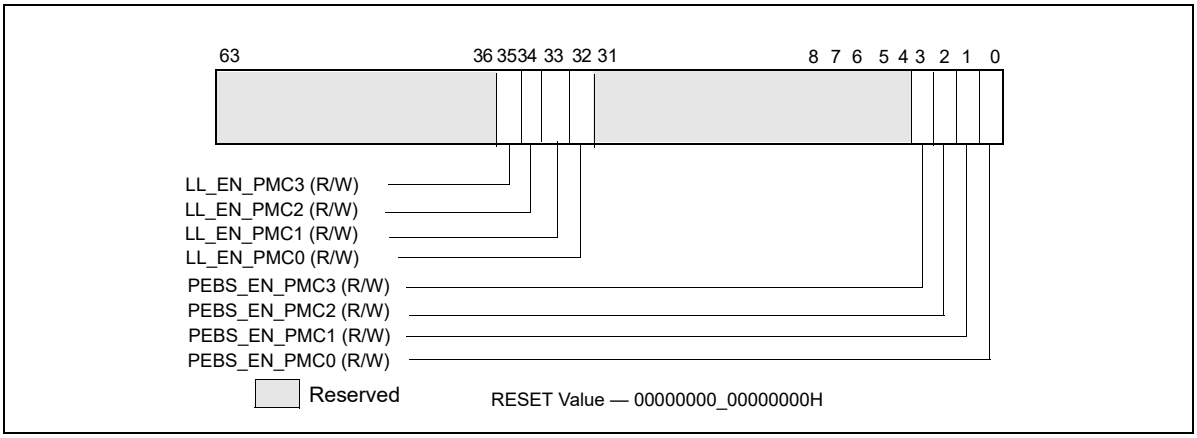


Figure 18-15. Layout of IA32_PEBS_ENABLE MSR

Upon occurrence of the next PEBS event, the PEBS hardware triggers an assist and causes a PEBS record to be written. The format of the PEBS record is indicated by the bit field IA32_PERF_CAPABILITIES[11:8] (see Figure 18-63).

The behavior of PEBS assists is reported by IA32_PERF_CAPABILITIES[6] (see Figure 18-63). The return instruction pointer (RIP) reported in the PEBS record will point to the instruction after (+1) the instruction that causes the PEBS assist. The machine state reported in the PEBS record is the machine state after the instruction that causes the PEBS assist is retired. For instance, if the instructions:

mov eax, [eax] ; causes PEBS assist

nop

are executed, the PEBS record will report the address of the nop, and the value of EAX in the PEBS record will show the value read from memory, not the target address of the read operation.

The PEBS record format is shown in Table 18-3, and each field in the PEBS record is 64 bits long. The PEBS record format, along with debug/store area storage format, does not change regardless of IA-32e mode is active or not. CPUID.01H:ECX.DTES64[bit 2] reports whether the processor's DS storage format support is mode-independent. When set, it uses 64-bit DS storage format.

Table 18-3. PEBS Record Format for Intel Core i7 Processor Family

Byte Offset	Field	Byte Offset	Field
00H	R/EFLAGS	58H	R9
08H	R/EIP	60H	R10
10H	R/EAX	68H	R11
18H	R/EBX	70H	R12
20H	R/ECX	78H	R13
28H	R/EDX	80H	R14
30H	R/ESI	88H	R15
38H	R/EDI	90H	IA32_PERF_GLOBAL_STATUS
40H	R/EBP	98H	Data Linear Address
48H	R/ESP	A0H	Data Source Encoding

Table 18-3. PEBS Record Format for Intel Core i7 Processor Family

Byte Offset	Field	Byte Offset	Field
50H	R8	A8H	Latency value (core cycles)

In IA-32e mode, the full 64-bit value is written to the register. If the processor is not operating in IA-32e mode, 32-bit value is written to registers with bits 63:32 zeroed. Registers not defined when the processor is not in IA-32e mode are written to zero.

Bytes AFH:90H are enhancement to the PEBS record format. Support for this enhanced PEBS record format is indicated by IA32_PERF_CAPABILITIES[11:8] encoding of 0001B.

The value written to bytes 97H:90H is the state of the IA32_PERF_GLOBAL_STATUS register before the PEBS assist occurred. This value is written so software can determine which counters overflowed when this PEBS record was written. Note that this field indicates the overflow status for all counters, regardless of whether they were programmed for PEBS or not.

Programming PEBS Facility

Only a subset of non-architectural performance events in the processor support PEBS. The subset of precise events are listed in Table 18-68. In addition to using IA32_PERFEVTSELx to specify event unit/mask settings and setting the EN_PMCx bit in the IA32_PEBS_ENABLE register for the respective counter, the software must also initialize the DS_BUFFER_MANAGEMENT_AREA data structure in memory to support capturing PEBS records for precise events.

NOTE

PEBS events are only valid when the following fields of IA32_PERFEVTSELx are all zero: AnyThread, Edge, Invert, CMask.

The beginning linear address of the DS_BUFFER_MANAGEMENT_AREA data structure must be programmed into the IA32_DS_AREA register. The layout of the DS_BUFFER_MANAGEMENT_AREA is shown in Figure 18-16.

- **PEBS Buffer Base**: This field is programmed with the linear address of the first byte of the PEBS buffer allocated by software. The processor reads this field to determine the base address of the PEBS buffer. Software should allocate this memory from the non-paged pool.
- **PEBS Index**: This field is initially programmed with the same value as the PEBS Buffer Base field, or the beginning linear address of the PEBS buffer. The processor reads this field to determine the location of the next PEBS record to write to. After a PEBS record has been written, the processor also updates this field with the address of the next PEBS record to be written. The figure above illustrates the state of PEBS Index after the first PEBS record is written.
- **PEBS Absolute Maximum**: This field represents the absolute address of the maximum length of the allocated PEBS buffer plus the starting address of the PEBS buffer. The processor will not write any PEBS record beyond the end of PEBS buffer, when **PEBS Index** equals **PEBS Absolute Maximum**. No signaling is generated when PEBS buffer is full. Software must reset the **PEBS Index** field to the beginning of the PEBS buffer address to continue capturing PEBS records.

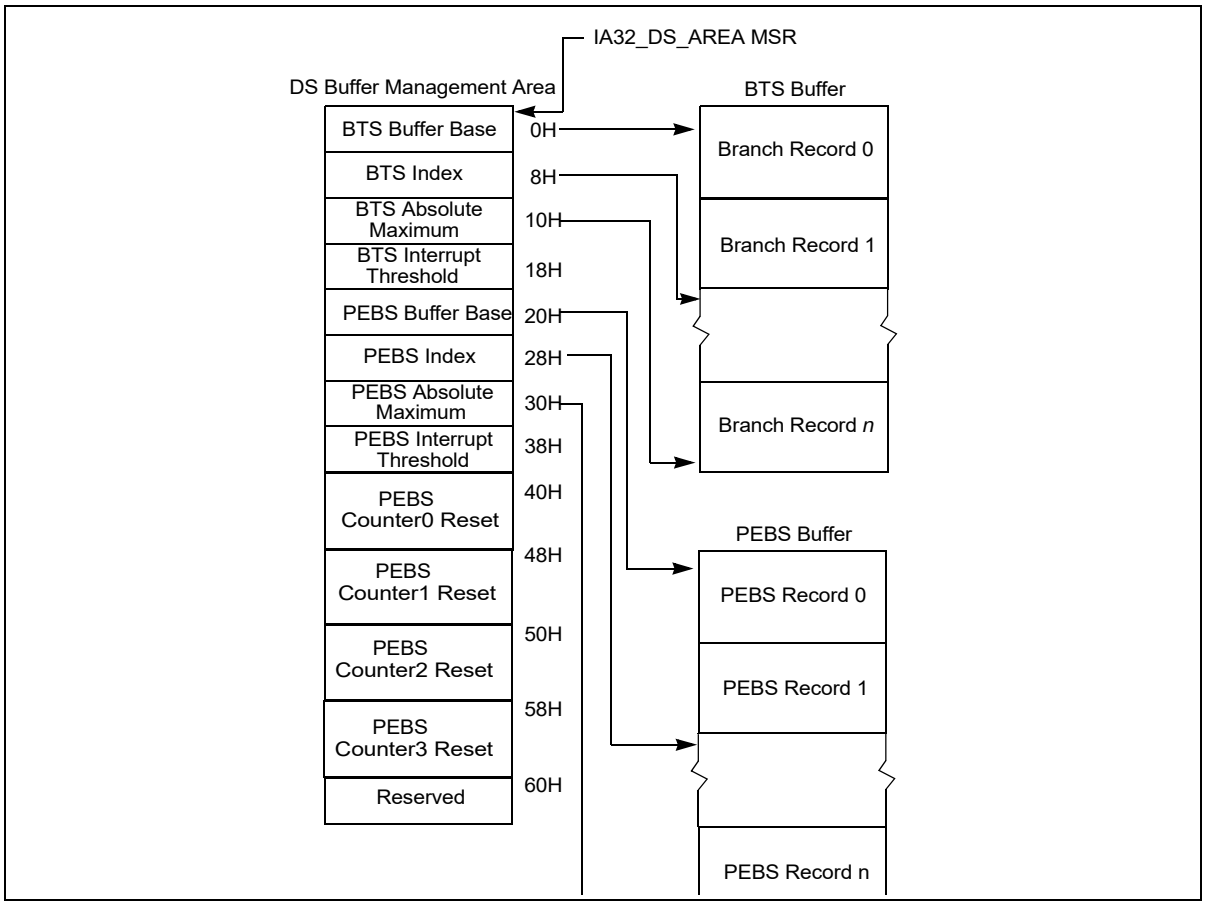


Figure 18-16. PEBS Programming Environment

- **PEBS Interrupt Threshold**: This field specifies the threshold value to trigger a performance interrupt and notify software that the PEBS buffer is nearly full. This field is programmed with the linear address of the first byte of the PEBS record within the PEBS buffer that represents the threshold record. After the processor writes a PEBS record and updates **PEBS Index**, if the **PEBS Index** reaches the threshold value of this field, the processor will generate a performance interrupt. This is the same interrupt that is generated by a performance counter overflow, as programmed in the Performance Monitoring Counters vector in the Local Vector Table of the Local APIC. When a performance interrupt due to PEBS buffer full is generated, the IA32_PERF_GLOBAL_STATUS.PEBS_Ovf bit will be set.
- **PEBS CounterX Reset**: This field allows software to set up PEBS counter overflow condition to occur at a rate useful for profiling workload, thereby generating multiple PEBS records to facilitate characterizing the profile the execution of test code. After each PEBS record is written, the processor checks each counter to see if it overflowed and was enabled for PEBS (the corresponding bit in IA32_PEBS_ENABLED was set). If these conditions are met, then the reset value for each overflowed counter is loaded from the DS Buffer Management Area. For example, if counter IA32_PMC0 caused a PEBS record to be written, then the value of "PEBS Counter 0 Reset" would be written to counter IA32_PMC0. If a counter is not enabled for PEBS, its value will not be modified by the PEBS assist.

Performance Counter Prioritization

Performance monitoring interrupts are triggered by a counter transitioning from maximum count to zero (assuming IA32_PerfEvtSelX.INT is set). This same transition will cause PEBS hardware to arm, but not trigger. PEBS hardware triggers upon detection of the first PEBS event after the PEBS hardware has been armed (a 0 to 1 transition of the counter). At this point, a PEBS assist will be undertaken by the processor.

Performance counters (fixed and general-purpose) are prioritized in index order. That is, counter IA32_PMC0 takes precedence over all other counters. Counter IA32_PMC1 takes precedence over counters IA32_PMC2 and IA32_PMC3, and so on. This means that if simultaneous overflows or PEBS assists occur, the appropriate action will be taken for the highest priority performance counter. For example, if IA32_PMC1 cause an overflow interrupt and IA32_PMC2 causes an PEBS assist simultaneously, then the overflow interrupt will be serviced first.

The PEBS threshold interrupt is triggered by the PEBS assist, and is by definition prioritized lower than the PEBS assist. Hardware will not generate separate interrupts for each counter that simultaneously overflows. General-purpose performance counters are prioritized over fixed counters.

If a counter is programmed with a precise (PEBS-enabled) event and programmed to generate a counter overflow interrupt, the PEBS assist is serviced before the counter overflow interrupt is serviced. If in addition the PEBS interrupt threshold is met, the

threshold interrupt is generated after the PEBS assist completes, followed by the counter overflow interrupt (two separate interrupts are generated).

Uncore counters may be programmed to interrupt one or more processor cores (see Section 18.3.1.2). It is possible for interrupts posted from the uncore facility to occur coincident with counter overflow interrupts from the processor core. Software must check core and uncore status registers to determine the exact origin of counter overflow interrupts.

18.3.1.1.2 Load Latency Performance Monitoring Facility

The load latency facility provides software a means to characterize the average load latency to different levels of cache/memory hierarchy. This facility requires processor supporting enhanced PEBS record format in the PEBS buffer, see Table 18-3. This field measures the load latency from load's first dispatch of till final data writeback from the memory subsystem. The latency is reported for retired demand load operations and in core cycles (it accounts for re-dispatches).

To use this feature software must assure:

- One of the IA32_PERFEVTSELx MSR is programmed to specify the event unit MEM_INST_RETIRED, and the LATENCY_ABOVE_THRESHOLD event mask must be specified (IA32_PerfEvtSelX[15:0] = 100H). The corresponding counter IA32_PMCx will accumulate event counts for architecturally visible loads which exceed the programmed latency threshold specified separately in a MSR. Stores are ignored when this event is programmed. The CMASK or INV fields of the IA32_PerfEvtSelX register used for counting load latency must be 0. Writing other values will result in undefined behavior.
- The MSR_PEBS_LD_LAT_THRESHOLD MSR is programmed with the desired latency threshold in core clock cycles. Loads with latencies greater than this value are eligible for counting and latency data reporting. The minimum value that may be programmed in this register is 3 (the minimum detectable load latency is 4 core clock cycles).
- The PEBS enable bit in the IA32_PEBS_ENABLE register is set for the corresponding IA32_PMCx counter register. This means that both the PEBS_EN_CTRX and LL_EN_CTRX bits must be set for the counter(s) of interest. For example, to enable load latency on counter IA32_PMC0, the IA32_PEBS_ENABLE register must be programmed with the 64-bit value 00000001_00000001H.

When the load-latency facility is enabled, load operations are randomly selected by hardware and tagged to carry information related to data source locality and latency. Latency and data source information of tagged loads are updated internally.

When a PEBS assist occurs, the last update of latency and data source information are captured by the assist and written as part of the PEBS record. The PEBS sample after value (SAV), specified in PEBS CounterX Reset, operates orthogonally to the tagging mechanism. Loads are randomly tagged to collect latency data. The SAV controls the number of tagged loads with latency information that will be written into the PEBS record field by the PEBS assists. The load latency data written to the PEBS record will be for the last tagged load operation which retired just before the PEBS assist was invoked.

The load-latency information written into a PEBS record (see Table 18-3, bytes AFH:98H) consists of:

- **Data Linear Address**: This is the linear address of the target of the load operation.
- **Latency Value**: This is the elapsed cycles of the tagged load operation between dispatch to GO, measured in processor core clock domain.

- **Data Source:** The encoded value indicates the origin of the data obtained by the load instruction. The encoding is shown in Table 18-4. In the descriptions local memory refers to system memory physically attached to a processor package, and remote memory referrals to system memory physically attached to another processor package.

Table 18-4. Data Source Encoding for Load Latency Record

Encoding	Description
00H	Unknown L3 cache miss
01H	Minimal latency core cache hit. This request was satisfied by the L1 data cache.
02H	Pending core cache HIT. Outstanding core cache miss to same cache-line address was already underway.
03H	This data request was satisfied by the L2.
04H	L3 HIT. Local or Remote home requests that hit L3 cache in the uncore with no coherency actions required (snooping).
05H	L3 HIT. Local or Remote home requests that hit the L3 cache and was serviced by another processor core with a cross core snoop where no modified copies were found. (clean).
06H	L3 HIT. Local or Remote home requests that hit the L3 cache and was serviced by another processor core with a cross core snoop where modified copies were found. (HITM).
07H[1]	Reserved/LLC Snoop HitM. Local or Remote home requests that hit the last level cache and was serviced by another core with a cross core snoop where modified copies found
08H	L3 MISS. Local homed requests that missed the L3 cache and was serviced by forwarded data following a cross package snoop where no modified copies found. (Remote home requests are not counted).
09H	Reserved
0AH	L3 MISS. Local home requests that missed the L3 cache and was serviced by local DRAM (go to shared state).
0BH	L3 MISS. Remote home requests that missed the L3 cache and was serviced by remote DRAM (go to shared state).
0CH	L3 MISS. Local home requests that missed the L3 cache and was serviced by local DRAM (go to exclusive state).
0DH	L3 MISS. Remote home requests that missed the L3 cache and was serviced by remote DRAM (go to exclusive state).
0EH	I/O, Request of input/output operation
0FH	The request was to un-cacheable memory.

NOTES:

1. Bit 7 is supported only for processor with CPUID DisplayFamily_DisplayModel signature of 06_2A, and 06_2E; otherwise it is reserved.

The layout of MSR_PEBS_LD_LAT_THRESHOLD is shown in Figure 18-17.

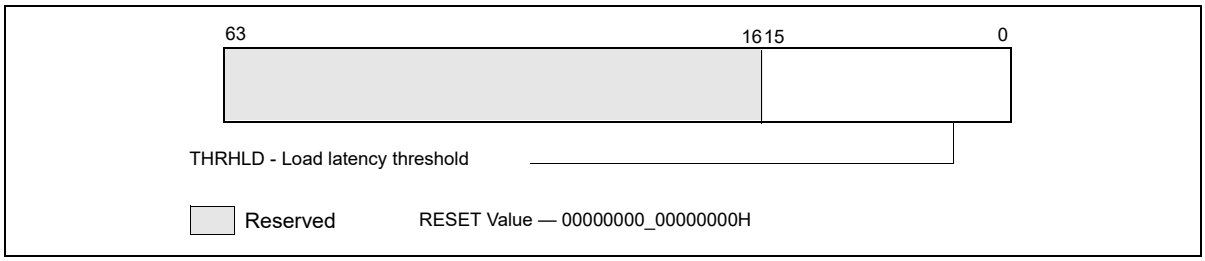


Figure 18-17. Layout of MSR_PEBS_LD_LAT MSR

Bits 15:0 specifies the threshold load latency in core clock cycles. Performance events with latencies greater than this value are counted in IA32_PMCx and their latency information is reported in the PEBS record. Otherwise, they are ignored. The minimum value that may be programmed in this field is 3.

18.3.1.1.3 Off-core Response Performance Monitoring in the Processor Core

Programming a performance event using the off-core response facility can choose any of the four IA32_PERFEVTSELx MSR with specific event codes and predefine mask bit value. Each event code for off-core response monitoring requires programming an associated configuration MSR, MSR_OFFCORE_RSP_0. There is only one off-core response configuration MSR. Table 18-5 lists the event code, mask value and additional off-core configuration MSR that must be programmed to count off-core response events using IA32_PMCx.

Table 18-5. Off-Core Response Event Encoding

Event code in IA32_PERFEVTSELx	**Mask Value in** IA32_PERFEVTSELx	**Required Off-core Response MSR**
B7H	01H	MSR_OFFCORE_RSP_0 (address 1A6H)

The layout of MSR_OFFCORE_RSP_0 is shown in Figure 18-18. Bits 7:0 specifies the request type of a transaction request to the uncore. Bits 15:8 specifies the response of the uncore subsystem.

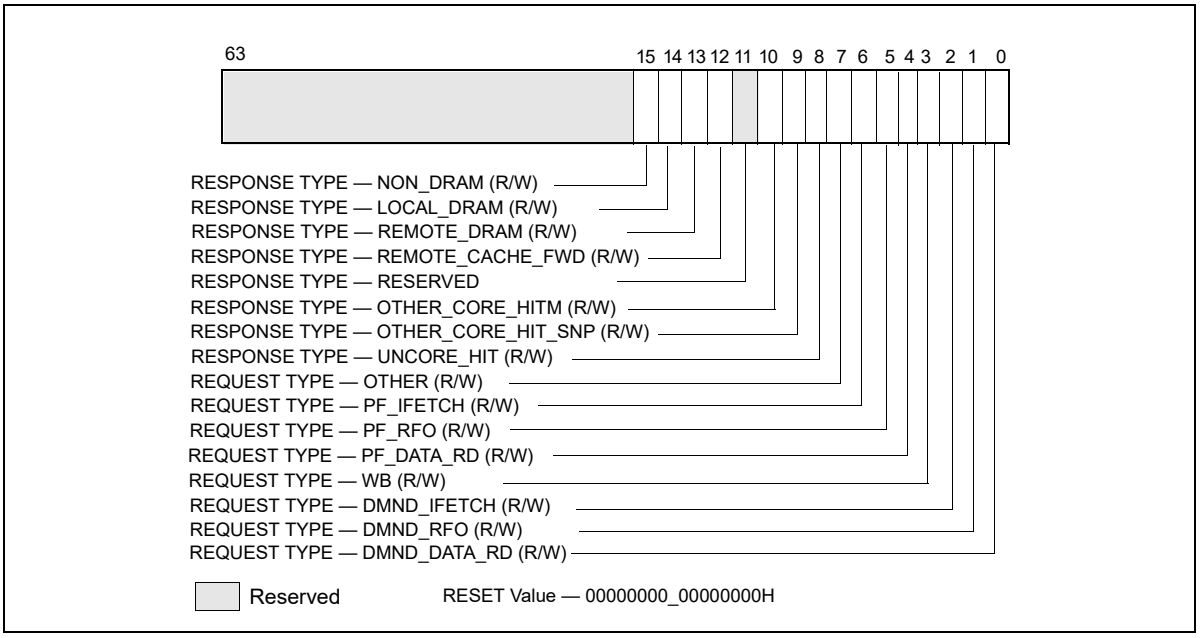


Figure 18-18. Layout of MSR_OFFCORE_RSP_0 and MSR_OFFCORE_RSP_1 to Configure Off-core Response Events

Table 18-6. MSR_OFFCORE_RSP_0 and MSR_OFFCORE_RSP_1 Bit Field Definition

Bit Name	**Offset**	**Description**
DMND_DATA_RD	0	(R/W). Counts the number of demand and DCU prefetch data reads of full and partial cachelines as well as demand data page table entry cacheline reads. Does not count L2 data read prefetches or instruction fetches.
DMND_RFO	1	(R/W). Counts the number of demand and DCU prefetch reads for ownership (RFO) requests generated by a write to data cacheline. Does not count L2 RFO.
DMND_IFETCH	2	(R/W). Counts the number of demand and DCU prefetch instruction cacheline reads. Does not count L2 code read prefetches.
WB	3	(R/W). Counts the number of writeback (modified to exclusive) transactions.
PF_DATA_RD	4	(R/W). Counts the number of data cacheline reads generated by L2 prefetchers.
PF_RFO	5	(R/W). Counts the number of RFO requests generated by L2 prefetchers.

Table 18-6. MSR_OFFCORE_RSP_0 and MSR_OFFCORE_RSP_1 Bit Field Definition (Contd.)

Bit Name	Offset	Description
PF_IFETCH	6	(R/W). Counts the number of code reads generated by L2 prefetchers.
OTHER	7	(R/W). Counts one of the following transaction types, including L3 invalidate, I/O, full or partial writes, WC or non-temporal stores, CLFLUSH, Fences, lock, unlock, split lock.
UNCORE_HIT	8	(R/W). L3 Hit: local or remote home requests that hit L3 cache in the uncore with no coherency actions required (snooping).
OTHER_CORE_HIT_SNP	9	(R/W). L3 Hit: local or remote home requests that hit L3 cache in the uncore and was serviced by another core with a cross core snoop where no modified copies were found (clean).
OTHER_CORE_HITM	10	(R/W). L3 Hit: local or remote home requests that hit L3 cache in the uncore and was serviced by another core with a cross core snoop where modified copies were found (HITM).
Reserved	11	Reserved
REMOTE_CACHE_FWD	12	(R/W). L3 Miss: local homed requests that missed the L3 cache and was serviced by forwarded data following a cross package snoop where no modified copies found. (Remote home requests are not counted)
REMOTE_DRAM	13	(R/W). L3 Miss: remote home requests that missed the L3 cache and were serviced by remote DRAM.
LOCAL_DRAM	14	(R/W). L3 Miss: local home requests that missed the L3 cache and were serviced by local DRAM.
NON_DRAM	15	(R/W). Non-DRAM requests that were serviced by IOH.

18.3.1.2 Performance Monitoring Facility in the Uncore

The "uncore" in Intel microarchitecture code name Nehalem refers to subsystems in the physical processor package that are shared by multiple processor cores. Some of the sub-systems in the uncore include the L3 cache, Intel QuickPath Interconnect link logic, and integrated memory controller. The performance monitoring facilities inside the uncore operates in the same clock domain as the uncore (U-clock domain), which is usually different from the processor core clock domain. The uncore performance monitoring facilities described in this section apply to Intel Xeon processor 5500 series and processors with the following CPUID signatures: 06_1AH, 06_1EH, 06_1FH (see Chapter 2, "Model-Specific Registers (MSRs)" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 4*). An overview of the uncore performance monitoring facilities is described separately.

The performance monitoring facilities available in the U-clock domain consist of:

- Eight General-purpose counters (MSR_UNCORE_PerfCntr0 through MSR_UNCORE_PerfCntr7). The counters are 48 bits wide. Each counter is associated with a configuration MSR, MSR_UNCORE_PerfEvtSelx, to specify event code, event mask and other event qualification fields. A set of global uncore performance counter enabling/overflow/status control MSRs are also provided for software.
- Performance monitoring in the uncore provides an address/opcode match MSR that provides event qualification control based on address value or QPI command opcode.
- One fixed-function counter, MSR_UNCORE_FixedCntr0. The fixed-function uncore counter increments at the rate of the U-clock when enabled.

  The frequency of the uncore clock domain can be determined from the uncore clock ratio which is available in the PCI configuration space register at offset C0H under device number 0 and Function 0.

18.3.1.2.1 Uncore Performance Monitoring Management Facility

MSR_UNCORE_PERF_GLOBAL_CTRL provides bit fields to enable/disable general-purpose and fixed-function counters in the uncore. Figure 18-19 shows the layout of MSR_UNCORE_PERF_GLOBAL_CTRL for an uncore that is shared by four processor cores in a physical package.

- EN_PCn (bit n, n = 0, 7): When set, enables counting for the general-purpose uncore counter MSR_UNCORE_PerfCntr n.
- EN_FC0 (bit 32): When set, enables counting for the fixed-function uncore counter MSR_UNCORE_FixedCntr0.

- EN_PMI_COREn (bit n, n = 0, 3 if four cores are present): When set, processor core n is programmed to receive an interrupt signal from any interrupt enabled uncore counter. PMI delivery due to an uncore counter overflow is enabled by setting IA32_DEBUGCTL.Offcore_PMI_EN to 1.
- PMI_FRZ (bit 63): When set, all U-clock uncore counters are disabled when any one of them signals a performance interrupt. Software must explicitly re-enable the counter by setting the enable bits in MSR_UNCORE_PERF_GLOBAL_CTRL upon exit from the ISR.

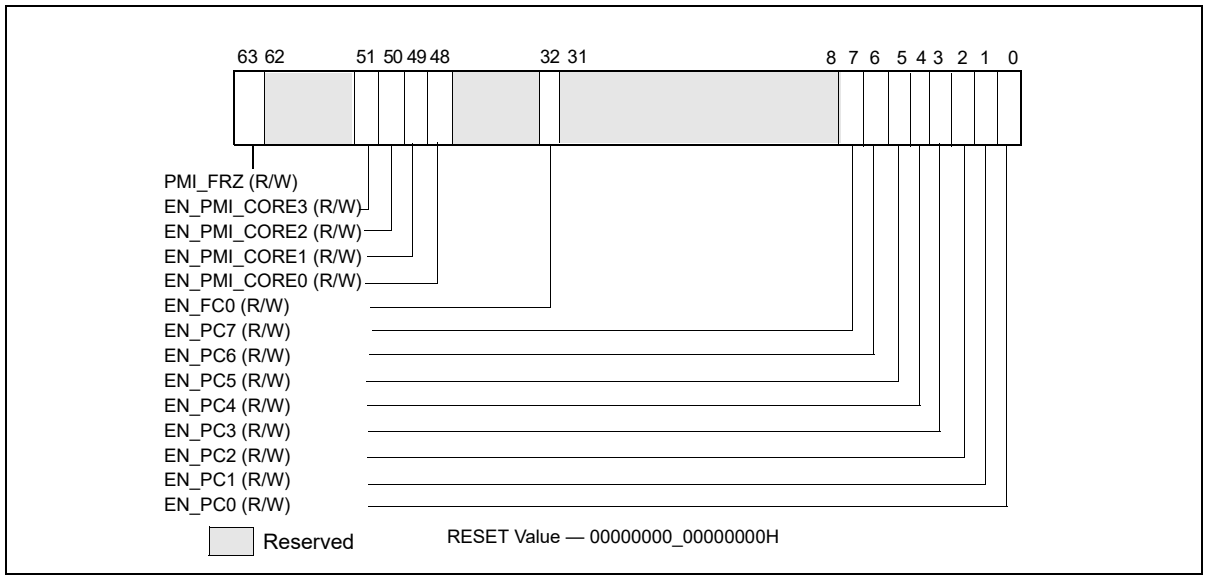


Figure 18-19. Layout of MSR_UNCORE_PERF_GLOBAL_CTRL MSR

MSR_UNCORE_PERF_GLOBAL_STATUS provides overflow status of the U-clock performance counters in the uncore. This is a read-only register. If an overflow status bit is set the corresponding counter has overflowed. The register provides a condition change bit (bit 63) which can be quickly checked by software to determine if a significant change has occurred since the last time the condition change status was cleared. Figure 18-20 shows the layout of MSR_UNCORE_PERF_GLOBAL_STATUS.

- OVF_PCn (bit n, n = 0, 7): When set, indicates general-purpose uncore counter MSR_UNCORE_PerfCntr n has overflowed.
- OVF_FC0 (bit 32): When set, indicates the fixed-function uncore counter MSR_UNCORE_FixedCntr0 has overflowed.
- OVF_PMI (bit 61): When set indicates that an uncore counter overflowed and generated an interrupt request.
- CHG (bit 63): When set indicates that at least one status bit in MSR_UNCORE_PERF_GLOBAL_STATUS register has changed state.

MSR_UNCORE_PERF_GLOBAL_OVF_CTRL allows software to clear the status bits in the UNCORE_PERF_GLOBAL_STATUS register. This is a write-only register, and individual status bits in the global status register are cleared by writing a binary one to the corresponding bit in this register. Writing zero to any bit position in this register has no effect on the uncore PMU hardware.

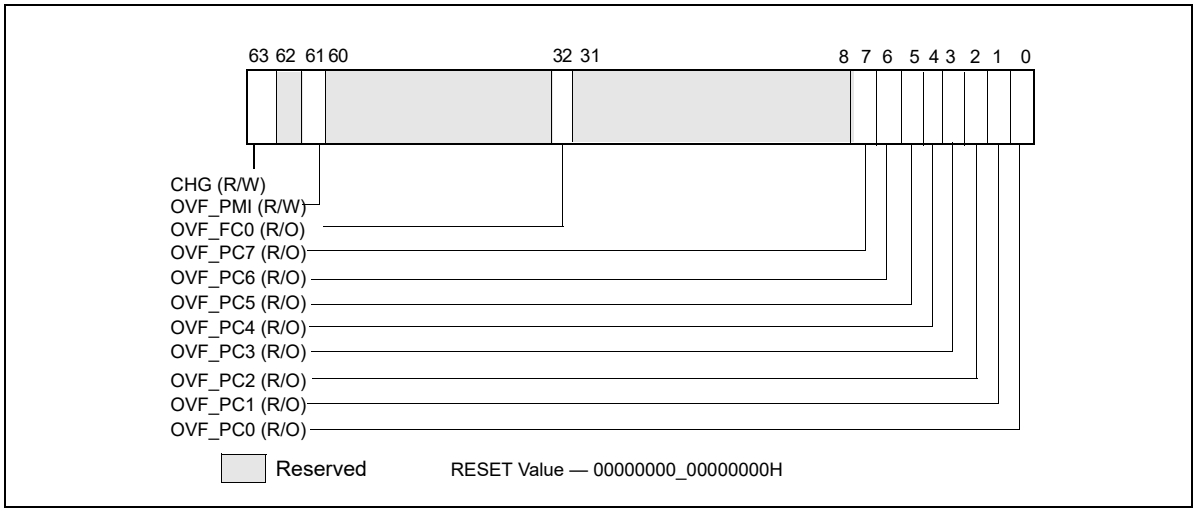


Figure 18-20. Layout of MSR_UNCORE_PERF_GLOBAL_STATUS MSR

Figure 18-21 shows the layout of MSR_UNCORE_PERF_GLOBAL_OVF_CTRL.

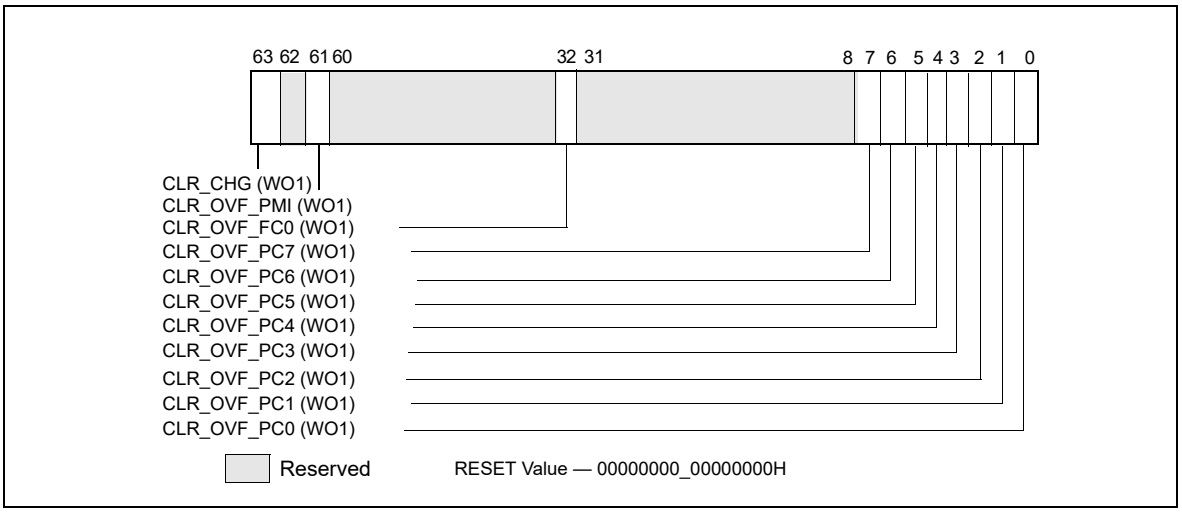


Figure 18-21. Layout of MSR_UNCORE_PERF_GLOBAL_OVF_CTRL MSR

- CLR_OVF_PCn (bit n, n = 0, 7): Set this bit to clear the overflow status for general-purpose uncore counter MSR_UNCORE_PerfCntr n. Writing a value other than 1 is ignored.
- CLR_OVF_FC0 (bit 32): Set this bit to clear the overflow status for the fixed-function uncore counter MSR_UNCORE_FixedCntr0. Writing a value other than 1 is ignored.
- CLR_OVF_PMI (bit 61): Set this bit to clear the OVF_PMI flag in MSR_UNCORE_PERF_GLOBAL_STATUS. Writing a value other than 1 is ignored.
- CLR_CHG (bit 63): Set this bit to clear the CHG flag in MSR_UNCORE_PERF_GLOBAL_STATUS register. Writing a value other than 1 is ignored.

18.3.1.2.2 Uncore Performance Event Configuration Facility

MSR_UNCORE_PerfEvtSel0 through MSR_UNCORE_PerfEvtSel7 are used to select performance event and configure the counting behavior of the respective uncore performance counter. Each uncore PerfEvtSel MSR is paired with an uncore performance counter. Each uncore counter must be locally configured using the corresponding MSR_UNCORE_PerfEvtSelx and counting must be enabled using the respective EN_PCx bit in MSR_UNCORE_PERF_GLOBAL_CTRL. Figure 18-22 shows the layout of MSR_UNCORE_PERFEVTSELx.

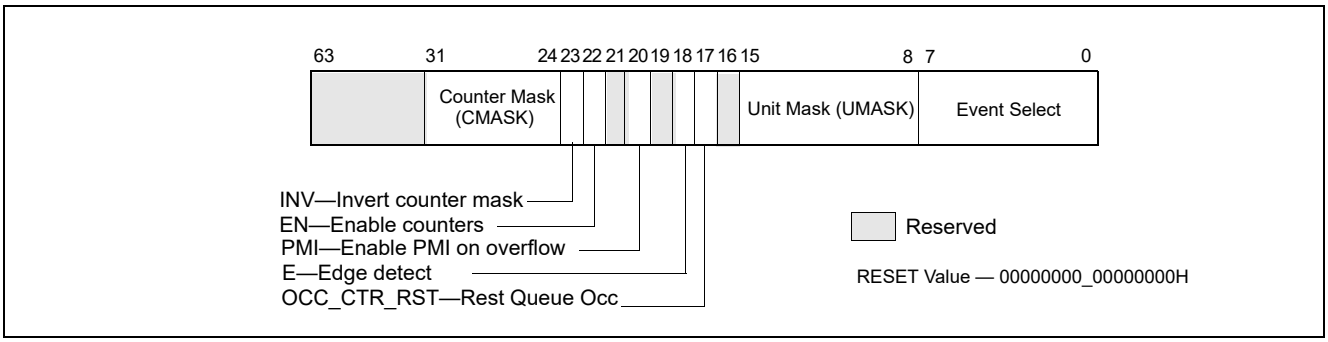


Figure 18-22. Layout of MSR_UNCORE_PERFEVTSELx MSRs

- Event Select (bits 7:0): Selects the event logic unit used to detect uncore events.
- Unit Mask (bits 15:8) : Condition qualifiers for the event selection logic specified in the Event Select field.
- OCC_CTR_RST (bit17): When set causes the queue occupancy counter associated with this event to be cleared (zeroed). Writing a zero to this bit will be ignored. It will always read as a zero.
- Edge Detect (bit 18): When set causes the counter to increment when a deasserted to asserted transition occurs for the conditions that can be expressed by any of the fields in this register.
- PMI (bit 20): When set, the uncore will generate an interrupt request when this counter overflowed. This request will be routed to the logical processors as enabled in the PMI enable bits (EN_PMI_COREx) in the register MSR_UNCORE_PERF_GLOBAL_CTRL.
- EN (bit 22): When clear, this counter is locally disabled. When set, this counter is locally enabled and counting starts when the corresponding EN_PCx bit in MSR_UNCORE_PERF_GLOBAL_CTRL is set.
- INV (bit 23): When clear, the Counter Mask field is interpreted as greater than or equal to. When set, the Counter Mask field is interpreted as less than.
- Counter Mask (bits 31:24): When this field is clear, it has no effect on counting. When set to a value other than zero, the logical processor compares this field to the event counts on each core clock cycle. If INV is clear and the event counts are greater than or equal to this field, the counter is incremented by one. If INV is set and the event counts are less than this field, the counter is incremented by one. Otherwise the counter is not incremented.

Figure 18-23 shows the layout of MSR_UNCORE_FIXED_CTR_CTRL.

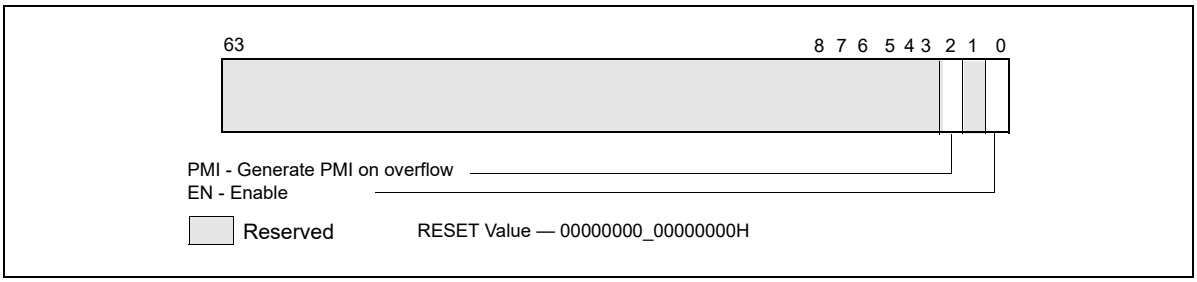


Figure 18-23. Layout of MSR_UNCORE_FIXED_CTR_CTRL MSR

- EN (bit 0): When clear, the uncore fixed-function counter is locally disabled. When set, it is locally enabled and counting starts when the EN_FC0 bit in MSR_UNCORE_PERF_GLOBAL_CTRL is set.
- PMI (bit 2): When set, the uncore will generate an interrupt request when the uncore fixed-function counter overflowed. This request will be routed to the logical processors as enabled in the PMI enable bits (EN_PMI_COREx) in the register MSR_UNCORE_PERF_GLOBAL_CTRL.

Both the general-purpose counters (MSR_UNCORE_PerfCntr) and the fixed-function counter (MSR_UNCORE_FixedCntr0) are 48 bits wide. They support both counting and interrupt based sampling usages. The event logic unit can filter event counts to specific regions of code or transaction types incoming to the home node logic.

18.3.1.2.3 Uncore Address/Opcode Match MSR

The Event Select field [7:0] of MSR_UNCORE_PERFEVTSELx is used to select different uncore event logic unit. When the event "ADDR_OPCODE_MATCH" is selected in the Event Select field, software can filter uncore performance events according to transaction address and certain transaction responses. The address filter and transaction response filtering requires the use of MSR_UNCORE_ADDR_OPCODE_MATCH register. The layout is shown in Figure 18-24.

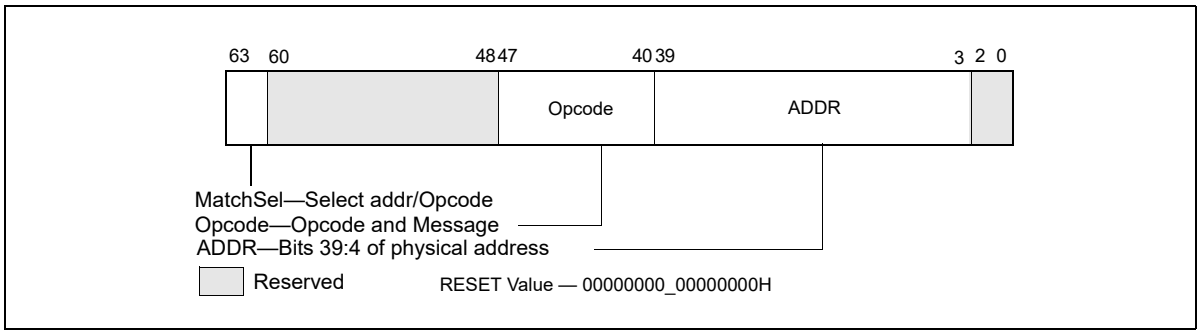


Figure 18-24. Layout of MSR_UNCORE_ADDR_OPCODE_MATCH MSR

- Addr (bits 39:3): The physical address to match if "MatchSel" field is set to select address match. The uncore performance counter will increment if the lowest 40-bit incoming physical address (excluding bits 2:0) for a transaction request matches bits 39:3.
- Opcode (bits 47:40) : Bits 47:40 allow software to filter uncore transactions based on QPI link message class/packed header opcode. These bits are consists two sub-fields:
  - — Bits 43:40 specify the QPI packet header opcode.
  - — Bits 47:44 specify the QPI message classes.

  Table 18-7 lists the encodings supported in the opcode field.

Table 18-7. Opcode Field Encoding for MSR_UNCORE_ADDR_OPCODE_MATCH

Opcode [43:40]	QPI Message Class		
	Home Request [47:44] = 0000B	Snoop Response [47:44] = 0001B	Data Response [47:44] = 1110B
		1	
DMND_IFETCH	2	2	
WB	3	3	
PF_DATA_RD	4	4	
PF_RFO	5	5	
PF_IFETCH	6	6	
OTHER	7	7	
NON_DRAM	15	15	

- MatchSel (bits 63:61): Software specifies the match criteria according to the following encoding:
  - 000B: Disable addr_opcode match hardware.
  - 100B: Count if only the address field matches.
  - 010B: Count if only the opcode field matches.
  - 110B: Count if either opcode field matches or the address field matches.
  - 001B: Count only if both opcode and address field match.
  - Other encoding are reserved.

18.3.1.3 Intel® Xeon® Processor 7500 Series Performance Monitoring Facility

The performance monitoring facility in the processor core of Intel® Xeon® processor 7500 series are the same as those supported in Intel Xeon processor 5500 series. The uncore subsystem in Intel Xeon processor 7500 series are significantly different The uncore performance monitoring facility consist of many distributed units associated with individual logic control units (referred to as boxes) within the uncore subsystem. A high level block diagram of the various box units of the uncore is shown in Figure 18-25.

Uncore PMUs are programmed via MSR interfaces. Each of the distributed uncore PMU units have several general-purpose counters. Each counter requires an associated event select MSR, and may require additional MSRs to configure sub-event conditions. The uncore PMU MSRs associated with each box can be categorized based on its functional scope: per-counter, per-box, or global across the uncore. The number counters available in each box type are different. Each box generally provides a set of MSRs to enable/disable, check status/overflow of multiple counters within each box.

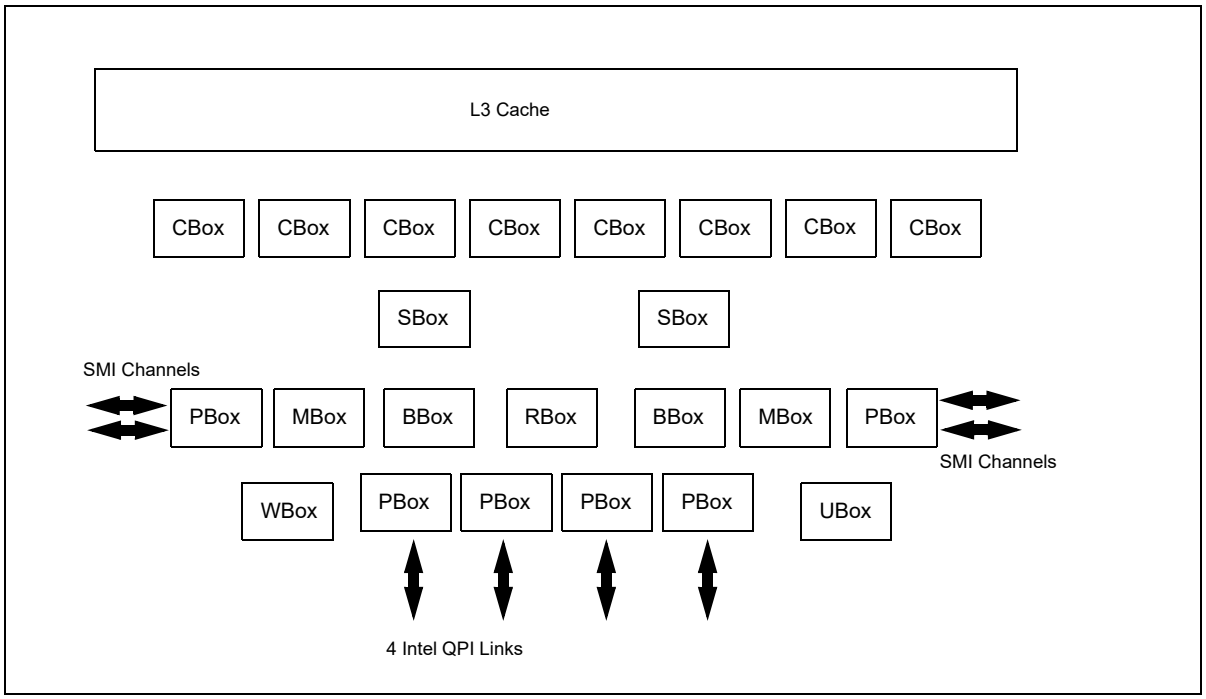


Figure 18-25. Distributed Units of the Uncore of Intel® Xeon® Processor 7500 Series

Table 18-8 summarizes the number MSRs for uncore PMU for each box.

Table 18-8. Uncore PMU MSR Summary

Box	# of Boxes	Counters per Box	Counter Width	General Purpose	Global Enable	Sub-control MSRs
C-Box	8	6	48	Yes	per-box	None
S-Box	2	4	48	Yes	per-box	Match/Mask
B-Box	2	4	48	Yes	per-box	Match/Mask
M-Box	2	6	48	Yes	per-box	Yes
R-Box	1	16 (2 port, 8 per port)	48	Yes	per-box	Yes
W-Box	1	4	48	Yes	per-box	None
		1	48	No	per-box	None
U-Box	1	1	48	Yes	uncore	None

The W-Box provides 4 general-purpose counters, each requiring an event select configuration MSR, similar to the general-purpose counters in other boxes. There is also a fixed-function counter that increments clockticks in the uncore clock domain.

For C,S,B,M,R, and W boxes, each box provides an MSR to enable/disable counting, configuring PMI of multiple counters within the same box, this is somewhat similar the "global control" programming interface, IA32_PERF_GLOBAL_CTRL, offered in the core PMU. Similarly status information and counter overflow control for multiple counters within the same box are also provided in C,S,B,M,R, and W boxes.

In the U-Box, MSR_U_PMON_GLOBAL_CTL provides overall uncore PMU enable/disable and PMI configuration control. The scope of status information in the U-box is at per-box granularity, in contrast to the per-box status information MSR (in the C,S,B,M,R, and W boxes) providing status information of individual counter overflow. The difference in scope also apply to the overflow control MSR in the U-Box versus those in the other Boxes.

The individual MSRs that provide uncore PMU interfaces are listed in Chapter 2, "Model-Specific Registers (MSRs)" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 4*, Table 2-16 under the general naming style of MSR_%box#%_PMON_%scope_function%, where %box#% designates the type of box and zero-based index if there are more the one box of the same type, %scope_function% follows the examples below:

- Multi-counter enabling MSRs: MSR_U_PMON_GLOBAL_CTL, MSR_S0_PMON_BOX_CTL, MSR_C7_PMON_BOX_CTL, etc.
- Multi-counter status MSRs: MSR_U_PMON_GLOBAL_STATUS, MSR_S0_PMON_BOX_STATUS, MSR_C7_PMON_BOX_STATUS, etc.
- Multi-counter overflow control MSRs: MSR_U_PMON_GLOBAL_OVF_CTL, MSR_S0_PMON_BOX_OVF_CTL, MSR_C7_PMON_BOX_OVF_CTL, etc.
- Performance counters MSRs: the scope is implicitly per counter, e.g. MSR_U_PMON_CTR, MSR_S0_PMON_CTR0, MSR_C7_PMON_CTR5, etc.
- Event select MSRs: the scope is implicitly per counter, e.g. MSR_U_PMON_EVNT_SEL, MSR_S0_PMON_EVNT_SEL0, MSR_C7_PMON_EVNT_SEL5, etc
- Sub-control MSRs: the scope is implicitly per-box granularity, e.g. MSR_M0_PMON_TIMESTAMP, MSR_R0_PMON_IPERF0_P1, MSR_S1_PMON_MATCH.

Details of uncore PMU MSR bit field definitions can be found in a separate document "Intel Xeon Processor 7500 Series Uncore Performance Monitoring Guide".

18.3.2 Performance Monitoring for Processors Based on Intel® Microarchitecture Code Name Westmere

All of the performance monitoring programming interfaces (architectural and non-architectural core PMU facilities, and uncore PMU) described in Section 18.6.3 also apply to processors based on Intel® microarchitecture code name Westmere.

Table 18-5 describes a non-architectural performance monitoring event (event code 0B7H) and associated MSR_OFFCORE_RSP_0 (address 1A6H) in the core PMU. This event and a second functionally equivalent offcore response event using event code 0BBH and MSR_OFFCORE_RSP_1 (address 1A7H) are supported in processors based on Intel microarchitecture code name Westmere. The event code and event mask definitions of Non-architectural performance monitoring events are listed in Table 19-28.

The load latency facility is the same as described in Section 18.3.1.1.2, but added enhancement to provide more information in the data source encoding field of each load latency record. The additional information relates to STLB_MISS and LOCK, see Table 18-13.

18.3.3 Intel® Xeon® Processor E7 Family Performance Monitoring Facility

The performance monitoring facility in the processor core of the Intel® Xeon® processor E7 family is the same as those supported in the Intel Xeon processor 5600 series[1]. The uncore subsystem in the Intel Xeon processor E7 family is similar to those of the Intel Xeon processor 7500 series. The high level construction of the uncore subsystem is similar to that shown in Figure 18-25, with the additional capability that up to 10 C-Box units are supported.

1. Exceptions are indicated for event code 0FH in Table 19-21; and valid bits of data source encoding field of each load latency record is limited to bits 5:4 of Table 18-13.

Table 18-9 summarizes the number MSRs for uncore PMU for each box.

Table 18-9. Uncore PMU MSR Summary for Intel® Xeon® Processor E7 Family

Box	# of Boxes	Counters per Box	Counter Width	General Purpose	Global Enable	Sub-control MSRs
C-Box	10	6	48	Yes	per-box	None
S-Box	2	4	48	Yes	per-box	Match/Mask
B-Box	2	4	48	Yes	per-box	Match/Mask
M-Box	2	6	48	Yes	per-box	Yes
R-Box	1	16 (2 port, 8 per port)	48	Yes	per-box	Yes
W-Box	1	4	48	Yes	per-box	None
		1	48	No	per-box	None
U-Box	1	1	48	Yes	uncore	None

Details of the uncore performance monitoring facility of Intel Xeon Processor E7 family is available in the "Intel® Xeon® Processor E7 Uncore Performance Monitoring Programming Reference Manual".

18.3.4 Performance Monitoring for Processors Based on Intel® Microarchitecture Code Name Sandy Bridge

Intel® Core™ i7-2xxx, Intel® Core™ i5-2xxx, Intel® Core™ i3-2xxx processor series, and Intel® Xeon® processor E3-1200 family are based on Intel microarchitecture code name Sandy Bridge; this section describes the performance monitoring facilities provided in the processor core. The core PMU supports architectural performance monitoring capability with version ID 3 (see Section 18.2.3) and a host of non-architectural monitoring capabilities.

Architectural performance monitoring version 3 capabilities are described in Section 18.2.3.

The core PMU's capability is similar to those described in Section 18.3.1.1 and Section 18.6.3, with some differences and enhancements relative to Intel microarchitecture code name Westmere summarized in Table 18-10.

Table 18-10. Core PMU Comparison

Box	Intel® microarchitecture code name Sandy Bridge	Intel® microarchitecture code name Westmere	Comment
# of Fixed counters per thread	3	3	Use CPUID to enumerate # of counters.
# of general-purpose counters per core	8	8	
Counter width (R,W)	R:48, W: 32/48	R:48, W:32	See Section 18.2.2.
# of programmable counters per thread	4 or (8 if a core not shared by two threads)	4	Use CPUID to enumerate # of counters.
PMI Overhead Mitigation	▪ Freeze_Perfmon_on_PMI with legacy semantics. ▪ Freeze_on_LBR with legacy semantics for branch profiling. ▪ Freeze_while_SMM.	▪ Freeze_Perfmon_on_PMI with legacy semantics. ▪ Freeze_on_LBR with legacy semantics for branch profiling. ▪ Freeze_while_SMM.	See Section 17.4.7.
Processor Event Based Sampling (PEBS) Events	See Table 18-12.	See Table 18-68.	IA32_PMC4-IA32_PMC7 do not support PEBS.

Table 18-10. Core PMU Comparison (Contd.)

Box	Intel® microarchitecture code name Sandy Bridge	Intel® microarchitecture code name Westmere	Comment
PEBS-Load Latency	See Section 18.3.4.4.2; ▪ Data source encoding ▪ STLB miss encoding ▪ Lock transaction encoding	Data source encoding	
PEBS-Precise Store	Section 18.3.4.4.3	No	
PEBS-PDIR	Yes (using precise INST_RETIRED.ALL).	No	
Off-core Response Event	MSR 1A6H and 1A7H, extended request and response types.	MSR 1A6H and 1A7H, limited response types.	Nehalem supports 1A6H only.

18.3.4.1 Global Counter Control Facilities In Intel® Microarchitecture Code Name Sandy Bridge

The number of general-purpose performance counters visible to a logical processor can vary across Processors based on Intel microarchitecture code name Sandy Bridge. Software must use CPUID to determine the number performance counters/event select registers (See Section 18.2.1.1).

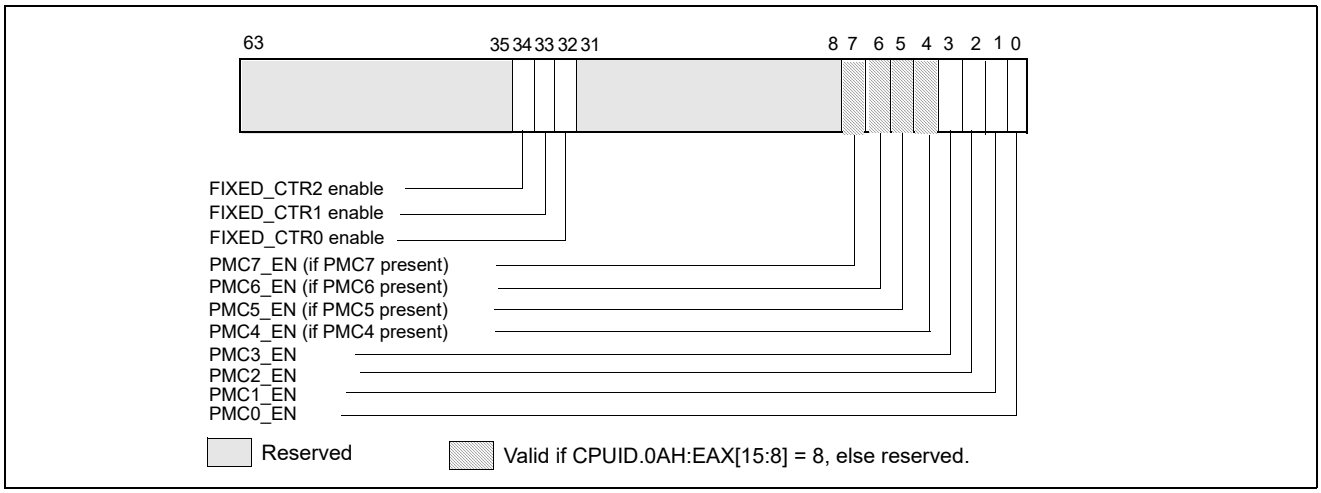


Figure 18-26. IA32_PERF_GLOBAL_CTRL MSR in Intel® Microarchitecture Code Name Sandy Bridge

Figure 18-42 depicts the layout of IA32_PERF_GLOBAL_CTRL MSR. The enable bits (PMC4_EN, PMC5_EN, PMC6_EN, PMC7_EN) corresponding to IA32_PMC4-IA32_PMC7 are valid only if CPUID.0AH:EAX[15:8] reports a value of '8'. If CPUID.0AH:EAX[15:8] = 4, attempts to set the invalid bits will cause #GP.

Each enable bit in IA32_PERF_GLOBAL_CTRL is AND'ed with the enable bits for all privilege levels in the respective IA32_PERFEVTSELx or IA32_PERF_FIXED_CTR_CTRL MSRs to start/stop the counting of respective counters. Counting is enabled if the AND'ed results is true; counting is disabled when the result is false.
IA32_PERF_GLOBAL_STATUS MSR provides single-bit status used by software to query the overflow condition of each performance counter. IA32_PERF_GLOBAL_STATUS[bit 62] indicates overflow conditions of the DS area data buffer (see Figure 18-27). A value of 1 in each bit of the PMCx_OVF field indicates an overflow condition has occurred in the associated counter.

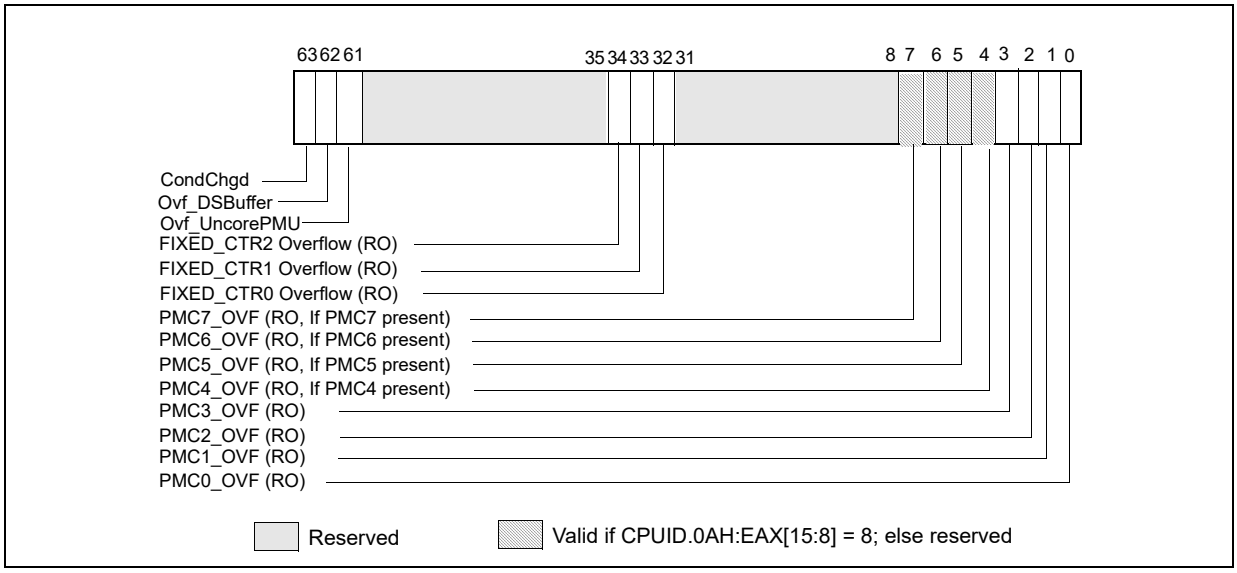


Figure 18-27. IA32_PERF_GLOBAL_STATUS MSR in Intel® Microarchitecture Code Name Sandy Bridge

When a performance counter is configured for PEBS, an overflow condition in the counter will arm PEBS. On the subsequent event following overflow, the processor will generate a PEBS event. On a PEBS event, the processor will perform bounds checks based on the parameters defined in the DS Save Area (see Section 17.4.9). Upon successful bounds checks, the processor will store the data record in the defined buffer area, clear the counter overflow status, and reload the counter. If the bounds checks fail, the PEBS will be skipped entirely. In the event that the PEBS buffer fills up, the processor will set the OvfBuffer bit in MSR_PERF_GLOBAL_STATUS.

IA32_PERF_GLOBAL_OVF_CTL MSR allows software to clear overflow the indicators for general-purpose or fixed-function counters via a single WRMSR (see Figure 18-28). Clear overflow indications when:

- Setting up new values in the event select and/or UMASK field for counting or interrupt based sampling.
- Reloading counter values to continue sampling.
- Disabling event counting or interrupt based sampling.

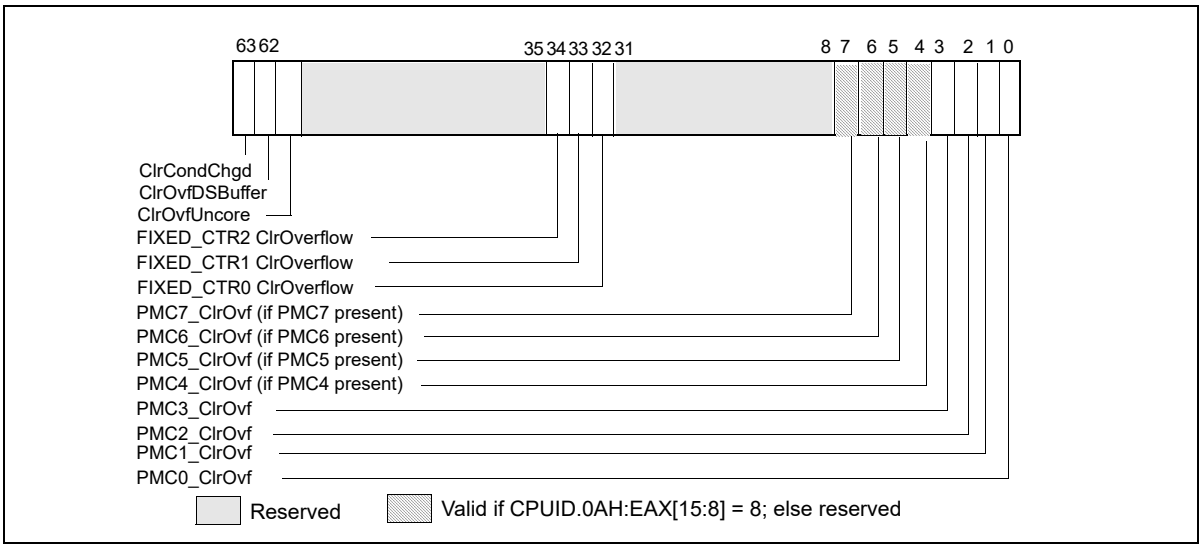


Figure 18-28. IA32_PERF_GLOBAL_OVF_CTRL MSR in Intel microarchitecture code name Sandy Bridge

18.3.4.2 Counter Coalescence

In processors based on Intel microarchitecture code name Sandy Bridge, each processor core implements eight general-purpose counters. CPUID.0AH:EAX[15:8] will report either 4 or 8 depending specific processor's product features.

If a processor core is shared by two logical processors, each logical processors can access 4 counters (IA32_PMC0-IA32_PMC3). This is the same as in the prior generation for processors based on Intel microarchitecture code name Nehalem.

If a processor core is not shared by two logical processors, all eight general-purpose counters are visible, and CPUID.0AH:EAX[15:8] reports 8. IA32_PMC4-IA32_PMC7 occupy MSR addresses 0C5H through 0C8H. Each counter is accompanied by an event select MSR (IA32_PERFEVTSEL4-IA32_PERFEVTSEL7).

If CPUID.0AH:EAX[15:8] report 4, access to IA32_PMC4-IA32_PMC7, IA32_PMC4-IA32_PMC7 will cause #GP. Writing 1's to bit position 7:4 of IA32_PERF_GLOBAL_CTRL, IA32_PERF_GLOBAL_STATUS, or IA32_PERF_GLOBAL_OVF_CTL will also cause #GP.

18.3.4.3 Full Width Writes to Performance Counters

Processors based on Intel microarchitecture code name Sandy Bridge support full-width writes to the general-purpose counters, IA32_PMCx. Support of full-width writes are enumerated by IA32_PERF_CAPABILITIES.FW_WRITES[13] (see Section 18.2.4).

The default behavior of IA32_PMCx is unchanged, i.e. WRMSR to IA32_PMCx results in a sign-extended 32-bit value of the input EAX written into IA32_PMCx. Full-width writes must issue WRMSR to a dedicated alias MSR address for each IA32_PMCx.

Software must check the presence of full-width write capability and the presence of the alias address IA32_A_PMCx by testing IA32_PERF_CAPABILITIES[13].

18.3.4.4 PEBS Support in Intel® Microarchitecture Code Name Sandy Bridge

Processors based on Intel microarchitecture code name Sandy Bridge support PEBS, similar to those offered in prior generation, with several enhanced features. The key components and differences of PEBS facility relative to Intel microarchitecture code name Westmere is summarized in Table 18-11.

Table 18-11. PEBS Facility Comparison

Box	Intel® microarchitecture code name Sandy Bridge	Intel® microarchitecture code name Westmere	Comment
Valid IA32_PMCx	PMC0-PMC3	PMC0-PMC3	No PEBS on PMC4-PMC7.
PEBS Buffer Programming	Section 18.3.1.1.1	Section 18.3.1.1.1	Unchanged
IA32_PEBS_ENABLE Layout	Figure 18-29	Figure 18-15	
PEBS record layout	Physical Layout same as Table 18-3.	Table 18-3	Enhanced fields at offsets 98H, A0H, A8H.
PEBS Events	See Table 18-12.	See Table 18-68.	IA32_PMC4-IA32_PMC7 do not support PEBS.
PEBS-Load Latency	See Table 18-13.	Table 18-4	
PEBS-Precise Store	Yes; see Section 18.3.4.4.3.	No	IA32_PMC3 only
PEBS-PDIR	Yes	No	IA32_PMC1 only
PEBS skid from EventingIP	1 (or 2 if micro+macro fusion)	1	
SAMPLING Restriction	Small SAV(CountDown) value incur higher overhead than prior generation.		

Only IA32_PMC0 through IA32_PMC3 support PEBS.

NOTE

PEBS events are only valid when the following fields of IA32_PERFEVTSELx are all zero: AnyThread, Edge, Invert, CMask.

In a PMU with PDIR capability, PEBS behavior is unpredictable if IA32_PERFEVTSELx or IA32_PMCx is changed for a PEBS-enabled counter while an event is being counted. To avoid this, changes to the programming or value of a PEBS-enabled counter should be performed when the counter is disabled.

In IA32_PEBS_ENABLE MSR, bit 63 is defined as PS_ENABLE: When set, this enables IA32_PMC3 to capture precise store information. Only IA32_PMC3 supports the precise store facility. In typical usage of PEBS, the bit fields in IA32_PEBS_ENABLE are written to when the agent software starts PEBS operation; the enabled bit fields should be modified only when re-programming another PEBS event or cleared when the agent uses the performance counters for non-PEBS operations.

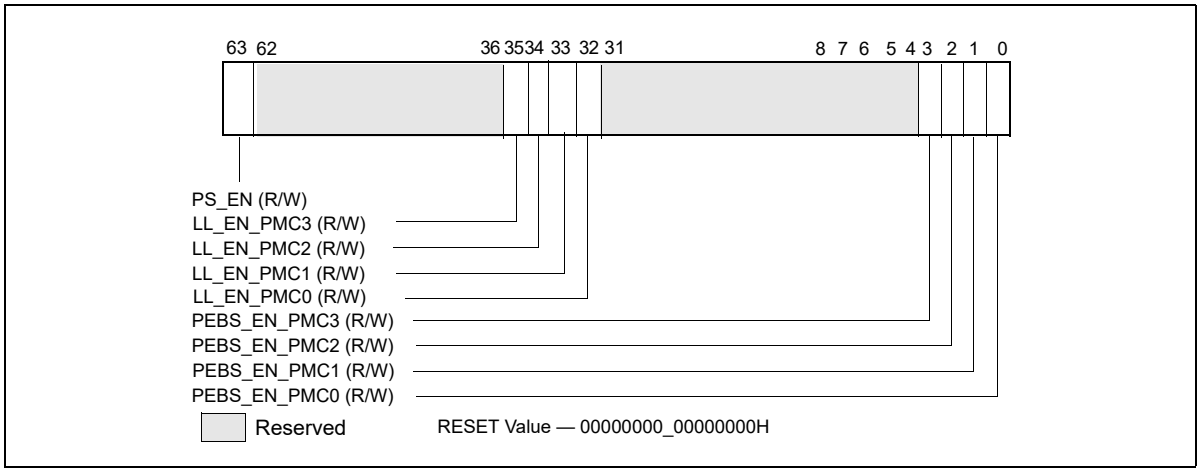


Figure 18-29. Layout of IA32_PEBS_ENABLE MSR

18.3.4.4.1 PEBS Record Format

The layout of PEBS records physically identical to those shown in Table 18-3, but the fields at offset 98H, A0H and A8H have been enhanced to support additional PEBS capabilities.

- Load/Store Data Linear Address (Offset 98H): This field will contain the linear address of the source of the load, or linear address of the destination of the store.
- Data Source /Store Status (Offset A0H):When load latency is enabled, this field will contain three piece of information (including an encoded value indicating the source which satisfied the load operation). The source field encodings are detailed in Table 18-4. When precise store is enabled, this field will contain information indicating the status of the store, as detailed in Table 19.
- Latency Value/0 (Offset A8H): When load latency is enabled, this field contains the latency in cycles to service the load. This field is not meaningful when precise store is enabled and will be written to zero in that case. Upon writing the PEBS record, microcode clears the overflow status bits in the IA32_PERF_GLOBAL_STATUS corresponding to those counters that both overflowed and were enabled in the IA32_PEBS_ENABLE register. The status bits of other counters remain unaffected.

The number PEBS events has expanded. The list of PEBS events supported in Intel microarchitecture code name Sandy Bridge is shown in Table 18-12.

Table 18-12. PEBS Performance Events for Intel® Microarchitecture Code Name Sandy Bridge

Event Name	Event Select	Sub-event	UMask
INST_RETIRED	C0H	PREC_DIST	01H[1]
UOPS_RETIRED	C2H	All	01H
		Retire_Slots	02H
BR_INST_RETIRED	C4H	Conditional	01H
		Near_Call	02H
		All_branches	04H
		Near_Return	08H
		Near_Taken	20H
BR_MISP_RETIRED	C5H	Conditional	01H
		Near_Call	02H
		All_branches	04H
		Not_Taken	10H
		Taken	20H
MEM_UOPS_RETIRED	D0H	STLB_MISS_LOADS	11H
		STLB_MISS_STORE	12H
		LOCK_LOADS	21H
		SPLIT_LOADS	41H
		SPLIT_STORES	42H
		ALL_LOADS	81H
		ALL_STORES	82H
MEM_LOAD_UOPS_RETIRED	D1H	L1_Hit	01H
		L2_Hit	02H
		L3_Hit	04H
		Hit_LFB	40H
MEM_LOAD_UOPS_LLC_HIT_RETIRED	D2H	XSNP_Miss	01H
		XSNP_Hit	02H
		XSNP_Hitm	04H
		XSNP_None	08H

NOTES:
1. Only available on IA32_PMC1.

18.3.4.4.2 Load Latency Performance Monitoring Facility

The load latency facility in Intel microarchitecture code name Sandy Bridge is similar to that in prior microarchitecture. It provides software a means to characterize the average load latency to different levels of cache/memory hierarchy. This facility requires processor supporting enhanced PEBS record format in the PEBS buffer, see Table 18-3 and Section 18.3.4.4.1. This field measures the load latency from load's first dispatch of till final data writeback from the memory subsystem. The latency is reported for retired demand load operations and in core cycles (it accounts for re-dispatches).

To use this feature software must assure:

- One of the IA32_PERFEVTSELx MSR is programmed to specify the event unit MEM_TRANS_RETIRED, and the LATENCY_ABOVE_THRESHOLD event mask must be specified (IA32_PerfEvtSelX[15:0] = 1CDH). The corresponding counter IA32_PMCx will accumulate event counts for architecturally visible loads which exceed the programmed latency threshold specified separately in a MSR. Stores are ignored when this event is

programmed. The CMASK or INV fields of the IA32_PerfEvtSelX register used for counting load latency must be 0. Writing other values will result in undefined behavior.

- The MSR_PEBS_LD_LAT_THRESHOLD MSR is programmed with the desired latency threshold in core clock cycles. Loads with latencies greater than this value are eligible for counting and latency data reporting. The minimum value that may be programmed in this register is 3 (the minimum detectable load latency is 4 core clock cycles).
- The PEBS enable bit in the IA32_PEBS_ENABLE register is set for the corresponding IA32_PMCx counter register. This means that both the PEBS_EN_CTRX and LL_EN_CTRX bits must be set for the counter(s) of interest. For example, to enable load latency on counter IA32_PMC0, the IA32_PEBS_ENABLE register must be programmed with the 64-bit value 00000001.00000001H.
- When Load latency event is enabled, no other PEBS event can be configured with other counters.

When the load-latency facility is enabled, load operations are randomly selected by hardware and tagged to carry information related to data source locality and latency. Latency and data source information of tagged loads are updated internally. The MEM_TRANS_RETIRED event for load latency counts only tagged retired loads. If a load is cancelled it will not be counted and the internal state of the load latency facility will not be updated. In this case the hardware will tag the next available load.

When a PEBS assist occurs, the last update of latency and data source information are captured by the assist and written as part of the PEBS record. The PEBS sample after value (SAV), specified in PEBS CounterX Reset, operates orthogonally to the tagging mechanism. Loads are randomly tagged to collect latency data. The SAV controls the number of tagged loads with latency information that will be written into the PEBS record field by the PEBS assists. The load latency data written to the PEBS record will be for the last tagged load operation which retired just before the PEBS assist was invoked.

The physical layout of the PEBS records is the same as shown in Table 18-3. The specificity of Data Source entry at offset A0H has been enhanced to report three pieces of information.

Table 18-13. Layout of Data Source Field of Load Latency Record

Field	Position	Description
Source	3:0	See Table 18-4
STLB_MISS	4	0: The load did not miss the STLB (hit the DTLB or STLB). 1: The load missed the STLB.
Lock	5	0: The load was not part of a locked transaction. 1: The load was part of a locked transaction.
Reserved	63:6	Reserved

The layout of MSR_PEBS_LD_LAT_THRESHOLD is the same as shown in Figure 18-17.

18.3.4.4.3 Precise Store Facility

Processors based on Intel microarchitecture code name Sandy Bridge offer a precise store capability that complements the load latency facility. It provides a means to profile store memory references in the system.

Precise stores leverage the PEBS facility and provide additional information about sampled stores. Having precise memory reference events with linear address information for both loads and stores can help programmers improve data structure layout, eliminate remote node references, and identify cache-line conflicts in NUMA systems.

Only IA32_PMC3 can be used to capture precise store information. After enabling this facility, counter overflows will initiate the generation of PEBS records as previously described in PEBS. Upon counter overflow hardware captures the linear address and other status information of the next store that retires. This information is then written to the PEBS record.

To enable the precise store facility, software must complete the following steps. Please note that the precise store facility relies on the PEBS facility, so the PEBS configuration requirements must be completed before attempting to capture precise store information.

- Complete the PEBS configuration steps.

- Program the MEM_TRANS_RETIRED.PRECISE_STORE event in IA32_PERFEVTSEL3. Only counter 3 (IA32_PMC3) supports collection of precise store information.
- Set IA32_PEBS_ENABLE[3] and IA32_PEBS_ENABLE[63]. This enables IA32_PMC3 as a PEBS counter and enables the precise store facility, respectively.

The precise store information written into a PEBS record affects entries at offset 98H, A0H and A8H of Table 18-3. The specificity of Data Source entry at offset A0H has been enhanced to report three piece of information.

Table 18-14. Layout of Precise Store Information In PEBS Record

Field	Offset	Description
Store Data Linear Address	98H	The linear address of the destination of the store.
Store Status	A0H	**L1D Hit** (Bit 0): The store hit the data cache closest to the core (lowest latency cache) if this bit is set, otherwise the store missed the data cache. **STLB Miss** (bit 4): The store missed the STLB if set, otherwise the store hit the STLB **Locked Access** (bit 5): The store was part of a locked access if set, otherwise the store was not part of a locked access.
Reserved	A8H	Reserved

18.3.4.4.4 Precise Distribution of Instructions Retired (PDIR)

Upon triggering a PEBS assist, there will be a finite delay between the time the counter overflows and when the microcode starts to carry out its data collection obligations. INST_RETIRED is a very common event that is used to sample where performance bottleneck happened and to help identify its location in instruction address space. Even if the delay is constant in core clock space, it invariably manifest as variable "skids" in instruction address space. This creates a challenge for programmers to profile a workload and pinpoint the location of bottlenecks.

The core PMU in processors based on Intel microarchitecture code name Sandy Bridge include a facility referred to as precise distribution of Instruction Retired (PDIR).

The PDIR facility mitigates the "skid" problem by providing an early indication of when the INST_RETIRED counter is about to overflow, allowing the machine to more precisely trap on the instruction that actually caused the counter overflow thus eliminating skid.

PDIR applies only to the INST_RETIRED.ALL precise event, and must use IA32_PMC1 with PerfEvtSel1 property configured and bit 1 in the IA32_PEBS_ENABLE set to 1. INST_RETIRED.ALL is a non-architectural performance event, it is not supported in prior generation microarchitectures. Additionally, on processors with CPUID DisplayFamily_DisplayModel signatures of 06_2A and 06_2D, the tool that programs PDIR should quiesce the rest of the programmable counters in the core when PDIR is active.

18.3.4.5 Off-core Response Performance Monitoring

The core PMU in processors based on Intel microarchitecture code name Sandy Bridge provides off-core response facility similar to prior generation. Off-core response can be programmed only with a specific pair of event select and counter MSR, and with specific event codes and predefine mask bit value in a dedicated MSR to specify attributes of the off-core transaction. Two event codes are dedicated for off-core response event programming. Each event code for off-core response monitoring requires programming an associated configuration MSR, MSR_OFFCORE_RSP_x. Table 18-15 lists the event code, mask value and additional off-core configuration MSR that must be programmed to count off-core response events using IA32_PMCx.

Table 18-15. Off-Core Response Event Encoding

Counter	Event code	UMask	Required Off-core Response MSR
PMC0-3	B7H	01H	MSR_OFFCORE_RSP_0 (address 1A6H)
PMC0-3	BBH	01H	MSR_OFFCORE_RSP_1 (address 1A7H)

The layout of MSR_OFFCORE_RSP_0 and MSR_OFFCORE_RSP_1 are shown in Figure 18-30 and Figure 18-31. Bits 15:0 specifies the request type of a transaction request to the uncore. Bits 30:16 specifies supplier information, bits 37:31 specifies snoop response information.

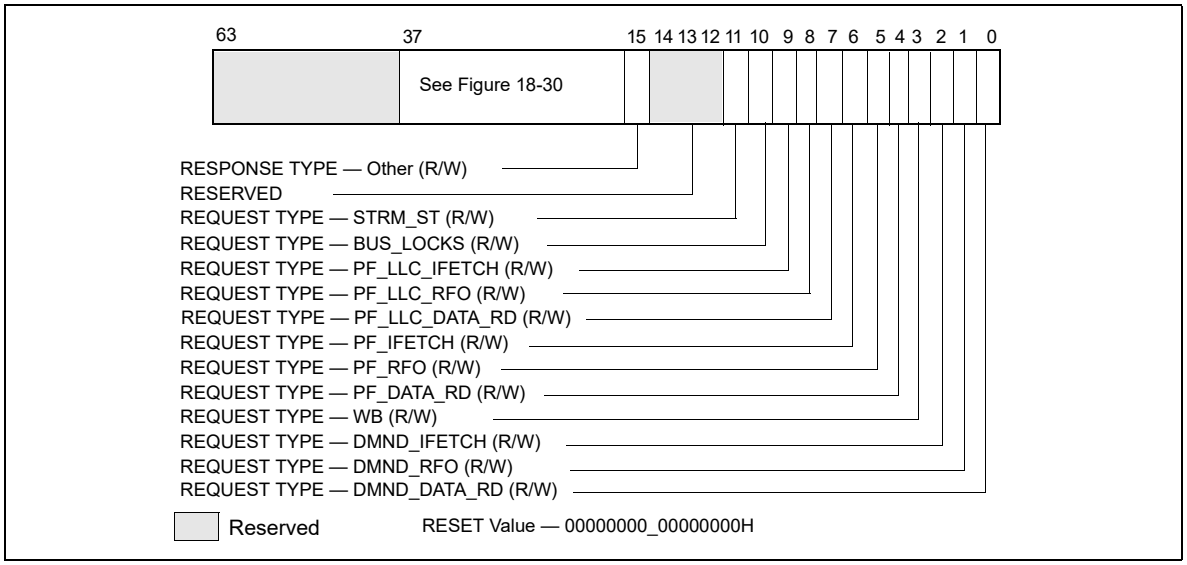


Figure 18-30. Request_Type Fields for MSR_OFFCORE_RSP_x

Table 18-16. MSR_OFFCORE_RSP_x Request_Type Field Definition

Bit Name	Offset	Description
DMND_DATA_RD	0	(R/W). Counts the number of demand data reads of full and partial cachelines as well as demand data page table entry cacheline reads. Does not count L2 data read prefetches or instruction fetches.
DMND_RFO	1	(R/W). Counts the number of demand and DCU prefetch reads for ownership (RFO) requests generated by a write to data cacheline. Does not count L2 RFO prefetches.
DMND_IFETCH	2	(R/W). Counts the number of demand and DCU prefetch instruction cacheline reads. Does not count L2 code read prefetches.
WB	3	(R/W). Counts the number of writeback (modified to exclusive) transactions.
PF_DATA_RD	4	(R/W). Counts the number of data cacheline reads generated by L2 prefetchers.
PF_RFO	5	(R/W). Counts the number of RFO requests generated by L2 prefetchers.
PF_IFETCH	6	(R/W). Counts the number of code reads generated by L2 prefetchers.
PF_LLC_DATA_RD	7	(R/W). L2 prefetcher to L3 for loads.
PF_LLC_RFO	8	(R/W). RFO requests generated by L2 prefetcher
PF_LLC_IFETCH	9	(R/W). L2 prefetcher to L3 for instruction fetches.
BUS_LOCKS	10	(R/W). Bus lock and split lock requests
STRM_ST	11	(R/W). Streaming store requests
OTHER	15	(R/W). Any other request that crosses IDI, including I/O.

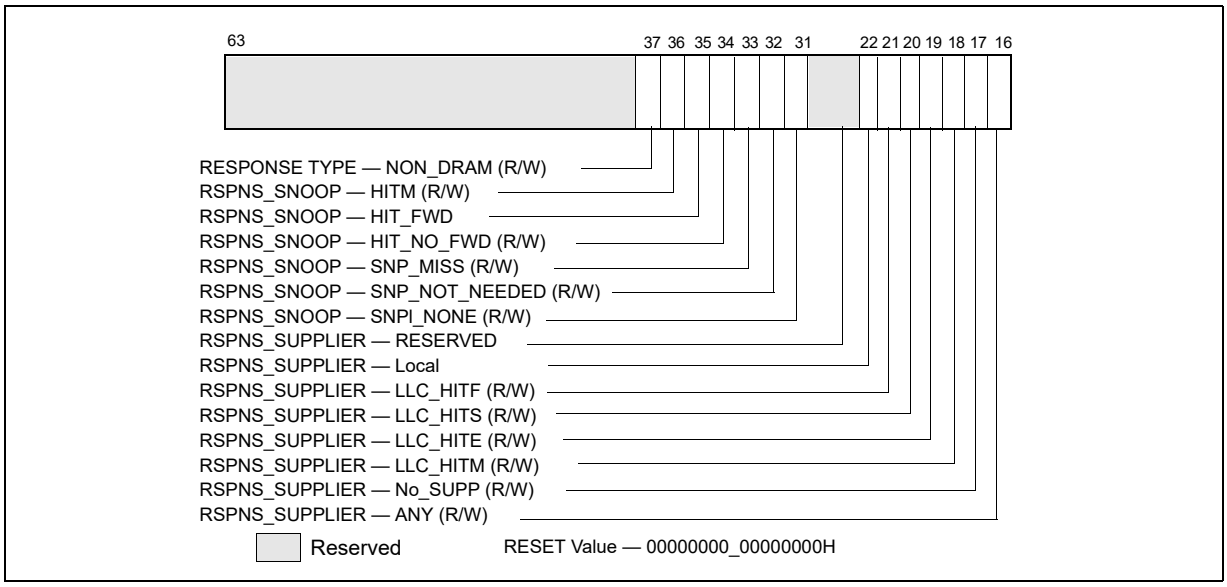


Figure 18-31. Response_Supplier and Snoop Info Fields for MSR_OFFCORE_RSP_x

To properly program this extra register, software must set at least one request type bit and a valid response type pattern. Otherwise, the event count reported will be zero. It is permissible and useful to set multiple request and response type bits in order to obtain various classes of off-core response events. Although MSR_OFFCORE_RSP_x allow an agent software to program numerous combinations that meet the above guideline, not all combinations produce meaningful data.

Table 18-17. MSR_OFFCORE_RSP_x Response Supplier Info Field Definition

Subtype	Bit Name	Offset	Description
Common	Any	16	(R/W). Catch all value for any response types.
Supplier Info	NO_SUPP	17	(R/W). No Supplier Information available
	LLC_HITM	18	(R/W). M-state initial lookup stat in L3.
	LLC_HITE	19	(R/W). E-state
	LLC_HITS	20	(R/W). S-state
	LLC_HITF	21	(R/W). F-state
	LOCAL	22	(R/W). Local DRAM Controller
	Reserved	30:23	Reserved

To specify a complete offcore response filter, software must properly program bits in the request and response type fields. A valid request type must have at least one bit set in the non-reserved bits of 15:0. A valid response type must be a non-zero value of the following expression:

ANY | [(‘OR’ of Supplier Info Bits) & (‘OR’ of Snoop Info Bits)]

If “ANY“ bit is set, the supplier and snoop info bits are ignored.

Table 18-18. MSR_OFFCORE_RSP_x Snoop Info Field Definition

Subtype	Bit Name	Offset	Description
Snoop Info	SNP_NONE	31	(R/W). No details on snoop-related information
	SNP_NOT_NEEDED	32	(R/W). No snoop was needed to satisfy the request.
	SNP_MISS	33	(R/W). A snoop was needed and it missed all snooped caches: -For LLC Hit, ReslHitI was returned by all cores -For LLC Miss, RspI was returned by all sockets and data was returned from DRAM.
	SNP_NO_FWD	34	(R/W). A snoop was needed and it hits in at least one snooped cache. Hit denotes a cache-line was valid before snoop effect. This includes: -Snoop Hit w/ Invalidation (LLC Hit, RFO) -Snoop Hit, Left Shared (LLC Hit/Miss, IFetch/Data_RD) -Snoop Hit w/ Invalidation and No Forward (LLC Miss, RFO Hit S) In the LLC Miss case, data is returned from DRAM.
	SNP_FWD	35	(R/W). A snoop was needed and data was forwarded from a remote socket. This includes: -Snoop Forward Clean, Left Shared (LLC Hit/Miss, IFetch/Data_RD/RFT).
	HITM	36	(R/W). A snoop was needed and it HitM-ed in local or remote cache. HitM denotes a cache-line was in modified state before effect as a results of snoop. This includes: -Snoop HitM w/ WB (LLC miss, IFetch/Data_RD) -Snoop Forward Modified w/ Invalidation (LLC Hit/Miss, RFO) -Snoop MtoS (LLC Hit, IFetch/Data_RD).
	NON_DRAM	37	(R/W). Target was non-DRAM system address. This includes MMIO transactions.

18.3.4.6 Uncore Performance Monitoring Facilities In Intel® Core™ i7-2xxx, Intel® Core™ i5-2xxx, Intel® Core™ i3-2xxx Processor Series

The uncore sub-system in Intel® Core™ i7-2xxx, Intel® Core™ i5-2xxx, Intel® Core™ i3-2xxx processor series provides a unified L3 that can support up to four processor cores. The L3 cache consists multiple slices, each slice interface with a processor via a coherence engine, referred to as a C-Box. Each C-Box provides dedicated facility of MSRs to select uncore performance monitoring events and each C-Box event select MSR is paired with a counter register, similar in style as those described in Section 18.3.1.2.2. The ARB unit in the uncore also provides its local performance counters and event select MSRs. The layout of the event select MSRs in the C-Boxes and the ARB unit are shown in Figure 18-32.

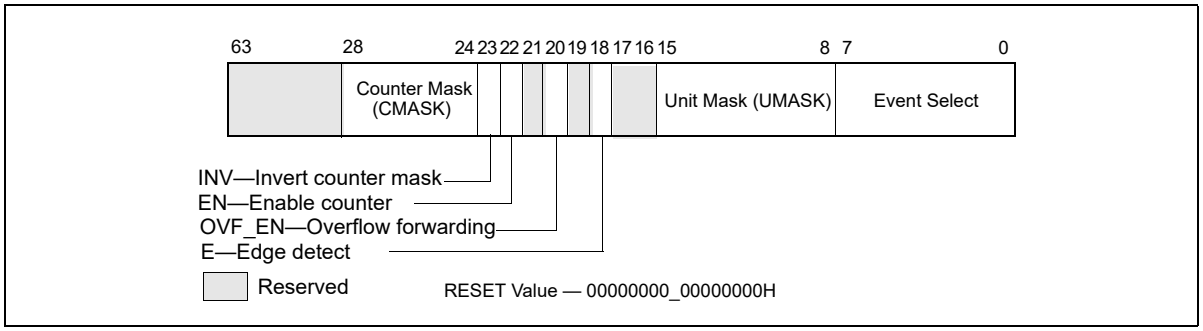


Figure 18-32. Layout of Uncore PERFEVTSEL MSR for a C-Box Unit or the ARB Unit

The bit fields of the uncore event select MSRs for a C-box unit or the ARB unit are summarized below:

- Event_Select (bits 7:0) and UMASK (bits 15:8): Specifies the microarchitectural condition to count in a local uncore PMU counter, see Table 19-18.
- E (bit 18): Enables edge detection filtering, if 1.
- OVF_EN (bit 20): Enables the overflow indicator from the uncore counter forwarded to MSR_UNC_PERF_GLOBAL_CTRL, if 1.
- EN (bit 22): Enables the local counter associated with this event select MSR.
- INV (bit 23): Event count increments with non-negative value if 0, with negated value if 1.
- CMASK (bits 28:24): Specifies a positive threshold value to filter raw event count input.

At the uncore domain level, there is a master set of control MSRs that centrally manages all the performance monitoring facility of uncore units. Figure 18-33 shows the layout of the uncore domain global control.

When an uncore counter overflows, a PMI can be routed to a processor core. Bits 3:0 of MSR_UNC_PERF_GLOBAL_CTRL can be used to select which processor core to handle the uncore PMI. Software must then write to bit 13 of IA32_DEBUGCTL (at address 1D9H) to enable this capability.

- PMI_SEL_Core#: Enables the forwarding of an uncore PMI request to a processor core, if 1. If bit 30 (WakePMI) is '1', a wake request is sent to the respective processor core prior to sending the PMI.
- EN: Enables the fixed uncore counter, the ARB counters, and the CBO counters in the uncore PMU, if 1. This bit is cleared if bit 31 (FREEZE) is set and any enabled uncore counters overflow.
- WakePMI: Controls sending a wake request to any halted processor core before issuing the uncore PMI request. If a processor core was halted and not sent a wake request, the uncore PMI will not be serviced by the processor core.
- FREEZE: Provides the capability to freeze all uncore counters when an overflow condition occurs in a unit counter. When this bit is set, and a counter overflow occurs, the uncore PMU logic will clear the global enable bit (bit 29).

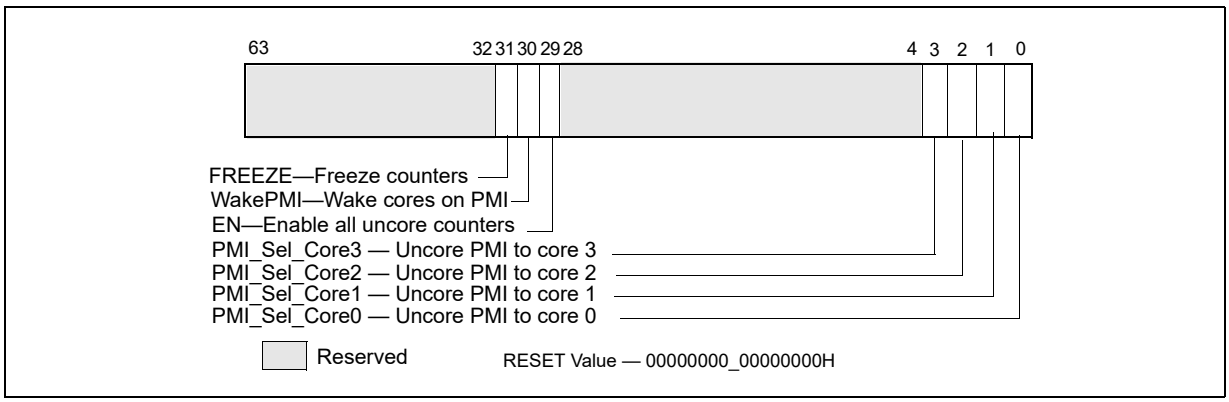


Figure 18-33. Layout of MSR_UNC_PERF_GLOBAL_CTRL MSR for Uncore

Additionally, there is also a fixed counter, counting uncore clockticks, for the uncore domain. Table 18-19 summarizes the number MSRs for uncore PMU for each box.

Table 18-19. Uncore PMU MSR Summary

Box	# of Boxes	Counters per Box	Counter Width	General Purpose	Global Enable	Comment
C-Box	SKU specific	2	44	Yes	Per-box	Up to 4, seeTable 2-20 MSR_UNC_CBO_CONFIG
ARB	1	2	44	Yes	Uncore	
Fixed Counter	N.A.	N.A.	48	No	Uncore	

18.3.4.6.1 Uncore Performance Monitoring Events

There are certain restrictions on the uncore performance counters in each C-Box. Specifically,

- Occupancy events are supported only with counter 0 but not counter 1.

Other uncore C-Box events can be programmed with either counter 0 or 1.

The C-Box uncore performance events described in Table 19-18 can collect performance characteristics of transactions initiated by processor core. In that respect, they are similar to various sub-events in the OFFCORE_RESPONSE family of performance events in the core PMU. Information such as data supplier locality (LLC HIT/MISS) and snoop responses can be collected via OFFCORE_RESPONSE and qualified on a per-thread basis.

On the other hand, uncore performance event logic can not associate its counts with the same level of per-thread qualification attributes as the core PMU events can. Therefore, whenever similar event programming capabilities are available from both core PMU and uncore PMU, the recommendation is that utilizing the core PMU events may be less affected by artifacts, complex interactions and other factors.

18.3.4.7 Intel® Xeon® Processor E5 Family Performance Monitoring Facility

The Intel® Xeon® Processor E5 Family (and Intel® Core™ i7-3930K Processor) are based on Intel microarchitecture code name Sandy Bridge-E. While the processor cores share the same microarchitecture as those of the Intel® Xeon® Processor E3 Family and 2nd generation Intel Core i7-2xxx, Intel Core i5-2xxx, Intel Core i3-2xxx processor series, the uncore subsystems are different. An overview of the uncore performance monitoring facilities of the Intel Xeon processor E5 family (and Intel Core i7-3930K processor) is described in Section 18.3.4.8.

Thus, the performance monitoring facilities in the processor core generally are the same as those described in Section 18.6.3 through Section 18.3.4.5. However, the MSR_OFFCORE_RSP_0/MSR_OFFCORE_RSP_1 Response Supplier Info field shown in Table 18-17 applies to Intel Core Processors with CPUID signature of DisplayFamily_DisplayModel encoding of 06_2AH; Intel Xeon processor with CPUID signature of DisplayFamily_DisplayModel encoding of 06_2DH supports an additional field for remote DRAM controller shown in Table 18-20. Additionally, the are some small differences in the non-architectural performance monitoring events (see Table 19-16).

Table 18-20. MSR_OFFCORE_RSP_x Supplier Info Field Definitions

Subtype	Bit Name	Offset	Description
Common	Any	16	(R/W). Catch all value for any response types.
Supplier Info	NO_SUPP	17	(R/W). No Supplier Information available
	LLC_HITM	18	(R/W). M-state initial lookup stat in L3.
	LLC_HITE	19	(R/W). E-state
	LLC_HITS	20	(R/W). S-state
	LLC_HITF	21	(R/W). F-state
	LOCAL	22	(R/W). Local DRAM Controller
	Remote	30:23	(R/W): Remote DRAM Controller (either all 0s or all 1s)

18.3.4.8 Intel® Xeon® Processor E5 Family Uncore Performance Monitoring Facility

The uncore subsystem in the Intel Xeon processor E5-2600 product family has some similarities with those of the Intel Xeon processor E7 family. Within the uncore subsystem, localized performance counter sets are provided at logic control unit scope. For example, each Cbox caching agent has a set of local performance counters, and the power controller unit (PCU) has its own local performance counters. Up to 8 C-Box units are supported in the uncore sub-system.

Table 18-21 summarizes the uncore PMU facilities providing MSR interfaces.

Table 18-21. Uncore PMU MSR Summary for Intel® Xeon® Processor E5 Family

Box	# of Boxes	Counters per Box	Counter Width	General Purpose	Global Enable	Sub-control MSRs
C-Box	8	4	44	Yes	per-box	None
PCU	1	4	48	Yes	per-box	Match/Mask
U-Box	1	2	44	Yes	uncore	None

Details of the uncore performance monitoring facility of Intel Xeon Processor E5 family is available in "Intel® Xeon® Processor E5 Uncore Performance Monitoring Programming Reference Manual". The MSR-based uncore PMU interfaces are listed in Table 2-23.

18.3.5 3rd Generation Intel® Core™ Processor Performance Monitoring Facility

The 3rd generation Intel® Core™ processor family and Intel® Xeon® processor E3-1200v2 product family are based on the Ivy Bridge microarchitecture. The performance monitoring facilities in the processor core generally are the same as those described in Section 18.6.3 through Section 18.3.4.5. The non-architectural performance monitoring events supported by the processor core are listed in Table 19-16.

18.3.5.1 Intel® Xeon® Processor E5 v2 and E7 v2 Family Uncore Performance Monitoring Facility

The uncore subsystem in the Intel Xeon processor E5 v2 and Intel Xeon Processor E7 v2 product families are based on the Ivy Bridge-E microarchitecture. There are some similarities with those of the Intel Xeon processor E5 family based on the Sandy Bridge microarchitecture. Within the uncore subsystem, localized performance counter sets are provided at logic control unit scope.

Details of the uncore performance monitoring facility of Intel Xeon Processor E5 v2 and Intel Xeon Processor E7 v2 families are available in "Intel® Xeon® Processor E5 v2 and E7 v2 Uncore Performance Monitoring Programming Reference Manual". The MSR-based uncore PMU interfaces are listed in Table 2-27.

18.3.6 4th Generation Intel® Core™ Processor Performance Monitoring Facility

The 4th generation Intel® Core™ processor and Intel® Xeon® processor E3-1200 v3 product family are based on the Haswell microarchitecture. The core PMU supports architectural performance monitoring capability with version ID 3 (see Section 18.2.3) and a host of non-architectural monitoring capabilities.

Architectural performance monitoring version 3 capabilities are described in Section 18.2.3.

The core PMU's capability is similar to those described in Section 18.6.3 through Section 18.3.4.5, with some differences and enhancements summarized in Table 18-22. Additionally, the core PMU provides some enhancement to support performance monitoring when the target workload contains instruction streams using Intel® Transactional Synchronization Extensions (TSX), see Section 18.3.6.5. For details of Intel TSX, see Chapter 16, "Programming with Intel® Transactional Synchronization Extensions" of *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*.

Table 18-22. Core PMU Comparison

Box	Intel® microarchitecture code name Haswell	Intel® microarchitecture code name Sandy Bridge	Comment
# of Fixed counters per thread	3	3	
# of general-purpose counters per core	8	8	
Counter width (R,W)	R:48, W: 32/48	R:48, W: 32/48	See Section 18.2.2.
# of programmable counters per thread	4 or (8 if a core not shared by two threads)	4 or (8 if a core not shared by two threads)	Use CPUID to enumerate # of counters.
PMI Overhead Mitigation	▪ Freeze_Perfmon_on_PMI with legacy semantics. ▪ Freeze_on_LBR with legacy semantics for branch profiling. ▪ Freeze_while_SMM.	▪ Freeze_Perfmon_on_PMI with legacy semantics. ▪ Freeze_on_LBR with legacy semantics for branch profiling. ▪ Freeze_while_SMM.	See Section 17.4.7.
Processor Event Based Sampling (PEBS) Events	See Table 18-12 and Section 18.3.6.5.1.	See Table 18-12.	IA32_PMC4-IA32_PMC7 do not support PEBS.
PEBS-Load Latency	See Section 18.3.4.4.2.	See Section 18.3.4.4.2.	
PEBS-Precise Store	No, replaced by Data Address profiling.	Section 18.3.4.4.3	
PEBS-PDIR	Yes (using precise INST_RETIRED.ALL)	Yes (using precise INST_RETIRED.ALL)	
PEBS-EventingIP	Yes	No	
Data Address Profiling	Yes	No	
LBR Profiling	Yes	Yes	
Call Stack Profiling	Yes, see Section 17.11.	No	Use LBR facility.
Off-core Response Event	MSR 1A6H and 1A7H; extended request and response types.	MSR 1A6H and 1A7H; extended request and response types.	
Intel TSX support for Perfmon	See Section 18.3.6.5.	No	

18.3.6.1 Processor Event Based Sampling (PEBS) Facility

The PEBS facility in the 4th Generation Intel Core processor is similar to those in processors based on Intel microarchitecture code name Sandy Bridge, with several enhanced features. The key components and differences of PEBS facility relative to Intel microarchitecture code name Sandy Bridge is summarized in Table 18-23.

Table 18-23. PEBS Facility Comparison

Box	Intel® microarchitecture code name Haswell	Intel® microarchitecture code name Sandy Bridge	Comment
Valid IA32_PMCx	PMC0-PMC3	PMC0-PMC3	No PEBS on PMC4-PMC7
PEBS Buffer Programming	Section 18.3.1.1.1	Section 18.3.1.1.1	Unchanged
IA32_PEBS_ENABLE Layout	Figure 18-15	Figure 18-29	
PEBS record layout	Table 18-24; enhanced fields at offsets 98H, A0H, A8H, B0H.	Table 18-3; enhanced fields at offsets 98H, A0H, A8H.	
Precise Events	See Table 18-12.	See Table 18-12.	IA32_PMC4-IA32_PMC7 do not support PEBS.
PEBS-Load Latency	See Table 18-13.	Table 18-13	
PEBS-Precise Store	No, replaced by data address profiling.	Yes; see Section 18.3.4.4.3.	
PEBS-PDIR	Yes	Yes	IA32_PMC1 only.
PEBS skid from EventingIP	1 (or 2 if micro+macro fusion)	1	
SAMPLING Restriction	Small SAV(CountDown) value incur higher overhead than prior generation.		

Only IA32_PMC0 through IA32_PMC3 support PEBS.

NOTE

PEBS events are only valid when the following fields of IA32_PERFEVTSELx are all zero: AnyThread, Edge, Invert, CMask.

In a PMU with PDIR capability, PEBS behavior is unpredictable if IA32_PERFEVTSELx or IA32_PMCx is changed for a PEBS-enabled counter while an event is being counted. To avoid this, changes to the programming or value of a PEBS-enabled counter should be performed when the counter is disabled.

18.3.6.2 PEBS Data Format

The PEBS record format for the 4th Generation Intel Core processor is shown in Table 18-24. The PEBS record format, along with debug/store area storage format, does not change regardless of whether IA-32e mode is active or not. CPUID.01H:ECX.DTES64[bit 2] reports whether the processor's DS storage format support is mode-independent. When set, it uses 64-bit DS storage format.

Table 18-24. PEBS Record Format for 4th Generation Intel Core Processor Family

Byte Offset	Field	Byte Offset	Field
00H	R/EFLAGS	60H	R10
08H	R/EIP	68H	R11
10H	R/EAX	70H	R12
18H	R/EBX	78H	R13
20H	R/ECX	80H	R14
28H	R/EDX	88H	R15
30H	R/ESI	90H	IA32_PERF_GLOBAL_STATUS
38H	R/EDI	98H	Data Linear Address
40H	R/EBP	A0H	Data Source Encoding
48H	R/ESP	A8H	Latency value (core cycles)
50H	R8	B0H	EventingIP
58H	R9	B8H	TX Abort Information (Section 18.3.6.5.1)

The layout of PEBS records are almost identical to those shown in Table 18-3. Offset B0H is a new field that records the eventing IP address of the retired instruction that triggered the PEBS assist.

The PEBS records at offsets 98H, A0H, and ABH record data gathered from three of the PEBS capabilities in prior processor generations: load latency facility (Section 18.3.4.4.2), PDIR (Section 18.3.4.4.4), and the equivalent capability of precise store in prior generation (see Section 18.3.6.3).

In the core PMU of the 4th generation Intel Core processor, load latency facility and PDIR capabilities are unchanged. However, precise store is replaced by an enhanced capability, data address profiling, that is not restricted to store address. Data address profiling also records information in PEBS records at offsets 98H, A0H, and ABH.

18.3.6.3 PEBS Data Address Profiling

The Data Linear Address facility is also abbreviated as DataLA. The facility is a replacement or extension of the precise store facility in previous processor generations. The DataLA facility complements the load latency facility by providing a means to profile load and store memory references in the system, leverages the PEBS facility, and provides additional information about sampled loads and stores. Having precise memory reference events with linear address information for both loads and stores provides information to improve data structure layout, eliminate remote node references, and identify cache-line conflicts in NUMA systems.

The DataLA facility in the 4th generation processor supports the following events configured to use PEBS:

Table 18-25. Precise Events That Supports Data Linear Address Profiling

Event Name	Event Name
MEM_UOPS_RETIRED.STLB_MISS_LOADS	MEM_UOPS_RETIRED.STLB_MISS_STORES
MEM_UOPS_RETIRED.LOCK_LOADS	MEM_UOPS_RETIRED.SPLIT_STORES
MEM_UOPS_RETIRED.SPLIT_LOADS	MEM_UOPS_RETIRED.ALL_STORES
MEM_UOPS_RETIRED.ALL_LOADS	MEM_LOAD_UOPS_LLC_MISS_RETIRED.LOCAL_DRAM
MEM_LOAD_UOPS_RETIRED.L1_HIT	MEM_LOAD_UOPS_RETIRED.L2_HIT
MEM_LOAD_UOPS_RETIRED.L3_HIT	MEM_LOAD_UOPS_RETIRED.L1_MISS
MEM_LOAD_UOPS_RETIRED.L2_MISS	MEM_LOAD_UOPS_RETIRED.L3_MISS
MEM_LOAD_UOPS_RETIRED.HIT_LFB	MEM_LOAD_UOPS_L3_HIT_RETIRED.XSNP_MISS

Table 18-25. Precise Events That Supports Data Linear Address Profiling (Contd.)

Event Name	Event Name
MEM_LOAD_UOPS_L3_HIT_RETIRED.XSNP_HIT	MEM_LOAD_UOPS_L3_HIT_RETIRED.XSNP_HITM
UOPS_RETIRED.ALL (if load or store is tagged)	MEM_LOAD_UOPS_LLC_HIT_RETIRED.XSNP_NONE

DataLA can use any one of the IA32_PMC0-IA32_PMC3 counters. Counter overflows will initiate the generation of PEBS records. Upon counter overflow, hardware captures the linear address and possible other status information of the retiring memory uop. This information is then written to the PEBS record that is subsequently generated.

To enable the DataLA facility, software must complete the following steps. Please note that the DataLA facility relies on the PEBS facility, so the PEBS configuration requirements must be completed before attempting to capture DataLA information.

- Complete the PEBS configuration steps.
- Program the an event listed in Table 18-25 using any one of IA32_PERFEVTSEL0-IA32_PERFEVTSEL3.
- Set the corresponding IA32_PEBS_ENABLE.PEBS_EN_CTRx bit. This enables the corresponding IA32_PMCx as a PEBS counter and enables the DataLA facility.

When the DataLA facility is enabled, the relevant information written into a PEBS record affects entries at offsets 98H, A0H and A8H, as shown in Table 18-26.

Table 18-26. Layout of Data Linear Address Information In PEBS Record

Field	Offset	Description
Data Linear Address	98H	The linear address of the load or the destination of the store.
Store Status	A0H	▪ **DCU Hit** (Bit 0): The store hit the data cache closest to the core (L1 cache) if this bit is set, otherwise the store missed the data cache. This information is valid only for the following store events: UOPS_RETIRED.ALL (if store is tagged), MEM_UOPS_RETIRED.STLB_MISS_STORES, MEM_UOPS_RETIRED.SPLIT_STORES, MEM_UOPS_RETIRED.ALL_STORES ▪ Other bits are zero, The STLB_MISS, LOCK bit information can be obtained by programming the corresponding store event in Table 18-25.
Reserved	A8H	Always zero.

18.3.6.3.1 EventingIP Record

The PEBS record layout for processors based on Intel microarchitecture code name Haswell adds a new field at offset 0B0H. This is the eventingIP field that records the IP address of the retired instruction that triggered the PEBS assist. The EIP/RIP field at offset 08H records the IP address of the next instruction to be executed following the PEBS assist.

18.3.6.4 Off-core Response Performance Monitoring

The core PMU facility to collect off-core response events are similar to those described in Section 18.3.4.5. The event codes are listed in Table 18-15. Each event code for off-core response monitoring requires programming an associated configuration MSR, MSR_OFFCORE_RSP_x. Software must program MSR_OFFCORE_RSP_x according to:

- Transaction request type encoding (bits 15:0): see Table 18-27.
- Supplier information (bits 30:16): see Table 18-28.
- Snoop response information (bits 37:31): see Table 18-18.

Table 18-27. MSR_OFFCORE_RSP_x Request_Type Definition (Haswell microarchitecture)

Bit Name	Offset	Description
DMND_DATA_RD	0	(R/W). Counts the number of demand data reads of full and partial cachelines as well as demand data page table entry cacheline reads. Does not count L2 data read prefetches or instruction fetches.
DMND_RFO	1	(R/W). Counts the number of demand and DCU prefetch reads for ownership (RFO) requests generated by a write to data cacheline. Does not count L2 RFO prefetches.
DMND_IFETCH	2	(R/W). Counts the number of demand and DCU prefetch instruction cacheline reads. Does not count L2 code read prefetches.
COREWB	3	(R/W). Counts the number of modified cachelines written back.
PF_DATA_RD	4	(R/W). Counts the number of data cacheline reads generated by L2 prefetchers.
PF_RFO	5	(R/W). Counts the number of RFO requests generated by L2 prefetchers.
PF_IFETCH	6	(R/W). Counts the number of code reads generated by L2 prefetchers.
PF_L3_DATA_RD	7	(R/W). Counts the number of data cacheline reads generated by L3 prefetchers.
PF_L3_RFO	8	(R/W). Counts the number of RFO requests generated by L3 prefetchers.
PF_L3_CODE_RD	9	(R/W). Counts the number of code reads generated by L3 prefetchers.
SPLIT_LOCK_UC_ LOCK	10	(R/W). Counts the number of lock requests that split across two cachelines or are to UC memory.
STRM_ST	11	(R/W). Counts the number of streaming store requests electronically.
Reserved	12-14	Reserved
OTHER	15	(R/W). Any other request that crosses IDI, including I/O.

The supplier information field listed in Table 18-28. The fields vary across products (according to CPUID signatures) and is noted in the description.

Table 18-28. MSR_OFFCORE_RSP_x Supplier Info Field Definition (CPUID Signature 06_3CH, 06_46H)

Subtype	Bit Name	Offset	Description
Common	Any	16	(R/W). Catch all value for any response types.
Supplier Info	NO_SUPP	17	(R/W). No Supplier Information available
	L3_HITM	18	(R/W). M-state initial lookup stat in L3.
	L3_HITE	19	(R/W). E-state
	L3_HITS	20	(R/W). S-state
	Reserved	21	Reserved
	LOCAL	22	(R/W). Local DRAM Controller
	Reserved	30:23	Reserved

Table 18-29. MSR_OFFCORE_RSP_x Supplier Info Field Definition (CPUID Signature 06_45H)

Subtype	Bit Name	Offset	Description
Common	Any	16	(R/W). Catch all value for any response types.
Supplier Info	NO_SUPP	17	(R/W). No Supplier Information available
	L3_HITM	18	(R/W). M-state initial lookup stat in L3.
	L3_HITE	19	(R/W). E-state
	L3_HITS	20	(R/W). S-state
	Reserved	21	Reserved
	L4_HIT_LOCAL_L4	22	(R/W). L4 Cache
	L4_HIT_REMOTE_HOP0_L4	23	(R/W). L4 Cache
	L4_HIT_REMOTE_HOP1_L4	24	(R/W). L4 Cache
	L4_HIT_REMOTE_HOP2P_L4	25	(R/W). L4 Cache
	Reserved	30:26	Reserved

18.3.6.4.1 Off-core Response Performance Monitoring in Intel Xeon Processors E5 v3 Series

Table 18-28 lists the supplier information field that apply to Intel Xeon processor E5 v3 series (CPUID signature 06_3FH).

Table 18-30. MSR_OFFCORE_RSP_x Supplier Info Field Definition

Subtype	Bit Name	Offset	Description
Common	Any	16	(R/W). Catch all value for any response types.
Supplier Info	NO_SUPP	17	(R/W). No Supplier Information available
	L3_HITM	18	(R/W). M-state initial lookup stat in L3.
	L3_HITE	19	(R/W). E-state
	L3_HITS	20	(R/W). S-state
	L3_HITF	21	(R/W). F-state
	LOCAL	22	(R/W). Local DRAM Controller
	Reserved	26:23	Reserved
	L3_MISS_REMOTE_HOP0	27	(R/W). Hop 0 Remote supplier
	L3_MISS_REMOTE_HOP1	28	(R/W). Hop 1 Remote supplier
	L3_MISS_REMOTE_HOP2P	29	(R/W). Hop 2 or more Remote supplier
	Reserved	30	Reserved

18.3.6.5 Performance Monitoring and Intel® TSX

Chapter 16 of *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1* describes the details of Intel® Transactional Synchronization Extensions (Intel TSX). This section describes performance monitoring support for Intel TSX.

If a processor supports Intel TSX, the core PMU enhances it's IA32_PERFEVTSELx MSR with two additional bit fields for event filtering. Support for Intel TSX is indicated by either (a) CPUID.(EAX=7, ECX=0):RTM[bit 11]=1, or (b) if CPUID.07H.EBX.HLE [bit 4] = 1. The TSX-enhanced layout of IA32_PERFEVTSELx is shown in Figure 18-34. The two additional bit fields are:

- **IN_TX** (bit 32): When set, the counter will only include counts that occurred inside a transactional region, regardless of whether that region was aborted or committed. This bit may only be set if the processor supports HLE or RTM.
- **IN_TXCP** (bit 33): When set, the counter will not include counts that occurred inside of an aborted transactional region. This bit may only be set if the processor supports HLE or RTM. This bit may only be set for IA32_PERFEVTSEL2.

When the IA32_PERFEVTSELx MSR is programmed with both IN_TX=0 and IN_TXCP=0 on a processor that supports Intel TSX, the result in a counter may include detectable conditions associated with a transaction code region for its aborted execution (if any) and completed execution.

In the initial implementation, software may need to take pre-caution when using the IN_TXCP bit. see Table 2-28.

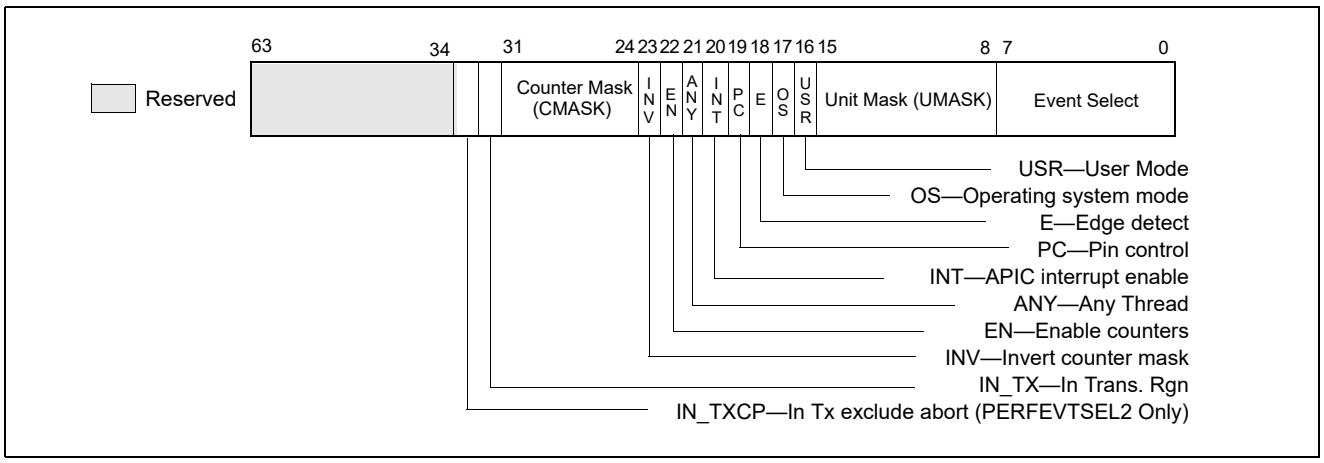


Figure 18-34. Layout of IA32_PERFEVTSELx MSRs Supporting Intel TSX

A common usage of setting IN_TXCP=1 is to capture the number of events that were discarded due to a transactional abort. With IA32_PMC2 configured to count in such a manner, then when a transactional region aborts, the value for that counter is restored to the value it had prior to the aborted transactional region. As a result, any updates performed to the counter during the aborted transactional region are discarded.

On the other hand, setting IN_TX=1 can be used to drill down on the performance characteristics of transactional code regions. When a PMCx is configured with the corresponding IA32_PERFEVTSELx.IN_TX=1, only eventing conditions that occur inside transactional code regions are propagated to the event logic and reflected in the counter result. Eventing conditions specified by IA32_PERFEVTSELx but occurring outside a transactional region are discarded. The following example illustrates using three counters to drill down cycles spent inside and outside of transactional regions:

- Program IA32_PERFEVTSEL2 to count Unhalted_Core_Cycles with (IN_TXCP=1, IN_TX=0), such that IA32_PMC2 will count cycles spent due to aborted TSX transactions;
- Program IA32_PERFEVTSEL0 to count Unhalted_Core_Cycles with (IN_TXCP=0, IN_TX=1), such that IA32_PMC0 will count cycles spent by the transactional code regions;
- Program IA32_PERFEVTSEL1 to count Unhalted_Core_Cycles with (IN_TXCP=0, IN_TX=0), such that IA32_PMC1 will count total cycles spent by the non-transactional code and transactional code regions.

Additionally, a number of performance events are solely focused on characterizing the execution of Intel TSX transactional code, they are listed in Table 19-10.

18.3.6.5.1 Intel TSX and PEBS Support

If a PEBS event would have occurred inside a transactional region, then the transactional region first aborts, and then the PEBS event is processed.

Two of the TSX performance monitoring events in Table 19-10 also support using PEBS facility to capture additional information. They are:

- HLE_RETIRED.ABORT ED (encoding C8H mask 04H),
- RTM_RETIRED.ABORTED (encoding C9H mask 04H).

A transactional abort (HLE_RETIRED.ABORTED,RTM_RETIRED.ABORTED) can also be programmed to cause PEBS events. In this scenario, a PEBS event is processed following the abort.

Pending a PEBS record inside of a transactional region will cause a transactional abort. If a PEBS record was pended at the time of the abort or on an overflow of the TSX PEBS events listed above, only the following PEBS entries will be valid (enumerated by PEBS entry offset B8H bits[33:32] to indicate an HLE abort or an RTM abort):

- Offset B0H: EventingIP,
- Offset B8H: TX Abort Information

These fields are set for all PEBS events.

- Offset 08H (RIP/EIP) corresponds to the instruction following the outermost XACQUIRE in HLE or the first instruction of the fallback handler of the outermost XBEGIN instruction in RTM. This is useful to identify the aborted transactional region.

In the case of HLE, an aborted transaction will restart execution deterministically at the start of the HLE region. In the case of RTM, an aborted transaction will transfer execution to the RTM fallback handler.

The layout of the TX Abort Information field is given in Table 18-31.

Table 18-31. TX Abort Information Field Definition

Bit Name	Offset	Description
Cycles_Last_TX	31:0	The number of cycles in the last TSX region, regardless of whether that region had aborted or committed.
HLE_Abort	32	If set, the abort information corresponds to an aborted HLE execution
RTM_Abort	33	If set, the abort information corresponds to an aborted RTM execution
Instruction_Abort	34	If set, the abort was associated with the instruction corresponding to the eventing IP (offset 0B0H) within the transactional region.
Non_Instruction_Abort	35	If set, the instruction corresponding to the eventing IP may not necessarily be related to the transactional abort.
Retry	36	If set, retrying the transactional execution may have succeeded.
Data_Conflict	37	If set, another logical processor conflicted with a memory address that was part of the transactional region that aborted.
Capacity Writes	38	If set, the transactional region aborted due to exceeding resources for transactional writes.
Capacity Reads	39	If set, the transactional region aborted due to exceeding resources for transactional reads.
Reserved	63:40	Reserved

18.3.6.6 Uncore Performance Monitoring Facilities in the 4th Generation Intel® Core™ Processors

The uncore sub-system in the 4th Generation Intel® Core™ processors provides its own performance monitoring facility. The uncore PMU facility provides dedicated MSRs to select uncore performance monitoring events in a similar manner as those described in Section 18.3.4.6.

The ARB unit and each C-Box provide local pairs of event select MSR and counter register. The layout of the event select MSRs in the C-Boxes are identical as shown in Figure 18-32.

At the uncore domain level, there is a master set of control MSRs that centrally manages all the performance monitoring facility of uncore units. Figure 18-33 shows the layout of the uncore domain global control.

Additionally, there is also a fixed counter, counting uncore clockticks, for the uncore domain. Table 18-19 summarizes the number MSRs for uncore PMU for each box.

Table 18-32. Uncore PMU MSR Summary

Box	# of Boxes	Counters per Box	Counter Width	General Purpose	Global Enable	Comment
C-Box	SKU specific	2	44	Yes	Per-box	Up to 4, seeTable 2-20 MSR_UNC_CBO_CONFIG
ARB	1	2	44	Yes	Uncore	
Fixed Counter	N.A.	N.A.	48	No	Uncore	

The uncore performance events for the C-Box and ARB units are listed in Table 19-11.

18.3.6.7 Intel® Xeon® Processor E5 v3 Family Uncore Performance Monitoring Facility

Details of the uncore performance monitoring facility of Intel Xeon Processor E5 v3 families are available in "Intel® Xeon® Processor E5 v3 Uncore Performance Monitoring Programming Reference Manual". The MSR-based uncore PMU interfaces are listed in Table 2-32.

18.3.7 5th Generation Intel® Core™ Processor and Intel® Core™ M Processor Performance Monitoring Facility

The 5th Generation Intel® Core™ processor and the Intel® Core™ M processor families are based on the Broadwell microarchitecture. The core PMU supports architectural performance monitoring capability with version ID 3 (see Section 18.2.3) and a host of non-architectural monitoring capabilities.

Architectural performance monitoring version 3 capabilities are described in Section 18.2.3.

The core PMU has the same capability as those described in Section 18.3.6. IA32_PERF_GLOBAL_STATUS provide a bit indicator (bit 55) for PMI handler to distinguish PMI due to output buffer overflow condition due to accumulating packet data from Intel Processor Trace.

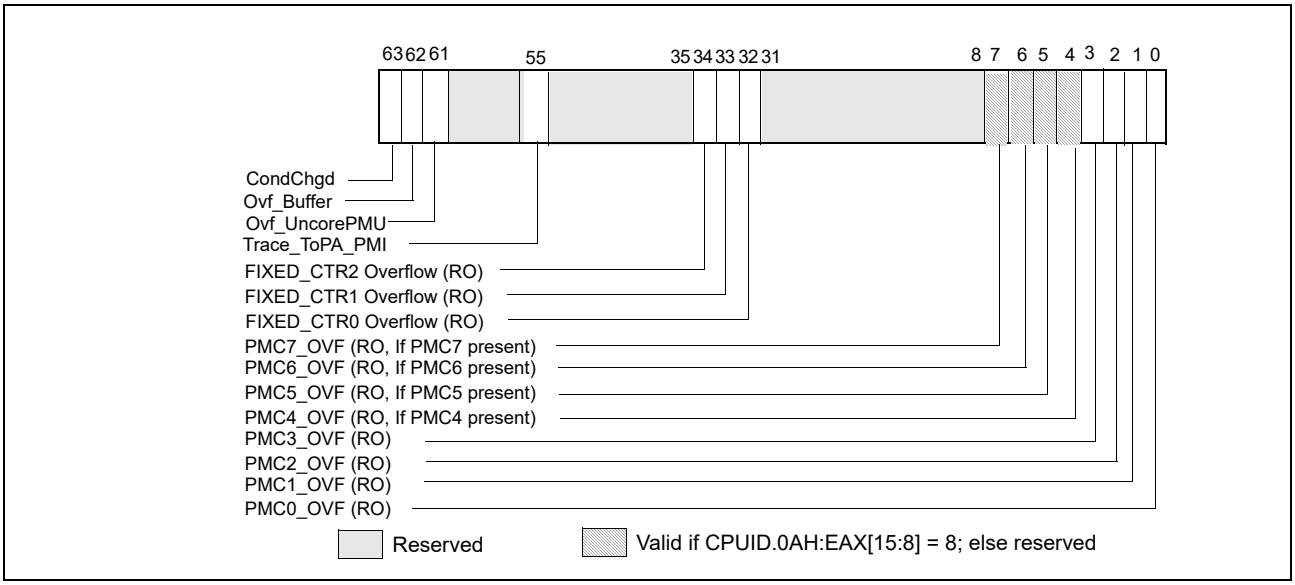


Figure 18-35. IA32_PERF_GLOBAL_STATUS MSR in Broadwell Microarchitecture

Details of Intel Processor Trace is described in Chapter 35, "Intel® Processor Trace".
IA32_PERF_GLOBAL_OVF_CTRL MSR provide a corresponding reset control bit.

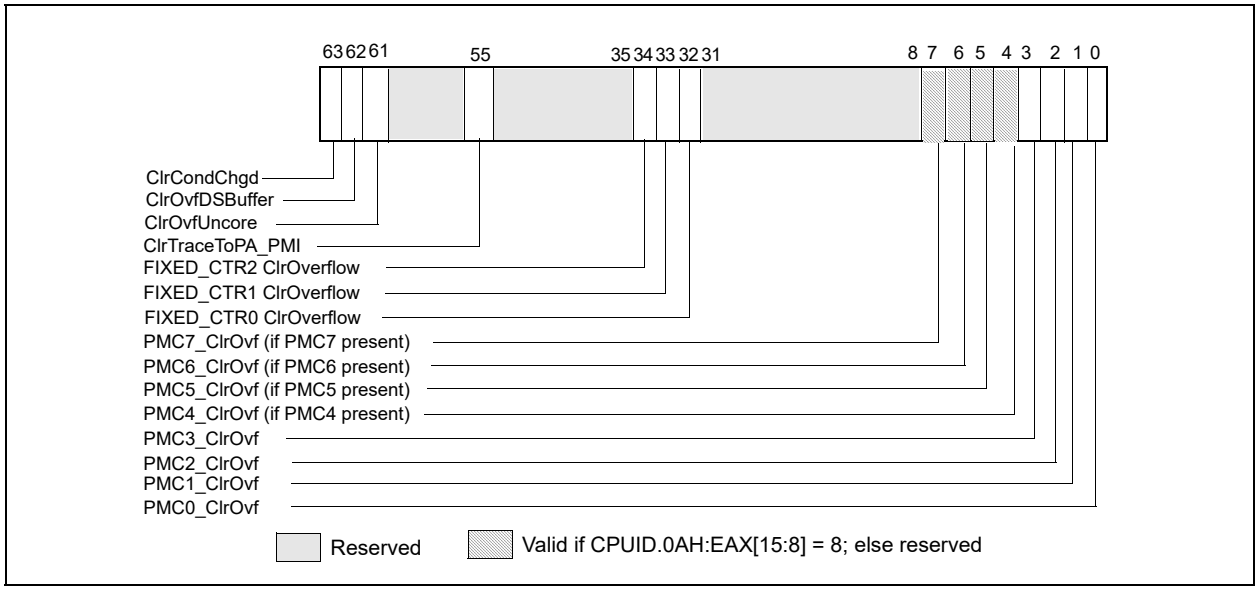


Figure 18-36. IA32_PERF_GLOBAL_OVF_CTRL MSR in Broadwell microarchitecture

The specifics of non-architectural performance events are listed in Chapter 19, "Performance Monitoring Events".

18.3.8 6th Generation Intel® Core™ Processor and 7th Generation Intel® Core™ Processor Performance Monitoring Facility

The 6th generation Intel® Core™ processor is based on the Skylake microarchitecture. The 7th generation Intel® Core™ processor is based on the Kaby Lake microarchitecture. The core PMU supports architectural performance monitoring capability with version ID 4 (see Section 18.2.4) and a host of non-architectural monitoring capabilities.

Architectural performance monitoring version 4 capabilities are described in Section 18.2.4.

The core PMU's capability is similar to those described in Section 18.6.3 through Section 18.3.4.5, with some differences and enhancements summarized in Table 18-22. Additionally, the core PMU provides some enhancement to support performance monitoring when the target workload contains instruction streams using Intel® Transactional Synchronization Extensions (TSX), see Section 18.3.6.5. For details of Intel TSX, see Chapter 16, "Programming with Intel® Transactional Synchronization Extensions" of *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*.

Performance monitoring result may be affected by side-band activity on processors that support Intel SGX, details are described in Chapter 42, "Enclave Code Debug and Profiling".

Table 18-33. Core PMU Comparison

Box	Intel® microarchitecture code name Skylake and Kaby Lake	Intel® microarchitecture code name Haswell and Broadwell	Comment
# of Fixed counters per thread	3	3	
# of general-purpose counters per core	8	8	
Counter width (R,W)	R:48, W: 32/48	R:48, W: 32/48	See Section 18.2.2.
# of programmable counters per thread	4 or (8 if a core not shared by two threads)	4 or (8 if a core not shared by two threads)	CPUID enumerates # of counters.
Architectural Perfmon version	4	3	See Section 18.2.4
PMI Overhead Mitigation	▪ Freeze_Perfmon_on_PMI with streamlined semantics. ▪ Freeze_on_LBR with streamlined semantics. ▪ Freeze_while_SMM.	▪ Freeze_Perfmon_on_PMI with legacy semantics. ▪ Freeze_on_LBR with legacy semantics for branch profiling. ▪ Freeze_while_SMM.	See Section 17.4.7. Legacy semantics not supported with version 4 or higher.
Counter and Buffer Overflow Status Management	▪ Query via IA32_PERF_GLOBAL_STATUS ▪ Reset via IA32_PERF_GLOBAL_STATUS_RESET ▪ Set via IA32_PERF_GLOBAL_STATUS_SET	▪ Query via IA32_PERF_GLOBAL_STATUS ▪ Reset via IA32_PERF_GLOBAL_OVF_CTRL	See Section 18.2.4.
IA32_PERF_GLOBAL_STATUS Indicators of Overflow/Overhead/Interference	▪ Individual counter overflow ▪ PEBS buffer overflow ▪ ToPA buffer overflow ▪ CTR_Frz, LBR_Frz, ASCI	▪ Individual counter overflow ▪ PEBS buffer overflow ▪ ToPA buffer overflow (applicable to Broadwell microarchitecture)	See Section 18.2.4.
Enable control in IA32_PERF_GLOBAL_STATUS	▪ CTR_Frz ▪ LBR_Frz	NA	See Section 18.2.4.1.
Perfmon Counter In-Use Indicator	Query IA32_PERF_GLOBAL_INUSE	NA	See Section 18.2.4.3.
Precise Events	See Table 18-36.	See Table 18-12.	IA32_PMC4-PMC7 do not support PEBS.
PEBS for front end events	See Section 18.3.8.1.4.	No	
LBR Record Format Encoding	000101b	000100b	Section 17.4.8.1
LBR Size	32 entries	16 entries	
LBR Entry	From_IP/To_IP/LBR_Info triplet	From_IP/To_IP pair	Section 17.12
LBR Timing	Yes	No	Section 17.12.1
Call Stack Profiling	Yes, see Section 17.11	Yes, see Section 17.11	Use LBR facility
Off-core Response Event	MSR 1A6H and 1A7H; Extended request and response types.	MSR 1A6H and 1A7H; Extended request and response types.	
Intel TSX support for Perfmon	See Section 18.3.6.5.	See Section 18.3.6.5.	

18.3.8.1 Processor Event Based Sampling (PEBS) Facility

The PEBS facility in the 6th and 7th generation Intel Core processors provides a number enhancement relative to PEBS in processors based on Haswell/Broadwell microarchitectures. The key components and differences of PEBS facility relative to Haswell/Broadwell microarchitecture is summarized in Table 18-34.

Table 18-34. PEBS Facility Comparison

Box	Intel® microarchitecture code name Skylake and Kaby Lake	Intel® microarchitecture code name Haswell and Broadwell	Comment
Valid IA32_PMCx	PMC0-PMC3	PMC0-PMC3	No PEBS on PMC4-PMC7.
PEBS Buffer Programming	Section 18.3.1.1.1	Section 18.3.1.1.1	Unchanged
IA32_PEBS_ENABLE Layout	Figure 18-15	Figure 18-15	
PEBS-EventingIP	Yes	Yes	
PEBS record format encoding	0011b	0010b	
PEBS record layout	Table 18-35; enhanced fields at offsets 98H- B8H; and TSC record field at C0H.	Table 18-24; enhanced fields at offsets 98H, A0H, A8H, B0H.	
Multi-counter PEBS resolution	PEBS record 90H resolves the eventing counter overflow.	PEBS record 90H reflects IA32_PERF_GLOBAL_STATUS.	
Precise Events	See Table 18-36.	See Table 18-12.	IA32_PMC4-IA32_PMC7 do not support PEBS.
PEBS-PDIR	Yes	Yes	IA32_PMC1 only.
PEBS-Load Latency	See Section 18.3.4.4.2.	See Section 18.3.4.4.2.	
Data Address Profiling	Yes	Yes	
FrontEnd event support	FrontEnd_Retried event and MSR_PEBS_FRONTEND.	No	IA32_PMC0-PMC3 only.

Only IA32_PMC0 through IA32_PMC3 support PEBS.

NOTES

Precise events are only valid when the following fields of IA32_PERFEVTSELx are all zero: AnyThread, Edge, Invert, CMask.

In a PMU with PDIR capability, PEBS behavior is unpredictable if IA32_PERFEVTSELx or IA32_PMCx is changed for a PEBS-enabled counter while an event is being counted. To avoid this, changes to the programming or value of a PEBS-enabled counter should be performed when the counter is disabled.

18.3.8.1.1 PEBS Data Format

The PEBS record format for the 6th and 7th generation Intel Core processors is reporting with encoding 0011b in IA32_PERF_CAPABILITIES[11:8]. The lay out is shown in Table 18-35. The PEBS record format, along with debug/store area storage format, does not change regardless of whether IA-32e mode is active or not. CPUID.01H:ECX.DTES64[bit 2] reports whether the processor's DS storage format support is mode-independent. When set, it uses 64-bit DS storage format.

Table 18-35. PEBS Record Format for 6th Generation Intel Core Processor and 7th Generation Intel Core Processor Families

Byte Offset	Field	Byte Offset	Field
00H	R/EFLAGS	68H	R11
08H	R/EIP	70H	R12
10H	R/EAX	78H	R13
18H	R/EBX	80H	R14
20H	R/ECX	88H	R15
28H	R/EDX	90H	Applicable Counter
30H	R/ESI	98H	Data Linear Address
38H	R/EDI	A0H	Data Source Encoding
40H	R/EBP	A8H	Latency value (core cycles)
48H	R/ESP	B0H	EventingIP
50H	R8	B8H	TX Abort Information (Section 18.3.6.5.1)
58H	R9	C0H	TSC
60H	R10		

The layout of PEBS records are largely identical to those shown in Table 18-24.

The PEBS records at offsets 98H, A0H, and ABH record data gathered from three of the PEBS capabilities in prior processor generations: load latency facility (Section 18.3.4.4.2), PDIR (Section 18.3.4.4.4), and data address profiling (Section 18.3.6.3).

In the core PMU of the 6th and 7th generation Intel Core processors, load latency facility and PDIR capabilities and data address profiling are unchanged relative to the 4th and 5th generation Intel Core processors. Similarly, precise store is replaced by data address profiling.

With format 0010b, a snapshot of the IA32_PERF_GLOBAL_STATUS may be useful to resolve the situations when more than one of IA32_PMICx have been configured to collect PEBS data and two consecutive overflows of the PEBS-enabled counters are sufficiently far apart in time. It is also possible for the image at 90H to indicate multiple PEBS-enabled counters have overflowed. In the latter scenario, software cannot to correlate the PEBS record entry to the multiple overflowed bits.

With PEBS record format encoding 0011b, offset 90H reports the “applicable counter” field, which is a multi-counter PEBS resolution index allowing software to correlate the PEBS record entry with the eventing PEBS overflow when multiple counters are configured to record PEBS records. Additionally, offset C0H captures a snapshot of the TSC that provides a time line annotation for each PEBS record entry.

18.3.8.1.2 PEBS Events

The list of precise events supported for PEBS in the Skylake and Kaby Lake microarchitectures is shown in Table 18-36.

Table 18-36. Precise Events for the Skylake and Kaby Lake Microarchitectures

Event Name	Event Select	Sub-event	UMask
INST_RETIRED	C0H	PREC_DIST[1]	01H
		ALL_CYCLES[2]	01H
OTHER_ASSISTS	C1H	ANY	3FH
BR_INST_RETIRED	C4H	CONDITIONAL	01H
		NEAR_CALL	02H
		ALL_BRANCHES	04H
		NEAR_RETURN	08H
		NEAR_TAKEN	20H
		FAR_BRACHES	40H
BR_MISP_RETIRED	C5H	CONDITIONAL	01H
		ALL_BRANCHES	04H
		NEAR_TAKEN	20H
FRONTEND_RETIRED	C6H	<Programmable[3]>	01H
HLE_RETIRED	C8H	ABORTED	04H
RTM_RETIRED	C9H	ABORTED	04H
MEM_INST_RETIRED[2]	D0H	LOCK_LOADS	21H
		SPLIT_LOADS	41H
		SPLIT_STORES	42H
		ALL_LOADS	81H
		ALL_STORES	82H
MEM_LOAD_RETIRED[4]	D1H	L1_HIT	01H
		L2_HIT	02H
		L3_HIT	04H
		L1_MISS	08H
		L2_MISS	10H
		L3_MISS	20H
		HIT_LFB	40H
MEM_LOAD_L3_HIT_RETIRED[2]	D2H	XSNP_MISS	01H
		XSNP_HIT	02H
		XSNP_HITM	04H
		XSNP_NONE	08H

NOTES:

1. Only available on IA32_PMC1.
2. INST_RETIRED.ALL_CYCLES is configured with additional parameters of cmask = 10 and INV = 1
3. Subevents are specified using MSR_PEBS_FRONTEND, see Section 18.3.8.2
4. Instruction with at least one load uop experiencing the condition specified in the UMask.

18.3.8.1.3 Data Address Profiling

The PEBS Data address profiling on the 6th and 7th generation Intel Core processors is largely unchanged from prior generation. When the DataLA facility is enabled, the relevant information written into a PEBS record affects entries at offsets 98H, A0H and A8H, as shown in Table 18-26.

Table 18-37. Layout of Data Linear Address Information In PEBS Record

Field	Offset	Description
Data Linear Address	98H	The linear address of the load or the destination of the store.
Store Status	A0H	▪ **DCU Hit** (Bit 0): The store hit the data cache closest to the core (L1 cache) if this bit is set, otherwise the store missed the data cache. This information is valid only for the following store events: UOPS_RETIRED.ALL (if store is tagged), MEM_INST_RETIRED.STLB_MISS_STORES, MEM_INST_RETIRED.ALL_STORES, MEM_INST_RETIRED.SPLIT_STORES. ▪ Other bits are zero.
Reserved	A8H	Always zero.

18.3.8.1.4 PEBS Facility for Front End Events

In the 6th and 7th generation Intel Core processors, the PEBS facility has been extended to allow capturing PEBS data for some microarchitectural conditions related to front end events. The frontend microarchitectural conditions supported by PEBS requires the following interfaces:

- The IA32_PERFEVTSELx MSR must select "FrontEnd_Retired" (C6H) in the EventSelect field (bits 7:0) and umask = 01H,
- The "FRONTEND_RETIRED" event employs a new MSR, MSR_PEBS_FRONTEND, to specify the supported frontend event details, see Table 18-38.
- Program the PEBS_EN_PMCx field of IA32_PEBS_ENABLE MSR as required.

Note the AnyThread field of IA32_PERFEVTSELx is ignored by the processor for the "FRONTEND_RETIRED" event.

The sub-event encodings supported by MSR_PEBS_FRONTEND.EVTSEL is given in Table 18-38.

Table 18-38. FrontEnd_Retired Sub-Event Encodings Supported by MSR_PEBS_FRONTEND.EVTSEL

Sub-Event Name	EVTSEL	Description
DSB_MISS	11H	Retired Instructions which experienced decode stream buffer (DSB) miss.
L1I_MISS	12H	The fetch of retired Instructions which experienced Instruction L1 Cache true miss[1]. Additional requests to the same cache line as an in-flight L1I cache miss will not be counted.
L2_MISS	13H	The fetch of retired Instructions which experienced L2 Cache true miss. Additional requests to the same cache line as an in-flight MLC cache miss will not be counted.
ITLB_MISS	14H	The fetch of retired Instructions which experienced ITLB true miss. Additional requests to the same cache line as an in-flight ITLB miss will not be counted.
STLB_MISS	15H	The fetch of retired Instructions which experienced STLB true miss. Additional requests to the same cache line as an in-flight STLB miss will not be counted.
IDQ_READ_BUBBLES	6H	An IDQ read bubble is defined as any one of the 4 allocation slots of IDQ that is not filled by the front-end on any cycle where there is no back end stall. Using the threshold and latency fields in MSR_PEBS_FRONTEND allows counting of IDQ read bubbles of various magnitude and duration. Latency controls the number of cycles and Threshold controls the number of allocation slots that contain bubbles. The event counts if and only if a sequence of at least FE_LATENCY consecutive cycles contain at least FE_TRESHOLD number of bubbles each.

NOTES:

1. A true miss is the first miss for a cacheline/page (excluding secondary misses that fall into same cacheline/page).

The layout of MSR_PEBS_FRONTEND is given in Table 18-39.

Table 18-39. MSR_PEBS_FRONTEND Layout

Bit Name	Offset	Description
EVTSEL	7:0	Encodes the sub-event within FrontEnd_Retired that can use PEBS facility, see Table 18-38.
IDQ_Bubble_Length	19:8	Specifies the threshold of continuously elapsed cycles for the specified width of bubbles when counting IDQ_READ_BUBBLES event.
IDQ_Bubble_Width	22:20	Specifies the threshold of simultaneous bubbles when counting IDQ_READ_BUBBLES event.
Reserved	63:23	Reserved

18.3.8.1.5 FRONTEND_RETIRED

The FRONTEND_RETIRED event is designed to help software developers identify exact instructions that caused front-end issues. There are some instances in which the event will, by design, the under-counting scenarios include the following:

- The event counts only retired (non-speculative) Frontend events, i.e. events from just true program execution path are counted.
- The event will count once per cacheline (at most). If a cacheline contains multiple instructions which caused front-end misses, the count will be only 1 for that line.
- If the multibyte sequence of an instruction spans across two cachelines and causes a miss it will be recorded once. If there were additional misses in the second cacheline, they will not be counted separately.
- If a multi-uop instruction exceeds the allocation width of one cycle, the bubbles associated with these uops will be counted once per that instruction.
- If 2 instructions are fused (macro-fusion), and either of them or both cause front-end misses, it will be counted once for the fused instruction.
- If a frontend (miss) event occurs outside instruction boundary (e.g. due to processor handling of architectural event), it may be reported for the next instruction to retire.

18.3.8.2 Off-core Response Performance Monitoring

The core PMU facility to collect off-core response events are similar to those described in Section 18.3.4.5. Each event code for off-core response monitoring requires programming an associated configuration MSR, MSR_OFFCORE_RSP_x. Software must program MSR_OFFCORE_RSP_x according to:

- Transaction request type encoding (bits 15:0): see Table 18-40.
- Supplier information (bits 30:16): see Table 18-41.
- Snoop response information (bits 37:31): see Table 18-42.

Table 18-40. MSR_OFFCORE_RSP_x Request_Type Definition (Skylake and Kaby Lake Microarchitectures)

Bit Name	Offset	Description
DMND_DATA_RD	0	(R/W). Counts the number of demand data reads of full and partial cachelines as well as demand data page table entry cacheline reads. Does not count hw or sw prefetches.
DMND_RFO	1	(R/W). Counts the number of demand reads for ownership (RFO) requests generated by a write to data cacheline. Does not count L2 RFO prefetches.
DMND_IFETCH	2	(R/W). Counts the number of demand and DCU prefetch instruction cacheline reads. Does not count L2 code read prefetches.
Reserved	6:3	Reserved
PF_L3_DATA_RD	7	(R/W). Counts the number of MLC prefetches into L3.
PF_L3_RFO	8	(R/W). Counts the number of RFO requests generated by MLC prefetches to L3.
Reserved	10:9	Reserved
STRM_ST	11	(R/W). Counts the number of streaming store requests.
Reserved	14:12	Reserved
OTHER	15	(R/W). Any other request that crosses IDI, including I/O.

Table 18-41 lists the supplier information field that applies to 6th and 7th generation Intel Core processors. (6th generation Intel Core processor CPUID signature: 06_4EH, 06_5EH; 7th generation Intel Core processor CPUID signature: 06_8EH, 06_9EH).

Table 18-41. MSR_OFFCORE_RSP_x Supplier Info Field Definition (CPUID Signature 06_4EH, 06_5EH and 06_8EH, 06_9EH)

Subtype	Bit Name	Offset	Description
Common	Any	16	(R/W). Catch all value for any response types.
Supplier Info	NO_SUPP	17	(R/W). No Supplier Information available.
	L3_HITM	18	(R/W). M-state initial lookup stat in L3.
	L3_HITE	19	(R/W). E-state
	L3_HITS	20	(R/W). S-state
	Reserved	21	Reserved
	L4_HIT	22	(R/W). L4 Cache (if L4 is present in the processor)
	Reserved	25:23	Reserved
	DRAM	26	(R/W). Local Node
	Reserved	29:27	Reserved
	SPL_HIT	30	(R/W). L4 cache super line hit (if L4 is present in the processor)

Table 18-42 lists the snoop information field that apply to processors with CPUID signatures 06_4EH, 06_5EH, 06_8EH, 06_9E, and 06_55H.

Table 18-42. MSR_OFFCORE_RSP_x Snoop Info Field Definition (CPUID Signatures 06_4EH, 06_5EH, 06_8EH, 06_9E and 06_55H)

Subtype	Bit Name	Offset	Description
Snoop Info	SNOOP_NONE	31	(R/W). No details on snoop-related information
	SNOOP_NOT_NEEDED	32	(R/W). No snoop was needed to satisfy the request.
	SNOOP_MISS	33	(R/W). A snoop was needed and it missed all snooped caches: -For LLC Hit, ReslHitI was returned by all cores -For LLC Miss, Rspl was returned by all sockets and data was returned from DRAM.
	SNOOP_HIT_NO_FWD	34	(R/W). A snoop was needed and it hits in at least one snooped cache. Hit denotes a cache-line was valid before snoop effect. This includes: -Snoop Hit w/ Invalidation (LLC Hit, RFO) -Snoop Hit, Left Shared (LLC Hit/Miss, IFetch/Data_RD) -Snoop Hit w/ Invalidation and No Forward (LLC Miss, RFO Hit S) In the LLC Miss case, data is returned from DRAM.
	SNOOP_HIT_WITH_FWD	35	(R/W). A snoop was needed and data was forwarded from a remote socket. This includes: -Snoop Forward Clean, Left Shared (LLC Hit/Miss, IFetch/Data_RD/RFT).
	SNOOP_HITM	36	(R/W). A snoop was needed and it HitM-ed in local or remote cache. HitM denotes a cache-line was in modified state before effect as a results of snoop. This includes: -Snoop HitM w/ WB (LLC miss, IFetch/Data_RD) -Snoop Forward Modified w/ Invalidation (LLC Hit/Miss, RFO) -Snoop MtoS (LLC Hit, IFetch/Data_RD).
	SNOOP_NON_DRAM	37	(R/W). Target was non-DRAM system address. This includes MMIO transactions.

18.3.8.2.1 Off-core Response Performance Monitoring for the Intel® Xeon® Processor Scalable Family

The following tables list the requestor and supplier information fields that apply to the Intel® Xeon® Processor Scalable Family.

- Transaction request type encoding (bits 15:0): see Table 18-43.
- Supplier information (bits 30:16): see Table 18-44.
- Snoop response information has not been changed and is the same as in (bits 37:31): see Table 18-42.

Table 18-43. MSR_OFFCORE_RSP_x Request_Type Definition (Intel® Xeon® Processor Scalable Family)

Bit Name	Offset	Description
DEMAND_DATA_RD	0	(R/W). Counts the number of demand data reads of full and partial cachelines as well as demand data page table entry cacheline reads. Does not count hw or sw prefetches.
DEMAND_RFO	1	(R/W). Counts the number of demand reads for ownership (RFO) requests generated by a write to data cacheline. Does not count L2 RFO prefetches.
DEMAND_CODE_RD	2	(R/W). Counts the number of demand and DCU prefetch instruction cacheline reads. Does not count L2 code read prefetches.
Reserved	3	Reserved.
PF_L2_DATA_RD	4	(R/W). Counts the number of prefetch data reads into L2.
PF_L2_RFO	5	(R/W). Counts the number of RFO Requests generated by the MLC prefetches to L2.
Reserved	6	Reserved.
PF_L3_DATA_RD	7	(R/W). Counts the number of MLC data read prefetches into L3.
PF_L3_RFO	8	(R/W). Counts the number of RFO requests generated by MLC prefetches to L3.
Reserved	9	Reserved.
PF_L1D_AND_SW	10	(R/W). Counts data cacheline reads generated by hardware L1 data cache prefetcher or software prefetch requests.
STREAMING_STORES	11	(R/W). Counts the number of streaming store requests.
Reserved	14:12	Reserved.
OTHER	15	(R/W). Any other request that crosses IDI, including I/O.

Table 18-44 lists the supplier information field that applies to the Intel Xeon Processor Scalable Family (CPUID signature: 06_55H).

Table 18-44. MSR_OFFCORE_RSP_x Supplier Info Field Definition (CPUID Signature 06_55H)

Subtype	Bit Name	Offset	Description
Common	Any	16	(R/W). Catch all value for any response types.
Supplier Info	SUPPLIER_NONE	17	(R/W). No Supplier Information available.
	L3_HIT_M	18	(R/W). M-state initial lookup stat in L3.
	L3_HIT_E	19	(R/W). E-state
	L3_HIT_S	20	(R/W). S-state
	L3_HIT_F	21	(R/W). F-state
	Reserved	25:22	Reserved.
	L3_MISS_LOCAL_DRAM	26	(R/W). L3 Miss: local home requests that missed the L3 cache and were serviced by local DRAM.
	L3_MISS_REMOTE_HOP0_DRAM	27	(R/W). Hop 0 Remote supplier.
	L3_MISS_REMOTE_HOP1_DRAM	28	(R/W). Hop 1 Remote supplier.
	L3_MISS_REMOTE_HOP2P_DRAM	29	(R/W). Hop 2 or more Remote supplier.
	Reserved	30	Reserved.

18.4 PERFORMANCE MONITORING (INTEL® XEON™ PHI PROCESSORS)

18.4.1 Intel® Xeon Phi™ Processor 7200/5200/3200 Performance Monitoring

The Intel® Xeon Phi™ processor 7200/5200/3200 series are based on the Knights Landing microarchitecture. The performance monitoring capabilities are distributed between its tiles (pair of processor cores) and untile (connecting many tiles in a physical processor package). Functional details of the tiles and untile of the Knights Landing microarchitecture can be found in Chapter 16 of *Intel® 64 and IA-32 Architectures Optimization Reference Manual*.

A complete description of the tile and untile PMU programming interfaces for Intel Xeon Phi processors based on the Knights Landing microarchitecture can be found in the Technical Document section at http://www.intel.com/content/www/us/en/processors/xeon/xeon-phi-detail.html.

A tile contains a pair of cores attached to a shared L2 cache and is similar to those found in Intel® Atom™ processors based on the Silvermont microarchitecture. The processor provides several new capabilities on top of the Silvermont performance monitoring facilities.

The processor supports architectural performance monitoring capability with version ID 3 (see Section 18.2.3) and a host of non-architectural performance monitoring capabilities. The processor provides two general-purpose performance counters (IA32_PMC0, IA32_PMC1) and three fixed-function performance counters (IA32_FIXED_CTR0, IA32_FIXED_CTR1, IA32_FIXED_CTR2).

Non-architectural performance monitoring in the processor also uses the IA32_PERFEVTSELx MSR to configure a set of non-architecture performance monitoring events to be counted by the corresponding general-purpose performance counter.

The bit fields within each IA32_PERFEVTSELx MSR are defined in Figure 18-6 and described in Section 18.2.1.1 and Section 18.2.3 in the SDM. The processor supports AnyThread counting in three architectural performance monitoring events.

18.4.1.1 Enhancements of Performance Monitoring in the Intel® Xeon Phi™ processor Tile

The Intel® Xeon Phi™ processor tile includes the following enhancements to the Silvermont microarchitecture.

- AnyThread support. This facility is limited to following three architectural events: Instructions Retired, Unhalted Core Cycles, Unhalted Reference Cycles using IA32_FIXED_CTR0-2 and Unhalted Core Cycles, Unhalted Reference Cycles using IA32_PERFEVTSELx.
- PEBS-DLA (Processor Event-Based Sampling-Data Linear Address) fields. The processor provides memory address in addition to the Silvermont PEBS record support on select events. The PEBS recording format as reported by IA32_PERF_CAPABILITIES [11:8] is 2.
- Off-core response counting facility. This facility in the processor core allows software to count certain transaction responses between the processor tile to subsystems outside the tile (untile). Counting off-core response requires additional event qualification configuration facility in conjunction with IA32_PERFEVTSELx. Two off-core response MSRs are provided to use in conjunction with specific event codes that must be specified with IA32_PERFEVTSELx. Two cores do not share the off-core response MSRs. Knights Landing expands off-core response capability to match the processor untile changes.
- Average request latency measurement. The off-core response counting facility can be combined to use two performance counters to count the occurrences and weighted cycles of transaction requests. This facility is updated to match the processor untile changes.

18.4.1.1.1 Processor Event-Based Sampling

The processor supports processor event based sampling (PEBS). PEBS is supported using IA32_PMC0 (see also Section 17.4.9, “BTS and DS Save Area”).

PEBS uses a debug store mechanism to store a set of architectural state information for the processor. The information provides architectural state of the instruction executed after the instruction that caused the event (See Section 18.6.2.4).

The list of PEBS events supported in the processor is shown in the following table.

Table 18-45. PEBS Performance Events for the Knights Landing Microarchitecture

Event Name	Event Select	Sub-event	UMask	Data Linear Address Support
BR_INST_RETIRED	C4H	ALL_BRANCHES	00H	No
		JCC	7EH	No
		TAKEN_JCC	FEH	No
		CALL	F9H	No
		REL_CALL	FDH	No
		IND_CALL	FBH	No
		NON_RETURN_IND	EBH	No
		FAR_BRANCH	BFH	No
		RETURN	F7H	No
BR_MISP_RETIRED	C5H	ALL_BRANCHES	00H	No
		JCC	7EH	No
		TAKEN_JCC	FEH	No
		IND_CALL	FBH	No
		NON_RETURN_IND	EBH	No
		RETURN	F7H	No
MEM_UOPS_RETIRED	04H	L2_HIT_LOADS	02H	Yes
		L2_MISS_LOADS	04H	Yes
		DLTB_MISS_LOADS	08H	Yes
RECYCLEQ	03H	LD_BLOCK_ST_FORWARD	01H	Yes
		LD_SPLITS	08H	Yes

The PEBS record format 2 supported by processors based on the Knights Landing microarchitecture is shown in Table 18-46, and each field in the PEBS record is 64 bits long.

Table 18-46. PEBS Record Format for the Knights Landing Microarchitecture

Byte Offset	Field	Byte Offset	Field
00H	R/EFLAGS	60H	R10
08H	R/EIP	68H	R11
10H	R/EAX	70H	R12
18H	R/EBX	78H	R13
20H	R/ECX	80H	R14
28H	R/EDX	88H	R15
30H	R/ESI	90H	IA32_PERF_GLOBAL_STATUS
38H	R/EDI	98H	PSDLA
40H	R/EBP	A0H	Reserved
48H	R/ESP	A8H	Reserved
50H	R8	B0H	EventingRIP
58H	R9	B8H	Reserved

18.4.1.1.2 Offcore Response Event

Event number 0B7H support offcore response monitoring using an associated configuration MSR, MSR_OFFCORE_RSP0 (address 1A6H) in conjunction with umask value 01H or MSR_OFFCORE_RSP1 (address 1A7H) in conjunction with umask value 02H. Table 18-47 lists the event code, mask value and additional off-core configuration MSR that must be programmed to count off-core response events using IA32_PMCx.

Table 18-47. OffCore Response Event Encoding

Counter	Event code	UMask	Required Off-core Response MSR
PMC0-1	B7H	01H	MSR_OFFCORE_RSP0 (address 1A6H)
PMC0-1	B7H	02H	MSR_OFFCORE_RSP1 (address 1A7H)

Some of the MSR_OFFCORE_RESP [0,1] register bits are not valid in this processor and their use is reserved. The layout of MSR_OFFCORE_RSP0 and MSR_OFFCORE_RSP1 registers are defined in Table 18-48. Bits 15:0 specifies the request type of a transaction request to the uncore. Bits 30:16 specifies supplier information, bits 37:31 specifies snoop response information.

Additionally, MSR_OFFCORE_RSP0 provides bit 38 to enable measurement of average latency of specific type of offcore transaction requests using two programmable counter simultaneously, see Section 18.5.2.3 for details.

Table 18-48. Bit fields of the MSR_OFFCORE_RESP [0, 1] Registers

Main	Sub-field	Bit	Name	Description
Request Type		0	DEMAND_DATA_RD	Demand cacheable data and L1 prefetch data reads.
		1	DEMAND_RFO	Demand cacheable data writes.
		2	DEMAND_CODE_RD	Demand code reads and prefetch code reads.
		3	Reserved	Reserved.
		4	Reserved	Reserved.
		5	PF_L2_RFO	L2 data RFO prefetches (includes PREFETCHW instruction).
		6	PF_L2_CODE_RD	L2 code HW prefetches.
		7	PARTIAL_READS	Partial reads (UC or WC).
		8	PARTIAL_WRITES	Partial writes (UC or WT or WP). Valid only for OFFCORE_RESP_1 event. Should only be used on PMC1. This bit is reserved for OFFCORE_RESP_0 event.
		9	UC_CODE_READS	UC code reads.
		10	BUS_LOCKS	Bus locks and split lock requests.
		11	FULL_STREAMING_STORES	Full streaming stores (WC). Valid only for OFFCORE_RESP_1 event. Should only be used on PMC1. This bit is reserved for OFFCORE_RESP_0 event.
		12	SW_PREFETCH	Software prefetches.
		13	PF_L1_DATA_RD	L1 data HW prefetches.
		14	PARTIAL_STREAMING_STORES	Partial streaming stores (WC). Valid only for OFFCORE_RESP_1 event. Should only be used on PMC1. This bit is reserved for OFFCORE_RESP_0 event.
		15	ANY_REQUEST	Account for any requests.
Response Type	Any	16	ANY_RESPONSE	Account for any response.
	Data Supply from Untile	17	NO_SUPP	No Supplier Details.
		18	Reserved	Reserved.

Table 18-48. Bit fields of the MSR_OFFCORE_RESP [0, 1] Registers (Contd.)

Main	Sub-field	Bit	Name	Description
		19	L2_HIT_OTHER_TILE_NEAR	Other tile L2 hit E Near.
		20	Reserved	Reserved.
		21	MCDRAM_NEAR	MCDRAM Local.
		22	MCDRAM_FAR_OR_L2_HIT_OTHER_TILE_FAR	MCDRAM Far or Other tile L2 hit far.
		23	DRAM_NEAR	DRAM Local.
		24	DRAM_FAR	DRAM Far.
	Data Supply from within same tile	25	L2_HITM_THIS_TILE	M-state.
		26	L2_HITE_THIS_TILE	E-state.
		27	L2_HITS_THIS_TILE	S-state.
		28	L2_HITF_THIS_TILE	F-state.
		29	Reserved	Reserved.
		30	Reserved	Reserved.
	Snoop Info; Only Valid in case of Data Supply from Untile	31	SNOOP_NONE	None of the cores were snooped.
		32	NO_SNOOP_NEEDED	No snoop was needed to satisfy the request.
		33	Reserved	Reserved.
		34	Reserved	Reserved.
		35	HIT_OTHER_TILE_FWD	Snoop request hit in the other tile with data forwarded.
		36	HITM_OTHER_TILE	A snoop was needed and it HitM-ed in other core's L1 cache. HitM denotes a cache-line was in modified state before effect as a result of snoop.
		37	NON_DRAM	Target was non-DRAM system address. This includes MMIO transactions.
Outstanding requests	Weighted cycles	38	OUTSTANDING (Valid only for MSR_OFFCORE_RESP0. Should only be used on PMC0. This bit is reserved for MSR_OFFCORE_RESP1).	If set, counts total number of weighted cycles of any outstanding offcore requests with data response. Valid only for OFFCORE_RESP_0 event. Should only be used on PMC0. This bit is reserved for OFFCORE_RESP_1 event.

18.4.1.1.3 Average Offcore Request Latency Measurement

Measurement of average latency of offcore transaction requests can be enabled using MSR_OFFCORE_RSP0.[bit 38] with the choice of request type specified in MSR_OFFCORE_RSP0.[bit 15:0].

Refer to Section 18.5.2.3, "Average Offcore Request Latency Measurement," for typical usage. Note that MSR_OFFCORE_RESPx registers are not shared between cores in Knights Landing. This allows one core to measure average latency while other core is measuring different offcore response events.

18.5 PERFORMANCE MONITORING (INTEL® ATOM™ PROCESSORS)

18.5.1 Performance Monitoring (45 nm and 32 nm Intel® Atom™ Processors)

45 nm and 32 nm Intel Atom processors report architectural performance monitoring versionID = 3 (supporting the aggregate capabilities of versionID 1, 2, and 3; see Section 18.2.3) and a host of non-architectural monitoring capabilities. These 45 nm and 32 nm Intel Atom processors provide two general-purpose performance counters (IA32_PMC0, IA32_PMC1) and three fixed-function performance counters (IA32_FIXED_CTR0, IA32_FIXED_CTR1, IA32_FIXED_CTR2).

Non-architectural performance monitoring in Intel Atom processor family uses the IA32_PERFEVTSELx MSR to configure a set of non-architecture performance monitoring events to be counted by the corresponding general-purpose performance counter. The list of non-architectural performance monitoring events is listed in Table 19-28.

Architectural and non-architectural performance monitoring events in 45 nm and 32 nm Intel Atom processors support thread qualification using bit 21 (AnyThread) of IA32_PERFEVTSELx MSR, i.e. if IA32_PERFEVTSELx.AnyThread =1, event counts include monitored conditions due to either logical processors in the same processor core.

The bit fields within each IA32_PERFEVTSELx MSR are defined in Figure 18-6 and described in Section 18.2.1.1 and Section 18.2.3.

Valid event mask (Umask) bits are listed in Chapter 19. The UMASK field may contain sub-fields that provide the same qualifying actions like those listed in Table 18-61, Table 18-62, Table 18-63, and Table 18-64. One or more of these sub-fields may apply to specific events on an event-by-event basis. Details are listed in Table 19-28 in Chapter 19, "Performance-Monitoring Events." Precise Event Based Monitoring is supported using IA32_PMC0 (see also Section 17.4.9, "BTS and DS Save Area").

18.5.2 Performance Monitoring for Silvermont Microarchitecture

Intel processors based on the Silvermont microarchitecture report architectural performance monitoring versionID = 3 (see Section 18.2.3) and a host of non-architectural monitoring capabilities. Intel processors based on the Silvermont microarchitecture provide two general-purpose performance counters (IA32_PMC0, IA32_PMC1) and three fixed-function performance counters (IA32_FIXED_CTR0, IA32_FIXED_CTR1, IA32_FIXED_CTR2). Intel Atom processors based on the Airmont microarchitecture support the same performance monitoring capabilities as those based on the Silvermont microarchitecture.

Non-architectural performance monitoring in the Silvermont microarchitecture uses the IA32_PERFEVTSELx MSR to configure a set of non-architecture performance monitoring events to be counted by the corresponding general-purpose performance counter. The list of non-architectural performance monitoring events is listed in Table 19-27.

The bit fields (except bit 21) within each IA32_PERFEVTSELx MSR are defined in Figure 18-6 and described in Section 18.2.1.1 and Section 18.2.3. Architectural and non-architectural performance monitoring events in the Silvermont microarchitecture ignore the AnyThread qualification regardless of its setting in IA32_PERFEVTSELx MSR.

18.5.2.1 Enhancements of Performance Monitoring in the Processor Core

The notable enhancements in the monitoring of performance events in the processor core include:

- The width of counter reported by CPUID.0AH:EAX[23:16] is 40 bits.
- Off-core response counting facility. This facility in the processor core allows software to count certain transaction responses between the processor core to sub-systems outside the processor core (uncore). Counting off-core response requires additional event qualification configuration facility in conjunction with IA32_PERFEVTSELx. Two off-core response MSRs are provided to use in conjunction with specific event codes that must be specified with IA32_PERFEVTSELx.
- Average request latency measurement. The off-core response counting facility can be combined to use two performance counters to count the occurrences and weighted cycles of transaction requests.

18.5.2.1.1 Processor Event Based Sampling (PEBS)

In the Silvermont microarchitecture, the PEBS facility can be used with precise events. PEBS is supported using IA32_PMC0 (see also Section 17.4.9).

PEBS uses a debug store mechanism to store a set of architectural state information for the processor. The information provides architectural state of the instruction executed after the instruction that caused the event (See Section 18.6.2.4).

The list of precise events supported in the Silvermont microarchitecture is shown in Table 18-49.

Table 18-49. PEBS Performance Events for the Silvermont Microarchitecture

Event Name	Event Select	Sub-event	UMask
BR_INST_RETIRED	C4H	ALL_BRANCHES	00H
		JCC	7EH
		TAKEN_JCC	FEH
		CALL	F9H
		REL_CALL	FDH
		IND_CALL	FBH
		NON_RETURN_IND	EBH
		FAR_BRANCH	BFH
		RETURN	F7H
BR_MISP_RETIRED	C5H	ALL_BRANCHES	00H
		JCC	7EH
		TAKEN_JCC	FEH
		IND_CALL	FBH
		NON_RETURN_IND	EBH
		RETURN	F7H
MEM_UOPS_RETIRED	04H	L2_HIT_LOADS	02H
		L2_MISS_LOADS	04H
		DLTB_MISS_LOADS	08H
		HITM	20H
REHABQ	03H	LD_BLOCK_ST_FORWARD	01H
		LD_SPLITS	08H

PEBS Record Format The PEBS record format supported by processors based on the Intel Silvermont microarchitecture is shown in Table 18-50, and each field in the PEBS record is 64 bits long.

Table 18-50. PEBS Record Format for the Silvermont Microarchitecture

Byte Offset	Field	Byte Offset	Field
00H	R/EFLAGS	60H	R10
08H	R/EIP	68H	R11
10H	R/EAX	70H	R12
18H	R/EBX	78H	R13
20H	R/ECX	80H	R14
28H	R/EDX	88H	R15
30H	R/ESI	90H	IA32_PERF_GLOBAL_STATUS
38H	R/EDI	98H	Reserved

Table 18-50. PEBS Record Format for the Silvermont Microarchitecture

Byte Offset	Field	Byte Offset	Field
40H	R/EBP	A0H	Reserved
48H	R/ESP	A8H	Reserved
50H	R8	B0H	EventingRIP
58H	R9	B8H	Reserved

18.5.2.2 Offcore Response Event

Event number 0B7H support offcore response monitoring using an associated configuration MSR, MSR_OFFCORE_RSP0 (address 1A6H) in conjunction with umask value 01H or MSR_OFFCORE_RSP1 (address 1A7H) in conjunction with umask value 02H. Table 18-51 lists the event code, mask value and additional off-core configuration MSR that must be programmed to count off-core response events using IA32_PMCx.

In the Silvermont microarchitecture, each MSR_OFFCORE_RSPx is shared by two processor cores.

Table 18-51. OffCore Response Event Encoding

Counter	Event code	UMask	Required Off-core Response MSR
PMC0-1	B7H	01H	MSR_OFFCORE_RSP0 (address 1A6H)
PMC0-1	B7H	02H	MSR_OFFCORE_RSP1 (address 1A7H)

The layout of MSR_OFFCORE_RSP0 and MSR_OFFCORE_RSP1 are shown in Figure 18-37 and Figure 18-38. Bits 15:0 specifies the request type of a transaction request to the uncore. Bits 30:16 specifies supplier information, bits 37:31 specifies snoop response information.

Additionally, MSR_OFFCORE_RSP0 provides bit 38 to enable measurement of average latency of specific type of offcore transaction requests using two programmable counter simultaneously, see Section 18.5.2.3 for details.

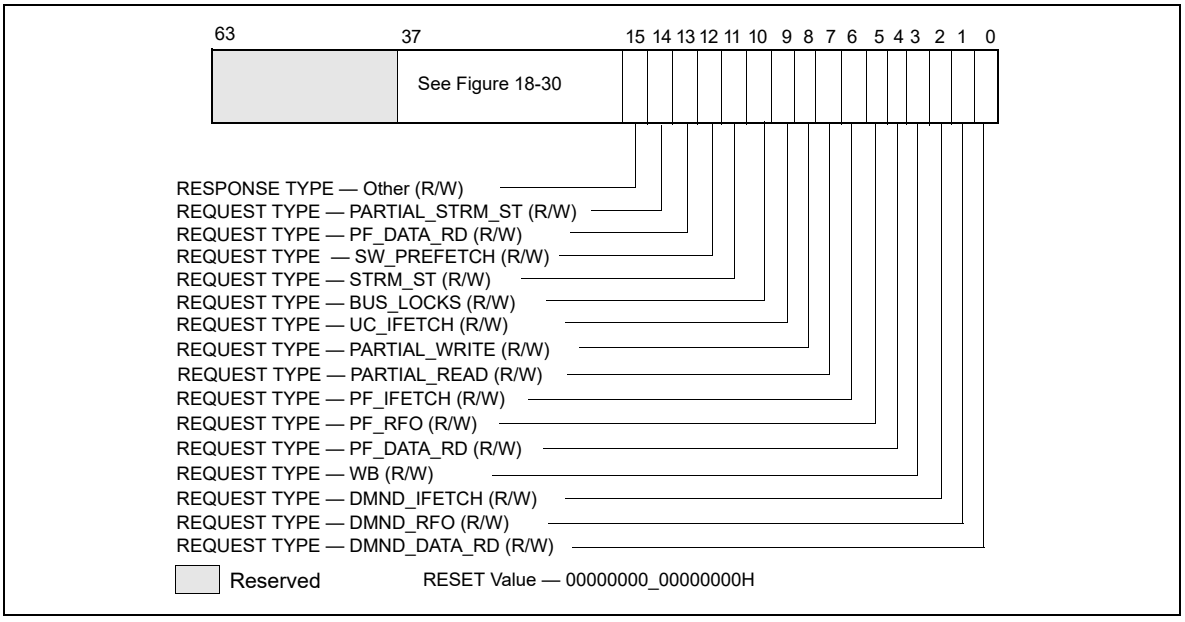


Figure 18-37. Request_Type Fields for MSR_OFFCORE_RSPx

Table 18-52. MSR_OFFCORE_RSPx Request_Type Field Definition

Bit Name	Offset	Description
DMND_DATA_RD	0	(R/W). Counts the number of demand and DCU prefetch data reads of full and partial cachelines as well as demand data page table entry cacheline reads. Does not count L2 data read prefetches or instruction fetches.
DMND_RFO	1	(R/W). Counts the number of demand and DCU prefetch reads for ownership (RFO) requests generated by a write to data cacheline. Does not count L2 RFO prefetches.
DMND_IFETCH	2	(R/W). Counts the number of demand and DCU prefetch instruction cacheline reads. Does not count L2 code read prefetches.
WB	3	(R/W). Counts the number of writeback (modified to exclusive) transactions.
PF_DATA_RD	4	(R/W). Counts the number of data cacheline reads generated by L2 prefetchers.
PF_RFO	5	(R/W). Counts the number of RFO requests generated by L2 prefetchers.
PF_IFETCH	6	(R/W). Counts the number of code reads generated by L2 prefetchers.
PARTIAL_READ	7	(R/W). Counts the number of demand reads of partial cache lines (including UC and WC).
PARTIAL_WRITE	8	(R/W). Counts the number of demand RFO requests to write to partial cache lines (includes UC, WT and WP)
UC_IFETCH	9	(R/W). Counts the number of UC instruction fetches.
BUS_LOCKS	10	(R/W). Bus lock and split lock requests
STRM_ST	11	(R/W). Streaming store requests
SW_PREFETCH	12	(R/W). Counts software prefetch requests
PF_DATA_RD	13	(R/W). Counts DCU hardware prefetcher data read requests
PARTIAL_STRM_ST	14	(R/W). Streaming store requests
ANY	15	(R/W). Any request that crosses IDI, including I/O.

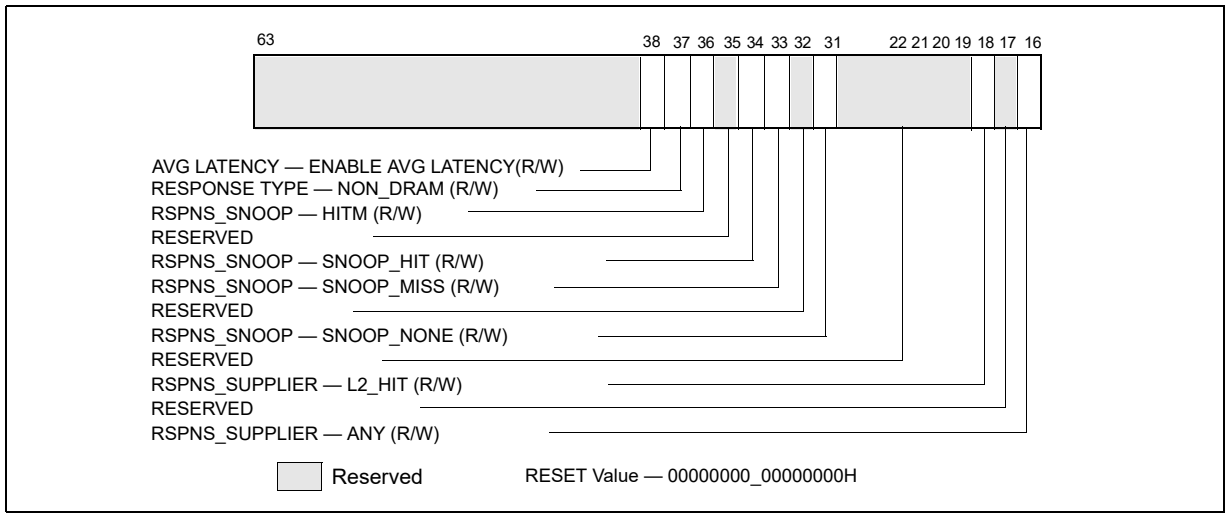


Figure 18-38. Response_Supplier and Snoop Info Fields for MSR_OFFCORE_RSPx

To properly program this extra register, software must set at least one request type bit (Table 18-52) and a valid response type pattern (Table 18-53, Table 18-54). Otherwise, the event count reported will be zero. It is permissible and useful to set multiple request and response type bits in order to obtain various classes of off-core response events. Although MSR_OFFCORE_RSPx allow an agent software to program numerous combinations that meet the above guideline, not all combinations produce meaningful data.

Table 18-53. MSR_OFFCORE_RSP_x Response Supplier Info Field Definition

Subtype	Bit Name	Offset	Description
Common	ANY_RESPONSE	16	(R/W). Catch all value for any response types.
Supplier Info	Reserved	17	Reserved
	L2_HIT	18	(R/W). Cache reference hit L2 in either M/E/S states.
	Reserved	30:19	Reserved

To specify a complete offcore response filter, software must properly program bits in the request and response type fields. A valid request type must have at least one bit set in the non-reserved bits of 15:0. A valid response type must be a non-zero value of the following expression:

ANY | [('OR' of Supplier Info Bits) & ('OR' of Snoop Info Bits)]

If "ANY" bit is set, the supplier and snoop info bits are ignored.

Table 18-54. MSR_OFFCORE_RSPx Snoop Info Field Definition

Subtype	Bit Name	Offset	Description
Snoop Info	SNP_NONE	31	(R/W). No details on snoop-related information.
	Reserved	32	Reserved
	SNOOP_MISS	33	(R/W). Counts the number of snoop misses when L2 misses.
	SNOOP_HIT	34	(R/W). Counts the number of snoops hit in the other module where no modified copies were found.
	Reserved	35	Reserved
	HITM	36	(R/W). Counts the number of snoops hit in the other module where modified copies were found in other core's L1 cache.
	NON_DRAM	37	(R/W). Target was non-DRAM system address. This includes MMIO transactions.
	AVG_LATENCY	38	(R/W). Enable average latency measurement by counting weighted cycles of outstanding offcore requests of the request type specified in bits 15:0 and any response (bits 37:16 cleared to 0). This bit is available in MSR_OFFCORE_RESP0. The weighted cycles is accumulated in the specified programmable counter IA32_PMCx and the occurrence of specified requests are counted in the other programmable counter.

18.5.2.3 Average Offcore Request Latency Measurement

Average latency for offcore transactions can be determined by using both MSR_OFFCORE_RSP registers. Using two performance monitoring counters, program the two OFFCORE_RESPONSE event encodings into the corresponding IA32_PERFEVTSELx MSRs. Count the weighted cycles via MSR_OFFCORE_RSP0 by programming a request type in MSR_OFFCORE_RSP0.[15:0] and setting MSR_OFFCORE_RSP0.OUTSTANDING[38] to 1, white setting the remaining bits to 0. Count the number of requests via MSR_OFFCORE_RSP1 by programming the same request type from MSR_OFFCORE_RSP0 into MSR_OFFCORE_RSP1[bit 15:0], and setting MSR_OFFCORE_RSP1.ANY_RESPONSE[16] = 1, while setting the remaining bits to 0. The average latency can be

obtained by dividing the value of the IA32_PMCx register that counted weight cycles by the register that counted requests.

18.5.3 Performance Monitoring for Goldmont Microarchitecture

Intel Atom processors based on the Goldmont microarchitecture report architectural performance monitoring versionID = 4 (see Section 18.2.4) and support non-architectural monitoring capabilities described in this section.

Architectural performance monitoring version 4 capabilities are described in Section 18.2.4.

The bit fields (except bit 21) within each IA32_PERFEVTSELx MSR are defined in Figure 18-6 and described in Section 18.2.1.1 and Section 18.2.3. The Goldmont microarchitecture does not support Hyper-Threading and thus architectural and non-architectural performance monitoring events ignore the AnyThread qualification regardless of its setting in the IA32_PERFEVTSELx MSR. However, Goldmont does not set the AnyThread deprecation bit (CPUID.0AH:EDX[15]).

The core PMU's capability is similar to that of the Silvermont microarchitecture described in Section 18.5.2 , with some differences and enhancements summarized in Table 18-55.

Table 18-55. Core PMU Comparison Between the Goldmont and Silvermont Microarchitectures

Box	The Goldmont microarchitecture	The Silvermont microarchitecture	Comment
# of Fixed counters per core	3	3	Use CPUID to enumerate # of counters.
# of general-purpose counters per core	4	2	
Counter width (R,W)	R:48, W: 32/48	R:40, W:32	See Section 18.2.2.
Architectural Performance Monitoring version ID	4	3	Use CPUID to enumerate # of counters.
PMI Overhead Mitigation	▪ Freeze_Perfmon_on_PMI with streamlined semantics. ▪ Freeze_LBR_on_PMI with streamlined semantics for branch profiling.	▪ Freeze_Perfmon_on_PMI with legacy semantics. ▪ Freeze_LBR_on_PMI with legacy semantics for branch profiling.	See Section 17.4.7. Legacy semantics not supported with version 4 or higher.
Counter and Buffer Overflow Status Management	▪ Query via IA32_PERF_GLOBAL_STATUS ▪ Reset via IA32_PERF_GLOBAL_STATUS_RESET ▪ Set via IA32_PERF_GLOBAL_STATUS_SET	▪ Query via IA32_PERF_GLOBAL_STATUS ▪ Reset via IA32_PERF_GLOBAL_OVF_CTRL	See Section 18.2.4.
IA32_PERF_GLOBAL_STATUS Indicators of Overflow/Overhead/Interference	▪ Individual counter overflow ▪ PEBS buffer overflow ▪ ToPA buffer overflow ▪ CTR_Frz, LBR_Frz	▪ Individual counter overflow ▪ PEBS buffer overflow	See Section 18.2.4.
Enable control in IA32_PERF_GLOBAL_STATUS	▪ CTR_Frz, ▪ LBR_Frz	No	See Section 18.2.4.1.
Perfmon Counter In-Use Indicator	Query IA32_PERF_GLOBAL_INUSE	No	See Section 18.2.4.3.
Processor Event Based Sampling (PEBS) Events	General-Purpose Counter 0 only. Supports all events (precise and non-precise). Precise events are listed in Table 18-56.	See Section 18.5.2.1.1. General-Purpose Counter 0 only. Only supports precise events (see Table 18-49).	IA32_PMC0 only.

Table 18-55. Core PMU Comparison Between the Goldmont and Silvermont Microarchitectures

Box	The Goldmont microarchitecture	The Silvermont microarchitecture	Comment
PEBS record format encoding	0011b	0010b	
Reduce skid PEBS	IA32_PMC0 only	No	
Data Address Profiling	Yes	No	
PEBS record layout	Table 18-57; enhanced fields at offsets 90H- 98H; and TSC record field at C0H.	Table 18-50.	
PEBS EventingIP	Yes	Yes	
Off-core Response Event	MSR 1A6H and 1A7H, each core has its own register.	MSR 1A6H and 1A7H, shared by a pair of cores.	Nehalem supports 1A6H only.

18.5.3.1 Processor Event Based Sampling (PEBS)

Processor event based sampling (PEBS) on the Goldmont microarchitecture is enhanced over prior generations with respect to sampling support of precise events and non-precise events. In the Goldmont microarchitecture, PEBS is supported using IA32_PMC0 for all events (see Section 17.4.9).

PEBS uses a debug store mechanism to store a set of architectural state information for the processor at the time the sample was generated.

Precise events work the same way on Goldmont microarchitecture as on the Silvermont microarchitecture. The record will be generated after an instruction that causes the event when the counter is already overflowed and will capture the architectural state at this point (see Section 18.6.2.4 and Section 17.4.9). The eventingIP in the record will indicate the instruction that caused the event. The list of precise events supported in the Goldmont microarchitecture is shown in Table 18-56.

In the Goldmont microarchitecture, the PEBS facility also supports the use of non-precise events to record processor state information into PEBS records with the same format as with precise events.

However, a non-precise event may not be attributable to a particular retired instruction or the time of instruction execution. When the counter overflows, a PEBS record will be generated at the next opportunity. Consider the event ICACHE.HIT. When the counter overflows, the processor is fetching future instructions. The PEBS record will be generated at the next opportunity and capture the state at the processor's current retirement point. It is likely that the instruction fetch that caused the event to increment was beyond that current retirement point. Other examples of non-precise events are CPU_CLK_UNHALTED.CORE_P and HARDWARE_INTERRUPTS.RECEIVED. CPU_CLK_UNHALTED.CORE_P will increment each cycle that the processor is awake. When the counter over-flows, there may be many instructions in various stages of execution. Additionally, zero, one or multiple instructions may be retired the cycle that the counter overflows. HARDWARE_INTERRUPTS.RECEIVED increments independent of any instructions being executed. For all non-precise events, the PEBS record will be generated at the next opportunity, after the counter has overflowed. The PEBS facility thus allows for identification of the instructions which were executing when the event overflowed.

After generating a record for a non-precise event, the PEBS facility reloads the counter and resumes execution, just as is done for precise events. Unlike interrupt-based sampling, which requires an interrupt service routine to collect the sample and reload the counter, the PEBS facility can collect samples even when interrupts are masked and without using NMI. Since a PEBS record is generated immediately when a counter for a non-precise event is enabled, it may also be generated after an overflow is set by an MSR write to IA32_PERF_GLOBAL_STATUS_SET.

Table 18-56. Precise Events Supported by the Goldmont Microarchitecture

Event Name	Event Select	Sub-event	UMask
LD_BLOCKS	03H	DATA_UNKNOWN	01H
		STORE_FORWARD	02H
		4K_ALIAS	04H
		UTLB_MISS	08H
		ALL_BLOCK	10H
MISALIGN_MEM_REF	13H	LOAD_PAGE_SPLIT	02H
		STORE_PAGE_SPLIT	04H
INST_RETIRED	C0H	ANY	00H
UOPS_RETITRED	C2H	ANY	00H
		LD_SPLITSMS	01H
BR_INST_RETIRED	C4H	ALL_BRANCHES	00H
		JCC	7EH
		TAKEN_JCC	FEH
		CALL	F9H
		REL_CALL	FDH
		IND_CALL	FBH
		NON_RETURN_IND	EBH
		FAR_BRANCH	BFH
		RETURN	F7H
BR_MISP_RETIRED	C5H	ALL_BRANCHES	00H
		JCC	7EH
		TAKEN_JCC	FEH
		IND_CALL	FBH
		NON_RETURN_IND	EBH
		RETURN	F7H
MEM_UOPS_RETIRED	D0H	ALL_LOADS	81H
		ALL_STORES	82H
		ALL	83H
		DLTB_MISS_LOADS	11H
		DLTB_MISS_STORES	12H
		DLTB_MISS	13H
MEM_LOAD_UOPS_RETIRED	D1H	L1_HIT	01H
		L2_HIT	02H
		L1_MISS	08H
		L2_MISS	10H
		HITM	20H
		WCB_HIT	40H
		DRAM_HIT	80H

The PEBS record format supported by processors based on the Intel Goldmont microarchitecture is shown in Table 18-57, and each field in the PEBS record is 64 bits long.

Table 18-57. PEBS Record Format for the Goldmont Microarchitecture

Byte Offset	Field	Byte Offset	Field
00H	R/EFLAGS	68H	R11
08H	R/EIP	70H	R12
10H	R/EAX	78H	R13
18H	R/EBX	80H	R14
20H	R/ECX	88H	R15
28H	R/EDX	90H	Applicable Counters
30H	R/ESI	98H	Data Linear Address
38H	R/EDI	A0H	Reserved
40H	R/EBP	A8H	Reserved
48H	R/ESP	B0H	EventingRIP
50H	R8	B8H	Reserved
58H	R9	C0H	TSC
60H	R10		

On Goldmont microarchitecture, all 64 bits of architectural registers are written into the PEBS record regardless of processor mode.

With PEBS record format encoding 0011b, offset 90H reports the "Applicable Counter" field, which indicates which counters actually requested generating a PEBS record. This allows software to correlate the PEBS record entry properly with the instruction that caused the event even when multiple counters are configured to record PEBS records and multiple bits are set in the field. Additionally, offset C0H captures a snapshot of the TSC that provides a time line annotation for each PEBS record entry.

18.5.3.1.1 PEBS Data Linear Address Profiling

Goldmont supports the Data Linear Address field introduced in Haswell. It does not support the Data Source Encoding or Latency Value fields that are also part of Data Address Profiling; those fields are present in the record but are reserved.

For Goldmont microarchitecture, the Data Linear Address field will record the linear address of memory accesses in the previous instruction (e.g. the one that triggered a precise event that caused the PEBS record to be generated). Goldmont microarchitecture may record a Data Linear Address for the instruction that caused the event even for events not related to memory accesses. This may differ from other microarchitectures.

18.5.3.1.2 Reduced Skid PEBS

For precise events, upon triggering a PEBS assist, there will be a finite delay between the time the counter overflows and when the microcode starts to carry out its data collection obligations. The Reduced Skid mechanism mitigates the “skid” problem by providing an early indication of when the counter is about to overflow, allowing the machine to more precisely trap on the instruction that actually caused the counter overflow thus greatly reducing skid.

This mechanism is a superset of the PDIR mechanism available in the Sandy Bridge microarchitecture. See Section 18.3.4.4.4

In the Goldmont microarchitecture, the mechanism applies to all precise events including, INST_RETIRED, except for UOPS_RETIRED. However, the Reduced Skid mechanism is disabled for any counter when the INV, ANY, E, or CMASK fields are set.

For the Reduced Skid mechanism to operate correctly, the performance monitoring counters should not be reconfigured or modified when they are running with PEBS enabled. The counters need to be disabled (e.g. via

IA32_PERF_GLOBAL_CTRL MSR) before changes to the configuration (e.g. what event is specified in IA32_PERFEVTSELx or whether PEBS is enabled for that counter via IA32_PEBS_ENABLE) or counter value (MSR write to IA32_PMCx and IA32_A_PMCx).

18.5.3.1.3 Enhancements to IA32_PERF_GLOBAL_STATUS.OvfDSBuffer[62]

In addition to IA32_PERF_GLOBAL_STATUS.OvfDSBuffer[62] being set when PEBS_Index reaches the PEBS_Interrupt_Theshold, the bit is also set when PEBS_Index is out of bounds. That is, the bit will be set when PEBS_Index < PEBS_Buffer_Base or PEBS_Index > PEBS_Absolute_Maximum. Note that when an out of bound condition is encountered, the overflow bits in IA32_PERF_GLOBAL_STATUS will be cleared according to Applicable Counters, however the IA32_PMCx values will not be reloaded with the Reset values stored in the DS_AREA.

18.5.3.2 Offcore Response Event

Event number 0B7H support offcore response monitoring using an associated configuration MSR, MSR_OFFCORE_RSP0 (address 1A6H) in conjunction with umask value 01H or MSR_OFFCORE_RSP1 (address 1A7H) in conjunction with umask value 02H. Table 18-51 lists the event code, mask value and additional off-core configuration MSR that must be programmed to count off-core response events using IA32_PMCx.

The Goldmont microarchitecture provides unique pairs of MSR_OFFCORE_RSPx registers per core.

The layout of MSR_OFFCORE_RSP0 and MSR_OFFCORE_RSP1 are organized as follows:

- Bits 15:0 specifies the request type of a transaction request to the uncore. This is described in Table 18-58.
- Bits 30:16 specifies common supplier information or an L2 Hit, and is described in Table 18-53.
- If L2 misses, then Bits 37:31 can be used to specify snoop response information and is described in Table 18-59.
- For outstanding requests, bit 38 can enable measurement of average latency of specific type of offcore transaction requests using two programmable counter simultaneously; see Section 18.5.2.3 for details.

Table 18-58. MSR_OFFCORE_RSPx Request_Type Field Definition

Bit Name	Offset	Description
DEMAND_DATA_RD	0	(R/W) Counts cacheline read requests due to demand reads (excludes prefetches).
DEMAND_RFO	1	(R/W) Counts cacheline read for ownership (RFO) requests due to demand writes (excludes prefetches).
DEMAND_CODE_RD	2	(R/W) Counts demand instruction cacheline and I-side prefetch requests that miss the instruction cache.
COREWB	3	(R/W) Counts writeback transactions caused by L1 or L2 cache evictions.
PF_L2_DATA_RD	4	(R/W) Counts data cacheline reads generated by hardware L2 cache prefetcher.
PF_L2_RFO	5	(R/W) Counts reads for ownership (RFO) requests generated by L2 prefetcher.
Reserved	6	Reserved.
PARTIAL_READS	7	(R/W) Counts demand data partial reads, including data in uncacheable (UC) or uncacheable (WC) write combining memory types.
PARTIAL_WRITES	8	(R/W) Counts partial writes, including uncacheable (UC), write through (WT) and write protected (WP) memory type writes.
UC_CODE_READS	9	(R/W) Counts code reads in uncacheable (UC) memory region.
BUS_LOCKS	10	(R/W) Counts bus lock and split lock requests.
FULL_STREAMING_STORES	11	(R/W) Counts full cacheline writes due to streaming stores.
SW_PREFETCH	12	(R/W) Counts cacheline requests due to software prefetch instructions.
PF_L1_DATA_RD	13	(R/W) Counts data cacheline reads generated by hardware L1 data cache prefetcher.
PARTIAL_STREAMING_STORES	14	(R/W) Counts partial cacheline writes due to streaming stores.

Table 18-58. MSR_OFFCORE_RSPx Request_Type Field Definition (Contd.)

Bit Name	Offset	Description
ANY_REQUEST	15	(R/W) Counts requests to the uncore subsystem.

To properly program this extra register, software must set at least one request type bit (Table 18-52) and a valid response type pattern (either Table 18-53 or Table 18-59). Otherwise, the event count reported will be zero. It is permissible and useful to set multiple request and response type bits in order to obtain various classes of off-core response events. Although MSR_OFFCORE_RSPx allow an agent software to program numerous combinations that meet the above guideline, not all combinations produce meaningful data.

Table 18-59. MSR_OFFCORE_RSPx For L2 Miss and Outstanding Requests

Subtype	Bit Name	Offset	Description
L2_MISS (Snoop Info)	Reserved	32:31	Reserved
	L2_MISS.SNOOP_MISS_OR_NO_SNOOP_NEEDED	33	(R/W). A true miss to this module, for which a snoop request missed the other module or no snoop was performed/needed.
	L2_MISS.HIT_OTHER_CORE_NO_FWD	34	(R/W) A snoop hit in the other processor module, but no data forwarding is required.
	Reserved	35	Reserved
	L2_MISS.HITM_OTHER_CORE	36	(R/W) Counts the number of snoops hit in the other module or other core's L1 where modified copies were found.
	L2_MISS.NON_DRAM	37	(R/W) Target was a non-DRAM system address. This includes MMIO transactions.
Outstanding requests[1]	OUTSTANDING	38	(R/W) Counts weighted cycles of outstanding offcore requests of the request type specified in bits 15:0, from the time the XQ receives the request and any response is received. Bits 37:16 must be set to 0. This bit is only available in MSR_OFFCORE_RESP0.

NOTES:

1. See Section 18.5.2.3, "Average Offcore Request Latency Measurement" for details on how to use this bit to extract average latency.

To specify a complete offcore response filter, software must properly program bits in the request and response type fields. A valid request type must have at least one bit set in the non-reserved bits of 15:0. A valid response type must be a non-zero value of the following expression:

[ANY 'OR' (L2 Hit)] 'XOR' (Snoop Info Bits) 'XOR' (Avg Latency)

18.5.3.3 Average Offcore Request Latency Measurement

In Goldmont microarchitecture, measurement of average latency of offcore transaction requests is the same as described in Section 18.5.2.3.

18.5.4 Performance Monitoring for Goldmont Plus Microarchitecture

Intel Atom processors based on the Goldmont Plus microarchitecture report architectural performance monitoring versionID = 4 and support non-architectural monitoring capabilities described in this section.

Architectural performance monitoring version 4 capabilities are described in Section 18.2.4.

Goldmont Plus performance monitoring capabilities are similar to Goldmont capabilities. The differences are in specific events and in which counters support PEBS. Goldmont Plus introduces the ability for fixed performance monitoring counters to generate PEBS records.

Goldmont Plus will set the AnyThread deprecation CPUID bit (CPUID.0AH:EDX[15]) to indicate that the Any-Thread bits in IA32_PERFEVTSELx and IA32_FIXED_CTR_CTRL have no effect.

The core PMU's capability is similar to that of the Goldmont microarchitecture described in Section 18.6.3, with some differences and enhancements summarized in Table 18-60.

Table 18-60. Core PMU Comparison Between the Goldmont Plus and Goldmont Microarchitectures

Box	Goldmont Plus Microarchitecture	Goldmont Microarchitecture	Comment
# of Fixed counters per core	3	3	No change.
# of general-purpose counters per core	4	4	No change.
Counter width (R,W)	R:48, W: 32/48	R:48, W: 32/48	No change.
Architectural Performance Monitoring version ID	4	4	No change.
Processor Event Based Sampling (PEBS) Events	All General-Purpose and Fixed counters. Each General-Purpose counter supports all events (precise and non-precise).	General-Purpose Counter 0 only. Supports all events (precise and non-precise). Precise events are listed in Table 18-56.	Goldmont Plus supports PEBS on all counters.
PEBS record format encoding	0011b	0011b	No change.

18.5.4.1 Extended PEBS

The Extended PEBS feature, introduced in Goldmont Plus microarchitecture, supports PEBS (Processor Event Based Sampling) on a fixed-function performance counters as well as all four general purpose counters (PMC0-3). PEBS can be enabled for the four general purpose counters using PEBS_EN_PMCi bits of IA32_PEBS_ENABLE (i = 0, 1, 2, 3). PEBS can be enabled for the 3 fixed function counters using the PEBS_EN_FIXEDi bits of IA32_PEBS_ENABLE (I = 0, 1, 2).

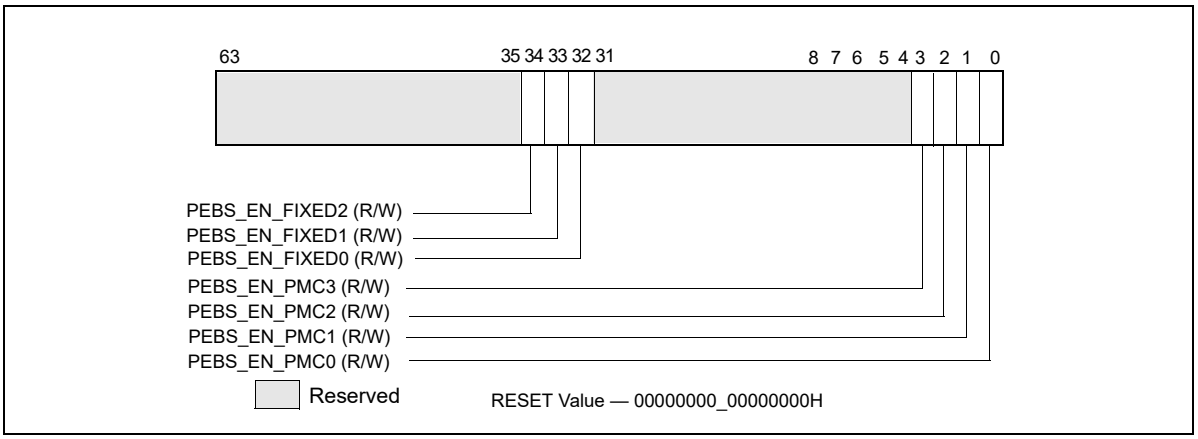


Figure 18-39. Layout of IA32_PEBS_ENABLE MSR

Similar to Goldmont microarchitecture, Goldmont Plus microarchitecture processors can generate PEBS record events on both precise as well as non-precise events.

A PEBS record due to a precise event will be generated after an instruction that causes the event when the counter has already overflowed. A PEBS record due to a non-precise event will occur at the next opportunity after the counter has overflowed, including immediately after an overflow is set by an MSR write.

IA32_FIXED_CTR0 counts instructions retired and is a precise event. IA32_FIXED_CTR1 counts unhalted core cycles and is a non-precise event. IA32_FIXED_CTR2 counts unhalted reference cycles and is a non-precise event.

The Applicable Counter field at offset 90H of the PEBS record indicates which counters caused the PEBS record to be generated. It is in the same format as the enable bits for each counter in IA32_PEBS_ENABLE. As an example, an Applicable Counter field with bits 2 and 32 set would indicate that both general purpose counter 2 and fixed function counter 0 generated the PEBS record.

- To properly use PEBS for the additional counters, software will need to set up the counter reset values in PEBS portion of the DS_BUFFER_MANAGEMENT_AREA data structure that is indicated by the IA32_DS_AREA register. The layout of the DS_BUFFER_MANAGEMENT_AREA for Goldmont Plus is shown in Figure 18-40. When a counter generates a PEBS records, the appropriate counter reset values will be loaded into that counter. In the above example where general purpose counter 2 and fixed function counter 0 generated the PEBS record, general purpose counter 2 would be reloaded with the value contained in PEBS GP Counter 2 Reset (offset 50H) and fixed function counter 0 would be reloaded with the value contained in PEBS Fixed Counter 0 Reset (offset 80H).

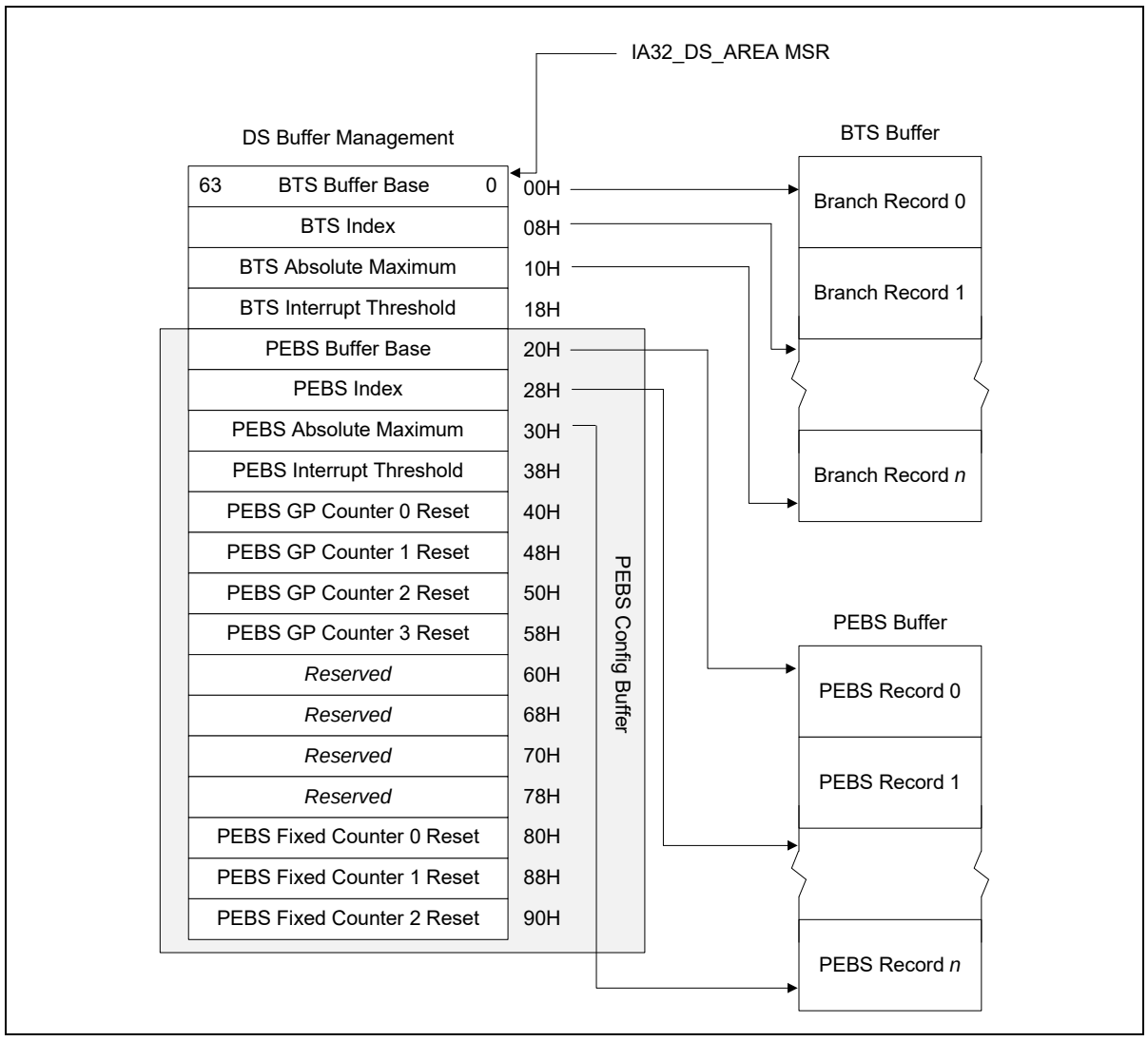


Figure 18-40. PEBS Programming Environment

18.5.4.2 Reduced Skid PEBS

Goldmont Plus microarchitecture processors supports the Reduced Skid PEBS feature described in Section 18.5.3.1.2 on the IA32_PMC0 counter. Although Goldmont Plus adds support for generating PEBS records for precise events on the other general-purpose and fixed-function performance counters, those counters do not support the Reduced Skid PEBS feature.

18.6 PERFORMANCE MONITORING (LEGACY INTEL PROCESSORS)

18.6.1 Performance Monitoring (Intel® Core™ Solo and Intel® Core™ Duo Processors)

In Intel Core Solo and Intel Core Duo processors, non-architectural performance monitoring events are programmed using the same facilities (see Figure 18-1) used for architectural performance events.

Non-architectural performance events use event select values that are model-specific. Event mask (Umask) values are also specific to event logic units. Some microarchitectural conditions detectable by a Umask value may have specificity related to processor topology (see Section 8.6, "Detecting Hardware Multi-Threading Support and Topology," in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*). As a result, the unit mask field (for example, IA32_PERFEVTSELx[bits 15:8]) may contain sub-fields that specify topology information of processor cores.

The sub-field layout within the Umask field may support two-bit encoding that qualifies the relationship between a microarchitectural condition and the originating core. This data is shown in Table 18-61. The two-bit encoding for core-specificity is only supported for a subset of Umask values (see Chapter 19, "Performance Monitoring Events") and for Intel Core Duo processors. Such events are referred to as core-specific events.

Table 18-61. Core Specificity Encoding within a Non-Architectural Umask

IA32_PERFEVTSELx MSRs	
Bit 15:14 Encoding	**Description**
11B	All cores
10B	Reserved
01B	This core
00B	Reserved

Some microarchitectural conditions allow detection specificity only at the boundary of physical processors. Some bus events belong to this category, providing specificity between the originating physical processor (a bus agent) versus other agents on the bus. Sub-field encoding for agent specificity is shown in Table 18-62.

Table 18-62. Agent Specificity Encoding within a Non-Architectural Umask

IA32_PERFEVTSELx MSRs	
Bit 13 Encoding	**Description**
0	This agent
1	Include all agents

Some microarchitectural conditions are detectable only from the originating core. In such cases, unit mask does not support core-specificity or agent-specificity encodings. These are referred to as core-only conditions.

Some microarchitectural conditions allow detection specificity that includes or excludes the action of hardware prefetches. A two-bit encoding may be supported to qualify hardware prefetch actions. Typically, this applies only to some L2 or bus events. The sub-field encoding for hardware prefetch qualification is shown in Table 18-63.

Table 18-63. HW Prefetch Qualification Encoding within a Non-Architectural Umask

IA32_PERFEVTSELx MSRs	
Bit 13:12 Encoding	**Description**
11B	All inclusive
10B	Reserved
01B	Hardware prefetch only
00B	Exclude hardware prefetch

Some performance events may (a) support none of the three event-specific qualification encodings (b) may support core-specificity and agent specificity simultaneously (c) or may support core-specificity and hardware prefetch qualification simultaneously. Agent-specificity and hardware prefetch qualification are mutually exclusive.

In addition, some L2 events permit qualifications that distinguish cache coherent states. The sub-field definition for cache coherency state qualification is shown in Table 18-64. If no bits in the MESI qualification sub-field are set for an event that requires setting MESI qualification bits, the event count will not increment.

Table 18-64. MESI Qualification Definitions within a Non-Architectural Umask

IA32_PERFEVTSELx MSRs	
Bit Position 11:8	**Description**
Bit 11	Counts modified state
Bit 10	Counts exclusive state
Bit 9	Counts shared state
Bit 8	Counts Invalid state

18.6.2 Performance Monitoring (Processors Based on Intel® Core™ Microarchitecture)

In addition to architectural performance monitoring, processors based on the Intel Core microarchitecture support non-architectural performance monitoring events.

Architectural performance events can be collected using general-purpose performance counters. Non-architectural performance events can be collected using general-purpose performance counters (coupled with two IA32_PERFEVTSELx MSRs for detailed event configurations), or fixed-function performance counters (see Section 18.6.2.1). IA32_PERFEVTSELx MSRs are architectural; their layout is shown in Figure 18-1. Starting with Intel Core 2 processor T 7700, fixed-function performance counters and associated counter control and status MSR becomes part of architectural performance monitoring version 2 facilities (see also Section 18.2.2).

Non-architectural performance events in processors based on Intel Core microarchitecture use event select values that are model-specific. Valid event mask (Umask) bits are listed in Chapter 19. The UMASK field may contain sub-fields identical to those listed in Table 18-61, Table 18-62, Table 18-63, and Table 18-64. One or more of these sub-fields may apply to specific events on an event-by-event basis. Details are listed in Table 19-25 in Chapter 19, "Performance-Monitoring Events."

In addition, the UMASK filed may also contain a sub-field that allows detection specificity related to snoop responses. Bits of the snoop response qualification sub-field are defined in Table 18-65.

Table 18-65. Bus Snoop Qualification Definitions within a Non-Architectural Umask

IA32_PERFEVTSELx MSRs	
Bit Position 11:8	**Description**
Bit 11	HITM response
Bit 10	Reserved

Table 18-65. Bus Snoop Qualification Definitions within a Non-Architectural Umask

IA32_PERFEVTSELx MSRs	
Bit Position 11:8	**Description**
Bit 9	HIT response
Bit 8	CLEAN response

There are also non-architectural events that support qualification of different types of snoop operation. The corresponding bit field for snoop type qualification are listed in Table 18-66.

Table 18-66. Snoop Type Qualification Definitions within a Non-Architectural Umask

IA32_PERFEVTSELx MSRs	
Bit Position 9:8	**Description**
Bit 9	CMP2I snoops
Bit 8	CMP2S snoops

No more than one sub-field of MESI, snoop response, and snoop type qualification sub-fields can be supported in a performance event.

NOTE

Software must write known values to the performance counters prior to enabling the counters. The content of general-purpose counters and fixed-function counters are undefined after INIT or RESET.

18.6.2.1 Fixed-function Performance Counters

Processors based on Intel Core microarchitecture provide three fixed-function performance counters. Bits beyond the width of the fixed counter are reserved and must be written as zeros. Model-specific fixed-function performance counters on processors that support Architectural Perfmon version 1 are 40 bits wide.

Each of the fixed-function counter is dedicated to count a pre-defined performance monitoring events. See Table 18-2 for details of the PMC addresses and what these events count.

Programming the fixed-function performance counters does not involve any of the IA32_PERFEVTSELx MSRs, and does not require specifying any event masks. Instead, the MSR MSR_PERF_FIXED_CTR_CTRL provides multiple sets of 4-bit fields; each 4-bit field controls the operation of a fixed-function performance counter (PMC). See Figures 18-41. Two sub-fields are defined for each control. See Figure 18-41; bit fields are:

- **Enable field (low 2 bits in each 4-bit control) —** When bit 0 is set, performance counting is enabled in the corresponding fixed-function performance counter to increment when the target condition associated with the architecture performance event occurs at ring 0.

  When bit 1 is set, performance counting is enabled in the corresponding fixed-function performance counter to increment when the target condition associated with the architecture performance event occurs at ring greater than 0.

  Writing 0 to both bits stops the performance counter. Writing 11B causes the counter to increment irrespective of privilege levels.

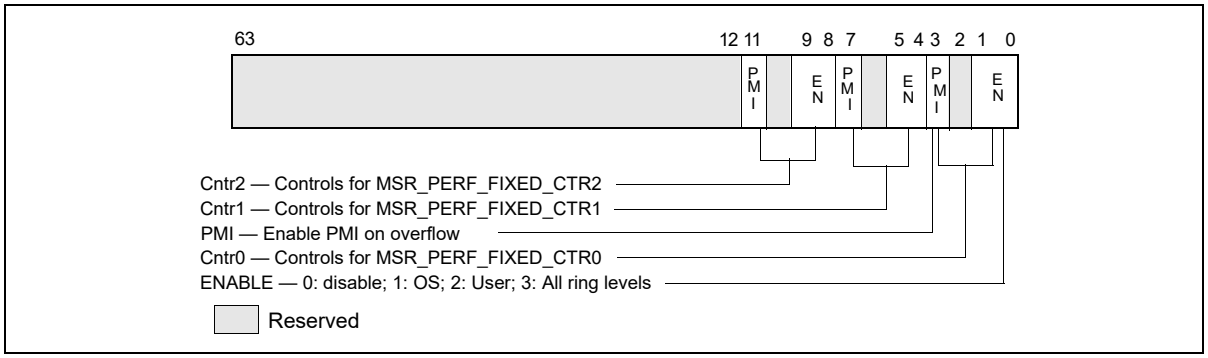


Figure 18-41. Layout of MSR_PERF_FIXED_CTR_CTRL MSR

- **PMI field (fourth bit in each 4-bit control)** — When set, the logical processor generates an exception through its local APIC on overflow condition of the respective fixed-function counter.

18.6.2.2 Global Counter Control Facilities

Processors based on Intel Core microarchitecture provides simplified performance counter control that simplifies the most frequent operations in programming performance events, i.e. enabling/disabling event counting and checking the status of counter overflows. This is done by the following three MSRs:

- MSR_PERF_GLOBAL_CTRL enables/disables event counting for all or any combination of fixed-function PMCs (MSR_PERF_FIXED_CTRx) or general-purpose PMCs via a single WRMSR.
- MSR_PERF_GLOBAL_STATUS allows software to query counter overflow conditions on any combination of fixed-function PMCs (MSR_PERF_FIXED_CTRx) or general-purpose PMCs via a single RDMSR.
- MSR_PERF_GLOBAL_OVF_CTRL allows software to clear counter overflow conditions on any combination of fixed-function PMCs (MSR_PERF_FIXED_CTRx) or general-purpose PMCs via a single WRMSR.

MSR_PERF_GLOBAL_CTRL MSR provides single-bit controls to enable counting in each performance counter (see Figure 18-42). Each enable bit in MSR_PERF_GLOBAL_CTRL is AND'ed with the enable bits for all privilege levels in the respective IA32_PERFEVTSELx or MSR_PERF_FIXED_CTR_CTRL MSRs to start/stop the counting of respective counters. Counting is enabled if the AND'ed results is true; counting is disabled when the result is false.

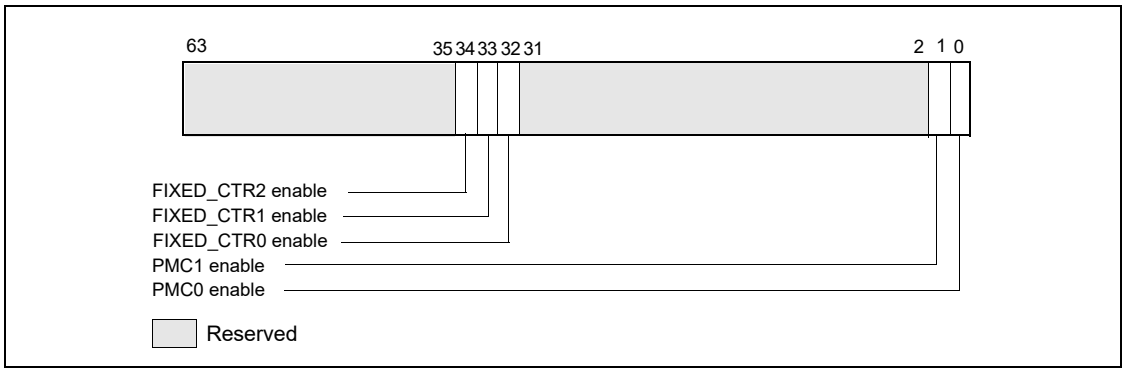


Figure 18-42. Layout of MSR_PERF_GLOBAL_CTRL MSR

MSR_PERF_GLOBAL_STATUS MSR provides single-bit status used by software to query the overflow condition of each performance counter. MSR_PERF_GLOBAL_STATUS[bit 62] indicates overflow conditions of the DS area data buffer. MSR_PERF_GLOBAL_STATUS[bit 63] provides a CondChgd bit to indicate changes to the state of performance monitoring hardware (see Figure 18-43). A value of 1 in bits 34:32, 1, 0 indicates an overflow condition has

occurred in the associated counter.

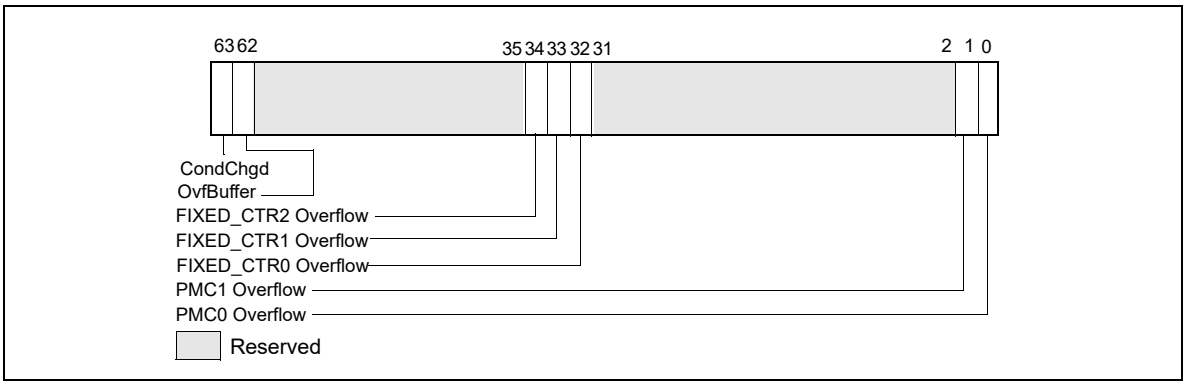


Figure 18-43. Layout of MSR_PERF_GLOBAL_STATUS MSR

When a performance counter is configured for PEBS, an overflow condition in the counter will arm PEBS. On the subsequent event following overflow, the processor will generate a PEBS event. On a PEBS event, the processor will perform bounds checks based on the parameters defined in the DS Save Area (see Section 17.4.9). Upon successful bounds checks, the processor will store the data record in the defined buffer area, clear the counter overflow status, and reload the counter. If the bounds checks fail, the PEBS will be skipped entirely. In the event that the PEBS buffer fills up, the processor will set the OvfBuffer bit in MSR_PERF_GLOBAL_STATUS.

MSR_PERF_GLOBAL_OVF_CTL MSR allows software to clear overflow the indicators for general-purpose or fixed-function counters via a single WRMSR (see Figure 18-44). Clear overflow indications when:

- Setting up new values in the event select and/or UMASK field for counting or interrupt-based event sampling.
- Reloading counter values to continue collecting next sample.
- Disabling event counting or interrupt-based event sampling.

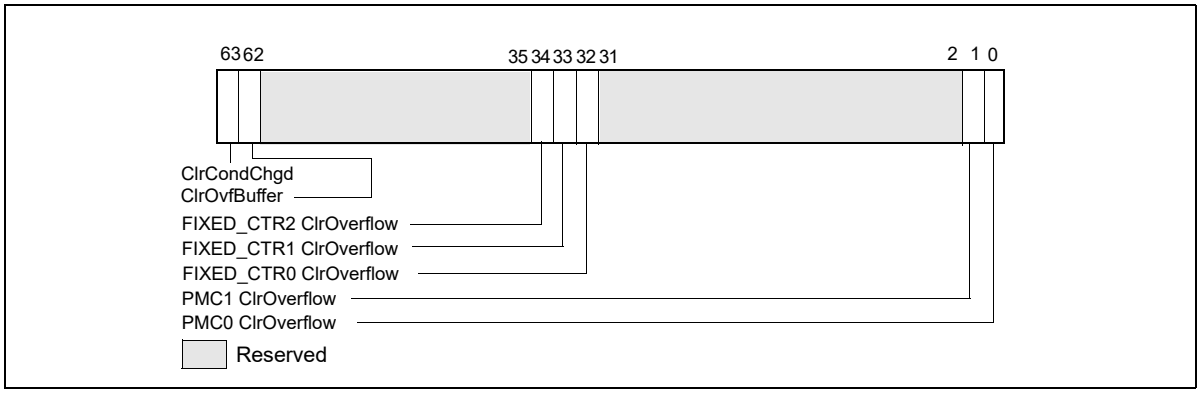


Figure 18-44. Layout of MSR_PERF_GLOBAL_OVF_CTRL MSR

18.6.2.3 At-Retirement Events

Many non-architectural performance events are impacted by the speculative nature of out-of-order execution. A subset of non-architectural performance events on processors based on Intel Core microarchitecture are enhanced with a tagging mechanism (similar to that found in Intel NetBurst® microarchitecture) that exclude contributions that arise from speculative execution. The at-retirement events available in processors based on Intel Core micro-architecture does not require special MSR programming control (see Section 18.6.3.6, "At-Retirement Counting"), but is limited to IA32_PMC0. See Table 18-67 for a list of events available to processors based on Intel Core micro-architecture.

Table 18-67. At-Retirement Performance Events for Intel Core Microarchitecture

Event Name	UMask	Event Select
ITLB_MISS_RETIRED	00H	C9H
MEM_LOAD_RETIRED.L1D_MISS	01H	CBH
MEM_LOAD_RETIRED.L1D_LINE_MISS	02H	CBH
MEM_LOAD_RETIRED.L2_MISS	04H	CBH
MEM_LOAD_RETIRED.L2_LINE_MISS	08H	CBH
MEM_LOAD_RETIRED.DTLB_MISS	10H	CBH

18.6.2.4 Processor Event Based Sampling (PEBS)

Processors based on Intel Core microarchitecture also support processor event based sampling (PEBS). This feature was introduced by processors based on Intel NetBurst microarchitecture.

PEBS uses a debug store mechanism and a performance monitoring interrupt to store a set of architectural state information for the processor. The information provides architectural state of the instruction executed after the instruction that caused the event (See Section 18.6.2.4.2 and Section 17.4.9).

In cases where the same instruction causes BTS and PEBS to be activated, PEBS is processed before BTS are processed. The PMI request is held until the processor completes processing of PEBS and BTS.

For processors based on Intel Core microarchitecture, precise events that can be used with PEBS are listed in Table 18-68. The procedure for detecting availability of PEBS is the same as described in Section 18.6.3.8.1.

Table 18-68. PEBS Performance Events for Intel Core Microarchitecture

Event Name	UMask	Event Select
INSTR_RETIRED.ANY_P	00H	C0H
X87_OPS_RETIRED.ANY	FEH	C1H
BR_INST_RETIRED.MISPRED	00H	C5H
SIMD_INST_RETIRED.ANY	1FH	C7H
MEM_LOAD_RETIRED.L1D_MISS	01H	CBH
MEM_LOAD_RETIRED.L1D_LINE_MISS	02H	CBH
MEM_LOAD_RETIRED.L2_MISS	04H	CBH
MEM_LOAD_RETIRED.L2_LINE_MISS	08H	CBH
MEM_LOAD_RETIRED.DTLB_MISS	10H	CBH

18.6.2.4.1 Setting up the PEBS Buffer

For processors based on Intel Core microarchitecture, PEBS is available using IA32_PMC0 only. Use the following procedure to set up the processor and IA32_PMC0 counter for PEBS:

1. Set up the precise event buffering facilities. Place values in the precise event buffer base, precise event index, precise event absolute maximum, precise event interrupt threshold, and precise event counter reset fields of the DS buffer management area. In processors based on Intel Core microarchitecture, PEBS records consist of 64-bit address entries. See Figure 17-8 to set up the precise event records buffer in memory.
2. Enable PEBS. Set the Enable PEBS on PMC0 flag (bit 0) in IA32_PEBS_ENABLE MSR.
3. Set up the IA32_PMC0 performance counter and IA32_PERFEVTSEL0 for an event listed in Table 18-68.

18.6.2.4.2 PEBS Record Format

The PEBS record format may be extended across different processor implementations. The IA32_PERF_CAPABILITES MSR defines a mechanism for software to handle the evolution of PEBS record format in processors that support architectural performance monitoring with version id equals 2 or higher. The bit fields of

IA32_PERF_CAPABILITES are defined in Table 2-2 of Chapter 2, "Model-Specific Registers (MSRs)" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 4*. The relevant bit fields that governs PEBS are:

- PEBSTrap [bit 6]: When set, PEBS recording is trap-like. After the PEBS-enabled counter has overflowed, PEBS record is recorded for the next PEBS-able event at the completion of the sampled instruction causing the PEBS event. When clear, PEBS recording is fault-like. The PEBS record is recorded before the sampled instruction causing the PEBS event.
- PEBSSaveArchRegs [bit 7]: When set, PEBS will save architectural register and state information according to the encoded value of the PEBSRecordFormat field. When clear, only the return instruction pointer and flags are recorded. On processors based on Intel Core microarchitecture, this bit is always 1
- PEBSRecordFormat [bits 11:8]: Valid encodings are:
  - 0000B: Only general-purpose registers, instruction pointer and RFLAGS registers are saved in each PEBS record (seeSection 18.6.3.8).
  - 0001B: PEBS record includes additional information of IA32_PERF_GLOBAL_STATUS and load latency data. (seeSection 18.3.1.1.1).
  - 0010B: PEBS record includes additional information of IA32_PERF_GLOBAL_STATUS, load latency data, and TSX tuning information. (seeSection 18.3.6.2).
  - 0011B: PEBS record includes additional information of load latency data, TSX tuning information, TSC data, and the applicable counter field replaces IA32_PERF_GLOBAL_STATUS at offset 90H. (see Section 18.3.8.1.1).

18.6.2.4.3 Writing a PEBS Interrupt Service Routine

The PEBS facilities share the same interrupt vector and interrupt service routine (called the DS ISR) with the Interrupt-based event sampling and BTS facilities. To handle PEBS interrupts, PEBS handler code must be included in the DS ISR. See Section 17.4.9.1, "64 Bit Format of the DS Save Area," for guidelines when writing the DS ISR.

The service routine can query MSR_PERF_GLOBAL_STATUS to determine which counter(s) caused of overflow condition. The service routine should clear overflow indicator by writing to MSR_PERF_GLOBAL_OVF_CTL.

A comparison of the sequence of requirements to program PEBS for processors based on Intel Core and Intel NetBurst microarchitectures is listed in Table 18-69.

Table 18-69. Requirements to Program PEBS

	For Processors based on Intel Core microarchitecture	For Processors based on Intel NetBurst microarchitecture
Verify PEBS support of processor/OS.	▪ IA32_MISC_ENABLE.EMON_AVAILABE (bit 7) is set. ▪ IA32_MISC_ENABLE.PEBS_UNAVAILABE (bit 12) is clear.	
Ensure counters are in disabled.	On initial set up or changing event configurations, write MSR_PERF_GLOBAL_CTRL MSR (38FH) with 0. On subsequent entries: ▪ Clear all counters if "Counter Freeze on PMI" is not enabled. ▪ If IA32_DebugCTL.Freeze is enabled, counters are automatically disabled. Counters MUST be stopped before writing.[1]	Optional
Disable PEBS.	Clear ENABLE PMC0 bit in IA32_PEBS_ENABLE MSR (3F1H).	Optional
Check overflow conditions.	Check MSR_PERF_GLOBAL_STATUS MSR (38EH) handle any overflow conditions.	Check OVF flag of each CCCR for overflow condition
Clear overflow status.	Clear MSR_PERF_GLOBAL_STATUS MSR (38EH) using IA32_PERF_GLOBAL_OVF_CTRL MSR (390H).	Clear OVF flag of each CCCR.
Write "sample-after" values.	Configure the counter(s) with the sample after value.	

Table 18-69. Requirements to Program PEBS (Contd.)

	For Processors based on Intel Core microarchitecture	For Processors based on Intel NetBurst microarchitecture
Configure specific counter configuration MSR.	▪ Set local enable bit 22 - 1. ▪ Do NOT set local counter PMI/INT bit, bit 20 - 0. ▪ Event programmed must be PEBS capable.	▪ Set appropriate OVF_PMI bits - 1. ▪ Only CCCR for MSR_IQ_COUNTER4 support PEBS.
Allocate buffer for PEBS states.	Allocate a buffer in memory for the precise information.	
Program the IA32_DS_AREA MSR.	Program the IA32_DS_AREA MSR.	
Configure the PEBS buffer management records.	Configure the PEBS buffer management records in the DS buffer management area.	
Configure/Enable PEBS.	Set Enable PMC0 bit in IA32_PEBS_ENABLE MSR (3F1H).	Configure MSR_PEBS_ENABLE, MSR_PEBS_MATRIX_VERT and MSR_PEBS_MATRIX_HORZ as needed.
Enable counters.	Set Enable bits in MSR_PERF_GLOBAL_CTRL MSR (38FH).	Set each CCCR enable bit 12 - 1.

NOTES:

1. Counters read while enabled are not guaranteed to be precise with event counts that occur in timing proximity to the RDMSR.

18.6.2.4.4 Re-configuring PEBS Facilities

When software needs to reconfigure PEBS facilities, it should allow a quiescent period between stopping the prior event counting and setting up a new PEBS event. The quiescent period is to allow any latent residual PEBS records to complete its capture at their previously specified buffer address (provided by IA32_DS_AREA).

18.6.3 Performance Monitoring (Processors Based on Intel NetBurst® Microarchitecture)

The performance monitoring mechanism provided in processors based on Intel NetBurst microarchitecture is different from that provided in the P6 family and Pentium processors. While the general concept of selecting, filtering, counting, and reading performance events through the WRMSR, RDMSR, and RDPMC instructions is unchanged, the setup mechanism and MSR layouts are incompatible with the P6 family and Pentium processor mechanisms. Also, the RDPMC instruction has been extended to support faster reading of counters and to read all performance counters available in processors based on Intel NetBurst microarchitecture.

The event monitoring mechanism consists of the following facilities:

- The IA32_MISC_ENABLE MSR, which indicates the availability in an Intel 64 or IA-32 processor of the performance monitoring and processor event-based sampling (PEBS) facilities.
- Event selection control (ESCR) MSRs for selecting events to be monitored with specific performance counters. The number available differs by family and model (43 to 45).
- 18 performance counter MSRs for counting events.
- 18 counter configuration control (CCCR) MSRs, with one CCCR associated with each performance counter. CCCRs sets up an associated performance counter for a specific method of counting.
- A debug store (DS) save area in memory for storing PEBS records.
- The IA32_DS_AREA MSR, which establishes the location of the DS save area.
- The debug store (DS) feature flag (bit 21) returned by the CPUID instruction, which indicates the availability of the DS mechanism.
- The MSR_PEBS_ENABLE MSR, which enables the PEBS facilities and replay tagging used in at-retirement event counting.
- A set of predefined events and event metrics that simplify the setting up of the performance counters to count specific events.

Table 18-70 lists the performance counters and their associated CCCRs, along with the ESCRs that select events to be counted for each performance counter. Predefined event metrics and events are listed in Chapter 19, "Perfor-

mance-Monitoring Events."

Table 18-70. Performance Counter MSRs and Associated CCCR and ESCR MSRs (Processors Based on Intel NetBurst Microarchitecture)

Counter			CCCR		ESCR		
Name	**No.**	**Addr**	**Name**	**Addr**	**Name**	**No.**	**Addr**
MSR_BPU_COUNTER0	0	300H	MSR_BPU_CCCR0	360H	MSR_BSU_ESCR0 MSR_FSB_ESCR0 MSR_MOB_ESCR0 MSR_PMH_ESCR0 MSR_BPU_ESCR0 MSR_IS_ESCR0 MSR_ITLB_ESCR0 MSR_IX_ESCR0	7 6 2 4 0 1 3 5	3A0H 3A2H 3AAH 3ACH 3B2H 3B4H 3B6H 3C8H
MSR_BPU_COUNTER1	1	301H	MSR_BPU_CCCR1	361H	MSR_BSU_ESCR0 MSR_FSB_ESCR0 MSR_MOB_ESCR0 MSR_PMH_ESCR0 MSR_BPU_ESCR0 MSR_IS_ESCR0 MSR_ITLB_ESCR0 MSR_IX_ESCR0	7 6 2 4 0 1 3 5	3A0H 3A2H 3AAH 3ACH 3B2H 3B4H 3B6H 3C8H
MSR_BPU_COUNTER2	2	302H	MSR_BPU_CCCR2	362H	MSR_BSU_ESCR1 MSR_FSB_ESCR1 MSR_MOB_ESCR1 MSR_PMH_ESCR1 MSR_BPU_ESCR1 MSR_IS_ESCR1 MSR_ITLB_ESCR1 MSR_IX_ESCR1	7 6 2 4 0 1 3 5	3A1H 3A3H 3ABH 3ADH 3B3H 3B5H 3B7H 3C9H
MSR_BPU_COUNTER3	3	303H	MSR_BPU_CCCR3	363H	MSR_BSU_ESCR1 MSR_FSB_ESCR1 MSR_MOB_ESCR1 MSR_PMH_ESCR1 MSR_BPU_ESCR1 MSR_IS_ESCR1 MSR_ITLB_ESCR1 MSR_IX_ESCR1	7 6 2 4 0 1 3 5	3A1H 3A3H 3ABH 3ADH 3B3H 3B5H 3B7H 3C9H
MSR_MS_COUNTER0	4	304H	MSR_MS_CCCR0	364H	MSR_MS_ESCR0 MSR_TBPU_ESCR0 MSR_TC_ESCR0	0 2 1	3C0H 3C2H 3C4H
MSR_MS_COUNTER1	5	305H	MSR_MS_CCCR1	365H	MSR_MS_ESCR0 MSR_TBPU_ESCR0 MSR_TC_ESCR0	0 2 1	3C0H 3C2H 3C4H
MSR_MS_COUNTER2	6	306H	MSR_MS_CCCR2	366H	MSR_MS_ESCR1 MSR_TBPU_ESCR1 MSR_TC_ESCR1	0 2 1	3C1H 3C3H 3C5H
MSR_MS_COUNTER3	7	307H	MSR_MS_CCCR3	367H	MSR_MS_ESCR1 MSR_TBPU_ESCR1 MSR_TC_ESCR1	0 2 1	3C1H 3C3H 3C5H
MSR_FLAME_COUNTER0	8	308H	MSR_FLAME_CCCR0	368H	MSR_FIRM_ESCR0 MSR_FLAME_ESCR0 MSR_DAC_ESCR0 MSR_SAAT_ESCR0 MSR_U2L_ESCR0	1 0 5 2 3	3A4H 3A6H 3A8H 3AEH 3B0H

Table 18-70. Performance Counter MSRs and Associated CCCR and ESCR MSRs (Processors Based on Intel NetBurst Microarchitecture) (Contd.)

Counter			CCCR		ESCR		
Name	**No.**	**Addr**	**Name**	**Addr**	**Name**	**No.**	**Addr**
MSR_FLAME_COUNTER1	9	309H	MSR_FLAME_CCCR1	369H	MSR_FIRM_ESCR0 MSR_FLAME_ESCR0 MSR_DAC_ESCR0 MSR_SAAT_ESCR0 MSR_U2L_ESCR0	1 0 5 2 3	3A4H 3A6H 3A8H 3AEH 3B0H
MSR_FLAME_COUNTER2	10	30AH	MSR_FLAME_CCCR2	36AH	MSR_FIRM_ESCR1 MSR_FLAME_ESCR1 MSR_DAC_ESCR1 MSR_SAAT_ESCR1 MSR_U2L_ESCR1	1 0 5 2 3	3A5H 3A7H 3A9H 3AFH 3B1H
MSR_FLAME_COUNTER3	11	30BH	MSR_FLAME_CCCR3	36BH	MSR_FIRM_ESCR1 MSR_FLAME_ESCR1 MSR_DAC_ESCR1 MSR_SAAT_ESCR1 MSR_U2L_ESCR1	1 0 5 2 3	3A5H 3A7H 3A9H 3AFH 3B1H
MSR_IQ_COUNTER0	12	30CH	MSR_IQ_CCCR0	36CH	MSR_CRU_ESCR0 MSR_CRU_ESCR2 MSR_CRU_ESCR4 MSR_IQ_ESCR0[1] MSR_RAT_ESCR0 MSR_SSU_ESCR0 MSR_ALF_ESCR0	4 5 6 0 2 3 1	3B8H 3CCH 3E0H 3BAH 3BCH 3BEH 3CAH
MSR_IQ_COUNTER1	13	30DH	MSR_IQ_CCCR1	36DH	MSR_CRU_ESCR0 MSR_CRU_ESCR2 MSR_CRU_ESCR4 MSR_IQ_ESCR0[1] MSR_RAT_ESCR0 MSR_SSU_ESCR0 MSR_ALF_ESCR0	4 5 6 0 2 3 1	3B8H 3CCH 3E0H 3BAH 3BCH 3BEH 3CAH
MSR_IQ_COUNTER2	14	30EH	MSR_IQ_CCCR2	36EH	MSR_CRU_ESCR1 MSR_CRU_ESCR3 MSR_CRU_ESCR5 MSR_IQ_ESCR1[1] MSR_RAT_ESCR1 MSR_ALF_ESCR1	4 5 6 0 2 1	3B9H 3CDH 3E1H 3BBH 3BDH 3CBH
MSR_IQ_COUNTER3	15	30FH	MSR_IQ_CCCR3	36FH	MSR_CRU_ESCR1 MSR_CRU_ESCR3 MSR_CRU_ESCR5 MSR_IQ_ESCR1[1] MSR_RAT_ESCR1 MSR_ALF_ESCR1	4 5 6 0 2 1	3B9H 3CDH 3E1H 3BBH 3BDH 3CBH
MSR_IQ_COUNTER4	16	310H	MSR_IQ_CCCR4	370H	MSR_CRU_ESCR0 MSR_CRU_ESCR2 MSR_CRU_ESCR4 MSR_IQ_ESCR0[1] MSR_RAT_ESCR0 MSR_SSU_ESCR0 MSR_ALF_ESCR0	4 5 6 0 2 3 1	3B8H 3CCH 3E0H 3BAH 3BCH 3BEH 3CAH
MSR_IQ_COUNTER5	17	311H	MSR_IQ_CCCR5	371H	MSR_CRU_ESCR1 MSR_CRU_ESCR3 MSR_CRU_ESCR5 MSR_IQ_ESCR1[1] MSR_RAT_ESCR1 MSR_ALF_ESCR1	4 5 6 0 2 1	3B9H 3CDH 3E1H 3BBH 3BDH 3CBH

NOTES:

1. MSR_IQ_ESCR0 and MSR_IQ_ESCR1 are available only on early processor builds (family 0FH, models 01H-02H). These MSRs are not available on later versions.

The types of events that can be counted with these performance monitoring facilities are divided into two classes: non-retirement events and at-retirement events.

- Non-retirement events (see Table 19-30) are events that occur any time during instruction execution (such as bus transactions or cache transactions).
- At-retirement events (see Table 19-31) are events that are counted at the retirement stage of instruction execution, which allows finer granularity in counting events and capturing machine state.

  The at-retirement counting mechanism includes facilities for tagging μops that have encountered a particular performance event during instruction execution. Tagging allows events to be sorted between those that occurred on an execution path that resulted in architectural state being committed at retirement as well as events that occurred on an execution path where the results were eventually cancelled and never committed to architectural state (such as, the execution of a mispredicted branch).

The Pentium 4 and Intel Xeon processor performance monitoring facilities support the three usage models described below. The first two models can be used to count both non-retirement and at-retirement events; the third model is used to count a subset of at-retirement events:

- **Event counting —** A performance counter is configured to count one or more types of events. While the counter is counting, software reads the counter at selected intervals to determine the number of events that have been counted between the intervals.
- **Interrupt-based event sampling —** A performance counter is configured to count one or more types of events and to generate an interrupt when it overflows. To trigger an overflow, the counter is preset to a modulus value that will cause the counter to overflow after a specific number of events have been counted.

  When the counter overflows, the processor generates a performance monitoring interrupt (PMI). The interrupt service routine for the PMI then records the return instruction pointer (RIP), resets the modulus, and restarts the counter. Code performance can be analyzed by examining the distribution of RIPs with a tool like the VTune™ Performance Analyzer.
- **Processor event-based sampling (PEBS) —** In PEBS, the processor writes a record of the architectural state of the processor to a memory buffer after the counter overflows. The records of architectural state provide additional information for use in performance tuning. Processor-based event sampling can be used to count only a subset of at-retirement events. PEBS captures more precise processor state information compared to interrupt based event sampling, because the latter need to use the interrupt service routine to re-construct the architectural states of processor.

The following sections describe the MSRs and data structures used for performance monitoring in the Pentium 4 and Intel Xeon processors.

18.6.3.1 ESCR MSRs

The 45 ESCR MSRs (see Table 18-70) allow software to select specific events to be countered. Each ESCR is usually associated with a pair of performance counters (see Table 18-70) and each performance counter has several ESCRs associated with it (allowing the events counted to be selected from a variety of events).

Figure 18-45 shows the layout of an ESCR MSR. The functions of the flags and fields are:

- **USR flag, bit 2 —** When set, events are counted when the processor is operating at a current privilege level (CPL) of 1, 2, or 3. These privilege levels are generally used by application code and unprotected operating system code.
- **OS flag, bit 3 —** When set, events are counted when the processor is operating at CPL of 0. This privilege level is generally reserved for protected operating system code. (When both the OS and USR flags are set, events are counted at all privilege levels.)

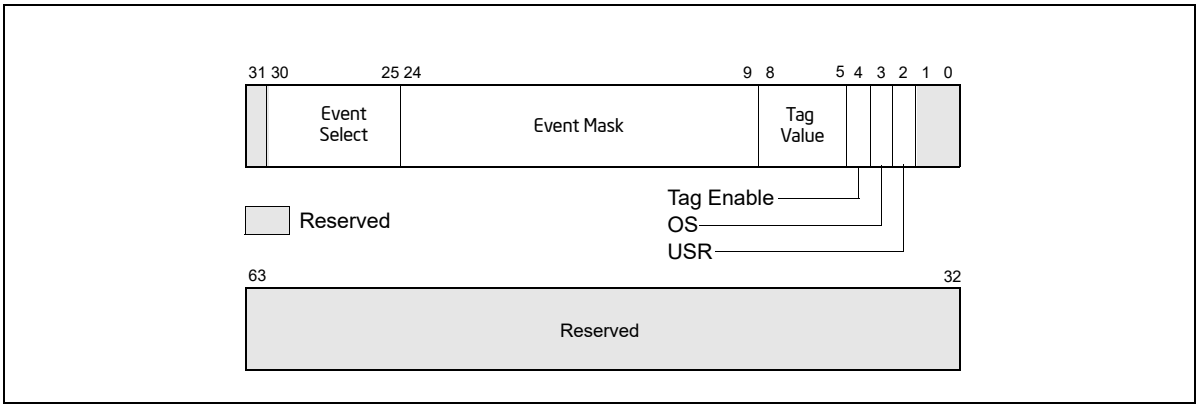


Figure 18-45. Event Selection Control Register (ESCR) for Pentium 4 and Intel Xeon Processors without Intel HT Technology Support

- **Tag enable, bit 4 —** When set, enables tagging of μops to assist in at-retirement event counting; when clear, disables tagging. See Section 18.6.3.6, "At-Retirement Counting."
- **Tag value field, bits 5 through 8 —** Selects a tag value to associate with a μop to assist in at-retirement event counting.
- **Event mask field, bits 9 through 24 —** Selects events to be counted from the event class selected with the event select field.
- **Event select field, bits 25 through 30) —** Selects a class of events to be counted. The events within this class that are counted are selected with the event mask field.

When setting up an ESCR, the event select field is used to select a specific class of events to count, such as retired branches. The event mask field is then used to select one or more of the specific events within the class to be counted. For example, when counting retired branches, four different events can be counted: branch not taken predicted, branch not taken mispredicted, branch taken predicted, and branch taken mispredicted. The OS and USR flags allow counts to be enabled for events that occur when operating system code and/or application code are being executed. If neither the OS nor USR flag is set, no events will be counted.

The ESCRs are initialized to all 0s on reset. The flags and fields of an ESCR are configured by writing to the ESCR using the WRMSR instruction. Table 18-70 gives the addresses of the ESCR MSRs.

Writing to an ESCR MSR does not enable counting with its associated performance counter; it only selects the event or events to be counted. The CCCR for the selected performance counter must also be configured. Configuration of the CCCR includes selecting the ESCR and enabling the counter.

18.6.3.2 Performance Counters

The performance counters in conjunction with the counter configuration control registers (CCCRs) are used for filtering and counting the events selected by the ESCRs. Processors based on Intel NetBurst microarchitecture provide 18 performance counters organized into 9 pairs. A pair of performance counters is associated with a particular subset of events and ESCR's (see Table 18-70). The counter pairs are partitioned into four groups:

- The BPU group, includes two performance counter pairs:
  - MSR_BPU_COUNTER0 and MSR_BPU_COUNTER1.
  - MSR_BPU_COUNTER2 and MSR_BPU_COUNTER3.
- The MS group, includes two performance counter pairs:
  - MSR_MS_COUNTER0 and MSR_MS_COUNTER1.
  - MSR_MS_COUNTER2 and MSR_MS_COUNTER3.
- The FLAME group, includes two performance counter pairs:
  - MSR_FLAME_COUNTER0 and MSR_FLAME_COUNTER1.

  - MSR_FLAME_COUNTER2 and MSR_FLAME_COUNTER3.
- The IQ group, includes three performance counter pairs:
  - MSR_IQ_COUNTER0 and MSR_IQ_COUNTER1.
  - MSR_IQ_COUNTER2 and MSR_IQ_COUNTER3.
  - MSR_IQ_COUNTER4 and MSR_IQ_COUNTER5.

The MSR_IQ_COUNTER4 counter in the IQ group provides support for the PEBS.

Alternate counters in each group can be cascaded: the first counter in one pair can start the first counter in the second pair and vice versa. A similar cascading is possible for the second counters in each pair. For example, within the BPU group of counters, MSR_BPU_COUNTER0 can start MSR_BPU_COUNTER2 and vice versa, and MSR_BPU_COUNTER1 can start MSR_BPU_COUNTER3 and vice versa (see Section 18.6.3.5.6, "Cascading Counters"). The cascade flag in the CCCR register for the performance counter enables the cascading of counters.

Each performance counter is 40-bits wide (see Figure 18-46). The RDPMC instruction is intended to allow reading of either the full counter-width (40-bits) or the low 32-bits of the counter. Reading the low 32-bits is faster than reading the full counter width and is appropriate in situations where the count is small enough to be contained in 32 bits.

The RDPMC instruction can be used by programs or procedures running at any privilege level and in virtual-8086 mode to read these counters. The PCE flag in control register CR4 (bit 8) allows the use of this instruction to be restricted to only programs and procedures running at privilege level 0.

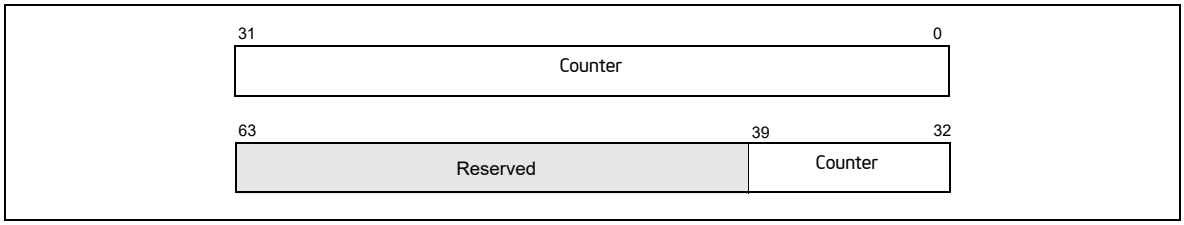


Figure 18-46. Performance Counter (Pentium 4 and Intel Xeon Processors)

The RDPMC instruction is not serializing or ordered with other instructions. Thus, it does not necessarily wait until all previous instructions have been executed before reading the counter. Similarly, subsequent instructions may begin execution before the RDPMC instruction operation is performed.

Only the operating system, executing at privilege level 0, can directly manipulate the performance counters, using the RDMSR and WRMSR instructions. A secure operating system would clear the PCE flag during system initialization to disable direct user access to the performance-monitoring counters, but provide a user-accessible programming interface that emulates the RDPMC instruction.

Some uses of the performance counters require the counters to be preset before counting begins (that is, before the counter is enabled). This can be accomplished by writing to the counter using the WRMSR instruction. To set a counter to a specified number of counts before overflow, enter a 2s complement negative integer in the counter. The counter will then count from the preset value up to -1 and overflow. Writing to a performance counter in a Pentium 4 or Intel Xeon processor with the WRMSR instruction causes all 40 bits of the counter to be written.

18.6.3.3 CCCR MSRs

Each of the 18 performance counters has one CCCR MSR associated with it (see Table 18-70). The CCCRs control the filtering and counting of events as well as interrupt generation. Figure 18-47 shows the layout of an CCCR MSR. The functions of the flags and fields are as follows:

- **Enable flag, bit 12 —** When set, enables counting; when clear, the counter is disabled. This flag is cleared on reset.
- **ESCR select field, bits 13 through 15 —** Identifies the ESCR to be used to select events to be counted with the counter associated with the CCCR.

- **Compare flag, bit 18 —** When set, enables filtering of the event count; when clear, disables filtering. The filtering method is selected with the threshold, complement, and edge flags.
- **Complement flag, bit 19 —** Selects how the incoming event count is compared with the threshold value. When set, event counts that are less than or equal to the threshold value result in a single count being delivered to the performance counter; when clear, counts greater than the threshold value result in a count being delivered to the performance counter (see Section 18.6.3.5.2, "Filtering Events"). The complement flag is not active unless the compare flag is set.
- **Threshold field, bits 20 through 23 —** Selects the threshold value to be used for comparisons. The processor examines this field only when the compare flag is set, and uses the complement flag setting to determine the type of threshold comparison to be made. The useful range of values that can be entered in this field depend on the type of event being counted (see Section 18.6.3.5.2, "Filtering Events").
- **Edge flag, bit 24 —** When set, enables rising edge (false-to-true) edge detection of the threshold comparison output for filtering event counts; when clear, rising edge detection is disabled. This flag is active only when the compare flag is set.

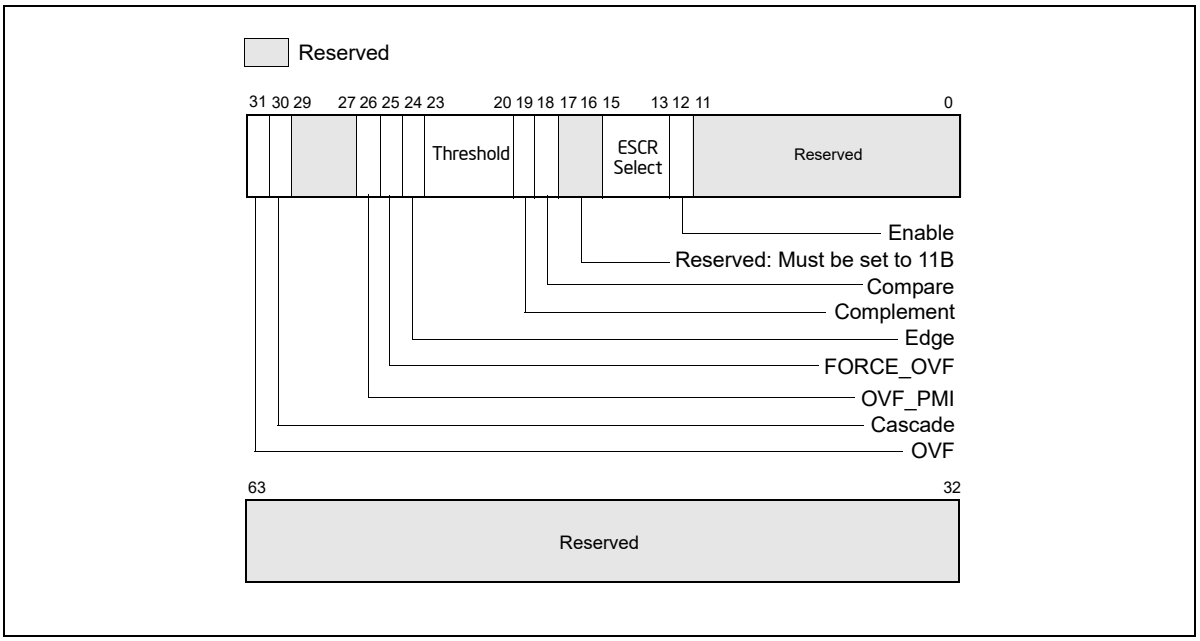


Figure 18-47. Counter Configuration Control Register (CCCR)

- **FORCE_OVF flag, bit 25 —** When set, forces a counter overflow on every counter increment; when clear, overflow only occurs when the counter actually overflows.
- **OVF_PMI flag, bit 26 —** When set, causes a performance monitor interrupt (PMI) to be generated when the counter overflows occurs; when clear, disables PMI generation. Note that the PMI is generated on the next event count after the counter has overflowed.
- **Cascade flag, bit 30 —** When set, enables counting on one counter of a counter pair when its alternate counter in the other the counter pair in the same counter group overflows (see Section 18.6.3.2, "Performance Counters," for further details); when clear, disables cascading of counters.
- **OVF flag, bit 31 —** Indicates that the counter has overflowed when set. This flag is a sticky flag that must be explicitly cleared by software.

The CCCRs are initialized to all 0s on reset.

The events that an enabled performance counter actually counts are selected and filtered by the following flags and fields in the ESCR and CCCR registers and in the qualification order given:

1. The event select and event mask fields in the ESCR select a class of events to be counted and one or more event types within the class, respectively.

2. The OS and USR flags in the ESCR selected the privilege levels at which events will be counted.
3. The ESCR select field of the CCCR selects the ESCR. Since each counter has several ESCRs associated with it, one ESCR must be chosen to select the classes of events that may be counted.
4. The compare and complement flags and the threshold field of the CCCR select an optional threshold to be used in qualifying an event count.
5. The edge flag in the CCCR allows events to be counted only on rising-edge transitions.

The qualification order in the above list implies that the filtered output of one "stage" forms the input for the next. For instance, events filtered using the privilege level flags can be further qualified by the compare and complement flags and the threshold field, and an event that matched the threshold criteria, can be further qualified by edge detection.

The uses of the flags and fields in the CCCRs are discussed in greater detail in Section 18.6.3.5, "Programming the Performance Counters for Non-Retirement Events."

18.6.3.4 Debug Store (DS) Mechanism

The debug store (DS) mechanism was introduced with processors based on Intel NetBurst microarchitecture to allow various types of information to be collected in memory-resident buffers for use in debugging and tuning programs. The DS mechanism can be used to collect two types of information: branch records and processor event-based sampling (PEBS) records. The availability of the DS mechanism in a processor is indicated with the DS feature flag (bit 21) returned by the CPUID instruction.

See Section 17.4.5, "Branch Trace Store (BTS)," and Section 18.6.3.8, "Processor Event-Based Sampling (PEBS)," for a description of these facilities. Records collected with the DS mechanism are saved in the DS save area. See Section 17.4.9, "BTS and DS Save Area."

18.6.3.5 Programming the Performance Counters for Non-Retirement Events

The basic steps to program a performance counter and to count events include the following:

1. Select the event or events to be counted.
2. For each event, select an ESCR that supports the event using the values in the ESCR restrictions row in Table 19-30, Chapter 19.
3. Match the CCCR Select value and ESCR name in Table 19-30 to a value listed in Table 18-70; select a CCCR and performance counter.
4. Set up an ESCR for the specific event or events to be counted and the privilege levels at which the are to be counted.
5. Set up the CCCR for the performance counter by selecting the ESCR and the desired event filters.
6. Set up the CCCR for optional cascading of event counts, so that when the selected counter overflows its alternate counter starts.
7. Set up the CCCR to generate an optional performance monitor interrupt (PMI) when the counter overflows. If PMI generation is enabled, the local APIC must be set up to deliver the interrupt to the processor and a handler for the interrupt must be in place.
8. Enable the counter to begin counting.

18.6.3.5.1 Selecting Events to Count

Table 19-31 in Chapter 19 lists a set of at-retirement events for processors based on Intel NetBurst microarchitecture. For each event listed in Table 19-31, setup information is provided. Table 18-71 gives an example of one of the events.

Table 18-71. Event Example

Event Name	Event Parameters	Parameter Value	Description
branch_retired			Counts the retirement of a branch. Specify one or more mask bits to select any combination of branch taken, not-taken, predicted and mispredicted.
	ESCR restrictions	MSR_CRU_ESCR2 MSR_CRU_ESCR3	See Table 15-3 for the addresses of the ESCR MSRs.
	Counter numbers per ESCR	ESCR2: 12, 13, 16 ESCR3: 14, 15, 17	The counter numbers associated with each ESCR are provided. The performance counters and corresponding CCCRs can be obtained from Table 15-3.
	ESCR Event Select	06H	ESCR[31:25]
	ESCR Event Mask	 Bit 0: MMNP 1: MMNM 2: MMTP 3: MMTM	ESCR[24:9] Branch Not-taken Predicted Branch Not-taken Mispredicted Branch Taken Predicted Branch Taken Mispredicted
	CCCR Select	05H	CCCR[15:13]
	Event Specific Notes		P6: EMON_BR_INST_RETIRED
	Can Support PEBS	No	
	Requires Additional MSRs for Tagging	No	

For Table 19-30 and Table 19-31, Chapter 19, the name of the event is listed in the Event Name column and parameters that define the event and other information are listed in the Event Parameters column. The Parameter Value and Description columns give specific parameters for the event and additional description information. Entries in the Event Parameters column are described below.

- **ESCR restrictions —** Lists the ESCRs that can be used to program the event. Typically only one ESCR is needed to count an event.
- **Counter numbers per ESCR —** Lists which performance counters are associated with each ESCR. Table 18-70 gives the name of the counter and CCCR for each counter number. Typically only one counter is needed to count the event.
- **ESCR event select —** Gives the value to be placed in the event select field of the ESCR to select the event.
- **ESCR event mask —** Gives the value to be placed in the Event Mask field of the ESCR to select sub-events to be counted. The parameter value column defines the documented bits with relative bit position offset starting from 0, where the absolute bit position of relative offset 0 is bit 9 of the ESCR. All undocumented bits are reserved and should be set to 0.
- **CCCR select —** Gives the value to be placed in the ESCR select field of the CCCR associated with the counter to select the ESCR to be used to define the event. This value is not the address of the ESCR; it is the number of the ESCR from the Number column in Table 18-70.
- **Event specific notes —** Gives additional information about the event, such as the name of the same or a similar event defined for the P6 family processors.
- **Can support PEBS —** Indicates if PEBS is supported for the event (only supplied for at-retirement events listed in Table 19-31.)
- **Requires additional MSR for tagging —** Indicates which if any additional MSRs must be programmed to count the events (only supplied for the at-retirement events listed in Table 19-31.)

NOTE

The performance-monitoring events listed in Chapter 19, "Performance-Monitoring Events," are intended to be used as guides for performance tuning. The counter values reported are not guaranteed to be absolutely accurate and should be used as a relative guide for tuning. Known discrepancies are documented where applicable.

The following procedure shows how to set up a performance counter for basic counting; that is, the counter is set up to count a specified event indefinitely, wrapping around whenever it reaches its maximum count. This procedure is continued through the following four sections.

Using information in Table 19-30, Chapter 19, an event to be counted can be selected as follows:

1. Select the event to be counted.
2. Select the ESCR to be used to select events to be counted from the ESCRs field.
3. Select the number of the counter to be used to count the event from the Counter Numbers Per ESCR field.
4. Determine the name of the counter and the CCCR associated with the counter, and determine the MSR addresses of the counter, CCCR, and ESCR from Table 18-70.
5. Use the WRMSR instruction to write the ESCR Event Select and ESCR Event Mask values into the appropriate fields in the ESCR. At the same time set or clear the USR and OS flags in the ESCR as desired.
6. Use the WRMSR instruction to write the CCCR Select value into the appropriate field in the CCCR.

NOTE

Typically all the fields and flags of the CCCR will be written with one WRMSR instruction; however, in this procedure, several WRMSR writes are used to more clearly demonstrate the uses of the various CCCR fields and flags.

This setup procedure is continued in the next section, Section 18.6.3.5.2, "Filtering Events."

18.6.3.5.2 Filtering Events

Each counter receives up to 4 input lines from the processor hardware from which it is counting events. The counter treats these inputs as binary inputs (input 0 has a value of 1, input 1 has a value of 2, input 3 has a value of 4, and input 3 has a value of 8). When a counter is enabled, it adds this binary input value to the counter value on each clock cycle. For each clock cycle, the value added to the counter can then range from 0 (no event) to 15.

For many events, only the 0 input line is active, so the counter is merely counting the clock cycles during which the 0 input is asserted. However, for some events two or more input lines are used. Here, the counters threshold setting can be used to filter events. The compare, complement, threshold, and edge fields control the filtering of counter increments by input value.

If the compare flag is set, then a "greater than" or a "less than or equal to" comparison of the input value vs. a threshold value can be made. The complement flag selects "less than or equal to" (flag set) or "greater than" (flag clear). The threshold field selects a threshold value of from 0 to 15. For example, if the complement flag is cleared and the threshold field is set to 6, than any input value of 7 or greater on the 4 inputs to the counter will cause the counter to be incremented by 1, and any value less than 7 will cause an increment of 0 (or no increment) of the counter. Conversely, if the complement flag is set, any value from 0 to 6 will increment the counter and any value from 7 to 15 will not increment the counter. Note that when a threshold condition has been satisfied, the input to the counter is always 1, not the input value that is presented to the threshold filter.

The edge flag provides further filtering of the counter inputs when a threshold comparison is being made. The edge flag is only active when the compare flag is set. When the edge flag is set, the resulting output from the threshold filter (a value of 0 or 1) is used as an input to the edge filter. Each clock cycle, the edge filter examines the last and current input values and sends a count to the counter only when it detects a "rising edge" event; that is, a false-to-true transition. Figure 18-48 illustrates rising edge filtering.

The following procedure shows how to configure a CCCR to filter events using the threshold filter and the edge filter. This procedure is a continuation of the setup procedure introduced in Section 18.6.3.5.1, "Selecting Events to Count."

7. (Optional) To set up the counter for threshold filtering, use the WRMSR instruction to write values in the CCCR compare and complement flags and the threshold field:
   — Set the compare flag.
   — Set or clear the complement flag for less than or equal to or greater than comparisons, respectively.
   — Enter a value from 0 to 15 in the threshold field.
8. (Optional) Select rising edge filtering by setting the CCCR edge flag.

This setup procedure is continued in the next section, Section 18.6.3.5.3, "Starting Event Counting."

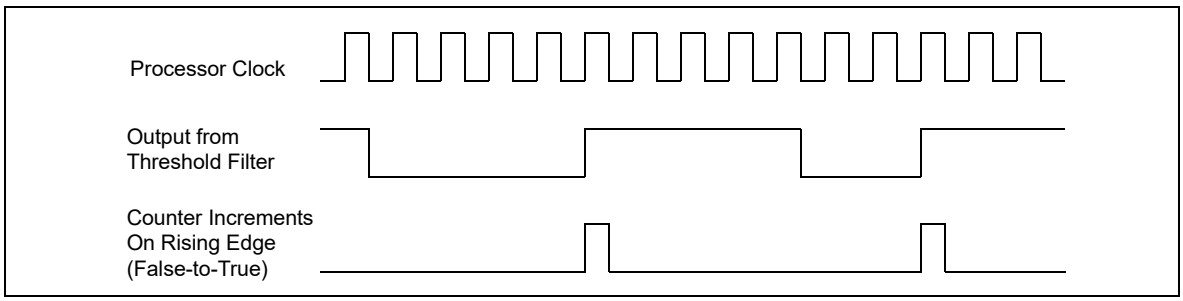


Figure 18-48. Effects of Edge Filtering

18.6.3.5.3 Starting Event Counting

Event counting by a performance counter can be initiated in either of two ways. The typical way is to set the enable flag in the counter's CCCR. Following the instruction to set the enable flag, event counting begins and continues until it is stopped (see Section 18.6.3.5.5, "Halting Event Counting").

The following procedural step shows how to start event counting. This step is a continuation of the setup procedure introduced in Section 18.6.3.5.2, "Filtering Events."

9. To start event counting, use the WRMSR instruction to set the CCCR enable flag for the performance counter.

This setup procedure is continued in the next section, Section 18.6.3.5.4, "Reading a Performance Counter's Count."

The second way that a counter can be started by using the cascade feature. Here, the overflow of one counter automatically starts its alternate counter (see Section 18.6.3.5.6, "Cascading Counters").

18.6.3.5.4 Reading a Performance Counter's Count

Performance counters can be read using either the RDPMC or RDMSR instructions. The enhanced functions of the RDPMC instruction (including fast read) are described in Section 18.6.3.2, "Performance Counters." These instructions can be used to read a performance counter while it is counting or when it is stopped.

The following procedural step shows how to read the event counter. This step is a continuation of the setup procedure introduced in Section 18.6.3.5.3, "Starting Event Counting."

10. To read a performance counters current event count, execute the RDPMC instruction with the counter number obtained from Table 18-70 used as an operand.

This setup procedure is continued in the next section, Section 18.6.3.5.5, "Halting Event Counting."

18.6.3.5.5 Halting Event Counting

After a performance counter has been started (enabled), it continues counting indefinitely. If the counter overflows (goes one count past its maximum count), it wraps around and continues counting. When the counter wraps

around, it sets its OVF flag to indicate that the counter has overflowed. The OVF flag is a sticky flag that indicates that the counter has overflowed at least once since the OVF bit was last cleared.

To halt counting, the CCCR enable flag for the counter must be cleared.

The following procedural step shows how to stop event counting. This step is a continuation of the setup procedure introduced in Section 18.6.3.5.4, "Reading a Performance Counter's Count."

11. To stop event counting, execute a WRMSR instruction to clear the CCCR enable flag for the performance counter.

To halt a cascaded counter (a counter that was started when its alternate counter overflowed), either clear the Cascade flag in the cascaded counter's CCCR MSR or clear the OVF flag in the alternate counter's CCCR MSR.

18.6.3.5.6 Cascading Counters

As described in Section 18.6.3.2, "Performance Counters," eighteen performance counters are implemented in pairs. Nine pairs of counters and associated CCCRs are further organized as four blocks: BPU, MS, FLAME, and IQ (see Table 18-70). The first three blocks contain two pairs each. The IQ block contains three pairs of counters (12 through 17) with associated CCCRs (MSR_IQ_CCCR0 through MSR_IQ_CCCR5).

The first 8 counter pairs (0 through 15) can be programmed using ESCRs to detect performance monitoring events. Pairs of ESCRs in each of the four blocks allow many different types of events to be counted. The cascade flag in the CCCR MSR allows nested monitoring of events to be performed by cascading one counter to a second counter located in another pair in the same block (see Figure 18-47 for the location of the flag).

Counters 0 and 1 form the first pair in the BPU block. Either counter 0 or 1 can be programmed to detect an event via MSR_MO B_ESCR0. Counters 0 and 2 can be cascaded in any order, as can counters 1 and 3. It's possible to set up 4 counters in the same block to cascade on two pairs of independent events. The pairing described also applies to subsequent blocks. Since the IQ PUB has two extra counters, cascading operates somewhat differently if 16 and 17 are involved. In the IQ block, counter 16 can only be cascaded from counter 14 (not from 12); counter 14 cannot be cascaded from counter 16 using the CCCR cascade bit mechanism. Similar restrictions apply to counter 17.

Example 18-1. Counting Events

Assume a scenario where counter X is set up to count 200 occurrences of event A; then counter Y is set up to count 400 occurrences of event B. Each counter is set up to count a specific event and overflow to the next counter. In the above example, counter X is preset for a count of -200 and counter Y for a count of -400; this setup causes the counters to overflow on the 200th and 400th counts respectively.

Continuing this scenario, counter X is set up to count indefinitely and wraparound on overflow. This is described in the basic performance counter setup procedure that begins in Section 18.6.3.5.1, "Selecting Events to Count." Counter Y is set up with the cascade flag in its associated CCCR MSR set to 1 and its enable flag set to 0.

To begin the nested counting, the enable bit for the counter X is set. Once enabled, counter X counts until it overflows. At this point, counter Y is automatically enabled and begins counting. Thus counter X overflows after 200 occurrences of event A. Counter Y then starts, counting 400 occurrences of event B before overflowing. When performance counters are cascaded, the counter Y would typically be set up to generate an interrupt on overflow. This is described in Section 18.6.3.5.8, "Generating an Interrupt on Overflow."

The cascading counters mechanism can be used to count a single event. The counting begins on one counter then continues on the second counter after the first counter overflows. This technique doubles the number of event counts that can be recorded, since the contents of the two counters can be added together.

18.6.3.5.7 EXTENDED CASCADING

Extended cascading is a model-specific feature in the Intel NetBurst microarchitecture with CPUID DisplayFamily_DisplayModel 0F_02, 0F_03, 0F_04, 0F_06. This feature uses bit 11 in CCCRs associated with the IQ

block. See Table 18-72.

Table 18-72. CCR Names and Bit Positions

CCCR Name:Bit Position	Bit Name	Description
MSR_IQ_CCCR1|2:11	Reserved	
MSR_IQ_CCCR0:11	CASCNT4INTO0	Allow counter 4 to cascade into counter 0
MSR_IQ_CCCR3:11	CASCNT5INTO3	Allow counter 5 to cascade into counter 3
MSR_IQ_CCCR4:11	CASCNT5INTO4	Allow counter 5 to cascade into counter 4
MSR_IQ_CCCR5:11	CASCNT4INTO5	Allow counter 4 to cascade into counter 5

The extended cascading feature can be adapted to the Interrupt based sampling usage model for performance monitoring. However, it is known that performance counters do not generate PMI in cascade mode or extended cascade mode due to an erratum. This erratum applies to processors with CPUID DisplayFamily_DisplayModel signature of 0F_02. For processors with CPUID DisplayFamily_DisplayModel signature of 0F_00 and 0F_01, the erratum applies to processors with stepping encoding greater than 09H.

Counters 16 and 17 in the IQ block are frequently used in processor event-based sampling or at-retirement counting of events indicating a stalled condition in the pipeline. Neither counter 16 or 17 can initiate the cascading of counter pairs using the cascade bit in a CCCR.

Extended cascading permits performance monitoring tools to use counters 16 and 17 to initiate cascading of two counters in the IQ block. Extended cascading from counter 16 and 17 is conceptually similar to cascading other counters, but instead of using CASCADE bit of a CCCR, one of the four CASCNTxINTOy bits is used.

Example 18-2. Scenario for Extended Cascading

A usage scenario for extended cascading is to sample instructions retired on logical processor 1 after the first 4096 instructions retired on logical processor 0. A procedure to program extended cascading in this scenario is outlined below:

1. Write the value 0 to counter 12.
2. Write the value 04000603H to MSR_CRU_ESCR0 (corresponding to selecting the NBOGNTAG and NBOGTAG event masks with qualification restricted to logical processor 1).
3. Write the value 04038800H to MSR_IQ_CCCR0. This enables CASCNT4INTO0 and OVF_PMI. An ISR can sample on instruction addresses in this case (do not set ENABLE, or CASCADE).
4. Write the value FFFFF000H into counter 16.1.
5. Write the value 0400060CH to MSR_CRU_ESCR2 (corresponding to selecting the NBOGNTAG and NBOGTAG event masks with qualification restricted to logical processor 0).
6. Write the value 00039000H to MSR_IQ_CCCR4 (set ENABLE bit, but not OVF_PMI).

Another use for cascading is to locate stalled execution in a multithreaded application. Assume MOB replays in thread B cause thread A to stall. Getting a sample of the stalled execution in this scenario could be accomplished by:

1. Set up counter B to count MOB replays on thread B.
2. Set up counter A to count resource stalls on thread A; set its force overflow bit and the appropriate CASCNTx-INTOy bit.
3. Use the performance monitoring interrupt to capture the program execution data of the stalled thread.

18.6.3.5.8 Generating an Interrupt on Overflow

Any performance counter can be configured to generate a performance monitor interrupt (PMI) if the counter overflows. The PMI interrupt service routine can then collect information about the state of the processor or program

when overflow occurred. This information can then be used with a tool like the Intel® VTune™ Performance Analyzer to analyze and tune program performance.

To enable an interrupt on counter overflow, the OVR_PMI flag in the counter's associated CCCR MSR must be set. When overflow occurs, a PMI is generated through the local APIC. (Here, the performance counter entry in the local vector table [LVT] is set up to deliver the interrupt generated by the PMI to the processor.)

The PMI service routine can use the OVF flag to determine which counter overflowed when multiple counters have been configured to generate PMIs. Also, note that these processors mask PMIs upon receiving an interrupt. Clear this condition before leaving the interrupt handler.

When generating interrupts on overflow, the performance counter being used should be preset to value that will cause an overflow after a specified number of events are counted plus 1. The simplest way to select the preset value is to write a negative number into the counter, as described in Section 18.6.3.5.6, "Cascading Counters." Here, however, if an interrupt is to be generated after 100 event counts, the counter should be preset to minus 100 plus 1 (-100 + 1), or -99. The counter will then overflow after it counts 99 events and generate an interrupt on the next (100th) event counted. The difference of 1 for this count enables the interrupt to be generated immediately after the selected event count has been reached, instead of waiting for the overflow to be propagation through the counter.

Because of latency in the microarchitecture between the generation of events and the generation of interrupts on overflow, it is sometimes difficult to generate an interrupt close to an event that caused it. In these situations, the FORCE_OVF flag in the CCCR can be used to improve reporting. Setting this flag causes the counter to overflow on every counter increment, which in turn triggers an interrupt after every counter increment.

18.6.3.5.9 Counter Usage Guideline

There are some instances where the user must take care to configure counting logic properly, so that it is not powered down. To use any ESCR, even when it is being used just for tagging, (any) one of the counters that the particular ESCR (or its paired ESCR) can be connected to should be enabled. If this is not done, 0 counts may result. Likewise, to use any counter, there must be some event selected in a corresponding ESCR (other than no_event, which generally has a select value of 0).

18.6.3.6 At-Retirement Counting

At-retirement counting provides a means counting only events that represent work committed to architectural state and ignoring work that was performed speculatively and later discarded.

One example of this speculative activity is branch prediction. When a branch misprediction occurs, the results of instructions that were decoded and executed down the mispredicted path are canceled. If a performance counter was set up to count all executed instructions, the count would include instructions whose results were canceled as well as those whose results committed to architectural state.

To provide finer granularity in event counting in these situations, the performance monitoring facilities provided in the Pentium 4 and Intel Xeon processors provide a mechanism for tagging events and then counting only those tagged events that represent committed results. This mechanism is called "at-retirement counting."

Tables 19-31 through 19-35 list predefined at-retirement events and event metrics that can be used to for tagging events when using at retirement counting. The following terminology is used in describing at-retirement counting:

- **Bogus, non-bogus, retire —** In at-retirement event descriptions, the term "bogus" refers to instructions or μops that must be canceled because they are on a path taken from a mispredicted branch. The terms "retired" and "non-bogus" refer to instructions or μops along the path that results in committed architectural state changes as required by the program being executed. Thus instructions and μops are either bogus or non-bogus, but not both. Several of the Pentium 4 and Intel Xeon processors' performance monitoring events (such as, Instruction_Retired and Uops_Retired in Table 19-31) can count instructions or μops that are retired based on the characterization of bogus" versus non-bogus.
- **Tagging —** Tagging is a means of marking μops that have encountered a particular performance event so they can be counted at retirement. During the course of execution, the same event can happen more than once per μop and a direct count of the event would not provide an indication of how many μops encountered that event.

  The tagging mechanisms allow a μop to be tagged once during its lifetime and thus counted once at retirement. The retired suffix is used for performance metrics that increment a count once per μop, rather than once per

event. For example, a μop may encounter a cache miss more than once during its life time, but a "Miss Retired" metric (that counts the number of retired μops that encountered a cache miss) will increment only once for that μop. A "Miss Retired" metric would be useful for characterizing the performance of the cache hierarchy for a particular instruction sequence. Details of various performance metrics and how these can be constructed using the Pentium 4 and Intel Xeon processors performance events are provided in the *Intel Pentium 4 Processor Optimization Reference Manual* (see Section 1.4, "Related Literature").

- **Replay —** To maximize performance for the common case, the Intel NetBurst microarchitecture aggressively schedules μops for execution before all the conditions for correct execution are guaranteed to be satisfied. In the event that all of these conditions are not satisfied, μops must be reissued. The mechanism that the Pentium 4 and Intel Xeon processors use for this reissuing of μops is called replay. Some examples of replay causes are cache misses, dependence violations, and unforeseen resource constraints. In normal operation, some number of replays is common and unavoidable. An excessive number of replays is an indication of a performance problem.
- **Assist —** When the hardware needs the assistance of microcode to deal with some event, the machine takes an assist. One example of this is an underflow condition in the input operands of a floating-point operation. The hardware must internally modify the format of the operands in order to perform the computation. Assists clear the entire machine of μops before they begin and are costly.

18.6.3.6.1 Using At-Retirement Counting

Processors based on Intel NetBurst microarchitecture allow counting both events and μops that encountered a specified event. For a subset of the at-retirement events listed in Table 19-31, a μop may be tagged when it encounters that event. The tagging mechanisms can be used in Interrupt-based event sampling, and a subset of these mechanisms can be used in PEBS. There are four independent tagging mechanisms, and each mechanism uses a different event to count μops tagged with that mechanism:

- **Front-end tagging —** This mechanism pertains to the tagging of μops that encountered front-end events (for example, trace cache and instruction counts) and are counted with the Front_end_event event.
- **Execution tagging —** This mechanism pertains to the tagging of μops that encountered execution events (for example, instruction types) and are counted with the Execution_Event event.
- **Replay tagging —** This mechanism pertains to tagging of μops whose retirement is replayed (for example, a cache miss) and are counted with the Replay_event event. Branch mispredictions are also tagged with this mechanism.
- **No tags —** This mechanism does not use tags. It uses the Instr_retired and the Uops_ retired events.

Each tagging mechanism is independent from all others; that is, a μop that has been tagged using one mechanism will not be detected with another mechanism's tagged-μop detector. For example, if μops are tagged using the front-end tagging mechanisms, the Replay_event will not count those as tagged μops unless they are also tagged using the replay tagging mechanism. However, execution tags allow up to four different types of μops to be counted at retirement through execution tagging.

The independence of tagging mechanisms does not hold when using PEBS. When using PEBS, only one tagging mechanism should be used at a time.

Certain kinds of μops that cannot be tagged, including I/O, uncacheable and locked accesses, returns, and far transfers.

Table 19-31 lists the performance monitoring events that support at-retirement counting: specifically the Front_end_event, Execution_event, Replay_event, Inst_retired and Uops_retired events. The following sections describe the tagging mechanisms for using these events to tag μop and count tagged μops.

18.6.3.6.2 Tagging Mechanism for Front_end_event

The Front_end_event counts μops that have been tagged as encountering any of the following events:

- **μop decode events —** Tagging μops for μop decode events requires specifying bits in the ESCR associated with the performance-monitoring event, Uop_type.
- **Trace cache events —** Tagging μops for trace cache events may require specifying certain bits in the MSR_TC_PRECISE_EVENT MSR (see Table 19-33).

Table 19-31 describes the Front_end_event and Table 19-33 describes metrics that are used to set up a Front_end_event count.

The MSRs specified in the Table 19-31 that are supported by the front-end tagging mechanism must be set and one or both of the NBOGUS and BOGUS bits in the Front_end_event event mask must be set to count events. None of the events currently supported requires the use of the MSR_TC_PRECISE_EVENT MSR.

18.6.3.6.3 Tagging Mechanism For Execution_event

Table 19-31 describes the Execution_event and Table 19-34 describes metrics that are used to set up an Execution_event count.

The execution tagging mechanism differs from other tagging mechanisms in how it causes tagging. One *upstream* ESCR is used to specify an event to detect and to specify a tag value (bits 5 through 8) to identify that event. A second *downstream* ESCR is used to detect μops that have been tagged with that tag value identifier using Execution_event for the event selection.

The upstream ESCR that counts the event must have its tag enable flag (bit 4) set and must have an appropriate tag value mask entered in its tag value field. The 4-bit tag value mask specifies which of tag bits should be set for a particular μop. The value selected for the tag value should coincide with the event mask selected in the downstream ESCR. For example, if a tag value of 1 is set, then the event mask of NBOGUS0 should be enabled, correspondingly in the downstream ESCR. The downstream ESCR detects and counts tagged μops. The normal (not tag value) mask bits in the downstream ESCR specify which tag bits to count. If any one of the tag bits selected by the mask is set, the related counter is incremented by one. This mechanism is summarized in the Table 19-34 metrics that are supported by the execution tagging mechanism. The tag enable and tag value bits are irrelevant for the downstream ESCR used to select the Execution_event.

The four separate tag bits allow the user to simultaneously but distinctly count up to four execution events at retirement. (This applies for interrupt-based event sampling. There are additional restrictions for PEBS as noted in Section 18.6.3.8.3, "Setting Up the PEBS Buffer.") It is also possible to detect or count combinations of events by setting multiple tag value bits in the upstream ESCR or multiple mask bits in the downstream ESCR. For example, use a tag value of 3H in the upstream ESCR and use NBOGUS0/NBOGUS1 in the downstream ESCR event mask.

18.6.3.7 Tagging Mechanism for Replay_event

Table 19-31 describes the Replay_event and Table 19-35 describes metrics that are used to set up an Replay_event count.

The replay mechanism enables tagging of μops for a subset of all replays before retirement. Use of the replay mechanism requires selecting the type of μop that may experience the replay in the MSR_PEBS_MATRIX_VERT MSR and selecting the type of event in the MSR_PEBS_ENABLE MSR. Replay tagging must also be enabled with the UOP_Tag flag (bit 24) in the MSR_PEBS_ENABLE MSR.

The Table 19-35 lists the metrics that are support the replay tagging mechanism and the at-retirement events that use the replay tagging mechanism, and specifies how the appropriate MSRs need to be configured. The replay tags defined in Table A-5 also enable Processor Event-Based Sampling (PEBS, see Section 17.4.9). Each of these replay tags can also be used in normal sampling by not setting Bit 24 nor Bit 25 in IA_32_PEBS_ENABLE_MSR. Each of these metrics requires that the Replay_Event (see Table 19-31) be used to count the tagged μops.

18.6.3.8 Processor Event-Based Sampling (PEBS)

The debug store (DS) mechanism in processors based on Intel NetBurst microarchitecture allow two types of information to be collected for use in debugging and tuning programs: PEBS records and BTS records. See Section 17.4.5, "Branch Trace Store (BTS)," for a description of the BTS mechanism.

PEBS permits the saving of precise architectural information associated with one or more performance events in the precise event records buffer, which is part of the DS save area (see Section 17.4.9, "BTS and DS Save Area"). To use this mechanism, a counter is configured to overflow after it has counted a preset number of events. After the counter overflows, the processor copies the current state of the general-purpose and EFLAGS registers and instruction pointer into a record in the precise event records buffer. The processor then resets the count in the performance counter and restarts the counter. When the precise event records buffer is nearly full, an interrupt is

generated, allowing the precise event records to be saved. A circular buffer is not supported for precise event records.

PEBS is supported only for a subset of the at-retirement events: Execution_event, Front_end_event, and Replay_event. Also, PEBS can only be carried out using the one performance counter, the MSR_IQ_COUNTER4 MSR.

In processors based on Intel Core microarchitecture, a similar PEBS mechanism is also supported using IA32_PMC0 and IA32_PERFEVTSEL0 MSRs (See Section 18.6.2.4).

18.6.3.8.1 Detection of the Availability of the PEBS Facilities

The DS feature flag (bit 21) returned by the CPUID instruction indicates (when set) the availability of the DS mechanism in the processor, which supports the PEBS (and BTS) facilities. When this bit is set, the following PEBS facilities are available:

- The PEBS_UNAVAILABLE flag in the IA32_MISC_ENABLE MSR indicates (when clear) the availability of the PEBS facilities, including the MSR_PEBS_ENABLE MSR.
- The enable PEBS flag (bit 24) in the MSR_PEBS_ENABLE MSR allows PEBS to be enabled (set) or disabled (clear).
- The IA32_DS_AREA MSR can be programmed to point to the DS save area.

18.6.3.8.2 Setting Up the DS Save Area

Section 17.4.9.2, "Setting Up the DS Save Area," describes how to set up and enable the DS save area. This procedure is common for PEBS and BTS.

18.6.3.8.3 Setting Up the PEBS Buffer

Only the MSR_IQ_COUNTER4 performance counter can be used for PEBS. Use the following procedure to set up the processor and this counter for PEBS:

1. Set up the precise event buffering facilities. Place values in the precise event buffer base, precise event index, precise event absolute maximum, and precise event interrupt threshold, and precise event counter reset fields of the DS buffer management area (see Figure 17-5) to set up the precise event records buffer in memory.
2. Enable PEBS. Set the Enable PEBS flag (bit 24) in MSR_PEBS_ENABLE MSR.
3. Set up the MSR_IQ_COUNTER4 performance counter and its associated CCCR and one or more ESCRs for PEBS as described in Tables 19-31 through 19-35.

18.6.3.8.4 Writing a PEBS Interrupt Service Routine

The PEBS facilities share the same interrupt vector and interrupt service routine (called the DS ISR) with the non-precise event-based sampling and BTS facilities. To handle PEBS interrupts, PEBS handler code must be included in the DS ISR. See Section 17.4.9.5, "Writing the DS Interrupt Service Routine," for guidelines for writing the DS ISR.

18.6.3.8.5 Other DS Mechanism Implications

The DS mechanism is not available in the SMM. It is disabled on transition to the SMM mode. Similarly the DS mechanism is disabled on the generation of a machine check exception and is cleared on processor RESET and INIT.

The DS mechanism is available in real address mode.

18.6.3.9 Operating System Implications

The DS mechanism can be used by the operating system as a debugging extension to facilitate failure analysis. When using this facility, a 25 to 30 times slowdown can be expected due to the effects of the trace store occurring on every taken branch.

Depending upon intended usage, the instruction pointers that are part of the branch records or the PEBS records need to have an association with the corresponding process. One solution requires the ability for the DS specific operating system module to be chained to the context switch. A separate buffer can then be maintained for each process of interest and the MSR pointing to the configuration area saved and setup appropriately on each context switch.

If the BTS facility has been enabled, then it must be disabled and state stored on transition of the system to a sleep state in which processor context is lost. The state must be restored on return from the sleep state.

It is required that an interrupt gate be used for the DS interrupt as opposed to a trap gate to prevent the generation of an endless interrupt loop.

Pages that contain buffers must have mappings to the same physical address for all processes/logical processors, such that any change to CR3 will not change DS addresses. If this requirement cannot be satisfied (that is, the feature is enabled on a per thread/process basis), then the operating system must ensure that the feature is enabled/disabled appropriately in the context switch code.

18.6.4 Performance Monitoring and Intel Hyper-Threading Technology in Processors Based on Intel NetBurst® Microarchitecture

The performance monitoring capability of processors based on Intel NetBurst microarchitecture and supporting Intel Hyper-Threading Technology is similar to that described in Section 18.6.3. However, the capability is extended so that:

- Performance counters can be programmed to select events qualified by logical processor IDs.
- Performance monitoring interrupts can be directed to a specific logical processor within the physical processor.

The sections below describe performance counters, event qualification by logical processor ID, and special purpose bits in ESCRs/CCCRs. They also describe MSR_PEBS_ENABLE, MSR_PEBS_MATRIX_VERT, and MSR_TC_PRECISE_EVENT.

18.6.4.1 ESCR MSRs

Figure 18-49 shows the layout of an ESCR MSR in processors supporting Intel Hyper-Threading Technology.

The functions of the flags and fields are as follows:

- **T1_USR flag, bit 0 —** When set, events are counted when thread 1 (logical processor 1) is executing at a current privilege level (CPL) of 1, 2, or 3. These privilege levels are generally used by application code and unprotected operating system code.

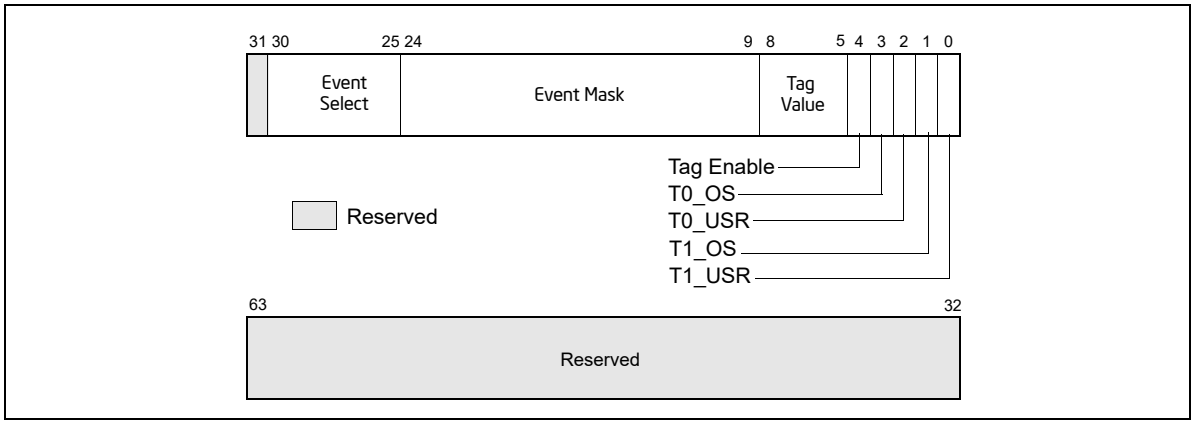


Figure 18-49. Event Selection Control Register (ESCR) for the Pentium 4 Processor, Intel Xeon Processor and Intel Xeon Processor MP Supporting Hyper-Threading Technology

- **T1_OS flag, bit 1 —** When set, events are counted when thread 1 (logical processor 1) is executing at CPL of 0. This privilege level is generally reserved for protected operating system code. (When both the T1_OS and T1_USR flags are set, thread 1 events are counted at all privilege levels.)
- **T0_USR flag, bit 2 —** When set, events are counted when thread 0 (logical processor 0) is executing at a CPL of 1, 2, or 3.
- **T0_OS flag, bit 3 —** When set, events are counted when thread 0 (logical processor 0) is executing at CPL of 0. (When both the T0_OS and T0_USR flags are set, thread 0 events are counted at all privilege levels.)
- **Tag enable, bit 4 —** When set, enables tagging of μops to assist in at-retirement event counting; when clear, disables tagging. See Section 18.6.3.6, "At-Retirement Counting."
- **Tag value field, bits 5 through 8 —** Selects a tag value to associate with a μop to assist in at-retirement event counting.
- **Event mask field, bits 9 through 24 —** Selects events to be counted from the event class selected with the event select field.
- **Event select field, bits 25 through 30) —** Selects a class of events to be counted. The events within this class that are counted are selected with the event mask field.

The T0_OS and T0_USR flags and the T1_OS and T1_USR flags allow event counting and sampling to be specified for a specific logical processor (0 or 1) within an Intel Xeon processor MP (See also: Section 8.4.5, "Identifying Logical Processors in an MP System," in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*).

Not all performance monitoring events can be detected within an Intel Xeon processor MP on a per logical processor basis (see Section 18.6.4.4, "Performance Monitoring Events"). Some sub-events (specified by an event mask bits) are counted or sampled without regard to which logical processor is associated with the detected event.

18.6.4.2 CCCR MSRs

Figure 18-50 shows the layout of a CCCR MSR in processors supporting Intel Hyper-Threading Technology. The functions of the flags and fields are as follows:

- **Enable flag, bit 12 —** When set, enables counting; when clear, the counter is disabled. This flag is cleared on reset
- **ESCR select field, bits 13 through 15 —** Identifies the ESCR to be used to select events to be counted with the counter associated with the CCCR.
- **Active thread field, bits 16 and 17 —** Enables counting depending on which logical processors are active (executing a thread). This field enables filtering of events based on the state (active or inactive) of the logical processors. The encodings of this field are as follows:

  00 — None. Count only when neither logical processor is active.

  01 — Single. Count only when one logical processor is active (either 0 or 1).

  10 — Both. Count only when both logical processors are active.

  11 — Any. Count when either logical processor is active.

  A halted logical processor or a logical processor in the "wait for SIPI" state is considered inactive.

- **Compare flag, bit 18 —** When set, enables filtering of the event count; when clear, disables filtering. The filtering method is selected with the threshold, complement, and edge flags.

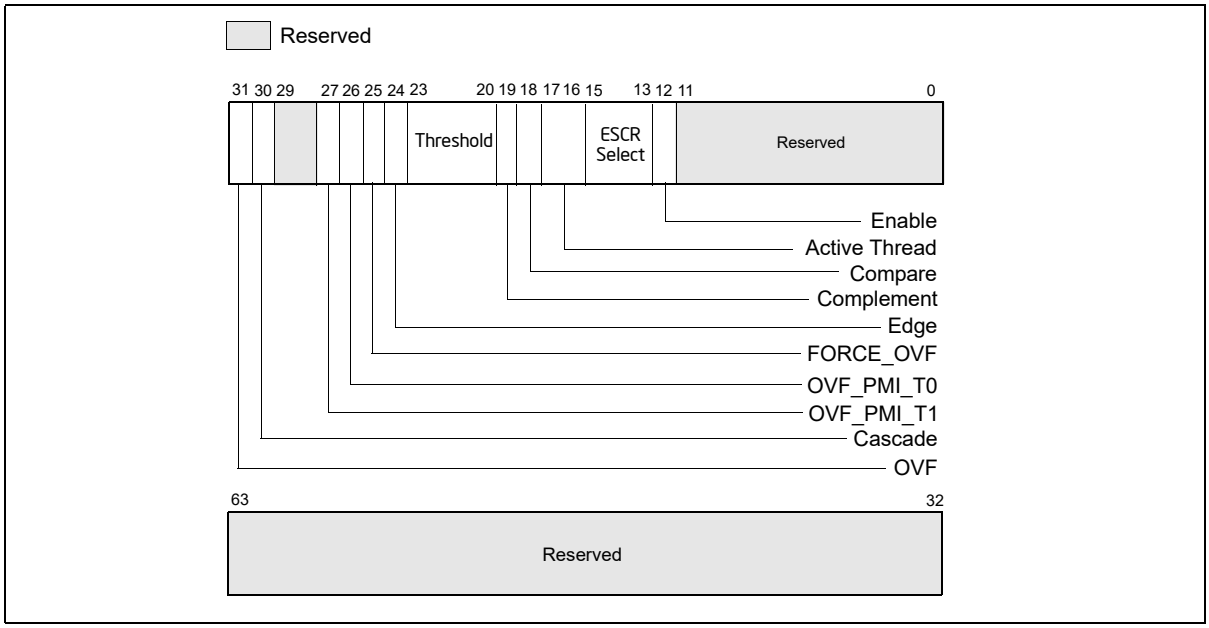


Figure 18-50. Counter Configuration Control Register (CCCR)

- **Complement flag, bit 19 —** Selects how the incoming event count is compared with the threshold value. When set, event counts that are less than or equal to the threshold value result in a single count being delivered to the performance counter; when clear, counts greater than the threshold value result in a count being delivered to the performance counter (see Section 18.6.3.5.2, "Filtering Events"). The compare flag is not active unless the compare flag is set.
- **Threshold field, bits 20 through 23 —** Selects the threshold value to be used for comparisons. The processor examines this field only when the compare flag is set, and uses the complement flag setting to determine the type of threshold comparison to be made. The useful range of values that can be entered in this field depend on the type of event being counted (see Section 18.6.3.5.2, "Filtering Events").
- **Edge flag, bit 24 —** When set, enables rising edge (false-to-true) edge detection of the threshold comparison output for filtering event counts; when clear, rising edge detection is disabled. This flag is active only when the compare flag is set.
- **FORCE_OVF flag, bit 25 —** When set, forces a counter overflow on every counter increment; when clear, overflow only occurs when the counter actually overflows.
- **OVF_PMI_T0 flag, bit 26 —** When set, causes a performance monitor interrupt (PMI) to be sent to logical processor 0 when the counter overflows occurs; when clear, disables PMI generation for logical processor 0. Note that the PMI is generate on the next event count after the counter has overflowed.
- **OVF_PMI_T1 flag, bit 27 —** When set, causes a performance monitor interrupt (PMI) to be sent to logical processor 1 when the counter overflows occurs; when clear, disables PMI generation for logical processor 1. Note that the PMI is generate on the next event count after the counter has overflowed.
- **Cascade flag, bit 30 —** When set, enables counting on one counter of a counter pair when its alternate counter in the other the counter pair in the same counter group overflows (see Section 18.6.3.2, "Performance Counters," for further details); when clear, disables cascading of counters.
- **OVF flag, bit 31 —** Indicates that the counter has overflowed when set. This flag is a sticky flag that must be explicitly cleared by software.

18.6.4.3 IA32_PEBS_ENABLE MSR

In a processor supporting Intel Hyper-Threading Technology and based on the Intel NetBurst microarchitecture, PEBS is enabled and qualified with two bits in the MSR_PEBS_ENABLE MSR: bit 25 (ENABLE_PEBS_MY_THR) and 26 (ENABLE_PEBS_OTH_THR) respectively. These bits do not explicitly identify a specific logical processor by logic

processor ID(T0 or T1); instead, they allow a software agent to enable PEBS for subsequent threads of execution on the same logical processor on which the agent is running ("my thread") or for the other logical processor in the physical package on which the agent is not running ("other thread").

PEBS is supported for only a subset of the at-retirement events: Execution_event, Front_end_event, and Replay_event. Also, PEBS can be carried out only with two performance counters: MSR_IQ_CCCR4 (MSR address 370H) for logical processor 0 and MSR_IQ_CCCR5 (MSR address 371H) for logical processor 1.

Performance monitoring tools should use a processor affinity mask to bind the kernel mode components that need to modify the ENABLE_PEBS_MY_THR and ENABLE_PEBS_OTH_THR bits in the MSR_PEBS_ENABLE MSR to a specific logical processor. This is to prevent these kernel mode components from migrating between different logical processors due to OS scheduling.

18.6.4.4 Performance Monitoring Events

All of the events listed in Table 19-30 and 19-31 are available in an Intel Xeon processor MP. When Intel Hyper-Threading Technology is active, many performance monitoring events can be can be qualified by the logical processor ID, which corresponds to bit 0 of the initial APIC ID. This allows for counting an event in any or all of the logical processors. However, not all the events have this logic processor specificity, or thread specificity.

Here, each event falls into one of two categories:

- **Thread specific (TS) —** The event can be qualified as occurring on a specific logical processor.
- **Thread independent (TI) —** The event cannot be qualified as being associated with a specific logical processor.

Table 19-36 gives logical processor specific information (TS or TI) for each of the events described in Tables 19-30 and 19-31. If for example, a TS event occurred in logical processor T0, the counting of the event (as shown in Table 18-73) depends only on the setting of the T0_USR and T0_OS flags in the ESCR being used to set up the event counter. The T1_USR and T1_OS flags have no effect on the count.

Table 18-73. Effect of Logical Processor and CPL Qualification for Logical-Processor-Specific (TS) Events

	T1_OS/T1_USR = 00	T1_OS/T1_USR = 01	T1_OS/T1_USR = 11	T1_OS/T1_USR = 10
T0_OS/T0_USR = 00	Zero count	Counts while T1 in USR	Counts while T1 in OS or USR	Counts while T1 in OS
T0_OS/T0_USR = 01	Counts while T0 in USR	Counts while T0 in USR or T1 in USR	Counts while (a) T0 in USR or (b) T1 in OS or (c) T1 in USR	Counts while (a) T0 in OS or (b) T1 in OS
T0_OS/T0_USR = 11	Counts while T0 in OS or USR	Counts while (a) T0 in OS or (b) T0 in USR or (c) T1 in USR	Counts irrespective of CPL, T0, T1	Counts while (a) T0 in OS or (b) or T0 in USR or (c) T1 in OS
T0_OS/T0_USR = 10	Counts T0 in OS	Counts T0 in OS or T1 in USR	Counts while (a)T0 in Os or (b) T1 in OS or (c) T1 in USR	Counts while (a) T0 in OS or (b) T1 in OS

When a bit in the event mask field is TI, the effect of specifying bit-0-3 of the associated ESCR are described in Table 15-6. For events that are marked as TI in Chapter 19, the effect of selectively specifying T0_USR, T0_OS, T1_USR, T1_OS bits is shown in Table 18-74.

Table 18-74. Effect of Logical Processor and CPL Qualification for Non-logical-Processor-specific (TI) Events

	T1_OS/T1_USR = 00	T1_OS/T1_USR = 01	T1_OS/T1_USR = 11	T1_OS/T1_USR = 10
T0_OS/T0_USR = 00	Zero count	Counts while (a) T0 in USR or (b) T1 in USR	Counts irrespective of CPL, T0, T1	Counts while (a) T0 in OS or (b) T1 in OS
T0_OS/T0_USR = 01	Counts while (a) T0 in USR or (b) T1 in USR	Counts while (a) T0 in USR or (b) T1 in USR	Counts irrespective of CPL, T0, T1	Counts irrespective of CPL, T0, T1
T0_OS/T0_USR = 11	Counts irrespective of CPL, T0, T1	Counts irrespective of CPL, T0, T1	Counts irrespective of CPL, T0, T1	Counts irrespective of CPL, T0, T1
T0_OS/T0_USR = 0	Counts while (a) T0 in OS or (b) T1 in OS	Counts irrespective of CPL, T0, T1	Counts irrespective of CPL, T0, T1	Counts while (a) T0 in OS or (b) T1 in OS

18.6.4.5 Counting Clocks on systems with Intel Hyper-Threading Technology in Processors Based on Intel NetBurst® Microarchitecture

18.6.4.5.1 Non-Halted Clockticks

Use the following procedure to program ESCRs and CCCRs to obtain non-halted clockticks on processors based on Intel NetBurst microarchitecture:

1. Select an ESCR for the global_power_events and specify the RUNNING sub-event mask and the desired T0_OS/T0_USR/T1_OS/T1_USR bits for the targeted processor.
2. Select an appropriate counter.
3. Enable counting in the CCCR for that counter by setting the enable bit.

18.6.4.5.2 Non-Sleep Clockticks

Performance monitoring counters can be configured to count clockticks whenever the performance monitoring hardware is not powered-down. To count Non-sleep Clockticks with a performance-monitoring counter, do the following:

1. Select one of the 18 counters.
2. Select any of the ESCRs whose events the selected counter can count. Set its event select to anything other than "no_event"; the counter may be disabled if this is not done.
3. Turn threshold comparison on in the CCCR by setting the compare bit to "1".
4. Set the threshold to "15" and the complement to "1" in the CCCR. Since no event can exceed this threshold, the threshold condition is met every cycle and the counter counts every cycle. Note that this overrides any qualification (e.g. by CPL) specified in the ESCR.
5. Enable counting in the CCCR for the counter by setting the enable bit.

In most cases, the counts produced by the non-halted and non-sleep metrics are equivalent if the physical package supports one logical processor and is not placed in a power-saving state. Operating systems may execute an HLT instruction and place a physical processor in a power-saving state.

On processors that support Intel Hyper-Threading Technology (Intel HT Technology), each physical package can support two or more logical processors. Current implementation of Intel HT Technology provides two logical processors for each physical processor. While both logical processors can execute two threads simultaneously, one logical processor may halt to allow the other logical processor to execute without sharing execution resources between two logical processors.

Non-halted Clockticks can be set up to count the number of processor clock cycles for each logical processor whenever the logical processor is not halted (the count may include some portion of the clock cycles for that logical processor to complete a transition to a halted state). Physical processors that support Intel HT Technology enter into a power-saving state if all logical processors halt.

The Non-sleep Clockticks mechanism uses a filtering mechanism in CCCRs. The mechanism will continue to increment as long as one logical processor is not halted or in a power-saving state. Applications may cause a processor to enter into a power-saving state by using an OS service that transfers control to an OS's idle loop. The idle loop then may place the processor into a power-saving state after an implementation-dependent period if there is no work for the processor.

18.6.5 Performance Monitoring and Dual-Core Technology

The performance monitoring capability of dual-core processors duplicates the microarchitectural resources of a single-core processor implementation. Each processor core has dedicated performance monitoring resources.

In the case of Pentium D processor, each logical processor is associated with dedicated resources for performance monitoring. In the case of Pentium processor Extreme edition, each processor core has dedicated resources, but two logical processors in the same core share performance monitoring resources (see Section 18.6.4, "Performance Monitoring and Intel Hyper-Threading Technology in Processors Based on Intel NetBurst® Microarchitecture").

18.6.6 Performance Monitoring on 64-bit Intel Xeon Processor MP with Up to 8-MByte L3 Cache

The 64-bit Intel Xeon processor MP with up to 8-MByte L3 cache has a CPUID signature of family [0FH], model [03H or 04H]. Performance monitoring capabilities available to Pentium 4 and Intel Xeon processors with the same values (see Section 18.1 and Section 18.6.4) apply to the 64-bit Intel Xeon processor MP with an L3 cache.

The level 3 cache is connected between the system bus and IOQ through additional control logic. See Figure 18-51.

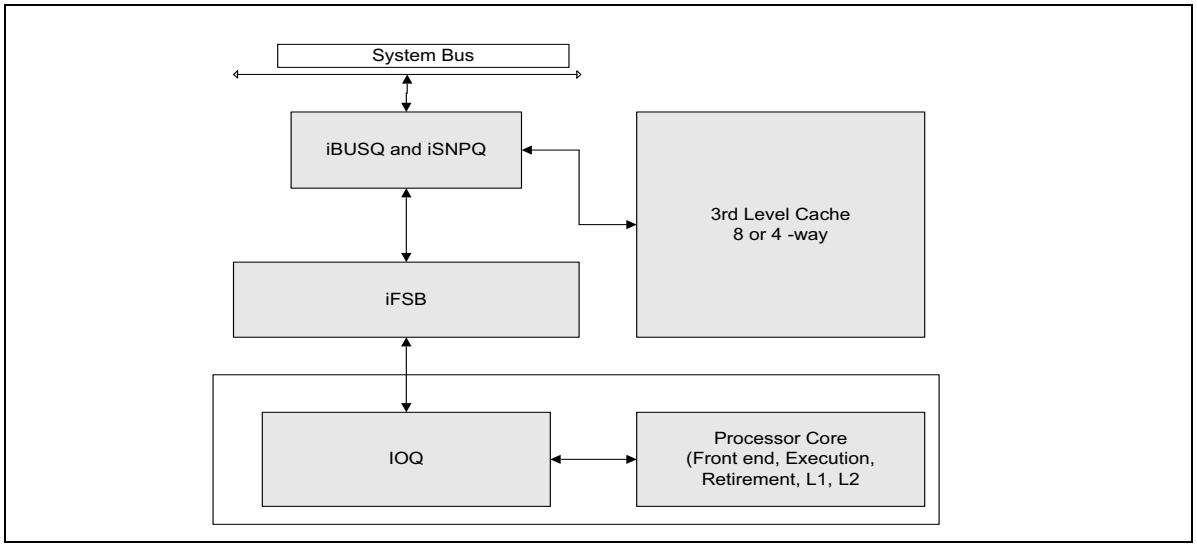


Figure 18-51. Block Diagram of 64-bit Intel Xeon Processor MP with 8-MByte L3

Additional performance monitoring capabilities and facilities unique to 64-bit Intel Xeon processor MP with an L3 cache are described in this section. The facility for monitoring events consists of a set of dedicated model-specific registers (MSRs), each dedicated to a specific event. Programming of these MSRs requires using RDMSR/WRMSR instructions with 64-bit values.

The lower 32-bits of the MSRs at addresses 107CC through 107D3 are treated as 32 bit performance counter registers. These performance counters can be accessed using RDPMC instruction with the index starting from 18 through 25. The EDX register returns zero when reading these 8 PMCs.

The performance monitoring capabilities consist of four events. These are:

- **IBUSQ event —** This event detects the occurrence of micro-architectural conditions related to the iBUSQ unit. It provides two MSRs: MSR_IFSB_IBUSQ0 and MSR_IFSB_IBUSQ1. Configure sub-event qualification and enable/disable functions using the high 32 bits of these MSRs. The low 32 bits act as a 32-bit event counter. Counting starts after software writes a non-zero value to one or more of the upper 32 bits. See Figure 18-52.

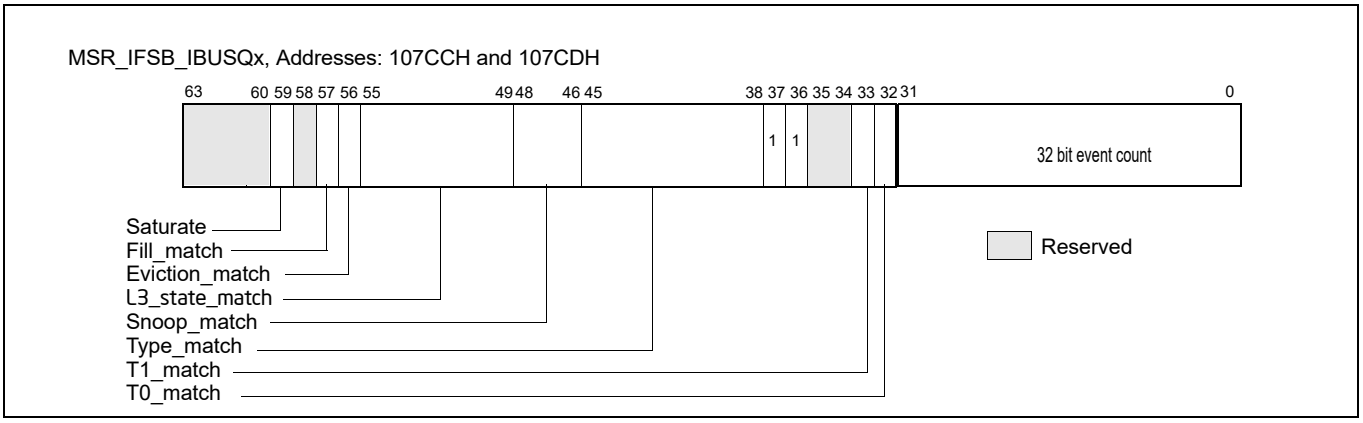


Figure 18-52. MSR_IFSB_IBUSQx, Addresses: 107CCH and 107CDH

- **ISNPQ event —** This event detects the occurrence of microarchitectural conditions related to the iSNPQ unit. It provides two MSRs: MSR_IFSB_ISNPQ0 and MSR_IFSB_ISNPQ1. Configure sub-event qualifications and enable/disable functions using the high 32 bits of the MSRs. The low 32-bits act as a 32-bit event counter. Counting starts after software writes a non-zero value to one or more of the upper 32-bits. See Figure 18-53.

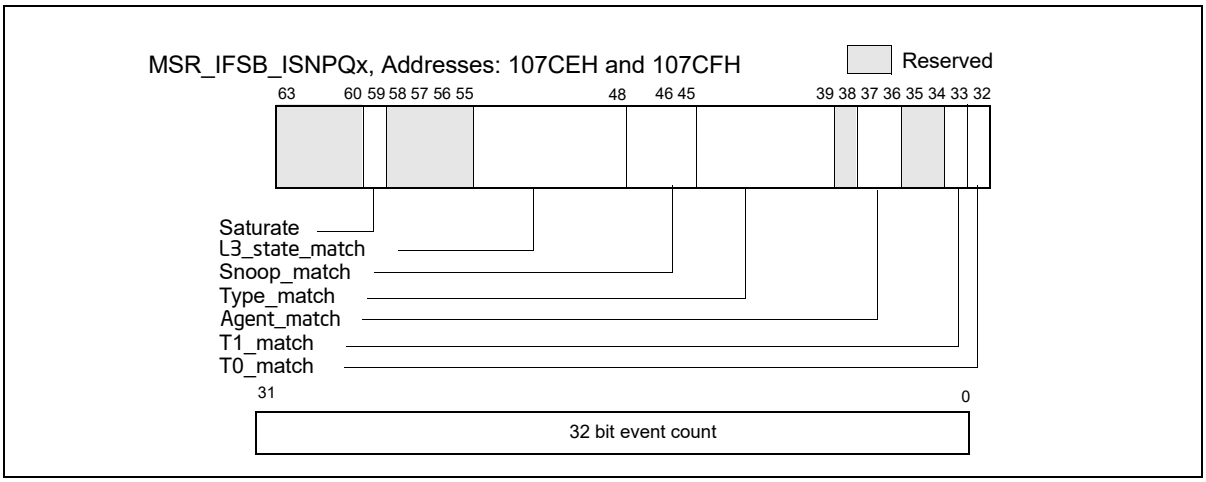


Figure 18-53. MSR_IFSB_ISNPQx, Addresses: 107CEH and 107CFH

- **EFSB event —** This event can detect the occurrence of micro-architectural conditions related to the iFSB unit or system bus. It provides two MSRs: MSR_EFSB_DRDY0 and MSR_EFSB_DRDY1. Configure sub-event qualifications and enable/disable functions using the high 32 bits of the 64-bit MSR. The low 32-bit act as a 32-bit event counter. Counting starts after software writes a non-zero value to one or more of the qualification bits in the upper 32-bits of the MSR. See Figure 18-54.

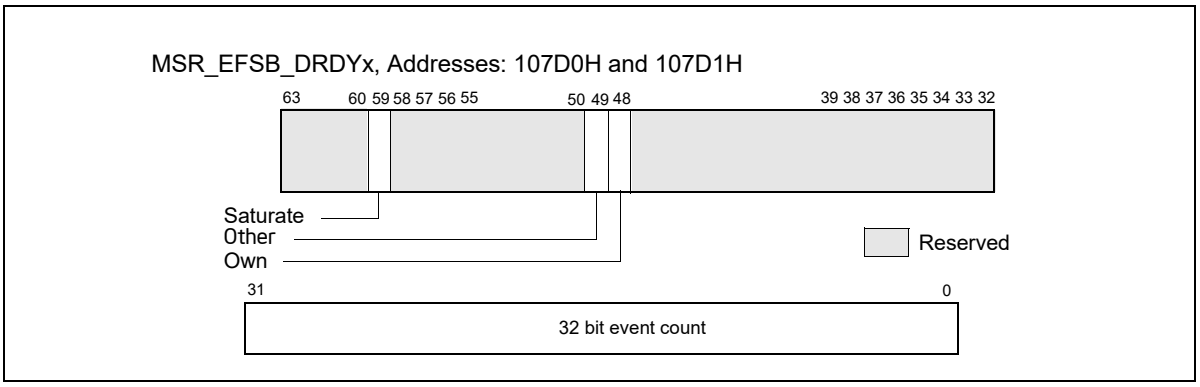


Figure 18-54. MSR_EFSB_DRDYx, Addresses: 107D0H and 107D1H

- **IBUSQ Latency event —** This event accumulates weighted cycle counts for latency measurement of transactions in the iBUSQ unit. The count is enabled by setting MSR_IFSB_CTRL6[bit 26] to 1; the count freezes after software sets MSR_IFSB_CTRL6[bit 26] to 0. MSR_IFSB_CNTR7 acts as a 64-bit event counter for this event. See Figure 18-55.

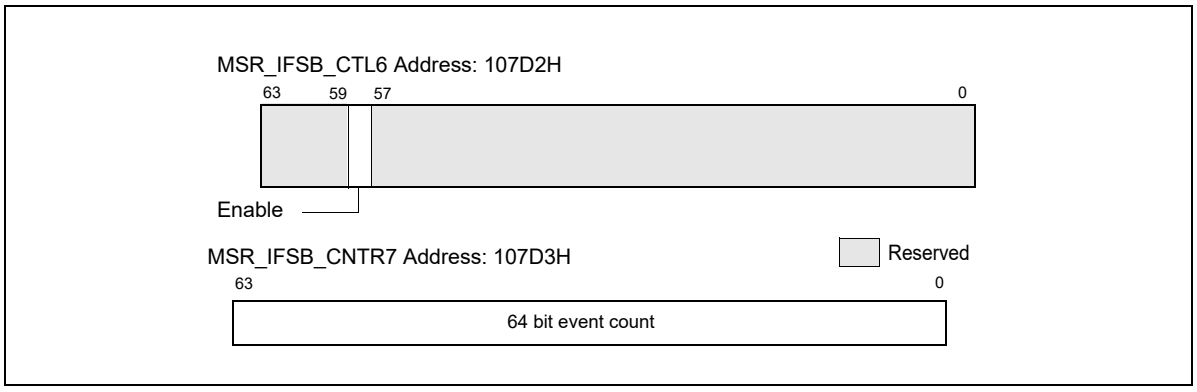


Figure 18-55. MSR_IFSB_CTL6, Address: 107D2H; MSR_IFSB_CNTR7, Address: 107D3H

18.6.7 Performance Monitoring on L3 and Caching Bus Controller Sub-Systems

The Intel Xeon processor 7400 series and Dual-Core Intel Xeon processor 7100 series employ a distinct L3/caching bus controller sub-system. These sub-system have a unique set of performance monitoring capability and programming interfaces that are largely common between these two processor families.

Intel Xeon processor 7400 series are based on 45 nm enhanced Intel Core microarchitecture. The CPUID signature is indicated by DisplayFamily_DisplayModel value of 06_1DH (see CPUID instruction in Chapter 3, "Instruction Set Reference, A-L" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A*). Intel Xeon processor 7400 series have six processor cores that share an L3 cache.

Dual-Core Intel Xeon processor 7100 series are based on Intel NetBurst microarchitecture, have a CPUID signature of family [0FH], model [06H] and a unified L3 cache shared between two cores. Each core in an Intel Xeon processor 7100 series supports Intel Hyper-Threading Technology, providing two logical processors per core.

Both Intel Xeon processor 7400 series and Intel Xeon processor 7100 series support multi-processor configurations using system bus interfaces. In Intel Xeon processor 7400 series, the L3/caching bus controller sub-system provides three Simple Direct Interface (SDI) to service transactions originated the XQ-replacement SDI logic in each dual-core modules. In Intel Xeon processor 7100 series, the IOQ logic in each processor core is replaced with a Simple Direct Interface (SDI) logic. The L3 cache is connected between the system bus and the SDI through addi-

tional control logic. See Figure 18-56 for the block configuration of six processor cores and the L3/Caching bus controller sub-system in Intel Xeon processor 7400 series. Figure 18-56 shows the block configuration of two processor cores (four logical processors) and the L3/Caching bus controller sub-system in Intel Xeon processor 7100 series.

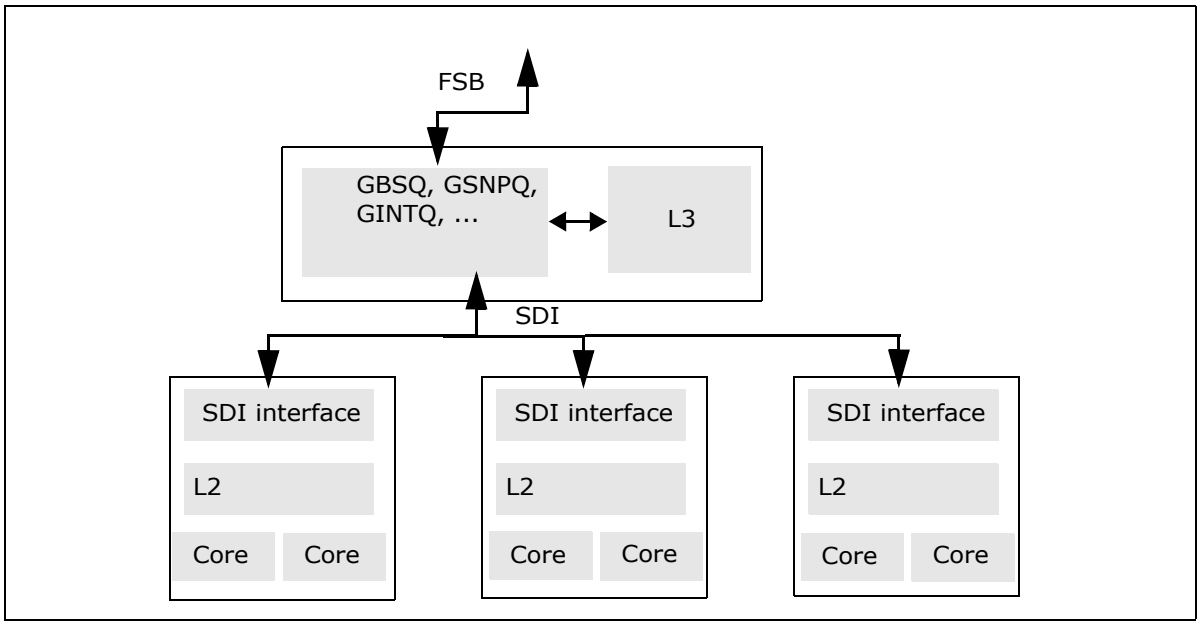


Figure 18-56. Block Diagram of Intel Xeon Processor 7400 Series

Almost all of the performance monitoring capabilities available to processor cores with the same CPUID signatures (see Section 18.1 and Section 18.6.4) apply to Intel Xeon processor 7100 series. The MSRs used by performance monitoring interface are shared between two logical processors in the same processor core.

The performance monitoring capabilities available to processor with DisplayFamily_DisplayModel signature 06_17H also apply to Intel Xeon processor 7400 series. Each processor core provides its own set of MSRs for performance monitoring interface.

The IOQ_allocation and IOQ_active_entries events are not supported in Intel Xeon processor 7100 series and 7400 series. Additional performance monitoring capabilities applicable to the L3/caching bus controller sub-system are described in this section.

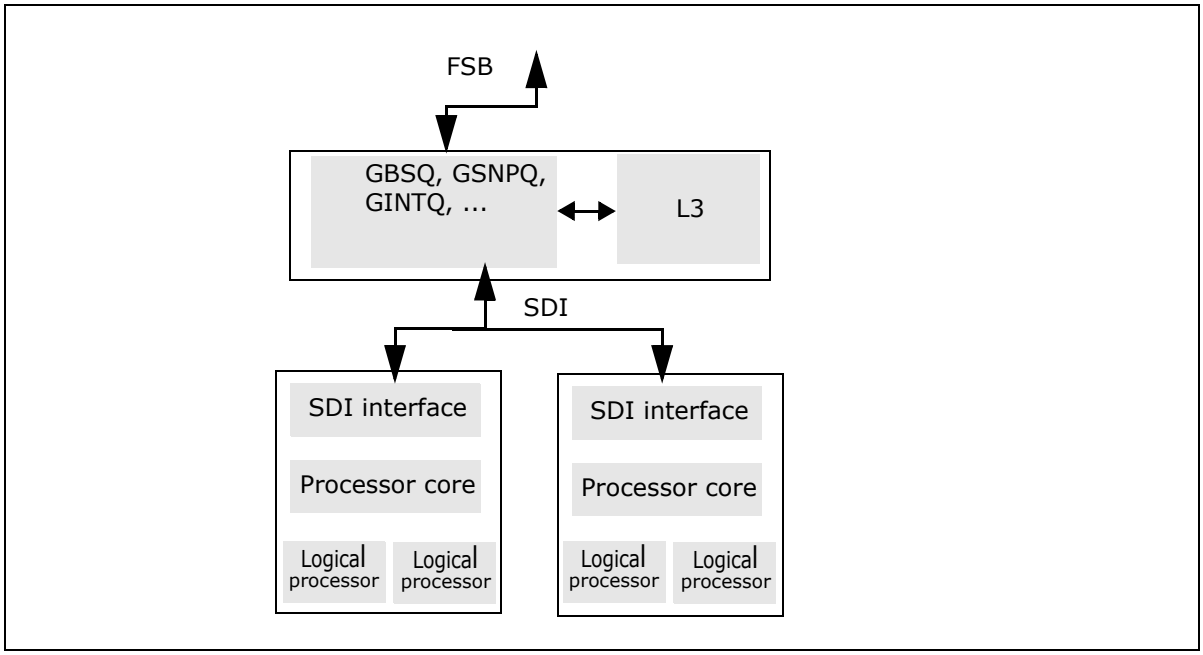


Figure 18-57. Block Diagram of Intel Xeon Processor 7100 Series

18.6.7.1 Overview of Performance Monitoring with L3/Caching Bus Controller

The facility for monitoring events consists of a set of dedicated model-specific registers (MSRs). There are eight event select/counting MSRs that are dedicated to counting events associated with specified microarchitectural conditions. Programming of these MSRs requires using RDMSR/WRMSR instructions with 64-bit values. In addition, an MSR MSR_EMON_L3_GL_CTL provides simplified interface to control freezing, resetting, re-enabling operation of any combination of these event select/counting MSRs.

The eight MSRs dedicated to count occurrences of specific conditions are further divided to count three sub-classes of microarchitectural conditions:

- Two MSRs (MSR_EMON_L3_CTR_CTL0 and MSR_EMON_L3_CTR_CTL1) are dedicated to counting GBSQ events. Up to two GBSQ events can be programmed and counted simultaneously.
- Two MSRs (MSR_EMON_L3_CTR_CTL2 and MSR_EMON_L3_CTR_CTL3) are dedicated to counting GSNPQ events. Up to two GBSQ events can be programmed and counted simultaneously.
- Four MSRs (MSR_EMON_L3_CTR_CTL4, MSR_EMON_L3_CTR_CTL5, MSR_EMON_L3_CTR_CTL6, and MSR_EMON_L3_CTR_CTL7) are dedicated to counting external bus operations.

The bit fields in each of eight MSRs share the following common characteristics:

- Bits 63:32 is the event control field that includes an event mask and other bit fields that control counter operation. The event mask field specifies details of the microarchitectural condition, and its definition differs across GBSQ, GSNPQ, FSB.
- Bits 31:0 is the event count field. If the specified condition is met during each relevant clock domain of the event logic, the matched condition signals the counter logic to increment the associated event count field. The lower 32-bits of these 8 MSRs at addresses 107CC through 107D3 are treated as 32 bit performance counter registers.

In Dual-Core Intel Xeon processor 7100 series, the uncore performance counters can be accessed using RDPMC instruction with the index starting from 18 through 25. The EDX register returns zero when reading these 8 PMCs.

In Intel Xeon processor 7400 series, RDPMC with ECX between 2 and 9 can be used to access the eight uncore performance counter/control registers.

18.6.7.2 GBSQ Event Interface

The layout of MSR_EMON_L3_CTR_CTL0 and MSR_EMON_L3_CTR_CTL1 is given in Figure 18-58. Counting starts after software writes a non-zero value to one or more of the upper 32 bits.

The event mask field (bits 58:32) consists of the following eight attributes:

- Agent_Select (bits 35:32): The definition of this field differs slightly between Intel Xeon processor 7100 and 7400.

  For Intel Xeon processor 7100 series, each bit specifies a logical processor in the physical package. The lower two bits corresponds to two logical processors in the first processor core, the upper two bits corresponds to two logical processors in the second processor core. 0FH encoding matches transactions from any logical processor.

  For Intel Xeon processor 7400 series, each bit of [34:32] specifies the SDI logic of a dual-core module as the originator of the transaction. A value of 0111B in bits [35:32] specifies transaction from any processor core.

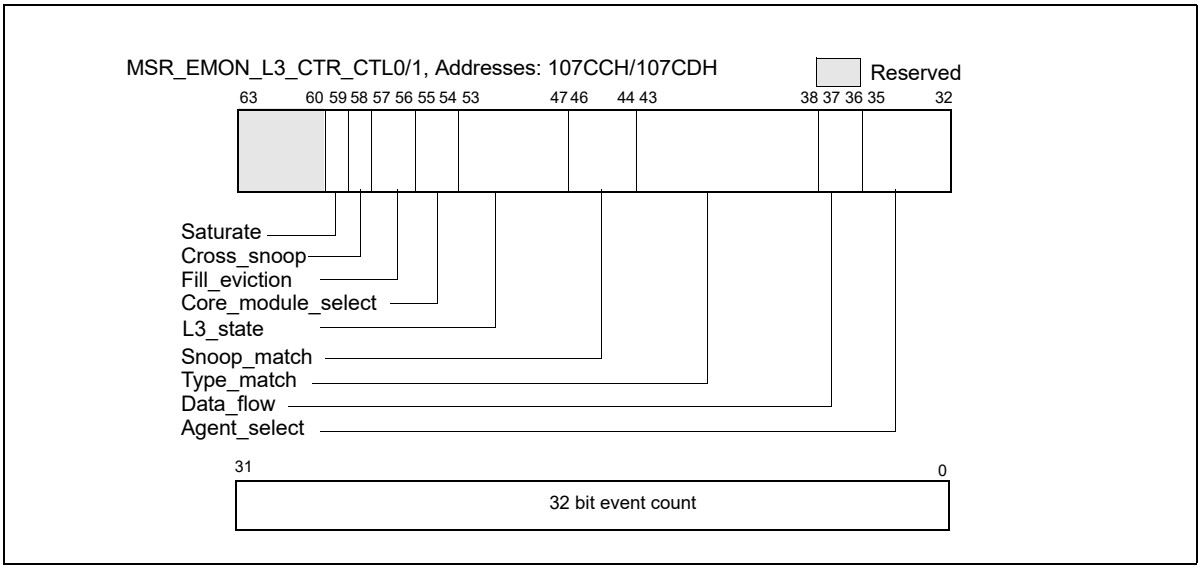


Figure 18-58. MSR_EMON_L3_CTR_CTL0/1, Addresses: 107CCH/107CDH

- Data_Flow (bits 37:36): Bit 36 specifies demand transactions, bit 37 specifies prefetch transactions.
- Type_Match (bits 43:38): Specifies transaction types. If all six bits are set, event count will include all transaction types.
- Snoop_Match: (bits 46:44): The three bits specify (in ascending bit position) clean snoop result, HIT snoop result, and HITM snoop results respectively.
- L3_State (bits 53:47): Each bit specifies an L2 coherency state.
- Core_Module_Select (bits 55:54): The valid encodings for L3 lookup differ slightly between Intel Xeon processor 7100 and 7400.

  For Intel Xeon processor 7100 series,

  — 00B: Match transactions from any core in the physical package

  — 01B: Match transactions from this core only

  — 10B: Match transactions from the other core in the physical package

  — 11B: Match transaction from both cores in the physical package

  For Intel Xeon processor 7400 series,

  — 00B: Match transactions from any dual-core module in the physical package

  — 01B: Match transactions from this dual-core module only

  — 10B: Match transactions from either one of the other two dual-core modules in the physical package

  - 11B: Match transaction from more than one dual-core modules in the physical package
- Fill_Eviction (bits 57:56): The valid encodings are
  - 00B: Match any transactions
  - 01B: Match transactions that fill L3
  - 10B: Match transactions that fill L3 without an eviction
  - 11B: Match transaction fill L3 with an eviction
- Cross_Snoop (bit 58): The encodings are \
  - 0B: Match any transactions
  - 1B: Match cross snoop transactions

For each counting clock domain, if all eight attributes match, event logic signals to increment the event count field.

18.6.7.3 GSNPQ Event Interface

The layout of MSR_EMON_L3_CTR_CTL2 and MSR_EMON_L3_CTR_CTL3 is given in Figure 18-59. Counting starts after software writes a non-zero value to one or more of the upper 32 bits.

The event mask field (bits 58:32) consists of the following six attributes:

- Agent_Select (bits 37:32): The definition of this field differs slightly between Intel Xeon processor 7100 and 7400.
- For Intel Xeon processor 7100 series, each of the lowest 4 bits specifies a logical processor in the physical package. The lowest two bits corresponds to two logical processors in the first processor core, the next two bits corresponds to two logical processors in the second processor core. Bit 36 specifies other symmetric agent transactions. Bit 37 specifies central agent transactions. 3FH encoding matches transactions from any logical processor.

  For Intel Xeon processor 7400 series, each of the lowest 3 bits specifies a dual-core module in the physical package. Bit 37 specifies central agent transactions.
- Type_Match (bits 43:38): Specifies transaction types. If all six bits are set, event count will include any transaction types.
- Snoop_Match: (bits 46:44): The three bits specify (in ascending bit position) clean snoop result, HIT snoop result, and HITM snoop results respectively.
- L2_State (bits 53:47): Each bit specifies an L3 coherency state.
- Core_Module_Select (bits 56:54): Bit 56 enables Core_Module_Select matching. If bit 56 is clear, Core_Module_Select encoding is ignored. The valid encodings for the lower two bits (bit 55, 54) differ slightly between Intel Xeon processor 7100 and 7400.

  For Intel Xeon processor 7100 series, if bit 56 is set, the valid encodings for the lower two bits (bit 55, 54) are
  - 00B: Match transactions from only one core (irrespective which core) in the physical package
  - 01B: Match transactions from this core and not the other core
  - 10B: Match transactions from the other core in the physical package, but not this core
  - 11B: Match transaction from both cores in the physical package

  For Intel Xeon processor 7400 series, if bit 56 is set, the valid encodings for the lower two bits (bit 55, 54) are
  - 00B: Match transactions from only one dual-core module (irrespective which module) in the physical package.
  - 01B: Match transactions from one or more dual-core modules.
  - 10B: Match transactions from two or more dual-core modules.
  - 11B: Match transaction from all three dual-core modules in the physical package.
- Block_Snoop (bit 57): specifies blocked snoop.

For each counting clock domain, if all six attributes match, event logic signals to increment the event count field.

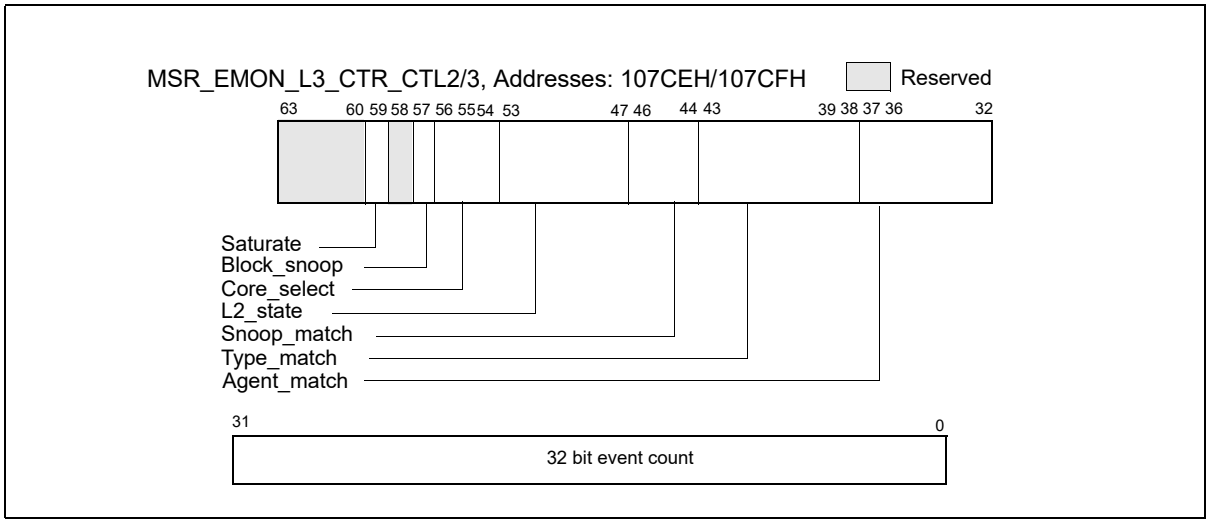


Figure 18-59. MSR_EMON_L3_CTR_CTL2/3, Addresses: 107CEH/107CFH

18.6.7.4 FSB Event Interface

The layout of MSR_EMON_L3_CTR_CTL4 through MSR_EMON_L3_CTR_CTL7 is given in Figure 18-60. Counting starts after software writes a non-zero value to one or more of the upper 32 bits.

The event mask field (bits 58:32) is organized as follows:

- Bit 58: must set to 1.
- FSB_Submask (bits 57:32): Specifies FSB-specific sub-event mask.

The FSB sub-event mask defines a set of independent attributes. The event logic signals to increment the associated event count field if one of the attribute matches. Some of the sub-event mask bit counts durations. A duration event increments at most once per cycle.

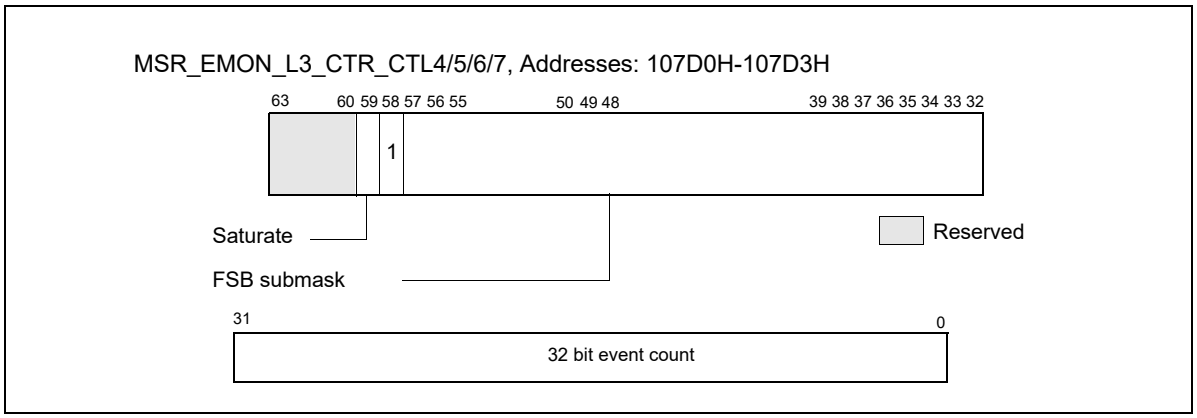


Figure 18-60. MSR_EMON_L3_CTR_CTL4/5/6/7, Addresses: 107D0H-107D3H

18.6.7.4.1 FSB Sub-Event Mask Interface

- FSB_type (bit 37:32): Specifies different FSB transaction types originated from this physical package.
- FSB_L_clear (bit 38): Count clean snoop results from any source for transaction originated from this physical package.
- FSB_L_hit (bit 39): Count HIT snoop results from any source for transaction originated from this physical package.

- FSB_L_hitm (bit 40): Count HITM snoop results from any source for transaction originated from this physical package.
- FSB_L_defer (bit 41): Count DEFER responses to this processor's transactions.
- FSB_L_retry (bit 42): Count RETRY responses to this processor's transactions.
- FSB_L_snoop_stall (bit 43): Count snoop stalls to this processor's transactions.
- FSB_DBSY (bit 44): Count DBSY assertions by this processor (without a concurrent DRDY).
- FSB_DRDY (bit 45): Count DRDY assertions by this processor.
- FSB_BNR (bit 46): Count BNR assertions by this processor.
- FSB_IOQ_empty (bit 47): Counts each bus clocks when the IOQ is empty.
- FSB_IOQ_full (bit 48): Counts each bus clocks when the IOQ is full.
- FSB_IOQ_active (bit 49): Counts each bus clocks when there is at least one entry in the IOQ.
- FSB_WW_data (bit 50): Counts back-to-back write transaction's data phase.
- FSB_WW_issue (bit 51): Counts back-to-back write transaction request pairs issued by this processor.
- FSB_WR_issue (bit 52): Counts back-to-back write-read transaction request pairs issued by this processor.
- FSB_RW_issue (bit 53): Counts back-to-back read-write transaction request pairs issued by this processor.
- FSB_other_DBSY (bit 54): Count DBSY assertions by another agent (without a concurrent DRDY).
- FSB_other_DRDY (bit 55): Count DRDY assertions by another agent.
- FSB_other_snoop_stall (bit 56): Count snoop stalls on the FSB due to another agent.
- FSB_other_BNR (bit 57): Count BNR assertions from another agent.

18.6.7.5 Common Event Control Interface

The MSR_EMON_L3_GL_CTL MSR provides simplified access to query overflow status of the GBSQ, GSNPQ, FSB event counters. It also provides control bit fields to freeze, unfreeze, or reset those counters. The following bit fields are supported:

- GL_freeze_cmd (bit 0): Freeze the event counters specified by the GL_event_select field.
- GL_unfreeze_cmd (bit 1): Unfreeze the event counters specified by the GL_event_select field.
- GL_reset_cmd (bit 2): Clear the event count field of the event counters specified by the GL_event_select field. The event select field is not affected.
- GL_event_select (bit 23:16): Selects one or more event counters to subject to specified command operations indicated by bits 2:0. Bit 16 corresponds to MSR_EMON_L3_CTR_CTL0, bit 23 corresponds to MSR_EMON_L3_CTR_CTL7.
- GL_event_status (bit 55:48): Indicates the overflow status of each event counters. Bit 48 corresponds to MSR_EMON_L3_CTR_CTL0, bit 55 corresponds to MSR_EMON_L3_CTR_CTL7.

In the event control field (bits 63:32) of each MSR, if the saturate control (bit 59, see Figure 18-58 for example) is set, the event logic forces the value FFFF_FFFFH into the event count field instead of incrementing it.

18.6.8 Performance Monitoring (P6 Family Processor)

The P6 family processors provide two 40-bit performance counters, allowing two types of events to be monitored simultaneously. These can either count events or measure duration. When counting events, a counter increments each time a specified event takes place or a specified number of events takes place. When measuring duration, it counts the number of processor clocks that occur while a specified condition is true. The counters can count events or measure durations that occur at any privilege level.

Table 19-39, Chapter 19, lists the events that can be counted with the P6 family performance monitoring counters.

NOTE

The performance-monitoring events listed in Chapter 19 are intended to be used as guides for performance tuning. Counter values reported are not guaranteed to be accurate and should be used as a relative guide for tuning. Known discrepancies are documented where applicable.

The performance-monitoring counters are supported by four MSRs: the performance event select MSRs (PerfEvtSel0 and PerfEvtSel1) and the performance counter MSRs (PerfCtr0 and PerfCtr1). These registers can be read from and written to using the RDMSR and WRMSR instructions, respectively. They can be accessed using these instructions only when operating at privilege level 0. The PerfCtr0 and PerfCtr1 MSRs can be read from any privilege level using the RDPMC (read performance-monitoring counters) instruction.

NOTE

The PerfEvtSel0, PerfEvtSel1, PerfCtr0, and PerfCtr1 MSRs and the events listed in Table 19-39 are model-specific for P6 family processors. They are not guaranteed to be available in other IA-32 processors.

18.6.8.1 PerfEvtSel0 and PerfEvtSel1 MSRs

The PerfEvtSel0 and PerfEvtSel1 MSRs control the operation of the performance-monitoring counters, with one register used to set up each counter. They specify the events to be counted, how they should be counted, and the privilege levels at which counting should take place. Figure 18-61 shows the flags and fields in these MSRs.

The functions of the flags and fields in the PerfEvtSel0 and PerfEvtSel1 MSRs are as follows:

- **Event select field (bits 0 through 7) —** Selects the event logic unit to detect certain microarchitectural conditions (see Table 19-39, for a list of events and their 8-bit codes).
- **Unit mask (UMASK) field (bits 8 through 15) —** Further qualifies the event logic unit selected in the event select field to detect a specific microarchitectural condition. For example, for some cache events, the mask is used as a MESI-protocol qualifier of cache states (see Table 19-39).

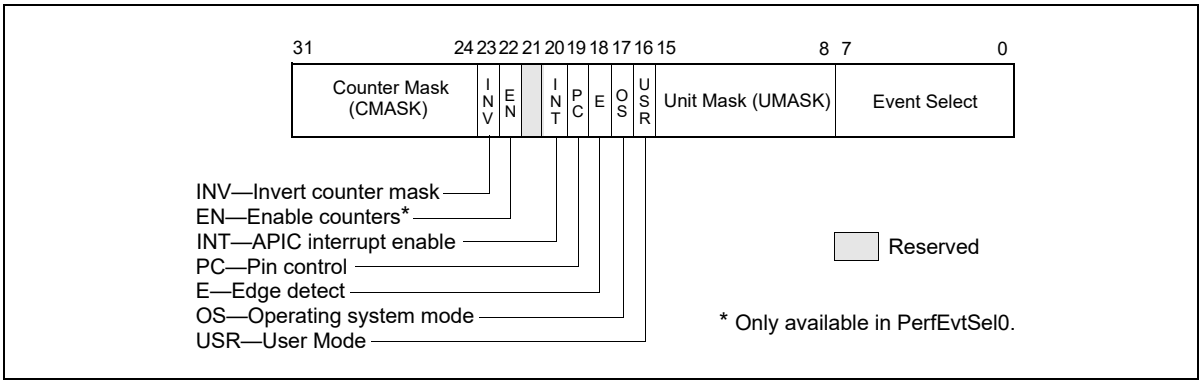


Figure 18-61. PerfEvtSel0 and PerfEvtSel1 MSRs

- **USR (user mode) flag (bit 16) —** Specifies that events are counted only when the processor is operating at privilege levels 1, 2 or 3. This flag can be used in conjunction with the OS flag.
- **OS (operating system mode) flag (bit 17) —** Specifies that events are counted only when the processor is operating at privilege level 0. This flag can be used in conjunction with the USR flag.
- **E (edge detect) flag (bit 18) —** Enables (when set) edge detection of events. The processor counts the number of deasserted to asserted transitions of any condition that can be expressed by the other fields. The mechanism is limited in that it does not permit back-to-back assertions to be distinguished. This mechanism allows software to measure not only the fraction of time spent in a particular state, but also the average length of time spent in such a state (for example, the time spent waiting for an interrupt to be serviced).

- **PC (pin control) flag (bit 19) —** When set, the processor toggles the PM*i* pins and increments the counter when performance-monitoring events occur; when clear, the processor toggles the PM*i* pins when the counter overflows. The toggling of a pin is defined as assertion of the pin for a single bus clock followed by deassertion.
- **INT (APIC interrupt enable) flag (bit 20) —** When set, the processor generates an exception through its local APIC on counter overflow.
- **EN (Enable Counters) Flag (bit 22) —** This flag is only present in the PerfEvtSel0 MSR. When set, performance counting is enabled in both performance-monitoring counters; when clear, both counters are disabled.
- **INV (invert) flag (bit 23) —** When set, inverts the counter-mask (CMASK) comparison, so that both greater than or equal to and less than comparisons can be made (0: greater than or equal; 1: less than). Note if counter-mask is programmed to zero, INV flag is ignored.
- **Counter mask (CMASK) field (bits 24 through 31) —** When nonzero, the processor compares this mask to the number of events counted during a single cycle. If the event count is greater than or equal to this mask, the counter is incremented by one. Otherwise the counter is not incremented. This mask can be used to count events only if multiple occurrences happen per clock (for example, two or more instructions retired per clock). If the counter-mask field is 0, then the counter is incremented each cycle by the number of events that occurred that cycle.

18.6.8.2 PerfCtr0 and PerfCtr1 MSRs

The performance-counter MSRs (PerfCtr0 and PerfCtr1) contain the event or duration counts for the selected events being counted. The RDPMC instruction can be used by programs or procedures running at any privilege level and in virtual-8086 mode to read these counters. The PCE flag in control register CR4 (bit 8) allows the use of this instruction to be restricted to only programs and procedures running at privilege level 0.

The RDPMC instruction is not serializing or ordered with other instructions. Thus, it does not necessarily wait until all previous instructions have been executed before reading the counter. Similarly, subsequent instructions may begin execution before the RDPMC instruction operation is performed.

Only the operating system, executing at privilege level 0, can directly manipulate the performance counters, using the RDMSR and WRMSR instructions. A secure operating system would clear the PCE flag during system initialization to disable direct user access to the performance-monitoring counters, but provide a user-accessible programming interface that emulates the RDPMC instruction.

The WRMSR instruction cannot arbitrarily write to the performance-monitoring counter MSRs (PerfCtr0 and PerfCtr1). Instead, the lower-order 32 bits of each MSR may be written with any value, and the high-order 8 bits are sign-extended according to the value of bit 31. This operation allows writing both positive and negative values to the performance counters.

18.6.8.3 Starting and Stopping the Performance-Monitoring Counters

The performance-monitoring counters are started by writing valid setup information in the PerfEvtSel0 and/or PerfEvtSel1 MSRs and setting the enable counters flag in the PerfEvtSel0 MSR. If the setup is valid, the counters begin counting following the execution of a WRMSR instruction that sets the enable counter flag. The counters can be stopped by clearing the enable counters flag or by clearing all the bits in the PerfEvtSel0 and PerfEvtSel1 MSRs. Counter 1 alone can be stopped by clearing the PerfEvtSel1 MSR.

18.6.8.4 Event and Time-Stamp Monitoring Software

To use the performance-monitoring counters and time-stamp counter, the operating system needs to provide an event-monitoring device driver. This driver should include procedures for handling the following operations:

- Feature checking.
- Initialize and start counters.
- Stop counters.
- Read the event counters.
- Read the time-stamp counter.

The event monitor feature determination procedure must check whether the current processor supports the performance-monitoring counters and time-stamp counter. This procedure compares the family and model of the processor returned by the CPUID instruction with those of processors known to support performance monitoring. (The Pentium and P6 family processors support performance counters.) The procedure also checks the MSR and TSC flags returned to register EDX by the CPUID instruction to determine if the MSRs and the RDTSC instruction are supported.

The initialize and start counters procedure sets the PerfEvtSel0 and/or PerfEvtSel1 MSRs for the events to be counted and the method used to count them and initializes the counter MSRs (PerfCtr0 and PerfCtr1) to starting counts. The stop counters procedure stops the performance counters (see Section 18.6.8.3, "Starting and Stopping the Performance-Monitoring Counters").

The read counters procedure reads the values in the PerfCtr0 and PerfCtr1 MSRs, and a read time-stamp counter procedure reads the time-stamp counter. These procedures would be provided in lieu of enabling the RDTSC and RDPMC instructions that allow application code to read the counters.

18.6.8.5 Monitoring Counter Overflow

The P6 family processors provide the option of generating a local APIC interrupt when a performance-monitoring counter overflows. This mechanism is enabled by setting the interrupt enable flag in either the PerfEvtSel0 or the PerfEvtSel1 MSR. The primary use of this option is for statistical performance sampling.

To use this option, the operating system should do the following things on the processor for which performance events are required to be monitored:

- Provide an interrupt vector for handling the counter-overflow interrupt.
- Initialize the APIC PERF local vector entry to enable handling of performance-monitor counter overflow events.
- Provide an entry in the IDT that points to a stub exception handler that returns without executing any instructions.
- Provide an event monitor driver that provides the actual interrupt handler and modifies the reserved IDT entry to point to its interrupt routine.

When interrupted by a counter overflow, the interrupt handler needs to perform the following actions:

- Save the instruction pointer (EIP register), code-segment selector, TSS segment selector, counter values and other relevant information at the time of the interrupt.
- Reset the counter to its initial setting and return from the interrupt.

An event monitor application utility or another application program can read the information collected for analysis of the performance of the profiled application.

18.6.9 Performance Monitoring (Pentium Processors)

The Pentium processor provides two 40-bit performance counters, which can be used to count events or measure duration. The counters are supported by three MSRs: the control and event select MSR (CESR) and the performance counter MSRs (CTR0 and CTR1). These can be read from and written to using the RDMSR and WRMSR instructions, respectively. They can be accessed using these instructions only when operating at privilege level 0.

Each counter has an associated external pin (PM0/BP0 and PM1/BP1), which can be used to indicate the state of the counter to external hardware.

NOTES

The CESR, CTR0, and CTR1 MSRs and the events listed in Table 19-40 are model-specific for the Pentium processor.

The performance-monitoring events listed in Chapter 19 are intended to be used as guides for performance tuning. Counter values reported are not guaranteed to be accurate and should be used as a relative guide for tuning. Known discrepancies are documented where applicable.

18.6.9.1 Control and Event Select Register (CESR)

The 32-bit control and event select MSR (CESR) controls the operation of performance-monitoring counters CTR0 and CTR1 and the associated pins (see Figure 18-62). To control each counter, the CESR register contains a 6-bit event select field (ES0 and ES1), a pin control flag (PC0 and PC1), and a 3-bit counter control field (CC0 and CC1). The functions of these fields are as follows:

- **ES0 and ES1 (event select) fields (bits 0-5, bits 16-21) —** Selects (by entering an event code in the field) up to two events to be monitored. See Table 19-40 for a list of available event codes.

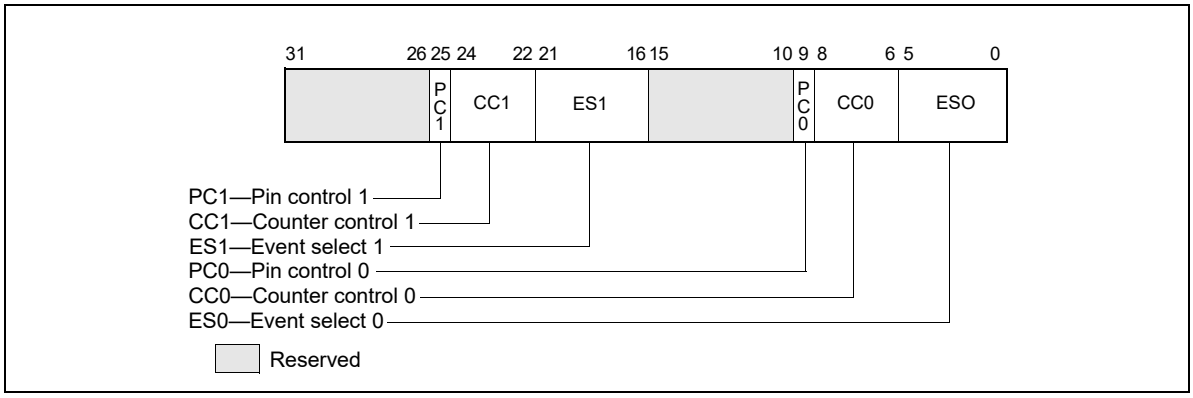


Figure 18-62. CESR MSR (Pentium Processor Only)

- **CC0 and CC1 (counter control) fields (bits 6-8, bits 22-24) —** Controls the operation of the counter. Control codes are as follows:

  000 — Count nothing (counter disabled).

  001 — Count the selected event while CPL is 0, 1, or 2.

  010 — Count the selected event while CPL is 3.

  011 — Count the selected event regardless of CPL.

  100 — Count nothing (counter disabled).

  101 — Count clocks (duration) while CPL is 0, 1, or 2.

  110 — Count clocks (duration) while CPL is 3.

  111 — Count clocks (duration) regardless of CPL.

  The highest order bit selects between counting events and counting clocks (duration); the middle bit enables counting when the CPL is 3; and the low-order bit enables counting when the CPL is 0, 1, or 2.

- **PC0 and PC1 (pin control) flags (bits 9, 25) —** Selects the function of the external performance-monitoring counter pin (PM0/BP0 and PM1/BP1). Setting one of these flags to 1 causes the processor to assert its associated pin when the counter has overflowed; setting the flag to 0 causes the pin to be asserted when the counter has been incremented. These flags permit the pins to be individually programmed to indicate the overflow or incremented condition. The external signalling of the event on the pins will lag the internal event by a few clocks as the signals are latched and buffered.

While a counter need not be stopped to sample its contents, it must be stopped and cleared or preset before switching to a new event. It is not possible to set one counter separately. If only one event needs to be changed, the CESR register must be read, the appropriate bits modified, and all bits must then be written back to CESR. At reset, all bits in the CESR register are cleared.

18.6.9.2 Use of the Performance-Monitoring Pins

When performance-monitor pins PM0/BP0 and/or PM1/BP1 are configured to indicate when the performance-monitor counter has incremented and an "occurrence event" is being counted, the associated pin is asserted (high) each time the event occurs. When a "duration event" is being counted, the associated PM pin is asserted for the

entire duration of the event. When the performance-monitor pins are configured to indicate when the counter has overflowed, the associated PM pin is asserted when the counter has overflowed.

When the PM0/BP0 and/or PM1/BP1 pins are configured to signal that a counter has incremented, it should be noted that although the counters may increment by 1 or 2 in a single clock, the pins can only indicate that the event occurred. Moreover, since the internal clock frequency may be higher than the external clock frequency, a single external clock may correspond to multiple internal clocks.

A "count up to" function may be provided when the event pin is programmed to signal an overflow of the counter. Because the counters are 40 bits, a carry out of bit 39 indicates an overflow. A counter may be preset to a specific value less then $2^{40} - 1$. After the counter has been enabled and the prescribed number of events has transpired, the counter will overflow.

Approximately 5 clocks later, the overflow is indicated externally and appropriate action, such as signaling an interrupt, may then be taken.

The PM0/BP0 and PM1/BP1 pins also serve to indicate breakpoint matches during in-circuit emulation, during which time the counter increment or overflow function of these pins is not available. After RESET, the PM0/BP0 and PM1/BP1 pins are configured for performance monitoring, however a hardware debugger may reconfigure these pins to indicate breakpoint matches.

18.6.9.3 Events Counted

Events that performance-monitoring counters can be set to count and record (using CTR0 and CTR1) are divided in two categories: occurrence and duration:

- **Occurrence events** — Counts are incremented each time an event takes place. If PM0/BP0 or PM1/BP1 pins are used to indicate when a counter increments, the pins are asserted each clock counters increment. But if an event happens twice in one clock, the counter increments by 2 (the pins are asserted only once).
- **Duration events** — Counters increment the total number of clocks that the condition is true. When used to indicate when counters increment, PM0/BP0 and/or PM1/BP1 pins are asserted for the duration.

18.7 COUNTING CLOCKS

The count of cycles, also known as clockticks, forms the basis for measuring how long a program takes to execute. Clockticks are also used as part of efficiency ratios like cycles per instruction (CPI). Processor clocks may stop ticking under circumstances like the following:

- The processor is halted when there is nothing for the CPU to do. For example, the processor may halt to save power while the computer is servicing an I/O request. When Intel Hyper-Threading Technology is enabled, both logical processors must be halted for performance-monitoring counters to be powered down.
- The processor is asleep as a result of being halted or because of a power-management scheme. There are different levels of sleep. In the some deep sleep levels, the time-stamp counter stops counting.

In addition, processor core clocks may undergo transitions at different ratios relative to the processor's bus clock frequency. Some of the situations that can cause processor core clock to undergo frequency transitions include:

- TM2 transitions.
- Enhanced Intel SpeedStep Technology transitions (P-state transitions).

For Intel processors that support TM2, the processor core clocks may operate at a frequency that differs from the Processor Base frequency (as indicated by processor frequency information reported by CPUID instruction). See Section 18.7.2 for more detail.

Due to the above considerations there are several important clocks referenced in this manual:

- **Base Clock —** The frequency of this clock is the frequency of the processor when the processor is not in turbo mode, and not being throttled via Intel SpeedStep.
- **Maximum Clock —** This is the maximum frequency of the processor when turbo mode is at the highest point.
- **Bus Clock —** These clockticks increment at a fixed frequency and help coordinate the bus on some systems.

- **Core Crystal Clock —** This is a clock that runs at fixed frequency; it coordinates the clocks on all packages across the system.
- **Non-halted Clockticks —** Measures clock cycles in which the specified logical processor is not halted and is not in any power-saving state. When Intel Hyper-Threading Technology is enabled, ticks can be measured on a per-logical-processor basis. There are also performance events on dual-core processors that measure clockticks per logical processor when the processor is not halted.
- **Non-sleep Clockticks —** Measures clock cycles in which the specified physical processor is not in a sleep mode or in a power-saving state. These ticks cannot be measured on a logical-processor basis.
- **Time-stamp Counter —** See Section 17.17, "Time-Stamp Counter".
- **Reference Clockticks —** TM2 or Enhanced Intel SpeedStep technology are two examples of processor features that can cause processor core clockticks to represent non-uniform tick intervals due to change of bus ratios. Performance events that counts clockticks of a constant reference frequency was introduced Intel Core Duo and Intel Core Solo processors. The mechanism is further enhanced on processors based on Intel Core microarchitecture.

Some processor models permit clock cycles to be measured when the physical processor is not in deep sleep (by using the time-stamp counter and the RDTSC instruction). Note that such ticks cannot be measured on a per-logical-processor basis. See Section 17.17, "Time-Stamp Counter," for detail on processor capabilities.

The first two methods use performance counters and can be set up to cause an interrupt upon overflow (for sampling). They may also be useful where it is easier for a tool to read a performance counter than to use a time stamp counter (the timestamp counter is accessed using the RDTSC instruction).

For applications with a significant amount of I/O, there are two ratios of interest:

- **Non-halted CPI —** Non-halted clockticks/instructions retired measures the CPI for phases where the CPU was being used. This ratio can be measured on a logical-processor basis when Intel Hyper-Threading Technology is enabled.
- **Nominal CPI —** Time-stamp counter ticks/instructions retired measures the CPI over the duration of a program, including those periods when the machine halts while waiting for I/O.

18.7.1 Non-Halted Reference Clockticks

Software can use UnHalted Reference Cycles on either a general purpose performance counter using event mask 0x3C and umask 0x01 or on fixed function performance counter 2 to count at a constant rate. These events count at a consistent rate irrespective of P-state, TM2, or frequency transitions that may occur to the processor. The UnHalted Reference Cycles event may count differently on the general purpose event and fixed counter.

18.7.2 Cycle Counting and Opportunistic Processor Operation

As a result of the state transitions due to opportunistic processor performance operation (see Chapter 14, "Power and Thermal Management"), a logical processor or a processor core can operate at frequency different from the Processor Base frequency.

The following items are expected to hold true irrespective of when opportunistic processor operation causes state transitions:

- The time stamp counter operates at a fixed-rate frequency of the processor.
- The IA32_MPERF counter increments at a fixed frequency irrespective of any transitions caused by opportunistic processor operation.
- The IA32_FIXED_CTR2 counter increments at the same TSC frequency irrespective of any transitions caused by opportunistic processor operation.
- The Local APIC timer operation is unaffected by opportunistic processor operation.
- The TSC, IA32_MPERF, and IA32_FIXED_CTR2 operate at close to the maximum non-turbo frequency, which is equal to the product of scalable bus frequency and maximum non-turbo ratio.

18.7.3 Determining the Processor Base Frequency

For Intel processors in which the nominal core crystal clock frequency is enumerated in CPUID.15H.ECX and the core crystal clock ratio is encoded in CPUID.15H (see Table 3-8 "Information Returned by CPUID Instruction"), the nominal TSC frequency can be determined by using the following equation:

Nominal TSC frequency = (CPUID.15H.ECX[31:0] * CPUID.15H.EBX[31:0]) ÷ CPUID.15H.EAX[31:0]

For Intel processors in which CPUID.15H.EBX[31:0] ÷ CPUID.0x15.EAX[31:0] is enumerated but CPUID.15H.ECX is not enumerated, Table 18-75 can be used to look up the nominal core crystal clock frequency.

Table 18-75. Nominal Core Crystal Clock Frequency

Processor Families/Processor Number Series[1]	Nominal Core Crystal Clock Frequency
Future Intel® Xeon® processors with CPUID signature 06_55H.	25 MHz
6th and 7th generation Intel® Core™ processors (does not include Intel® Xeon® processors).	24 MHz
Next Generation Intel® Atom™ processors based on Goldmont Microarchitecture with CPUID signature 06_5CH (does not include Intel Xeon processors).	19.2 MHz

NOTES:

1. For any processor in which CPUID.15H is enumerated and MSR_PLATFORM_INFO[15:8] (which gives the scalable bus frequency) is available, a more accurate frequency can be obtained by using CPUID.15H.

18.7.3.1 For Intel® Processors Based on Microarchitecture Code Name Sandy Bridge, Ivy Bridge, Haswell and Broadwell

The scalable bus frequency is encoded in the bit field MSR_PLATFORM_INFO[15:8] and the nominal TSC frequency can be determined by multiplying this number by a bus speed of 100 MHz.

18.7.3.2 For Intel® Processors Based on Microarchitecture Code Name Nehalem

The scalable bus frequency is encoded in the bit field MSR_PLATFORM_INFO[15:8] and the nominal TSC frequency can be determined by multiplying this number by a bus speed of 133.33 MHz.

18.7.3.3 For Intel® Atom™ Processors Based on the Silvermont Microarchitecture (Including Intel Processors Based on Airmont Microarchitecture)

The scalable bus frequency is encoded in the bit field MSR_PLATFORM_INFO[15:8] and the nominal TSC frequency can be determined by multiplying this number by the scalable bus frequency. The scalable bus frequency is encoded in the bit field MSR_FSB_FREQ[2:0] for Intel Atom processors based on the Silvermont microarchitecture, and in bit field MSR_FSB_FREQ[3:0] for processors based on the Airmont microarchitecture; see Chapter 2, "Model-Specific Registers (MSRs)" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 4*.

18.7.3.4 For Intel® Core™ 2 Processor Family and for Intel® Xeon® Processors Based on Intel Core Microarchitecture

For processors based on Intel Core microarchitecture, the scalable bus frequency is encoded in the bit field MSR_FSB_FREQ[2:0] at (0CDH), see Chapter 2, "Model-Specific Registers (MSRs)" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 4*. The maximum resolved bus ratio can be read from the following bit field:

- If XE operation is disabled, the maximum resolved bus ratio can be read in MSR_PLATFORM_ID[12:8]. It corresponds to the Processor Base frequency.
- IF XE operation is enabled, the maximum resolved bus ratio is given in MSR_PERF_STATUS[44:40], it corresponds to the maximum XE operation frequency configured by BIOS.

XE operation of an Intel 64 processor is implementation specific. XE operation can be enabled only by BIOS. If MSR_PERF_STATUS[31] is set, XE operation is enabled. The MSR_PERF_STATUS[31] field is read-only.

18.8 IA32_PERF_CAPABILITIES MSR ENUMERATION

The layout of IA32_PERF_CAPABILITIES MSR is shown in Figure 18-63, it provides enumeration of a variety of interfaces:

- IA32_PERF_CAPABILITIES.LBR_FMT[bits 5:0]: encodes the LBR format, details are described in Section 17.4.8.1.
- IA32_PERF_CAPABILITIES.PEBSTrap[6]: Trap/Fault-like indicator of PEBS recording assist, see Section 18.6.2.4.2.
- IA32_PERF_CAPABILITIES.PEBSArchRegs[7]: Indicator of PEBS assist save architectural registers, see Section 18.6.2.4.2.
- IA32_PERF_CAPABILITIES.PEBS_FMT[bits 11:8]: Specifies the encoding of the layout of PEBS records, see Section 18.6.2.4.2.
- IA32_PERF_CAPABILITIES.SMM_FRZ[12]: Indicates IA32_DEBUGCTL.FREEZE_WHILE_SMM is supported if 1, see Section 18.8.1.
- IA32_PERF_CAPABILITIES.FULL_WRITE[13]: Indicates the processor supports IA32_A_PMCx interface for updating bits 32 and above of IA32_PMCx, see Section 18.2.5.

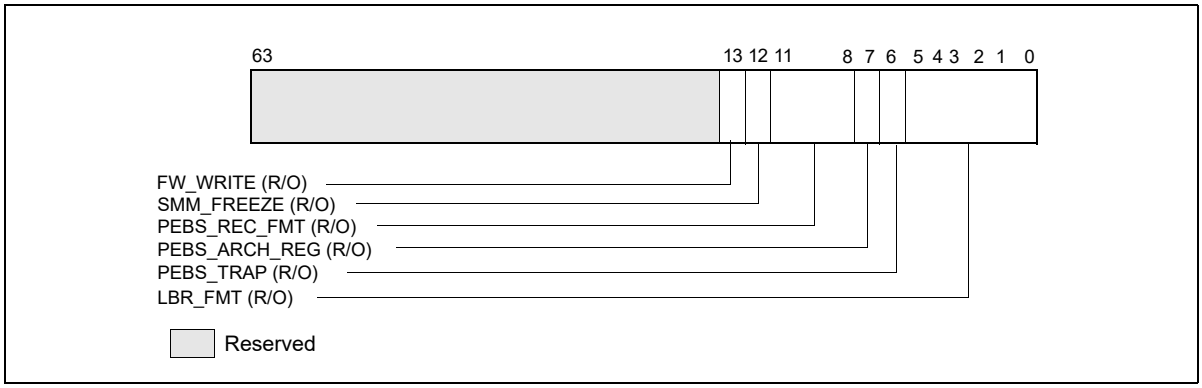


Figure 18-63. Layout of IA32_PERF_CAPABILITIES MSR

18.8.1 Filtering of SMM Handler Overhead

When performance monitoring facilities and/or branch profiling facilities (see Section 17.5, "Last Branch, Interrupt, and Exception Recording (Intel® Core™ 2 Duo and Intel® Atom™ Processors)") are enabled, these facilities capture event counts, branch records and branch trace messages occurring in a logical processor. The occurrence of interrupts, instruction streams due to various interrupt handlers all contribute to the results recorded by these facilities.

If CPUID.01H:ECX.PDCM[bit 15] is 1, the processor supports the IA32_PERF_CAPABILITIES MSR. If IA32_PERF_CAPABILITIES.FREEZE_WHILE_SMM[Bit 12] is 1, the processor supports the ability for system software using performance monitoring and/or branch profiling facilities to filter out the effects of servicing system management interrupts.

If the FREEZE_WHILE_SMM capability is enabled on a logical processor and after an SMI is delivered, the processor will clear all the enable bits of IA32_PERF_GLOBAL_CTRL, save a copy of the content of IA32_DEBUGCTL and disable LBR, BTF, TR, and BTS fields of IA32_DEBUGCTL before transferring control to the SMI handler.

The enable bits of IA32_PERF_GLOBAL_CTRL will be set to 1, the saved copy of IA32_DEBUGCTL prior to SMI delivery will be restored , after the SMI handler issues RSM to complete its servicing.

It is the responsibility of the SMM code to ensure the state of the performance monitoring and branch profiling facilities are preserved upon entry or until prior to exiting the SMM. If any of this state is modified due to actions by the SMM code, the SMM code is required to restore such state to the values present at entry to the SMM handler.

System software is allowed to set IA32_DEBUGCTL.FREEZE_WHILE_SMM_EN[bit 14] to 1 only supported as indicated by IA32_PERF_CAPABILITIES.FREEZE_WHILE_SMM[Bit 12] reporting 1.

15.Updates to Chapter 19, Volume 3B

Change bars show changes to Chapter 19 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3B:* System Programming Guide, Part 2.

Changes to this chapter: Addition of Section 19.2 "Performance Monitoring Events for Intel® Xeon® Processor Scalable Family". Updates to Table 19-27 "Performance Events for Silvermont Microarchitecture".

CHAPTER 19
PERFORMANCE-MONITORING EVENTS

This chapter lists the performance-monitoring events that can be monitored with the Intel 64 or IA-32 processors. The ability to monitor performance events and the events that can be monitored in these processors are mostly model-specific, except for architectural performance events, described in Section 19.1.

Non-architectural performance events (i.e. model-specific events) are listed for each generation of microarchitecture:

- Section 19.2 - Processors based on Skylake microarchitecture
- Section 19.3 - Processors based on Skylake and Kaby Lake microarchitectures
- Section 19.4 - Processors based on Knights Landing microarchitecture
- Section 19.5 - Processors based on Broadwell microarchitecture
- Section 19.6 - Processors based on Haswell microarchitecture
- Section 19.6.1 - Processors based on Haswell-E microarchitecture
- Section 19.7 - Processors based on Ivy Bridge microarchitecture
- Section 19.7.1 - Processors based on Ivy Bridge-E microarchitecture
- Section 19.8 - Processors based on Sandy Bridge microarchitecture
- Section 19.9 - Processors based on Intel® microarchitecture code name Nehalem
- Section 19.10 - Processors based on Intel® microarchitecture code name Westmere
- Section 19.11 - Processors based on Enhanced Intel® Core™ microarchitecture
- Section 19.12 - Processors based on Intel® Core™ microarchitecture
- Section 19.13 - Processors based on the Goldmont microarchitecture
- Section 19.14 - Processors based on the Silvermont microarchitecture
- Section 19.14.1 - Processors based on the Airmont microarchitecture
- Section 19.15 - 45 nm and 32 nm Intel® Atom™ Processors
- Section 19.16 - Intel® Core™ Solo and Intel® Core™ Duo processors
- Section 19.17 - Processors based on Intel NetBurst® microarchitecture
- Section 19.18 - Pentium® M family processors
- Section 19.19 - P6 family processors
- Section 19.20 - Pentium® processors

NOTE

These performance-monitoring events are intended to be used as guides for performance tuning. The counter values reported by the performance-monitoring events are approximate and believed to be useful as relative guides for tuning software. Known discrepancies are documented where applicable.

All performance event encodings not documented in the appropriate tables for the given processor are considered reserved, and their use will result in undefined counter updates with associated overflow actions.

The event tables listed this chapter provide information for tool developers to support architectural and non-architectural performance monitoring events. The tables are up to date at processor launch, but are subject to changes. The most up to date event tables and additional details of performance event implementation for end-user (including additional details beyond event code/umask) can found at the “perfmon” repository provided by The Intel Open Source Technology Center (https://download.01.org/perfmon/).

19.1 ARCHITECTURAL PERFORMANCE-MONITORING EVENTS

Architectural performance events are introduced in Intel Core Solo and Intel Core Duo processors. They are also supported on processors based on Intel Core microarchitecture. Table 19-1 lists pre-defined architectural performance events that can be configured using general-purpose performance counters and associated event-select registers.

Table 19-1. Architectural Performance Events

Event Num.	Event Mask Name	Umask Value	Description
3CH	UnHalted Core Cycles	00H	Counts core clock cycles whenever the logical processor is in C0 state (not halted). The frequency of this event varies with state transitions in the core.
3CH	UnHalted Reference Cycles[1]	01H	Counts at a fixed frequency whenever the logical processor is in C0 state (not halted).
C0H	Instructions Retired	00H	Counts when the last uop of an instruction retires.
2EH	LLC Reference	4FH	Counts requests originating from the core that reference a cache line in the last level on-die cache.
2EH	LLC Misses	41H	Counts each cache miss condition for references to the last level on-die cache.
C4H	Branch Instruction Retired	00H	Counts when the last uop of a branch instruction retires.
C5H	Branch Misses Retired	00H	Counts when the last uop of a branch instruction retires which corrected misprediction of the branch prediction hardware at execution time.

NOTES:

1. Current implementations count at core crystal clock, TSC, or bus clock frequency.

Fixed-function performance counters count only events defined in Table 19-2.

Table 19-2. Fixed-Function Performance Counter and Pre-defined Performance Events

Fixed-Function Performance Counter	Address	Event Mask Mnemonic	Description
IA32_PERF_FIXED_CTR0	309H	Inst_Retired.Any	This event counts the number of instructions that retire execution. For instructions that consist of multiple micro-ops, this event counts the retirement of the last micro-op of the instruction. The counter continues counting during hardware interrupts, traps, and inside interrupt handlers.
IA32_PERF_FIXED_CTR1	30AH	CPU_CLK_UNHALTED.THREAD/CPU_CLK_UNHALTED.CORE/CPU_CLK_UNHALTED.THREAD_ANY	The CPU_CLK_UNHALTED.THREAD event counts the number of core cycles while the logical processor is not in a halt state. If there is only one logical processor in a processor core, CPU_CLK_UNHALTED.CORE counts the unhalted cycles of the processor core. If there are more than one logical processor in a processor core, CPU_CLK_UNHALTED.THREAD_ANY is supported by programming IA32_FIXED_CTR_CTRL[bit 6]AnyThread = 1. The core frequency may change from time to time due to transitions associated with Enhanced Intel SpeedStep Technology or TM2. For this reason this event may have a changing ratio with regards to time.

Table 19-2. Fixed-Function Performance Counter and Pre-defined Performance Events (Contd.)

Fixed-Function Performance Counter	Address	Event Mask Mnemonic	Description
IA32_PERF_FIXED_CTR2	30BH	CPU_CLK_UNHALTED.REF_TSC	This event counts the number of reference cycles at the TSC rate when the core is not in a halt state and not in a TM stop-clock state. The core enters the halt state when it is running the HLT instruction or the MWAIT instruction. This event is not affected by core frequency changes (e.g., P states) but counts at the same frequency as the time stamp counter. This event can approximate elapsed time while the core was not in a halt state and not in a TM stopclock state.

19.2 PERFORMANCE MONITORING EVENTS FOR INTEL® XEON® PROCESSOR SCALABLE FAMILY

The Intel® Xeon® Processor Scalable Family is based on the Skylake microarchitecture. These processors support the architectural performance-monitoring events listed in Table 19-1. Fixed counters in the core PMU support the architecture events defined in Table 19-2. Non-architectural performance-monitoring events in the processor core are listed in Table 19-4. The events in Table 19-4 apply to processors with CPUID signature of DisplayFamily_DisplayModel encoding with the following value: 06_55H .

The comment column in Table 19-4 uses abbreviated letters to indicate additional conditions applicable to the Event Mask Mnemonic. For event umasks listed in Table 19-4 that do not show "AnyT", users should refrain from programming "AnyThread =1" in IA32_PERF_EVTSELx.

Table 19-3. Non-Architectural Performance Events of the Processor Core Supported in Intel® Xeon® Processor Scalable Family with Skylake Microarchitecture

Event Num.	Umask Value	Event Mask Mnemonic	Description	Comment
00H	01H	INST_RETIRED.ANY	Counts the number of instructions retired from execution. For instructions that consist of multiple micro-ops, Counts the retirement of the last micro-op of the instruction. Counting continues during hardware interrupts, traps, and inside interrupt handlers. Notes: INST_RETIRED.ANY is counted by a designated fixed counter, leaving the four (eight when Hyperthreading is disabled) programmable counters available for other events. INST_RETIRED.ANY_P is counted by a programmable counter and it is an architectural performance event. Counting: Faulting executions of GETSEC/VM entry/VM Exit/MWait will not count as retired instructions.	Fixed Counter
00H	02H	CPU_CLK_UNHALTED.THREAD	Counts the number of core cycles while the thread is not in a halt state. The thread enters the halt state when it is running the HLT instruction. This event is a component in many key event ratios. The core frequency may change from time to time due to transitions associated with Enhanced Intel SpeedStep Technology or TM2. For this reason this event may have a changing ratio with regards to time. When the core frequency is constant, this event can approximate elapsed time while the core was not in the halt state. It is counted on a dedicated fixed counter, leaving the four (eight when Hyperthreading is disabled) programmable counters available for other events.	Fixed Counter

Table 19-3. Non-Architectural Performance Events of the Processor Core Supported in Intel® Xeon® Processor Scalable Family with Skylake Microarchitecture (Contd.)

Event Num.	Umask Value	Event Mask Mnemonic	Description	Comment
00H	02H	CPU_CLK_UNHALTED.THREAD_ANY	Core cycles when at least one thread on the physical core is not in halt state.	AnyThread=1
00H	03H	CPU_CLK_UNHALTED.REF_TSC	Counts the number of reference cycles when the core is not in a halt state. The core enters the halt state when it is running the HLT instruction or the MWAIT instruction. This event is not affected by core frequency changes (for example, P states, TM2 transitions) but has the same incrementing frequency as the time stamp counter. This event can approximate elapsed time while the core was not in a halt state. This event has a constant ratio with the CPU_CLK_UNHALTED.REF_XCLK event. It is counted on a dedicated fixed counter, leaving the four (eight when Hyperthreading is disabled) programmable counters available for other events. Note: On all current platforms this event stops counting during 'throttling (TM)' states duty off periods the processor is 'halted'. The counter update is done at a lower clock rate then the core clock the overflow status bit for this counter may appear 'sticky'. After the counter has overflowed and software clears the overflow status bit and resets the counter to less than MAX. The reset value to the counter is not clocked immediately so the overflow status bit will flip "high (1)" and generate another PMI (if enabled) after which the reset value gets clocked into the counter. Therefore, software will get the interrupt, read the overflow status bit '1 for bit 34 while the counter value is less than MAX. Software should ignore this case.	Fixed Counter
03H	02H	LD_BLOCKS.STORE_FORWARD	Counts how many times the load operation got the true Block-on-Store blocking code preventing store forwarding. This includes cases when: a. preceding store conflicts with the load (incomplete overlap), b. store forwarding is impossible due to u-arch limitations, c. preceding lock RMW operations are not forwarded, d. store has the no-forward bit set (uncacheable/page-split/masked stores), e. all-blocking stores are used (mostly, fences and port I/O), and others. The most common case is a load blocked due to its address range overlapping with a preceding smaller uncompleted store. Note: This event does not take into account cases of out-of-SW-control (for example, SbTailHit), unknown physical STA, and cases of blocking loads on store due to being non-WB memory type or a lock. These cases are covered by other events. See the table of not supported store forwards in the Optimization Guide.	
03H	08H	LD_BLOCKS.NO_SR	The number of times that split load operations are temporarily blocked because all resources for handling the split accesses are in use.	

Table 19-3. Non-Architectural Performance Events of the Processor Core Supported in Intel® Xeon® Processor Scalable Family with Skylake Microarchitecture (Contd.)

Event Num.	Umask Value	Event Mask Mnemonic	Description	Comment
07H	01H	LD_BLOCKS_PARTIAL.ADDRESS_ALIAS	Counts false dependencies in MOB when the partial comparison upon loose net check and dependency was resolved by the Enhanced Loose net mechanism. This may not result in high performance penalties. Loose net checks can fail when loads and stores are 4k aliased.	
08H	01H	DTLB_LOAD_MISSES.MISS_CAUSES_A_WALK	Counts demand data loads that caused a page walk of any page size (4K/2M/4M/1G). This implies it missed in all TLB levels, but the walk need not have completed.	
08H	02H	DTLB_LOAD_MISSES.WALK_COMPLETED_4K	Counts demand data loads that caused a completed page walk (4K page size). This implies it missed in all TLB levels. The page walk can end with or without a fault.	
08H	04H	DTLB_LOAD_MISSES.WALK_COMPLETED_2M_4M	Counts demand data loads that caused a completed page walk (2M and 4M page sizes). This implies it missed in all TLB levels. The page walk can end with or without a fault.	
08H	08H	DTLB_LOAD_MISSES.WALK_COMPLETED_1G	Counts load misses in all DTLB levels that cause a completed page walk (1G page size). The page walk can end with or without a fault.	
08H	0EH	DTLB_LOAD_MISSES.WALK_COMPLETED	Counts demand data loads that caused a completed page walk of any page size (4K/2M/4M/1G). This implies it missed in all TLB levels. The page walk can end with or without a fault.	
08H	10H	DTLB_LOAD_MISSES.WALK_PENDING	Counts 1 per cycle for each PMH that is busy with a page walk for a load. EPT page walk duration are excluded in Skylake microarchitecture.	
08H	10H	DTLB_LOAD_MISSES.WALK_ACTIVE	Counts cycles when at least one PMH (Page Miss Handler) is busy with a page walk for a load.	CounterMask=1 CMSK1
08H	20H	DTLB_LOAD_MISSES.STLB_HIT	Counts loads that miss the DTLB (Data TLB) and hit the STLB (Second level TLB).	
0DH	01H	INT_MISC.RECOVERY_CYCLES	Core cycles the Resource allocator was stalled due to recovery from an earlier branch misprediction or machine clear event.	
0DH	01H	INT_MISC.RECOVERY_CYCLES_ANY	Core cycles the allocator was stalled due to recovery from earlier clear event for any thread running on the physical core (e.g. misprediction or memory nuke).	AnyThread=1 AnyT
0DH	80H	INT_MISC.CLEAR_RESTEER_CYCLES	Cycles the issue-stage is waiting for front-end to fetch from resteered path following branch misprediction or machine clear events.	
0EH	01H	UOPS_ISSUED.ANY	Counts the number of uops that the Resource Allocation Table (RAT) issues to the Reservation Station (RS).	
0EH	01H	UOPS_ISSUED.STALL_CYCLES	Counts cycles during which the Resource Allocation Table (RAT) does not issue any uops to the reservation station (RS) for the current thread.	CounterMask=1 Invert=1 CMSK1, INV

Table 19-3. Non-Architectural Performance Events of the Processor Core Supported in Intel® Xeon® Processor Scalable Family with Skylake Microarchitecture (Contd.)

Event Num.	Umask Value	Event Mask Mnemonic	Description	Comment
0EH	02H	UOPS_ISSUED.VECTOR_WIDTH_MISMATCH	Counts the number of Blend Uops issued by the Resource Allocation Table (RAT) to the reservation station (RS) in order to preserve upper bits of vector registers. Starting with the Skylake microarchitecture, these Blend uops are needed since every Intel SSE instruction executed in Dirty Upper State needs to preserve bits 128-255 of the destination register. For more information, refer to Mixing Intel AVX and Intel SSE Code section of the Optimization Guide.	
0EH	20H	UOPS_ISSUED.SLOW_LEA	Number of slow LEA uops being allocated. A uop is generally considered SlowLea if it has 3 sources (e.g. 2 sources + immediate) regardless if as a result of LEA instruction or not.	
14H	01H	ARITH.DIVIDER_ACTIVE	Cycles when divide unit is busy executing divide or square root operations. Accounts for integer and floating-point operations.	CounterMask=1
24H	21H	L2_RQSTS.DEMAND_DATA_RD_MISS	Counts the number of demand Data Read requests that miss L2 cache. Only not rejected loads are counted.	
24H	22H	L2_RQSTS.RFO_MISS	Counts the RFO (Read-for-Ownership) requests that miss L2 cache.	
24H	24H	L2_RQSTS.CODE_RD_MISS	Counts L2 cache misses when fetching instructions.	
24H	27H	L2_RQSTS.ALL_DEMAND_MISS	Demand requests that miss L2 cache.	
24H	38H	L2_RQSTS.PF_MISS	Counts requests from the L1/L2/L3 hardware prefetchers or Load software prefetches that miss L2 cache.	
24H	3FH	L2_RQSTS.MISS	All requests that miss L2 cache.	
24H	41H	L2_RQSTS.DEMAND_DATA_RD_HIT	Counts the number of demand Data Read requests that hit L2 cache. Only non rejected loads are counted.	
24H	42H	L2_RQSTS.RFO_HIT	Counts the RFO (Read-for-Ownership) requests that hit L2 cache.	
24H	44H	L2_RQSTS.CODE_RD_HIT	Counts L2 cache hits when fetching instructions, code reads.	
24H	D8H	L2_RQSTS.PF_HIT	Counts requests from the L1/L2/L3 hardware prefetchers or Load software prefetches that hit L2 cache.	
24H	E1H	L2_RQSTS.ALL_DEMAND_DATA_RD	Counts the number of demand Data Read requests (including requests from L1D hardware prefetchers). These loads may hit or miss L2 cache. Only non rejected loads are counted.	
24H	E2H	L2_RQSTS.ALL_RFO	Counts the total number of RFO (read for ownership) requests to L2 cache. L2 RFO requests include both L1D demand RFO misses as well as L1D RFO prefetches.	
24H	E4H	L2_RQSTS.ALL_CODE_RD	Counts the total number of L2 code requests.	
24H	E7H	L2_RQSTS.ALL_DEMAND_REFERENCES	Demand requests to L2 cache.	
24H	F8H	L2_RQSTS.ALL_PF	Counts the total number of requests from the L2 hardware prefetchers.	

Table 19-3. Non-Architectural Performance Events of the Processor Core Supported in Intel® Xeon® Processor Scalable Family with Skylake Microarchitecture (Contd.)

Event Num.	Umask Value	Event Mask Mnemonic	Description	Comment
24H	FFH	L2_RQSTS.REFERENCES	All L2 requests.	
28H	07H	CORE_POWER.LVL0_TURBO_LICENSE	Core cycles where the core was running with power-delivery for baseline license level 0. This includes non-AVX codes, SSE, AVX 128-bit, and low-current AVX 256-bit codes.	
28H	18H	CORE_POWER.LVL1_TURBO_LICENSE	Core cycles where the core was running with power-delivery for license level 1. This includes high current AVX 256-bit instructions as well as low current AVX 512-bit instructions.	
28H	20H	CORE_POWER.LVL2_TURBO_LICENSE	Core cycles where the core was running with power-delivery for license level 2 (introduced in Skylake Server microarchitecture). This includes high current AVX 512-bit instructions.	
28H	40H	CORE_POWER.THROTTLE	Core cycles the out-of-order engine was throttled due to a pending power level request.	
2EH	41H	LONGEST_LAT_CACHE.MISS	Counts core-originated cacheable requests that miss the L3 cache (Longest Latency cache). Requests include data and code reads, Reads-for-Ownership (RFOs), speculative accesses and hardware prefetches from L1 and L2. It does not include all misses to the L3.	See Table 19-1.
2EH	4FH	LONGEST_LAT_CACHE.REFERENCE	Counts core-originated cacheable requests to the L3 cache (Longest Latency cache). Requests include data and code reads, Reads-for-Ownership (RFOs), speculative accesses and hardware prefetches from L1 and L2. It does not include all accesses to the L3.	See Table 19-1.
3CH	00H	CPU_CLK_UNHALTED.THREAD_P	This is an architectural event that counts the number of thread cycles while the thread is not in a halt state. The thread enters the halt state when it is running the HLT instruction. The core frequency may change from time to time due to power or thermal throttling. For this reason, this event may have a changing ratio with regards to wall clock time.	See Table 19-1.
3CH	00H	CPU_CLK_UNHALTED.THREAD_P_ANY	Core cycles when at least one thread on the physical core is not in halt state.	AnyThread=1 AnyT
3CH	00H	CPU_CLK_UNHALTED.RING0_TRANS	Counts when the Current Privilege Level (CPL) transitions from ring 1, 2 or 3 to ring 0 (Kernel).	EdgeDetect=1 CounterMask=1
3CH	01H	CPU_CLK_THREAD_UNHALTED.REF_XCLK	Core crystal clock cycles when the thread is unhalted.	See Table 19-1.
3CH	01H	CPU_CLK_THREAD_UNHALTED.REF_XCLK_ANY	Core crystal clock cycles when at least one thread on the physical core is unhalted.	AnyThread=1 AnyT
3CH	01H	CPU_CLK_UNHALTED.REF_XCLK	Core crystal clock cycles when the thread is unhalted.	See Table 19-1.
3CH	01H	CPU_CLK_UNHALTED.REF_XCLK_ANY	Core crystal clock cycles when at least one thread on the physical core is unhalted.	AnyThread=1 AnyT
3CH	02H	CPU_CLK_THREAD_UNHALTED.ONE_THREAD_ACTIVE	Core crystal clock cycles when this thread is unhalted and the other thread is halted.	
3CH	02H	CPU_CLK_UNHALTED.ONE_THREAD_ACTIVE	Core crystal clock cycles when this thread is unhalted and the other thread is halted.	

Table 19-3. Non-Architectural Performance Events of the Processor Core Supported in Intel® Xeon® Processor Scalable Family with Skylake Microarchitecture (Contd.)

Event Num.	Umask Value	Event Mask Mnemonic	Description	Comment
48H	01H	L1D_PEND_MISS.PENDING	Counts duration of L1D miss outstanding, that is each cycle number of Fill Buffers (FB) outstanding required by Demand Reads. FB either is held by demand loads, or it is held by non-demand loads and gets hit at least once by demand. The valid outstanding interval is defined until the FB deallocation by one of the following ways: from FB allocation, if FB is allocated by demand from the demand Hit FB, if it is allocated by hardware or software prefetch.Note: In the L1D, a Demand Read contains cacheable or noncacheable demand loads, including ones causing cache-line splits and reads due to page walks resulted from any request type.	
48H	01H	L1D_PEND_MISS.PENDING_CYCLES	Counts duration of L1D miss outstanding in cycles.	CounterMask=1 CMSK1
48H	01H	L1D_PEND_MISS.PENDING_CYCLES_ANY	Cycles with L1D load Misses outstanding from any thread on physical core.	CounterMask=1 AnyThread=1 CMSK1, AnyT
48H	02H	L1D_PEND_MISS.FB_FULL	Number of times a request needed a FB (Fill Buffer) entry but there was no entry available for it. A request includes cacheable/uncacheable demands that are load, store or SW prefetch instructions.	
49H	01H	DTLB_STORE_MISSES.MISS_CAUSES_A_WALK	Counts demand data stores that caused a page walk of any page size (4K/2M/4M/1G). This implies it missed in all TLB levels, but the walk need not have completed.	
49H	02H	DTLB_STORE_MISSES.WALK_COMPLETED_4K	Counts demand data stores that caused a completed page walk (4K page size). This implies it missed in all TLB levels. The page walk can end with or without a fault.	
49H	04H	DTLB_STORE_MISSES.WALK_COMPLETED_2M_4M	Counts demand data stores that caused a completed page walk (2M and 4M page sizes). This implies it missed in all TLB levels. The page walk can end with or without a fault.	
49H	08H	DTLB_STORE_MISSES.WALK_COMPLETED_1G	Counts store misses in all DTLB levels that cause a completed page walk (1G page size). The page walk can end with or without a fault.	
49H	0EH	DTLB_STORE_MISSES.WALK_COMPLETED	Counts demand data stores that caused a completed page walk of any page size (4K/2M/4M/1G). This implies it missed in all TLB levels. The page walk can end with or without a fault.	
49H	10H	DTLB_STORE_MISSES.WALK_PENDING	Counts 1 per cycle for each PMH that is busy with a page walk for a store. EPT page walk duration are excluded in Skylake microarchitecture.	
49H	10H	DTLB_STORE_MISSES.WALK_ACTIVE	Counts cycles when at least one PMH (Page Miss Handler) is busy with a page walk for a store.	CounterMask=1 CMSK1
49H	20H	DTLB_STORE_MISSES.STLB_HIT	Stores that miss the DTLB (Data TLB) and hit the STLB (2nd Level TLB).	

Table 19-3. Non-Architectural Performance Events of the Processor Core Supported in Intel® Xeon® Processor Scalable Family with Skylake Microarchitecture (Contd.)

Event Num.	Umask Value	Event Mask Mnemonic	Description	Comment
4CH	01H	LOAD_HIT_PRE.SW_PF	Counts all not software-prefetch load dispatches that hit the fill buffer (FB) allocated for the software prefetch. It can also be incremented by some lock instructions. So it should only be used with profiling so that the locks can be excluded by ASM (Assembly File) inspection of the nearby instructions.	
4FH	10H	EPT.WALK_PENDING	Counts cycles for each PMH (Page Miss Handler) that is busy with an EPT (Extended Page Table) walk for any request type.	
51H	01H	L1D.REPLACEMENT	Counts L1D data line replacements including opportunistic replacements, and replacements that require stall-for-replace or block-for-replace.	
54H	01H	TX_MEM.ABORT_CONFLICT	Number of times a TSX line had a cache conflict.	
54H	02H	TX_MEM.ABORT_CAPACITY	Number of times a transactional abort was signaled due to a data capacity limitation for transactional reads or writes.	
54H	04H	TX_MEM.ABORT_HLE_STORE_TO_ELIDED_LOCK	Number of times a TSX Abort was triggered due to a non-release/commit store to lock.	
54H	08H	TX_MEM.ABORT_HLE_ELISION_BUFFER_NOT_EMPTY	Number of times a TSX Abort was triggered due to commit but Lock Buffer not empty.	
54H	10H	TX_MEM.ABORT_HLE_ELISION_BUFFER_MISMATCH	Number of times a TSX Abort was triggered due to release/commit but data and address mismatch.	
54H	20H	TX_MEM.ABORT_HLE_ELISION_BUFFER_UNSUPPORTED_ALIGNMENT	Number of times a TSX Abort was triggered due to attempting an unsupported alignment from Lock Buffer.	
54H	40H	TX_MEM.HLE_ELISION_BUFFER_FULL	Number of times we could not allocate Lock Buffer.	
5DH	01H	TX_EXEC.MISC1	Unfriendly TSX abort triggered by a flowmarker.	
5DH	02H	TX_EXEC.MISC2	Unfriendly TSX abort triggered by a vzeroupper instruction.	
5DH	04H	TX_EXEC.MISC3	Unfriendly TSX abort triggered by a nest count that is too deep.	
5DH	08H	TX_EXEC.MISC4	RTM region detected inside HLE.	
5DH	10H	TX_EXEC.MISC5	Counts the number of times an HLE XACQUIRE instruction was executed inside an RTM transactional region.	
5EH	01H	RS_EVENTS.EMPTY_CYCLES	Counts cycles during which the reservation station (RS) is empty for the thread.; Note: In ST-mode, not active thread should drive 0. This is usually caused by severely costly branch mispredictions, or allocator/FE issues.	
5EH	01H	RS_EVENTS.EMPTY_END	Counts end of periods where the Reservation Station (RS) was empty. Could be useful to precisely locate front-end Latency Bound issues.	EdgeDetect=1 CounterMask=1 Invert=1 CMSK1, INV

Table 19-3. Non-Architectural Performance Events of the Processor Core Supported in Intel® Xeon® Processor Scalable Family with Skylake Microarchitecture (Contd.)

Event Num.	Umask Value	Event Mask Mnemonic	Description	Comment
60H	01H	OFFCORE_REQUESTS_OUTSTANDING.DEMAND_DATA_RD	Counts the number of offcore outstanding Demand Data Read transactions in the super queue (SQ) every cycle. A transaction is considered to be in the Offcore outstanding state between L2 miss and transaction completion sent to requestor. See the corresponding Umask under OFFCORE_REQUESTS. Note: A prefetch promoted to Demand is counted from the promotion point.	
60H	01H	OFFCORE_REQUESTS_OUTSTANDING.CYCLES_WITH_DEMAND_DATA_RD	Counts cycles when offcore outstanding Demand Data Read transactions are present in the super queue (SQ). A transaction is considered to be in the Offcore outstanding state between L2 miss and transaction completion sent to requestor (SQ de-allocation).	CounterMask=1 CMSK1
60H	01H	OFFCORE_REQUESTS_OUTSTANDING.DEMAND_DATA_RD_GE_6	Cycles with at least 6 offcore outstanding Demand Data Read transactions in uncore queue.	CounterMask=6 CMSK6
60H	02H	OFFCORE_REQUESTS_OUTSTANDING.DEMAND_CODE_RD	Counts the number of offcore outstanding Code Reads transactions in the super queue every cycle. The 'Offcore outstanding' state of the transaction lasts from the L2 miss until the sending transaction completion to requestor (SQ deallocation). See the corresponding Umask under OFFCORE_REQUESTS.	CounterMask=1 CMSK1
60H	02H	OFFCORE_REQUESTS_OUTSTANDING.CYCLES_WITH_DEMAND_CODE_RD	Counts the number of offcore outstanding Code Reads transactions in the super queue every cycle. The 'Offcore outstanding' state of the transaction lasts from the L2 miss until the sending transaction completion to requestor (SQ deallocation). See the corresponding Umask under OFFCORE_REQUESTS.	CMSK1
60H	04H	OFFCORE_REQUESTS_OUTSTANDING.DEMAND_RFO	Counts the number of offcore outstanding RFO (store) transactions in the super queue (SQ) every cycle. A transaction is considered to be in the Offcore outstanding state between L2 miss and transaction completion sent to requestor (SQ de-allocation). See corresponding Umask under OFFCORE_REQUESTS.	CounterMask=1 CMSK1
60H	04H	OFFCORE_REQUESTS_OUTSTANDING.CYCLES_WITH_DEMAND_RFO	Counts the number of offcore outstanding demand rfo Reads transactions in the super queue every cycle. The 'Offcore outstanding' state of the transaction lasts from the L2 miss until the sending transaction completion to requestor (SQ deallocation). See the corresponding Umask under OFFCORE_REQUESTS.	CMSK1
60H	08H	OFFCORE_REQUESTS_OUTSTANDING.ALL_DATA_RD	Counts the number of offcore outstanding cacheable Core Data Read transactions in the super queue every cycle. A transaction is considered to be in the Offcore outstanding state between L2 miss and transaction completion sent to requestor (SQ de-allocation). See corresponding Umask under OFFCORE_REQUESTS.	
60H	08H	OFFCORE_REQUESTS_OUTSTANDING.CYCLES_WITH_DATA_RD	Counts cycles when offcore outstanding cacheable Core Data Read transactions are present in the super queue. A transaction is considered to be in the Offcore outstanding state between L2 miss and transaction completion sent to requestor (SQ de-allocation). See corresponding Umask under OFFCORE_REQUESTS.	CounterMask=1 CMSK1

Table 19-3. Non-Architectural Performance Events of the Processor Core Supported in Intel® Xeon® Processor Scalable Family with Skylake Microarchitecture (Contd.)

Event Num.	Umask Value	Event Mask Mnemonic	Description	Comment
60H	10H	OFFCORE_REQUESTS_OUTSTANDING.L3_MISS_DEMAND_DATA_RD	Counts number of Offcore outstanding Demand Data Read requests that miss L3 cache in the superQ every cycle.	
60H	10H	OFFCORE_REQUESTS_OUTSTANDING.CYCLES_WITH_L3_MISS_DEMAND_DATA_RD	Cycles with at least 1 Demand Data Read requests who miss L3 cache in the superQ.	CounterMask=1 CMSK1
60H	10H	OFFCORE_REQUESTS_OUTSTANDING.L3_MISS_DEMAND_DATA_RD_GE_6	Cycles with at least 6 Demand Data Read requests that miss L3 cache in the superQ.	CounterMask=6 CMSK6
79H	04H	IDQ.MITE_UOPS	Counts the number of uops delivered to Instruction Decode Queue (IDQ) from the MITE path. Counting includes uops that may 'bypass' the IDQ. This also means that uops are not being delivered from the Decode Stream Buffer (DSB).	
79H	04H	IDQ.MITE_CYCLES	Counts cycles during which uops are being delivered to Instruction Decode Queue (IDQ) from the MITE path. Counting includes uops that may 'bypass' the IDQ.	CounterMask=1 CMSK1
79H	08H	IDQ.DSB_UOPS	Counts the number of uops delivered to Instruction Decode Queue (IDQ) from the Decode Stream Buffer (DSB) path. Counting includes uops that may 'bypass' the IDQ.	
79H	08H	IDQ.DSB_CYCLES	Counts cycles during which uops are being delivered to Instruction Decode Queue (IDQ) from the Decode Stream Buffer (DSB) path. Counting includes uops that may 'bypass' the IDQ.	CounterMask=1 CMSK1
79H	10H	IDQ.MS_DSB_CYCLES	Counts cycles during which uops initiated by Decode Stream Buffer (DSB) are being delivered to Instruction Decode Queue (IDQ) while the Microcode Sequencer (MS) is busy. Counting includes uops that may 'bypass' the IDQ.	CounterMask=1
79H	18H	IDQ.ALL_DSB_CYCLES_4_UOPS	Counts the number of cycles 4 uops were delivered to Instruction Decode Queue (IDQ) from the Decode Stream Buffer (DSB) path. Count includes uops that may 'bypass' the IDQ.	CounterMask=4 CMSK4
79H	18H	IDQ.ALL_DSB_CYCLES_ANY_UOPS	Counts the number of cycles uops were delivered to Instruction Decode Queue (IDQ) from the Decode Stream Buffer (DSB) path. Count includes uops that may 'bypass' the IDQ.	CounterMask=1 CMSK1
79H	20H	IDQ.MS_MITE_UOPS	Counts the number of uops initiated by MITE and delivered to Instruction Decode Queue (IDQ) while the Microcode Sequencer (MS) is busy. Counting includes uops that may 'bypass' the IDQ.	
79H	24H	IDQ.ALL_MITE_CYCLES_4_UOPS	Counts the number of cycles 4 uops were delivered to the Instruction Decode Queue (IDQ) from the MITE (legacy decode pipeline) path. Counting includes uops that may 'bypass' the IDQ. During these cycles uops are not being delivered from the Decode Stream Buffer (DSB).	CounterMask=4 CMSK4

Table 19-3. Non-Architectural Performance Events of the Processor Core Supported in Intel® Xeon® Processor Scalable Family with Skylake Microarchitecture (Contd.)

Event Num.	Umask Value	Event Mask Mnemonic	Description	Comment
79H	24H	IDQ.ALL_MITE_CYCLES_ANY_UOPS	Counts the number of cycles uops were delivered to the Instruction Decode Queue (IDQ) from the MITE (legacy decode pipeline) path. Counting includes uops that may 'bypass' the IDQ. During these cycles uops are not being delivered from the Decode Stream Buffer (DSB).	CounterMask=1 CMSK1
79H	30H	IDQ.MS_CYCLES	Counts cycles during which uops are being delivered to Instruction Decode Queue (IDQ) while the Microcode Sequencer (MS) is busy. Counting includes uops that may 'bypass' the IDQ. Uops maybe initiated by Decode Stream Buffer (DSB) or MITE.	CounterMask=1 CMSK1
79H	30H	IDQ.MS_SWITCHES	Number of switches from DSB (Decode Stream Buffer) or MITE (legacy decode pipeline) to the Microcode Sequencer.	EdgeDetect=1 CounterMask=1 EDGE
79H	30H	IDQ.MS_UOPS	Counts the total number of uops delivered by the Microcode Sequencer (MS). Any instruction over 4 uops will be delivered by the MS. Some instructions such as transcendentals may additionally generate uops from the MS.	
80H	04H	ICACHE_16B.IFDATA_STALL	Cycles where a code line fetch is stalled due to an L1 instruction cache miss. The legacy decode pipeline works at a 16 Byte granularity.	
83H	01H	ICACHE_64B.IFTAG_HIT	Instruction fetch tag lookups that hit in the instruction cache (L1I). Counts at 64-byte cache-line granularity.	
83H	02H	ICACHE_64B.IFTAG_MISS	Instruction fetch tag lookups that miss in the instruction cache (L1I). Counts at 64-byte cache-line granularity.	
83H	04H	ICACHE_64B.IFTAG_STALL	Cycles where a code fetch is stalled due to L1 instruction cache tag miss.	
85H	01H	ITLB_MISSES.MISS_CAUSES_A_WALK	Counts page walks of any page size (4K/2M/4M/1G) caused by a code fetch. This implies it missed in the ITLB and further levels of TLB, but the walk need not have completed.	
85H	02H	ITLB_MISSES.WALK_COMPLETED_4K	Counts completed page walks (4K page size) caused by a code fetch. This implies it missed in the ITLB and further levels of TLB. The page walk can end with or without a fault.	
85H	04H	ITLB_MISSES.WALK_COMPLETED_2M_4M	Counts completed page walks of any page size (4K/2M/4M/1G) caused by a code fetch. This implies it missed in the ITLB and further levels of TLB. The page walk can end with or without a fault.	
85H	08H	ITLB_MISSES.WALK_COMPLETED_1G	Counts store misses in all DTLB levels that cause a completed page walk (1G page size). The page walk can end with or without a fault.	
85H	0EH	ITLB_MISSES.WALK_COMPLETED	Counts completed page walks (2M and 4M page sizes) caused by a code fetch. This implies it missed in the ITLB and further levels of TLB. The page walk can end with or without a fault.	

Table 19-3. Non-Architectural Performance Events of the Processor Core Supported in Intel® Xeon® Processor Scalable Family with Skylake Microarchitecture (Contd.)

Event Num.	Umask Value	Event Mask Mnemonic	Description	Comment
85H	10H	ITLB_MISSES.WALK_PENDING	Counts 1 per cycle for each PMH that is busy with a page walk for an instruction fetch request. EPT page walk duration are excluded in Skylake microarchitecture.	
85H	10H	ITLB_MISSES.WALK_ACTIVE	Cycles when at least one PMH is busy with a page walk for code (instruction fetch) request. EPT page walk duration are excluded in Skylake microarchitecture.	CounterMask=1
85H	20H	ITLB_MISSES.STLB_HIT	Instruction fetch requests that miss the ITLB and hit the STLB.	
87H	01H	ILD_STALL.LCP	Counts cycles that the Instruction Length decoder (ILD) stalls occurred due to dynamically changing prefix length of the decoded instruction (by operand size prefix instruction 0x66, address size prefix instruction 0x67 or REX.W for Intel64). Count is proportional to the number of prefixes in a 16B-line. This may result in a three-cycle penalty for each LCP (Length changing prefix) in a 16-byte chunk.	
9CH	01H	IDQ_UOPS_NOT_DELIVERED.CORE	Counts the number of uops not delivered to Resource Allocation Table (RAT) per thread adding "4 – x" ? when Resource Allocation Table (RAT) is not stalled and Instruction Decode Queue (IDQ) delivers x uops to Resource Allocation Table (RAT) (where x belongs to {0,1,2,3}). Counting does not cover cases when: a. IDQ-Resource Allocation Table (RAT) pipe serves the other thread. b. Resource Allocation Table (RAT) is stalled for the thread (including uop drops and clear BE conditions). c. Instruction Decode Queue (IDQ) delivers four uops.	
9CH	01H	IDQ_UOPS_NOT_DELIVERED.CYCLES_0_UOPS_DELIV.CORE	Counts, on the per-thread basis, cycles when no uops are delivered to Resource Allocation Table (RAT). IDQ_Uops_Not_Delivered.core =4.	CounterMask=4 CMSK4
9CH	01H	IDQ_UOPS_NOT_DELIVERED.CYCLES_LE_1_UOP_DELIV.CORE	Counts, on the per-thread basis, cycles when less than 1 uop is delivered to Resource Allocation Table (RAT). IDQ_Uops_Not_Delivered.core >= 3.	CounterMask=3 CMSK3
9CH	01H	IDQ_UOPS_NOT_DELIVERED.CYCLES_LE_2_UOP_DELIV.CORE	Cycles with less than 2 uops delivered by the front end.	CounterMask=2 CMSK2
9CH	01H	IDQ_UOPS_NOT_DELIVERED.CYCLES_LE_3_UOP_DELIV.CORE	Cycles with less than 3 uops delivered by the front end.	CounterMask=1 CMSK1
9CH	01H	IDQ_UOPS_NOT_DELIVERED.CYCLES_FE_WAS_OK	Counts cycles FE delivered 4 uops or Resource Allocation Table (RAT) was stalling FE.	CounterMask=1 Invert=1 CMSK, INV
A1H	01H	UOPS_DISPATCHED_PORT.PORT_0	Counts, on the per-thread basis, cycles during which at least one uop is dispatched from the Reservation Station (RS) to port 0.	
A1H	02H	UOPS_DISPATCHED_PORT.PORT_1	Counts, on the per-thread basis, cycles during which at least one uop is dispatched from the Reservation Station (RS) to port 1.	
A1H	04H	UOPS_DISPATCHED_PORT.PORT_2	Counts, on the per-thread basis, cycles during which at least one uop is dispatched from the Reservation Station (RS) to port 2.	

Table 19-3. Non-Architectural Performance Events of the Processor Core Supported in Intel® Xeon® Processor Scalable Family with Skylake Microarchitecture (Contd.)

Event Num.	Umask Value	Event Mask Mnemonic	Description	Comment
A1H	08H	UOPS_DISPATCHED_PORT.PORT_3	Counts, on the per-thread basis, cycles during which at least one uop is dispatched from the Reservation Station (RS) to port 3.	
A1H	10H	UOPS_DISPATCHED_PORT.PORT_4	Counts, on the per-thread basis, cycles during which at least one uop is dispatched from the Reservation Station (RS) to port 4.	
A1H	20H	UOPS_DISPATCHED_PORT.PORT_5	Counts, on the per-thread basis, cycles during which at least one uop is dispatched from the Reservation Station (RS) to port 5.	
A1H	40H	UOPS_DISPATCHED_PORT.PORT_6	Counts, on the per-thread basis, cycles during which at least one uop is dispatched from the Reservation Station (RS) to port 6.	
A1H	80H	UOPS_DISPATCHED_PORT.PORT_7	Counts, on the per-thread basis, cycles during which at least one uop is dispatched from the Reservation Station (RS) to port 7.	
A2H	01H	RESOURCE_STALLS.ANY	Counts resource-related stall cycles. Reasons for stalls can be as follows: a. *any* u-arch structure got full (LB, SB, RS, ROB, BOB, LM, Physical Register Reclaim Table (PRRT), or Physical History Table (PHT) slots). b. *any* u-arch structure got empty (like INT/SIMD FreeLists). c. FPU control word (FPCW), MXCSR.and others. This counts cycles that the pipeline back end blocked uop delivery from the front end.	
A2H	08H	RESOURCE_STALLS.SB	Counts allocation stall cycles caused by the store buffer (SB) being full. This counts cycles that the pipeline back end blocked uop delivery from the front end.	
A3H	01H	CYCLE_ACTIVITY.CYCLES_L2_MISS	Cycles while L2 cache miss demand load is outstanding.	CounterMask=1 CMSK1
A3H	02H	CYCLE_ACTIVITY.CYCLES_L3_MISS	Cycles while L3 cache miss demand load is outstanding.	CounterMask=2 CMSK2
A3H	04H	CYCLE_ACTIVITY.STALLS_TOTAL	Total execution stalls.	CounterMask=4 CMSK4
A3H	05H	CYCLE_ACTIVITY.STALLS_L2_MISS	Execution stalls while L2 cache miss demand load is outstanding.	CounterMask=5 CMSK5
A3H	06H	CYCLE_ACTIVITY.STALLS_L3_MISS	Execution stalls while L3 cache miss demand load is outstanding.	CounterMask=6 CMSK6
A3H	08H	CYCLE_ACTIVITY.CYCLES_L1D_MISS	Cycles while L1 cache miss demand load is outstanding.	CounterMask=8 CMSK8
A3H	0CH	CYCLE_ACTIVITY.STALLS_L1D_MISS	Execution stalls while L1 cache miss demand load is outstanding.	CounterMask=12 CMSK12
A3H	10H	CYCLE_ACTIVITY.CYCLES_MEM_ANY	Cycles while memory subsystem has an outstanding load.	CounterMask=16 CMSK16
A3H	14H	CYCLE_ACTIVITY.STALLS_MEM_ANY	Execution stalls while memory subsystem has an outstanding load.	CounterMask=20 CMSK20
A6H	01H	EXE_ACTIVITY.EXE_BOUND_0_PORTS	Counts cycles during which no uops were executed on all ports and Reservation Station (RS) was not empty.	

Table 19-3. Non-Architectural Performance Events of the Processor Core Supported in Intel® Xeon® Processor Scalable Family with Skylake Microarchitecture (Contd.)

Event Num.	Umask Value	Event Mask Mnemonic	Description	Comment
A6H	02H	EXE_ACTIVITY.1_PORTS_UTIL	Counts cycles during which a total of 1 uop was executed on all ports and Reservation Station (RS) was not empty.	
A6H	04H	EXE_ACTIVITY.2_PORTS_UTIL	Counts cycles during which a total of 2 uops were executed on all ports and Reservation Station (RS) was not empty.	
A6H	08H	EXE_ACTIVITY.3_PORTS_UTIL	Cycles total of 3 uops are executed on all ports and Reservation Station (RS) was not empty.	
A6H	10H	EXE_ACTIVITY.4_PORTS_UTIL	Cycles total of 4 uops are executed on all ports and Reservation Station (RS) was not empty.	
A6H	40H	EXE_ACTIVITY.BOUND_ON_STORES	Cycles where the Store Buffer was full and no outstanding load.	
A8H	01H	LSD.UOPS	Number of uops delivered to the back-end by the LSD (Loop Stream Detector).	
A8H	01H	LSD.CYCLES_ACTIVE	Counts the cycles when at least one uop is delivered by the LSD (Loop-stream detector).	CounterMask=1 CMSK1
A8H	01H	LSD.CYCLES_4_UOPS	Counts the cycles when 4 uops are delivered by the LSD (Loop-stream detector).	CounterMask=4 CMSK4
ABH	02H	DSB2MITE_SWITCHES.PENALTY_CYCLES	Counts Decode Stream Buffer (DSB)-to-MITE switch true penalty cycles. These cycles do not include uops routed through because of the switch itself, for example, when Instruction Decode Queue (IDQ) pre-allocation is unavailable, or Instruction Decode Queue (IDQ) is full. SBD-to-MITE switch true penalty cycles happen after the merge mux (MM) receives Decode Stream Buffer (DSB) Sync-indication until receiving the first MITE uop. MM is placed before Instruction Decode Queue (IDQ) to merge uops being fed from the MITE and Decode Stream Buffer (DSB) paths. Decode Stream Buffer (DSB) inserts the Sync-indication whenever a Decode Stream Buffer (DSB)-to-MITE switch occurs.Penalty: A Decode Stream Buffer (DSB) hit followed by a Decode Stream Buffer (DSB) miss can cost up to six cycles in which no uops are delivered to the IDQ. Most often, such switches from the Decode Stream Buffer (DSB) to the legacy pipeline cost 0 to 2 cycles.	
AEH	01H	ITLB.ITLB_FLUSH	Counts the number of flushes of the big or small ITLB pages. Counting include both TLB Flush (covering all sets) and TLB Set Clear (set-specific).	
B0H	01H	OFFCORE_REQUESTS.DEMAND_DATA_RD	Counts the Demand Data Read requests sent to uncore. Use it in conjunction with OFFCORE_REQUESTS_OUTSTANDING to determine average latency in the uncore.	
B0H	02H	OFFCORE_REQUESTS.DEMAND_CODE_RD	Counts both cacheable and non-cacheable code read requests.	
B0H	04H	OFFCORE_REQUESTS.DEMAND_RFO	Counts the demand RFO (read for ownership) requests including regular RFOs, locks, ItoM.	

Table 19-3. Non-Architectural Performance Events of the Processor Core Supported in Intel® Xeon® Processor Scalable Family with Skylake Microarchitecture (Contd.)

Event Num.	Umask Value	Event Mask Mnemonic	Description	Comment
B0H	08H	OFFCORE_REQUESTS.ALL_DATA_RD	Counts the demand and prefetch data reads. All Core Data Reads include cacheable 'Demands' and L2 prefetchers (not L3 prefetchers). Counting also covers reads due to page walks resulted from any request type.	
B0H	10H	OFFCORE_REQUESTS.L3_MISS_DEMAND_DATA_RD	Demand Data Read requests who miss L3 cache.	
B0H	80H	OFFCORE_REQUESTS.ALL_REQUESTS	Counts memory transactions reached the super queue including requests initiated by the core, all L3 prefetches, page walks, etc.	
B1H	01H	UOPS_EXECUTED.THREAD	Number of uops to be executed per-thread each cycle.	
B1H	01H	UOPS_EXECUTED.STALL_CYCLES	Counts cycles during which no uops were dispatched from the Reservation Station (RS) per thread.	CounterMask=1 Invert=1 CMSK, INV
B1H	01H	UOPS_EXECUTED.CYCLES_GE_1_UOP_EXEC	Cycles where at least 1 uop was executed per-thread.	CounterMask=1 CMSK1
B1H	01H	UOPS_EXECUTED.CYCLES_GE_2_UOPS_EXEC	Cycles where at least 2 uops were executed per-thread.	CounterMask=2 CMSK2
B1H	01H	UOPS_EXECUTED.CYCLES_GE_3_UOPS_EXEC	Cycles where at least 3 uops were executed per-thread.	CounterMask=3 CMSK3
B1H	01H	UOPS_EXECUTED.CYCLES_GE_4_UOPS_EXEC	Cycles where at least 4 uops were executed per-thread.	CounterMask=4 CMSK4
B1H	02H	UOPS_EXECUTED.CORE	Number of uops executed from any thread.	
B1H	02H	UOPS_EXECUTED.CORE_CYCLES_GE_1	Cycles at least 1 micro-op is executed from any thread on physical core.	CounterMask=1 CMSK1
B1H	02H	UOPS_EXECUTED.CORE_CYCLES_GE_2	Cycles at least 2 micro-op is executed from any thread on physical core.	CounterMask=2 CMSK2
B1H	02H	UOPS_EXECUTED.CORE_CYCLES_GE_3	Cycles at least 3 micro-op is executed from any thread on physical core.	CounterMask=3 CMSK3
B1H	02H	UOPS_EXECUTED.CORE_CYCLES_GE_4	Cycles at least 4 micro-op is executed from any thread on physical core.	CounterMask=4 CMSK4
B1H	02H	UOPS_EXECUTED.CORE_CYCLES_NONE	Cycles with no micro-ops executed from any thread on physical core.	CounterMask=1 Invert=1 CMSK1, INV
B1H	10H	UOPS_EXECUTED.X87	Counts the number of x87 uops executed.	
B2H	01H	OFFCORE_REQUESTS_BUFFER.SQ_FULL	Counts the number of cases when the offcore requests buffer cannot take more entries for the core. This can happen when the superqueue does not contain eligible entries, or when L1D writeback pending FIFO requests is full. Note: Writeback pending FIFO has six entries.	
BDH	01H	TLB_FLUSH.DTLB_THREAD	Counts the number of DTLB flush attempts of the thread-specific entries.	
BDH	20H	TLB_FLUSH.STLB_ANY	Counts the number of any STLB flush attempts (such as entire, VPID, PCID, InvPage, CR3 write, etc.).	
C0H	00H	INST_RETIRED.ANY_P	Counts the number of instructions (EOMs) retired. Counting covers macro-fused instructions individually (that is, increments by two).	See Table 19-1.

Table 19-3. Non-Architectural Performance Events of the Processor Core Supported in Intel® Xeon® Processor Scalable Family with Skylake Microarchitecture (Contd.)

Event Num.	Umask Value	Event Mask Mnemonic	Description	Comment
C0H	01H	INST_RETIRED.PREC_DIST	A version of INST_RETIRED that allows for a more unbiased distribution of samples across instructions retired. It utilizes the Precise Distribution of Instructions Retired (PDIR) feature to mitigate some bias in how retired instructions get sampled.	Precise event capable Requires PEBS on General Counter 1(PDIR).
C1H	3FH	OTHER_ASSISTS.ANY	Number of times a microcode assist is invoked by HW other than FP-assist. Examples include AD (page Access Dirty) and AVX* related assists.	
C2H	01H	UOPS_RETIRED.STALL_CYCLES	This is a non-precise version (that is, does not use PEBS) of the event that counts cycles without actually retired uops.	CounterMask=1 Invert=1 CMSK1, INV
C2H	01H	UOPS_RETIRED.TOTAL_CYCLES	Number of cycles using always true condition (uops_ret < 16) applied to non PEBS uops retired event.	CounterMask=10 Invert=1 CMSK10, INV
C2H	02H	UOPS_RETIRED.RETIRE_SLOTS	Counts the retirement slots used.	
C3H	01H	MACHINE_CLEARS.COUNT	Number of machine clears (nukes) of any type.	EdgeDetect=1 CounterMask=1 CMSK1, EDG
C3H	02H	MACHINE_CLEARS.MEMORY_ORDERING	Counts the number of memory ordering Machine Clears detected. Memory Ordering Machine Clears can result from one of the following: a. memory disambiguation, b. external snoop, or c. cross SMT-HW-thread snoop (stores) hitting load buffer.	
C3H	04H	MACHINE_CLEARS.SMC	Counts self-modifying code (SMC) detected, which causes a machine clear.	
C4H	00H	BR_INST_RETIRED.ALL_BRANCHES	Counts all (macro) branch instructions retired.	Precise event capable. See Table 19-1.
C4H	01H	BR_INST_RETIRED.CONDITIONAL	This is a non-precise version (that is, does not use PEBS) of the event that counts conditional branch instructions retired.	Precise event capable. PS
C4H	02H	BR_INST_RETIRED.NEAR_CALL	This is a non-precise version (that is, does not use PEBS) of the event that counts both direct and indirect near call instructions retired.	Precise event capable. PS
C4H	08H	BR_INST_RETIRED.NEAR_RETURN	This is a non-precise version (that is, does not use PEBS) of the event that counts return instructions retired.	Precise event capable. PS
C4H	10H	BR_INST_RETIRED.NOT_TAKEN	This is a non-precise version (that is, does not use PEBS) of the event that counts not taken branch instructions retired.	
C4H	20H	BR_INST_RETIRED.NEAR_TAKEN	This is a non-precise version (that is, does not use PEBS) of the event that counts taken branch instructions retired.	Precise event capable. PS
C4H	40H	BR_INST_RETIRED.FAR_BRANCH	This is a non-precise version (that is, does not use PEBS) of the event that counts far branch instructions retired.	Precise event capable. PS

Table 19-3. Non-Architectural Performance Events of the Processor Core Supported in Intel® Xeon® Processor Scalable Family with Skylake Microarchitecture (Contd.)

Event Num.	Umask Value	Event Mask Mnemonic	Description	Comment
C5H	00H	BR_MISP_RETIRED.ALL_BRANCHES	Counts all the retired branch instructions that were mispredicted by the processor. A branch misprediction occurs when the processor incorrectly predicts the destination of the branch. When the misprediction is discovered at execution, all the instructions executed in the wrong (speculative) path must be discarded, and the processor must start fetching from the correct path.	Precise event capable. See Table 19-1.
C5H	01H	BR_MISP_RETIRED.CONDITIONAL	This is a non-precise version (that is, does not use PEBS) of the event that counts mispredicted conditional branch instructions retired.	Precise event capable. PS
C5H	02H	BR_MISP_RETIRED.NEAR_CALL	Counts both taken and not taken retired mispredicted direct and indirect near calls, including both register and memory indirect.	Precise event capable.
C5H	20H	BR_MISP_RETIRED.NEAR_TAKEN	Number of near branch instructions retired that were mispredicted and taken.	Precise event capable. PS
C6H	01H	FRONTEND_RETIRED.DSB_MISS	Counts retired Instructions that experienced DSB (Decode stream buffer, i.e. the decoded instruction-cache) miss.	Precise event capable.
C6H	01H	FRONTEND_RETIRED.L1I_MISS	Retired Instructions who experienced Instruction L1 Cache true miss.	Precise event capable.
C6H	01H	FRONTEND_RETIRED.L2_MISS	Retired Instructions who experienced Instruction L2 Cache true miss.	Precise event capable.
C6H	01H	FRONTEND_RETIRED.ITLB_MISS	Counts retired Instructions that experienced iTLB (Instruction TLB) true miss.	Precise event capable.
C6H	01H	FRONTEND_RETIRED.STLB_MISS	Counts retired Instructions that experienced STLB (2nd level TLB) true miss.	Precise event capable.
C6H	01H	FRONTEND_RETIRED.LATENCY_GE_2	Retired instructions that are fetched after an interval where the front end delivered no uops for a period of 2 cycles which was not interrupted by a back-end stall.	Precise event capable.
C6H	01H	FRONTEND_RETIRED.LATENCY_GE_4	Retired instructions that are fetched after an interval where the front end delivered no uops for a period of 4 cycles which was not interrupted by a back-end stall.	Precise event capable.
C6H	01H	FRONTEND_RETIRED.LATENCY_GE_8	Counts retired instructions that are delivered to the back end after a front-end stall of at least 8 cycles. During this period the front end delivered no uops.	Precise event capable.
C6H	01H	FRONTEND_RETIRED.LATENCY_GE_16	Counts retired instructions that are delivered to the back end after a front-end stall of at least 16 cycles. During this period the front end delivered no uops.	Precise event capable.
C6H	01H	FRONTEND_RETIRED.LATENCY_GE_32	Counts retired instructions that are delivered to the back end after a front-end stall of at least 32 cycles. During this period the front end delivered no uops.	Precise event capable.
C6H	01H	FRONTEND_RETIRED.LATENCY_GE_64	Retired instructions that are fetched after an interval where the front end delivered no uops for a period of 64 cycles which was not interrupted by a back-end stall.	Precise event capable.

Table 19-3. Non-Architectural Performance Events of the Processor Core Supported in Intel® Xeon® Processor Scalable Family with Skylake Microarchitecture (Contd.)

Event Num.	Umask Value	Event Mask Mnemonic	Description	Comment
C6H	01H	FRONTEND_RETIRED.LATENCY_GE_128	Retired instructions that are fetched after an interval where the front end delivered no uops for a period of 128 cycles which was not interrupted by a back-end stall.	Precise event capable.
C6H	01H	FRONTEND_RETIRED.LATENCY_GE_256	Retired instructions that are fetched after an interval where the front end delivered no uops for a period of 256 cycles which was not interrupted by a back-end stall.	Precise event capable.
C6H	01H	FRONTEND_RETIRED.LATENCY_GE_512	Retired instructions that are fetched after an interval where the front end delivered no uops for a period of 512 cycles which was not interrupted by a back-end stall.	Precise event capable.
C6H	01H	FRONTEND_RETIRED.LATENCY_GE_2_BUBBLES_GE_1	Counts retired instructions that are delivered to the back end after the front end had at least 1 bubble-slot for a period of 2 cycles. A bubble-slot is an empty issue-pipeline slot while there was no RAT stall.	Precise event capable.
C6H	01H	FRONTEND_RETIRED.LATENCY_GE_2_BUBBLES_GE_2	Retired instructions that are fetched after an interval where the front end had at least 2 bubble-slots for a period of 2 cycles which was not interrupted by a back-end stall.	Precise event capable.
C6H	01H	FRONTEND_RETIRED.LATENCY_GE_2_BUBBLES_GE_3	Retired instructions that are fetched after an interval where the front end had at least 3 bubble-slots for a period of 2 cycles which was not interrupted by a back-end stall.	Precise event capable.
C7H	01H	FP_ARITH_INST_RETIRED.SCALAR_DOUBLE	Number of SSE/AVX computational scalar double precision floating-point instructions retired. Each count represents 1 computation. Applies to SSE* and AVX* scalar double precision floating-point instructions: ADD SUB MUL DIV MIN MAX SQRT FM(N)ADD/SUB. FM(N)ADD/SUB instructions count twice as they perform multiple calculations per element.	Software may treat each count as one DP FLOP.
C7H	02H	FP_ARITH_INST_RETIRED.SCALAR_SINGLE	Number of SSE/AVX computational scalar single precision floating-point instructions retired. Each count represents 1 computation. Applies to SSE* and AVX* scalar single precision floating-point instructions: ADD SUB MUL DIV MIN MAX RCP RSQRT SQRT FM(N)ADD/SUB. FM(N)ADD/SUB instructions count twice as they perform multiple calculations per element.	Software may treat each count as one SP FLOP.
C7H	04H	FP_ARITH_INST_RETIRED.128B_PACKED_DOUBLE	Number of SSE/AVX computational 128-bit packed double precision floating-point instructions retired. Each count represents 2 computations. Applies to SSE* and AVX* packed double precision floating-point instructions: ADD SUB MUL DIV MIN MAX SQRT DPP FM(N)ADD/SUB. DPP and FM(N)ADD/SUB instructions count twice as they perform multiple calculations per element.	Software may treat each count as two DP FLOPs.

Table 19-3. Non-Architectural Performance Events of the Processor Core Supported in Intel® Xeon® Processor Scalable Family with Skylake Microarchitecture (Contd.)

Event Num.	Umask Value	Event Mask Mnemonic	Description	Comment
C7H	08H	FP_ARITH_INST_RETIRED.128B_PACKED_SINGLE	Number of SSE/AVX computational 128-bit packed single precision floating-point instructions retired. Each count represents 4 computations. Applies to SSE* and AVX* packed single precision floating-point instructions: ADD SUB MUL DIV MIN MAX RCP RSQRT SQRT DPP FM(N)ADD/SUB. DPP and FM(N)ADD/SUB instructions count twice as they perform multiple calculations per element.	Software may treat each count as four SP FLOPs.
C7H	10H	FP_ARITH_INST_RETIRED.256B_PACKED_DOUBLE	Number of SSE/AVX computational 256-bit packed double precision floating-point instructions retired. Each count represents 4 computations. Applies to SSE* and AVX* packed double precision floating-point instructions: ADD SUB MUL DIV MIN MAX SQRT DPP FM(N)ADD/SUB. DPP and FM(N)ADD/SUB instructions count twice as they perform multiple calculations per element.	Software may treat each count as four DP FLOPs.
C7H	20H	FP_ARITH_INST_RETIRED.256B_PACKED_SINGLE	Number of SSE/AVX computational 256-bit packed single precision floating-point instructions retired. Each count represents 8 computations. Applies to SSE* and AVX* packed single precision floating-point instructions: ADD SUB MUL DIV MIN MAX RCP RSQRT SQRT DPP FM(N)ADD/SUB. DPP and FM(N)ADD/SUB instructions count twice as they perform multiple calculations per element.	Software may treat each count as eight SP FLOPs.
C7H	40H	FP_ARITH_INST_RETIRED.512B_PACKED_DOUBLE	Number of Packed Double-Precision FP arithmetic instructions (use operation multiplier of 8).	Only applicable when AVX-512 is enabled.
C7H	80H	FP_ARITH_INST_RETIRED.512B_PACKED_SINGLE	Number of Packed Single-Precision FP arithmetic instructions (use operation multiplier of 16).	Only applicable when AVX-512 is enabled.
C8H	01H	HLE_RETIRED.START	Number of times we entered an HLE region. Does not count nested transactions.	
C8H	02H	HLE_RETIRED.COMMIT	Number of times HLE commit succeeded.	
C8H	04H	HLE_RETIRED.ABORTED	Number of times HLE abort was triggered.	Precise event capable.
C8H	08H	HLE_RETIRED.ABORTED_MEM	Number of times an HLE execution aborted due to various memory events (e.g., read/write capacity and conflicts).	
C8H	10H	HLE_RETIRED.ABORTED_TIMER	Number of times an HLE execution aborted due to hardware timer expiration.	
C8H	20H	HLE_RETIRED.ABORTED_UNFRIENDLY	Number of times an HLE execution aborted due to HLE-unfriendly instructions and certain unfriendly events (such as AD assists etc.).	
C8H	40H	HLE_RETIRED.ABORTED_MEMTYPE	Number of times an HLE execution aborted due to incompatible memory type.	
C8H	80H	HLE_RETIRED.ABORTED_EVENTS	Number of times an HLE execution aborted due to unfriendly events (such as interrupts).	
C9H	01H	RTM_RETIRED.START	Number of times we entered an RTM region. Does not count nested transactions.	
C9H	02H	RTM_RETIRED.COMMIT	Number of times RTM commit succeeded.	
C9H	04H	RTM_RETIRED.ABORTED	Number of times RTM abort was triggered.	Precise event capable.

Table 19-3. Non-Architectural Performance Events of the Processor Core Supported in Intel® Xeon® Processor Scalable Family with Skylake Microarchitecture (Contd.)

Event Num.	Umask Value	Event Mask Mnemonic	Description	Comment
C9H	08H	RTM_RETIRED.ABORTED_MEM	Number of times an RTM execution aborted due to various memory events (e.g. read/write capacity and conflicts).	
C9H	10H	RTM_RETIRED.ABORTED_TIMER	Number of times an RTM execution aborted due to uncommon conditions.	
C9H	20H	RTM_RETIRED.ABORTED_UNFRIENDLY	Number of times an RTM execution aborted due to HLE-unfriendly instructions.	
C9H	40H	RTM_RETIRED.ABORTED_MEMTYPE	Number of times an RTM execution aborted due to incompatible memory type.	
C9H	80H	RTM_RETIRED.ABORTED_EVENTS	Number of times an RTM execution aborted due to none of the previous 4 categories (e.g. interrupt).	
CAH	1EH	FP_ASSIST.ANY	Counts cycles with any input and output SSE or x87 FP assist. If an input and output assist are detected on the same cycle the event increments by 1.	CounterMask=1 CMSK1
CBH	01H	HW_INTERRUPTS.RECEIVED	Counts the number of hardware interruptions received by the processor.	
CCH	20H	ROB_MISC_EVENTS.LBR_INSERTS	Increments when an entry is added to the Last Branch Record (LBR) array (or removed from the array in case of RETURNs in call stack mode). The event requires LBR enable via IA32_DEBUGCTL MSR and branch type selection via MSR_LBR_SELECT.	
CDH	01H	MEM_TRANS_RETIRED.LOAD_LATENCY_GT_4	Counts loads when the latency from first dispatch to completion is greater than 4 cycles. Reported latency may be longer than just the memory latency.	Precise event capable. Specify threshold in MSR 3F6H.
CDH	01H	MEM_TRANS_RETIRED.LOAD_LATENCY_GT_8	Counts loads when the latency from first dispatch to completion is greater than 8 cycles. Reported latency may be longer than just the memory latency.	Precise event capable. Specify threshold in MSR 3F6H.
CDH	01H	MEM_TRANS_RETIRED.LOAD_LATENCY_GT_16	Counts loads when the latency from first dispatch to completion is greater than 16 cycles. Reported latency may be longer than just the memory latency.	Precise event capable. Specify threshold in MSR 3F6H.
CDH	01H	MEM_TRANS_RETIRED.LOAD_LATENCY_GT_32	Counts loads when the latency from first dispatch to completion is greater than 32 cycles. Reported latency may be longer than just the memory latency.	Precise event capable. Specify threshold in MSR 3F6H.
CDH	01H	MEM_TRANS_RETIRED.LOAD_LATENCY_GT_64	Counts loads when the latency from first dispatch to completion is greater than 64 cycles. Reported latency may be longer than just the memory latency.	Precise event capable. Specify threshold in MSR 3F6H.
CDH	01H	MEM_TRANS_RETIRED.LOAD_LATENCY_GT_128	Counts loads when the latency from first dispatch to completion is greater than 128 cycles. Reported latency may be longer than just the memory latency.	Precise event capable. Specify threshold in MSR 3F6H.
CDH	01H	MEM_TRANS_RETIRED.LOAD_LATENCY_GT_256	Counts loads when the latency from first dispatch to completion is greater than 256 cycles. Reported latency may be longer than just the memory latency.	Precise event capable. Specify threshold in MSR 3F6H.
CDH	01H	MEM_TRANS_RETIRED.LOAD_LATENCY_GT_512	Counts loads when the latency from first dispatch to completion is greater than 512 cycles. Reported latency may be longer than just the memory latency.	Precise event capable. Specify threshold in MSR 3F6H.
D0H	11H	MEM_INST_RETIRED.STLB_MISS_LOADS	Retired load instructions that miss the STLB.	Precise event capable. PSDLA

Table 19-3. Non-Architectural Performance Events of the Processor Core Supported in Intel® Xeon® Processor Scalable Family with Skylake Microarchitecture (Contd.)

Event Num.	Umask Value	Event Mask Mnemonic	Description	Comment
D0H	12H	MEM_INST_RETIRED.STLB_MISS_STORES	Retired store instructions that miss the STLB.	Precise event capable. PSDLA
D0H	21H	MEM_INST_RETIRED.LOCK_LOADS	Retired load instructions with locked access.	Precise event capable. PSDLA
D0H	41H	MEM_INST_RETIRED.SPLIT_LOADS	Counts retired load instructions that split across a cacheline boundary.	Precise event capable. PSDLA
D0H	42H	MEM_INST_RETIRED.SPLIT_STORES	Counts retired store instructions that split across a cacheline boundary.	Precise event capable. PSDLA
D0H	81H	MEM_INST_RETIRED.ALL_LOADS	All retired load instructions.	Precise event capable. PSDLA
D0H	82H	MEM_INST_RETIRED.ALL_STORES	All retired store instructions.	Precise event capable. PSDLA
D1H	01H	MEM_LOAD_RETIRED.L1_HIT	Counts retired load instructions with at least one uop that hit in the L1 data cache. This event includes all SW prefetches and lock instructions regardless of the data source.	Precise event capable. PSDLA
D1H	02H	MEM_LOAD_RETIRED.L2_HIT	Retired load instructions with L2 cache hits as data sources.	Precise event capable. PSDLA
D1H	04H	MEM_LOAD_RETIRED.L3_HIT	Counts retired load instructions with at least one uop that hit in the L3 cache.	Precise event capable. PSDLA
D1H	08H	MEM_LOAD_RETIRED.L1_MISS	Counts retired load instructions with at least one uop that missed in the L1 cache.	Precise event capable. PSDLA
D1H	10H	MEM_LOAD_RETIRED.L2_MISS	Retired load instructions missed L2 cache as data sources.	Precise event capable. PSDLA
D1H	20H	MEM_LOAD_RETIRED.L3_MISS	Counts retired load instructions with at least one uop that missed in the L3 cache.	Precise event capable. PSDLA
D1H	40H	MEM_LOAD_RETIRED.FB_HIT	Counts retired load instructions with at least one uop was load missed in L1 but hit FB (Fill Buffers) due to preceding miss to the same cache line with data not ready.	Precise event capable. PSDLA
D2H	01H	MEM_LOAD_L3_HIT_RETIRED.XSNP_MISS	Retired load instructions which data sources were L3 hit and cross-core snoop missed in on-pkg core cache.	Precise event capable. PSDLA
D2H	02H	MEM_LOAD_L3_HIT_RETIRED.XSNP_HIT	Retired load instructions which data sources were L3 and cross-core snoop hits in on-pkg core cache.	Precise event capable. PSDLA
D2H	04H	MEM_LOAD_L3_HIT_RETIRED.XSNP_HITM	Retired load instructions which data sources were HitM responses from shared L3.	Precise event capable. PSDLA
D2H	08H	MEM_LOAD_L3_HIT_RETIRED.XSNP_NONE	Retired load instructions which data sources were hits in L3 without snoops required.	Precise event capable. PSDLA
D3H	01H	MEM_LOAD_L3_MISS_RETIRED.LOCAL_DRAM	Retired load instructions which data sources missed L3 but serviced from local DRAM.	Precise event capable.
D3H	02H	MEM_LOAD_L3_MISS_RETIRED.REMOTE_DRAM	Retired load instructions which data sources missed L3 but serviced from remote dram.	Precise event capable.
D3H	04H	MEM_LOAD_L3_MISS_RETIRED.REMOTE_HITM	Retired load instructions whose data sources was remote HITM.	Precise event capable.
D3H	08H	MEM_LOAD_L3_MISS_RETIRED.REMOTE_FWD	Retired load instructions whose data sources was forwarded from a remote cache.	

Table 19-3. Non-Architectural Performance Events of the Processor Core Supported in Intel® Xeon® Processor Scalable Family with Skylake Microarchitecture (Contd.)

Event Num.	Umask Value	Event Mask Mnemonic	Description	Comment
D4H	04H	MEM_LOAD_MISC_RETIRED.UC	Retired instructions with at least 1 uncacheable load or lock.	Precise event capable.
E6H	01H	BACLEARS.ANY	Counts the number of times the front-end is resteered when it finds a branch instruction in a fetch line. This occurs for the first time a branch instruction is fetched or when the branch is not tracked by the BPU (Branch Prediction Unit) anymore.	
F0H	40H	L2_TRANS.L2_WB	Counts L2 writebacks that access L2 cache.	
F1H	1FH	L2_LINES_IN.ALL	Counts the number of L2 cache lines filling the L2. Counting does not cover rejects.	
F2H	01H	L2_LINES_OUT.SILENT	Counts the number of lines that are silently dropped by L2 cache when triggered by an L2 cache fill. These lines are typically in Shared state. A non-threaded event.	
F2H	02H	L2_LINES_OUT.NON_SILENT	Counts the number of lines that are evicted by L2 cache when triggered by an L2 cache fill. Those lines can be either in modified state or clean state. Modified lines may either be written back to L3 or directly written to memory and not allocated in L3. Clean lines may either be allocated in L3 or dropped.	
F2H	04H	L2_LINES_OUT.USELESS_PREF	Counts the number of lines that have been hardware prefetched but not used and now evicted by L2 cache.	
F2H	04H	L2_LINES_OUT.USELESS_HWPF	Counts the number of lines that have been hardware prefetched but not used and now evicted by L2 cache.	
F4H	10H	SQ_MISC.SPLIT_LOCK	Counts the number of cache line split locks sent to the uncore.	
FEH	02H	IDI_MISC.WB_UPGRADE	Counts number of cache lines that are allocated and written back to L3 with the intention that they are more likely to be reused shortly.	
FEH	04H	IDI_MISC.WB_DOWNGRADE	Counts number of cache lines that are dropped and not written back to L3 as they are deemed to be less likely to be reused shortly.	

19.3 PERFORMANCE MONITORING EVENTS FOR 6TH GENERATION INTEL® CORE™ PROCESSOR AND 7TH GENERATION INTEL® CORE™ PROCESSOR

6th Generation Intel® Core™ processors are based on the Skylake microarchitecture. They support the architectural performance-monitoring events listed in Table 19-1. Fixed counters in the core PMU support the architecture events defined in Table 19-2. Non-architectural performance-monitoring events in the processor core are listed in Table 19-4. The events in Table 19-4 apply to processors with CPUID signature of DisplayFamily_DisplayModel encoding with the following values: 06_4EH and 06_5EH. Table 19-10 lists performance events supporting Intel TSX (see Section 18.3.6.5) and the events are applicable to processors based on Skylake microarchitecture. Where Skylake microarchitecture implements TSX-related event semantics that differ from Table 19-10, they are listed in Table 19-5.

7th Generation Intel® Core™ processors are based on the Kaby Lake microarchitecture. Non-architectural performance-monitoring events in the processor core are listed in Table 19-4. The events in Table 19-4 apply to processors with CPUID signature of DisplayFamily_DisplayModel encoding with the following values: 06_8EH and 06_9EH.

The comment column in Table 19-4 uses abbreviated letters to indicate additional conditions applicable to the Event Mask Mnemonic. For event umasks listed in Table 19-4 that do not show "AnyT", users should refrain from programming "AnyThread =1" in IA32_PERF_EVTSELx.

Table 19-4. Non-Architectural Performance Events of the Processor Core Supported by Skylake Microarchitecture and Kaby Lake Microarchitecture

Event Num.	Umask Value	Event Mask Mnemonic	Description	Comment
03H	02H	LD_BLOCKS.STORE_FORWARD	Loads blocked by overlapping with store buffer that cannot be forwarded.	
03H	08H	LD_BLOCKS.NO_SR	The number of times that split load operations are temporarily blocked because all resources for handling the split accesses are in use.	
07H	01H	LD_BLOCKS_PARTIAL.ADDRESS_ALIAS	False dependencies in MOB due to partial compare on address.	
08H	01H	DTLB_LOAD_MISSES.MISS_CAUSES_A_WALK	Load misses in all TLB levels that cause a page walk of any page size.	
08H	0EH	DTLB_LOAD_MISSES.WALK_COMPLETED	Load misses in all TLB levels causes a page walk that completes. (All page sizes.)	
08H	10H	DTLB_LOAD_MISSES.WALK_PENDING	Counts 1 per cycle for each PMH that is busy with a page walk for a load.	
08H	10H	DTLB_LOAD_MISSES.WALK_ACTIVE	Cycles when at least one PMH is busy with a walk for a load.	CMSK1
08H	20H	DTLB_LOAD_MISSES.STLB_HIT	Loads that miss the DTLB but hit STLB.	
0DH	01H	INT_MISC.RECOVERY_CYCLES	Core cycles the allocator was stalled due to recovery from earlier machine clear event for this thread (for example, misprediction or memory order conflict).	
0DH	01H	INT_MISC.RECOVERY_CYCLES_ANY	Core cycles the allocator was stalled due to recovery from earlier machine clear event for any logical thread in this processor core.	AnyT
0DH	80H	INT_MISC.CLEAR_RESTEER_CYCLES	Cycles the issue-stage is waiting for front end to fetch from resteered path following branch misprediction or machine clear events.	
0EH	01H	UOPS_ISSUED.ANY	The number of uops issued by the RAT to RS.	
0EH	01H	UOPS_ISSUED.STALL_CYCLES	Cycles when the RAT does not issue uops to RS for the thread.	CMSK1, INV
0EH	02H	UOPS_ISSUED.VECTOR_WIDTH_MISMATCH	Uops inserted at issue-stage in order to preserve upper bits of vector registers.	
0EH	20H	UOPS_ISSUED.SLOW_LEA	Number of slow LEA or similar uops allocated. Such uop has 3 sources (for example, 2 sources + immediate) regardless of whether it is a result of LEA instruction or not.	
14H	01H	ARITH.FPU_DIVIDER_ACTIVE	Cycles when divider is busy executing divide or square root operations. Accounts for FP operations including integer divides.	
24H	21H	L2_RQSTS.DEMAND_DATA_RD_MISS	Demand Data Read requests that missed L2, no rejects.	
24H	22H	L2_RQSTS.RFO_MISS	RFO requests that missed L2.	
24H	24H	L2_RQSTS.CODE_RD_MISS	L2 cache misses when fetching instructions.	
24H	27H	L2_RQSTS.ALL_DEMAND_MISS	Demand requests that missed L2.	

Table 19-4. Non-Architectural Performance Events of the Processor Core Supported by Skylake Microarchitecture and Kaby Lake Microarchitecture (Contd.)

Event Num.	Umask Value	Event Mask Mnemonic	Description	Comment
24H	38H	L2_RQSTS.PF_MISS	Requests from the L1/L2/L3 hardware prefetchers or load software prefetches that miss L2 cache.	
24H	3FH	L2_RQSTS.MISS	All requests that missed L2.	
24H	41H	L2_RQSTS.DEMAND_DATA_RD_HIT	Demand Data Read requests that hit L2 cache.	
24H	42H	L2_RQSTS.RFO_HIT	RFO requests that hit L2 cache.	
24H	44H	L2_RQSTS.CODE_RD_HIT	L2 cache hits when fetching instructions.	
24H	D8H	L2_RQSTS.PF_HIT	Prefetches that hit L2.	
24H	E1H	L2_RQSTS.ALL_DEMAND_DATA_RD	All demand data read requests to L2.	
24H	E2H	L2_RQSTS.ALL_RFO	All L RFO requests to L2.	
24H	E4H	L2_RQSTS.ALL_CODE_RD	All L2 code requests.	
24H	E7H	L2_RQSTS.ALL_DEMAND_REFERENCES	All demand requests to L2.	
24H	F8H	L2_RQSTS.ALL_PF	All requests from the L1/L2/L3 hardware prefetchers or load software prefetches.	
24H	EFH	L2_RQSTS.REFERENCES	All requests to L2.	
2EH	4FH	LONGEST_LAT_CACHE.REFERENCE	This event counts requests originating from the core that reference a cache line in the L3 cache.	See Table 19-1.
2EH	41H	LONGEST_LAT_CACHE.MISS	This event counts each cache miss condition for references to the L3 cache.	See Table 19-1.
3CH	00H	CPU_CLK_UNHALTED.THREAD_P	Cycles while the logical processor is not in a halt state.	See Table 19-1.
3CH	00H	CPU_CLK_UNHALTED.THREAD_P_ANY	Cycles while at least one logical processor is not in a halt state.	AnyT
3CH	01H	CPU_CLK_THREAD_UNHALTED.REF_XCLK	Core crystal clock cycles when the thread is unhalted.	See Table 19-1.
3CH	01H	CPU_CLK_THREAD_UNHALTED.REF_XCLK_ANY	Core crystal clock cycles when at least one thread on the physical core is unhalted.	AnyT
3CH	02H	CPU_CLK_THREAD_UNHALTED.ONE_THREAD_ACTIVE	Core crystal clock cycles when this thread is unhalted and the other thread is halted.	
48H	01H	L1D_PEND_MISS.PENDING	Increments the number of outstanding L1D misses every cycle.	
48H	01H	L1D_PEND_MISS.PENDING_CYCLES	Cycles with at least one outstanding L1D misses from this logical processor.	CMSK1
48H	01H	L1D_PEND_MISS.PENDING_CYCLES_ANY	Cycles with at least one outstanding L1D misses from any logical processor in this core.	CMSK1, AnyT
48H	02H	L1D_PEND_MISS.FB_FULL	Number of times a request needed a FB entry but there was no entry available for it. That is, the FB unavailability was the dominant reason for blocking the request. A request includes cacheable/uncacheable demand that is load, store or SW prefetch. HWP are excluded.	
49H	01H	DTLB_STORE_MISSES.MISS_CAUSES_A_WALK	Store misses in all TLB levels that cause page walks.	

Table 19-4. Non-Architectural Performance Events of the Processor Core Supported by Skylake Microarchitecture and Kaby Lake Microarchitecture (Contd.)

Event Num.	Umask Value	Event Mask Mnemonic	Description	Comment
49H	0EH	DTLB_STORE_MISSES.WALK_COMPLETED	Counts completed page walks in any TLB levels due to store misses (all page sizes).	
49H	10H	DTLB_STORE_MISSES.WALK_PENDING	Counts 1 per cycle for each PMH that is busy with a page walk for a store.	
49H	10H	DTLB_STORE_MISSES.WALK_ACTIVE	Cycles when at least one PMH is busy with a page walk for a store.	CMSK1
49H	20H	DTLB_STORE_MISSES.STLB_HIT	Store misses that missed DTLB but hit STLB.	
4CH	01H	LOAD_HIT_PRE.HW_PF	Demand load dispatches that hit fill buffer allocated for software prefetch.	
4FH	10H	EPT.WALK_PENDING	Counts 1 per cycle for each PMH that is busy with an EPT walk for any request type.	
51H	01H	L1D.REPLACEMENT	Counts the number of lines brought into the L1 data cache.	
5EH	01H	RS_EVENTS.EMPTY_CYCLES	Cycles the RS is empty for the thread.	
5EH	01H	RS_EVENTS.EMPTY_END	Counts end of periods where the Reservation Station (RS) was empty. Could be useful to precisely locate Front-end Latency Bound issues.	CMSK1, INV
60H	01H	OFFCORE_REQUESTS_OUTSTANDING.DEMAND_DATA_RD	Increment each cycle of the number of offcore outstanding Demand Data Read transactions in SQ to uncore.	
60H	01H	OFFCORE_REQUESTS_OUTSTANDING.CYCLES_WITH_DEMAND_DATA_RD	Cycles with at least one offcore outstanding Demand Data Read transactions in SQ to uncore.	CMSK1
60H	01H	OFFCORE_REQUESTS_OUTSTANDING.DEMAND_DATA_RD_GE_6	Cycles with at least 6 offcore outstanding Demand Data Read transactions in SQ to uncore.	CMSK6
60H	02H	OFFCORE_REQUESTS_OUTSTANDING.DEMAND_CODE_RD	Increment each cycle of the number of offcore outstanding demand code read transactions in SQ to uncore.	
60H	02H	OFFCORE_REQUESTS_OUTSTANDING.CYCLES_WITH_DEMAND_CODE_RD	Cycles with at least one offcore outstanding demand code read transactions in SQ to uncore.	CMSK1
60H	04H	OFFCORE_REQUESTS_OUTSTANDING.DEMAND_RFO	Increment each cycle of the number of offcore outstanding RFO store transactions in SQ to uncore. Set Cmask=1 to count cycles.	
60H	04H	OFFCORE_REQUESTS_OUTSTANDING.CYCLES_WITH_DEMAND_RFO	Cycles with at least one offcore outstanding RFO transactions in SQ to uncore.	CMSK1
60H	08H	OFFCORE_REQUESTS_OUTSTANDING.ALL_DATA_RD	Increment each cycle of the number of offcore outstanding cacheable data read transactions in SQ to uncore. Set Cmask=1 to count cycles.	
60H	08H	OFFCORE_REQUESTS_OUTSTANDING.CYCLES_WITH_DATA_RD	Cycles with at least one offcore outstanding data read transactions in SQ to uncore.	CMSK1
60H	10H	OFFCORE_REQUESTS_OUTSTANDING.L3_MISS_DEMAND_DATA_RD	Increment each cycle of the number of offcore outstanding demand data read requests from SQ that missed L3.	

Table 19-4. Non-Architectural Performance Events of the Processor Core Supported by Skylake Microarchitecture and Kaby Lake Microarchitecture (Contd.)

Event Num.	Umask Value	Event Mask Mnemonic	Description	Comment
60H	10H	OFFCORE_REQUESTS_OUTSTANDING.CYCLES_WITH_L3_MISS_DEMAND_DATA_RD	Cycles with at least one offcore outstanding demand data read requests from SQ that missed L3.	CMSK1
60H	10H	OFFCORE_REQUESTS_OUTSTANDING.L3_MISS_DEMAND_DATA_RD_GE_6	Cycles with at least one offcore outstanding demand data read requests from SQ that missed L3.	CMSK6
63H	02H	LOCK_CYCLES.CACHE_LOCK_DURATION	Cycles in which the L1D is locked.	
79H	04H	IDQ.MITE_UOPS	Increment each cycle # of uops delivered to IDQ from MITE path.	
79H	04H	IDQ.MITE_CYCLES	Cycles when uops are being delivered to IDQ from MITE path.	CMSK1
79H	08H	IDQ.DSB_UOPS	Increment each cycle. # of uops delivered to IDQ from DSB path.	
79H	08H	IDQ.DSB_CYCLES	Cycles when uops are being delivered to IDQ from DSB path.	CMSK1
79H	10H	IDQ.MS_DSB_UOPS	Increment each cycle # of uops delivered to IDQ by DSB when MS_busy.	
79H	18H	IDQ.ALL_DSB_CYCLES_ANY_UOPS	Cycles DSB is delivered at least one uops.	CMSK1
79H	18H	IDQ.ALL_DSB_CYCLES_4_UOPS	Cycles DSB is delivered four uops.	CMSK4
79H	20H	IDQ.MS_MITE_UOPS	Increment each cycle # of uops delivered to IDQ by MITE when MS_busy.	
79H	24H	IDQ.ALL_MITE_CYCLES_ANY_UOPS	Counts cycles MITE is delivered at least one uops.	CMSK1
79H	24H	IDQ.ALL_MITE_CYCLES_4_UOPS	Counts cycles MITE is delivered four uops.	CMSK4
79H	30H	IDQ.MS_UOPS	Increment each cycle # of uops delivered to IDQ while MS is busy.	
79H	30H	IDQ.MS_SWITCHES	Number of switches from DSB or MITE to MS.	EDG
79H	30H	IDQ.MS_CYCLES	Cycles MS is delivered at least one uops.	CMSK1
80H	04H	ICACHE_16B.IFDATA_STALL	Cycles where a code fetch is stalled due to L1 instruction cache miss.	
80H	04H	ICACHE_64B.IFDATA_STALL	Cycles where a code fetch is stalled due to L1 instruction cache tag miss.	
83H	01H	ICACHE_64B.IFTAG_HIT	Instruction fetch tag lookups that hit in the instruction cache (L1I). Counts at 64-byte cache-line granularity.	
83H	02H	ICACHE_64B.IFTAG_MISS	Instruction fetch tag lookups that miss in the instruction cache (L1I). Counts at 64-byte cache-line granularity.	
85H	01H	ITLB_MISSES.MISS_CAUSES_A_WALK	Misses at all ITLB levels that cause page walks.	
85H	0EH	ITLB_MISSES.WALK_COMPLETED	Counts completed page walks in any TLB level due to code fetch misses (all page sizes).	
85H	10H	ITLB_MISSES.WALK_PENDING	Counts 1 per cycle for each PMH that is busy with a page walk for an instruction fetch request.	

Table 19-4. Non-Architectural Performance Events of the Processor Core Supported by Skylake Microarchitecture and Kaby Lake Microarchitecture (Contd.)

Event Num.	Umask Value	Event Mask Mnemonic	Description	Comment
85H	20H	ITLB_MISSES.STLB_HIT	ITLB misses that hit STLB.	
87H	01H	ILD_STALL.LCP	Stalls caused by changing prefix length of the instruction.	
9CH	01H	IDQ_UOPS_NOT_DELIVERED.CORE	Count issue pipeline slots where no uop was delivered from the front end to the back end when there is no back-end stall.	
9CH	01H	IDQ_UOPS_NOT_DELIVERED.CYCLES_0_UOP_DELIV.CORE	Cycles which 4 issue pipeline slots had no uop delivered from the front end to the back end when there is no back-end stall.	CMSK4
9CH	01H	IDQ_UOPS_NOT_DELIVERED.CYCLES_LE_n_UOP_DELIV.CORE	Cycles which "4-n" issue pipeline slots had no uop delivered from the front end to the back end when there is no back-end stall.	Set CMSK = 4-n; n = 1, 2, 3
9CH	01H	IDQ_UOPS_NOT_DELIVERED.CYCLES_FE_WAS_OK	Cycles which front end delivered 4 uops or the RAT was stalling FE.	CMSK, INV
A1H	01H	UOPS_DISPATCHED_PORT.PORT_0	Counts the number of cycles in which a uop is dispatched to port 0.	
A1H	02H	UOPS_DISPATCHED_PORT.PORT_1	Counts the number of cycles in which a uop is dispatched to port 1.	
A1H	04H	UOPS_DISPATCHED_PORT.PORT_2	Counts the number of cycles in which a uop is dispatched to port 2.	
A1H	08H	UOPS_DISPATCHED_PORT.PORT_3	Counts the number of cycles in which a uop is dispatched to port 3.	
A1H	10H	UOPS_DISPATCHED_PORT.PORT_4	Counts the number of cycles in which a uop is dispatched to port 4.	
A1H	20H	UOPS_DISPATCHED_PORT.PORT_5	Counts the number of cycles in which a uop is dispatched to port 5.	
A1H	40H	UOPS_DISPATCHED_PORT.PORT_6	Counts the number of cycles in which a uop is dispatched to port 6.	
A1H	80H	UOPS_DISPATCHED_PORT.PORT_7	Counts the number of cycles in which a uop is dispatched to port 7.	
A2H	01H	RESOURCE_STALLS.ANY	Resource-related stall cycles.	
A2H	08H	RESOURCE_STALLS.SB	Cycles stalled due to no store buffers available (not including draining form sync).	
A3H	01H	CYCLE_ACTIVITY.CYCLES_L2_MISS	Cycles while L2 cache miss demand load is outstanding.	CMSK1
A3H	02H	CYCLE_ACTIVITY.CYCLES_L3_MISS	Cycles while L3 cache miss demand load is outstanding.	CMSK2
A3H	04H	CYCLE_ACTIVITY.STALLS_TOTAL	Total execution stalls.	CMSK4
A3H	05H	CYCLE_ACTIVITY.STALLS_L2_MISS	Execution stalls while L2 cache miss demand load is outstanding.	CMSK5
A3H	06H	CYCLE_ACTIVITY.STALLS_L3_MISS	Execution stalls while L3 cache miss demand load is outstanding.	CMSK6
A3H	08H	CYCLE_ACTIVITY.CYCLES_L1D_MISS	Cycles while L1 data cache miss demand load is outstanding.	CMSK8
A3H	0CH	CYCLE_ACTIVITY.STALLS_L1D_MISS	Execution stalls while L1 data cache miss demand load is outstanding.	CMSK12

Table 19-4. Non-Architectural Performance Events of the Processor Core Supported by Skylake Microarchitecture and Kaby Lake Microarchitecture (Contd.)

Event Num.	Umask Value	Event Mask Mnemonic	Description	Comment
A3H	10H	CYCLE_ACTIVITY.CYCLES_MEM_ANY	Cycles while memory subsystem has an outstanding load.	CMSK16
A3H	14H	CYCLE_ACTIVITY.STALLS_MEM_ANY	Execution stalls while memory subsystem has an outstanding load.	CMSK20
A6H	01H	EXE_ACTIVITY.EXE_BOUND_0_PORTS	Cycles for which no uops began execution, the Reservation Station was not empty, the Store Buffer was full and there was no outstanding load.	
A6H	02H	EXE_ACTIVITY.1_PORTS_UTIL	Cycles for which one uop began execution on any port, and the Reservation Station was not empty.	
A6H	04H	EXE_ACTIVITY.2_PORTS_UTIL	Cycles for which two uops began execution, and the Reservation Station was not empty.	
A6H	08H	EXE_ACTIVITY.3_PORTS_UTIL	Cycles for which three uops began execution, and the Reservation Station was not empty.	
A6H	04H	EXE_ACTIVITY.4_PORTS_UTIL	Cycles for which four uops began execution, and the Reservation Station was not empty.	
A6H	40H	EXE_ACTIVITY.BOUND_ON_STORES	Cycles where the Store Buffer was full and no outstanding load.	
A8H	01H	LSD.UOPS	Number of uops delivered by the LSD.	
A8H	01H	LSD.CYCLES_ACTIVE	Cycles with at least one uop delivered by the LSD and none from the decoder.	CMSK1
A8H	01H	LSD.CYCLES_4_UOPS	Cycles with 4 uops delivered by the LSD and none from the decoder.	CMSK4
ABH	02H	DSB2MITE_SWITCHES.PENALTY_CYCLES	DSB-to-MITE switch true penalty cycles.	
AEH	01H	ITLB.ITLB_FLUSH	Flushing of the Instruction TLB (ITLB) pages, includes 4k/2M/4M pages.	
B0H	01H	OFFCORE_REQUESTS.DEMAND_DATA_RD	Demand data read requests sent to uncore.	
B0H	02H	OFFCORE_REQUESTS.DEMAND_CODE_RD	Demand code read requests sent to uncore.	
B0H	04H	OFFCORE_REQUESTS.DEMAND_RFO	Demand RFO read requests sent to uncore, including regular RFOs, locks, ItoM.	
B0H	08H	OFFCORE_REQUESTS.ALL_DATA_RD	Data read requests sent to uncore (demand and prefetch).	
B0H	10H	OFFCORE_REQUESTS.L3_MISS_DEMAND_DATA_RD	Demand data read requests that missed L3.	
B0H	80H	OFFCORE_REQUESTS.ALL_REQUESTS	Any memory transaction that reached the SQ.	
B1H	01H	UOPS_EXECUTED.THREAD	Counts the number of uops that begin execution across all ports.	
B1H	01H	UOPS_EXECUTED.STALL_CYCLES	Cycles where there were no uops that began execution.	CMSK, INV
B1H	01H	UOPS_EXECUTED.CYCLES_GE_1_UOP_EXEC	Cycles where there was at least one uop that began execution.	CMSK1
B1H	01H	UOPS_EXECUTED.CYCLES_GE_2_UOP_EXEC	Cycles where there were at least two uops that began execution.	CMSK2

Table 19-4. Non-Architectural Performance Events of the Processor Core Supported by Skylake Microarchitecture and Kaby Lake Microarchitecture (Contd.)

Event Num.	Umask Value	Event Mask Mnemonic	Description	Comment
B1H	01H	UOPS_EXECUTED.CYCLES_GE_3_UOP_EXEC	Cycles where there were at least three uops that began execution.	CMSK3
B1H	01H	UOPS_EXECUTED.CYCLES_GE_4_UOP_EXEC	Cycles where there were at least four uops that began execution.	CMSK4
B1H	02H	UOPS_EXECUTED.CORE	Counts the number of uops from any logical processor in this core that begin execution.	
B1H	02H	UOPS_EXECUTED.CORE_CYCLES_GE_1	Cycles where there was at least one uop, from any logical processor in this core, that began execution.	CMSK1
B1H	02H	UOPS_EXECUTED.CORE_CYCLES_GE_2	Cycles where there were at least two uops, from any logical processor in this core, that began execution.	CMSK2
B1H	02H	UOPS_EXECUTED.CORE_CYCLES_GE_3	Cycles where there were at least three uops, from any logical processor in this core, that began execution.	CMSK3
B1H	02H	UOPS_EXECUTED.CORE_CYCLES_GE_4	Cycles where there were at least four uops, from any logical processor in this core, that began execution.	CMSK4
B1H	02H	UOPS_EXECUTED.CORE_CYCLES_NONE	Cycles where there were no uops from any logical processor in this core that began execution.	CMSK1, INV
B1H	10H	UOPS_EXECUTED.X87	Counts the number of X87 uops that begin execution.	
B2H	01H	OFF_CORE_REQUEST_BUFFER.SQ_FULL	Offcore requests buffer cannot take more entries for this core.	
B7H	01H	OFF_CORE_RESPONSE_0	See Section 18.3.4.5, "Off-core Response Performance Monitoring".	Requires MSR 01A6H
BBH	01H	OFF_CORE_RESPONSE_1	See Section 18.3.4.5, "Off-core Response Performance Monitoring".	Requires MSR 01A7H
BDH	01H	TLB_FLUSH.DTLB_THREAD	DTLB flush attempts of the thread-specific entries.	
BDH	01H	TLB_FLUSH.STLB_ANY	STLB flush attempts.	
C0H	00H	INST_RETIRED.ANY_P	Number of instructions at retirement.	See Table 19-1.
C0H	01H	INST_RETIRED.PREC_DIST	Precise instruction retired event with HW to reduce effect of PEBS shadow in IP distribution.	PMC1 only;
C0H	01H	INST_RETIRED.TOTAL_CYCLES	Number of cycles using always true condition applied to PEBS instructions retired event.	CMSK10, PS
C1H	3FH	OTHER_ASSISTS.ANY	Number of times a microcode assist is invoked by HW other than FP-assist. Examples include AD (page Access Dirty) and AVX* related assists.	
C2H	01H	UOPS_RETIRED.STALL_CYCLES	Cycles without actually retired uops.	CMSK1, INV
C2H	01H	UOPS_RETIRED.TOTAL_CYCLES	Cycles with less than 10 actually retired uops.	CMSK10, INV
C2H	02H	UOPS_RETIRED.RETIRE_SLOTS	Retirement slots used.	
C3H	01H	MACHINE_CLEARS.COUNT	Number of machine clears of any type.	CMSK1, EDG
C3H	02H	MACHINE_CLEARS.MEMORY_ORDERING	Counts the number of machine clears due to memory order conflicts.	
C3H	04H	MACHINE_CLEARS.SMC	Number of self-modifying-code machine clears detected.	
C4H	00H	BR_INST_RETIRED.ALL_BRANCHES	Branch instructions that retired.	See Table 19-1.
C4H	01H	BR_INST_RETIRED.CONDITIONAL	Counts the number of conditional branch instructions retired.	PS

Table 19-4. Non-Architectural Performance Events of the Processor Core Supported by Skylake Microarchitecture and Kaby Lake Microarchitecture (Contd.)

Event Num.	Umask Value	Event Mask Mnemonic	Description	Comment
C4H	02H	BR_INST_RETIRED.NEAR_CALL	Direct and indirect near call instructions retired.	PS
C4H	04H	BR_INST_RETIRED.ALL_BRANCHES	Counts the number of branch instructions retired.	PS
C4H	08H	BR_INST_RETIRED.NEAR_RETURN	Counts the number of near return instructions retired.	PS
C4H	10H	BR_INST_RETIRED.NOT_TAKEN	Counts the number of not taken branch instructions retired.	
C4H	20H	BR_INST_RETIRED.NEAR_TAKEN	Number of near taken branches retired.	PS
C4H	40H	BR_INST_RETIRED.FAR_BRANCH	Number of far branches retired.	PS
C5H	00H	BR_MISP_RETIRED.ALL_BRANCHES	Mispredicted branch instructions at retirement.	See Table 19-1.
C5H	01H	BR_MISP_RETIRED.CONDITIONAL	Mispredicted conditional branch instructions retired.	PS
C5H	04H	BR_MISP_RETIRED.ALL_BRANCHES	Mispredicted macro branch instructions retired.	PS
C5H	20H	BR_MISP_RETIRED.NEAR_TAKEN	Number of near branch instructions retired that were mispredicted and taken.	PS
C6H	01H	FRONTEND_RETIRED.DSB_MISS	Retired instructions which experienced DSB miss. Specify MSR_PEBS_FRONTEND.EVTSEL=11H.	PS
C6H	01H	FRONTEND_RETIRED.L1I_MISS	Retired instructions which experienced instruction L1 cache true miss. Specify MSR_PEBS_FRONTEND.EVTSEL=12H.	PS
C6H	01H	FRONTEND_RETIRED.L2_MISS	Retired instructions which experienced L2 cache true miss. Specify MSR_PEBS_FRONTEND.EVTSEL=13H.	PS
C6H	01H	FRONTEND_RETIRED.ITLB_MISS	Retired instructions which experienced ITLB true miss. Specify MSR_PEBS_FRONTEND.EVTSEL=14H.	PS
C6H	01H	FRONTEND_RETIRED.STLB_MISS	Retired instructions which experienced STLB true miss. Specify MSR_PEBS_FRONTEND.EVTSEL=15H.	PS
C6H	01H	FRONTEND_RETIRED.LATENCY_GE_16	Retired instructions that are fetched after an interval where the front end delivered no uops for at least 16 cycles. Specify the following fields in MSR_PEBS_FRONTEND: EVTSEL=16H, IDQ_Bubble_Length =16, IDQ_Bubble_Width = 4.	PS
C6H	01H	FRONTEND_RETIRED.LATENCY_GE_2_BUBBLES_GE_m	Retired instructions that are fetched after an interval where the front end had 'm' IDQ slots delivered, no uops for at least 2 cycles. Specify the following fields in MSR_PEBS_FRONTEND: EVTSEL=16H, IDQ_Bubble_Length =2, IDQ_Bubble_Width = m.	PS, m = 1, 2, 3
C7H	01H	FP_ARITH_INST_RETIRED.SCALAR_DOUBLE	Number of double-precision, floating-point, scalar SSE/AVX computational instructions that are retired. Each scalar FMA instruction counts as 2.	Software may treat each count as one DP FLOP.
C7H	02H	FP_ARITH_INST_RETIRED.SCALAR_SINGLE	Number of single-precision, floating-point, scalar SSE/AVX computational instructions that are retired. Each scalar FMA instruction counts as 2.	Software may treat each count as one SP FLOP.

Table 19-4. Non-Architectural Performance Events of the Processor Core Supported by Skylake Microarchitecture and Kaby Lake Microarchitecture (Contd.)

Event Num.	Umask Value	Event Mask Mnemonic	Description	Comment
C7H	04H	FP_ARITH_INST_RETIRED.128B_PACKED_DOUBLE	Number of double-precision, floating-point, 128-bit SSE/AVX computational instructions that are retired. Each 128-bit FMA or (V)DPPD instruction counts as 2.	Software may treat each count as two DP FLOPs.
C7H	08H	FP_ARITH_INST_RETIRED.128B_PACKED_SINGLE	Number of single-precision, floating-point, 128-bit SSE/AVX computational instructions that are retired. Each 128-bit FMA or (V)DPPS instruction counts as 2.	Software may treat each count as four SP FLOPs.
C7H	10H	FP_ARITH_INST_RETIRED.256B_PACKED_DOUBLE	Number of double-precision, floating-point, 256-bit SSE/AVX computational instructions that are retired. Each 256-bit FMA instruction counts as 2.	Software may treat each count as four DP FLOPs.
C7H	20H	FP_ARITH_INST_RETIRED.256B_PACKED_SINGLE	Number of single-precision, floating-point, 256-bit SSE/AVX computational instructions that are retired. Each 256-bit FMA or VDPPS instruction counts as 2.	Software may treat each count as eight SP FLOPs.
CAH	1EH	FP_ASSIST.ANY	Cycles with any input/output SSE* or FP assists.	CMSK1
CBH	01H	HW_INTERRUPTS.RECEIVED	Number of hardware interrupts received by the processor.	
CDH	01H	MEM_TRANS_RETIRED.LOAD_LATENCY	Randomly sampled loads whose latency is above a user defined threshold. A small fraction of the overall loads are sampled due to randomization.	Specify threshold in MSR 3F6H. PSDLA
D0H	11H	MEM_INST_RETIRED.STLB_MISS_LOADS	Retired load instructions that miss the STLB.	PSDLA
D0H	12H	MEM_INST_RETIRED.STLB_MISS_STORES	Retired store instructions that miss the STLB.	PSDLA
D0H	21H	MEM_INST_RETIRED.LOCK_LOADS	Retired load instructions with locked access.	PSDLA
D0H	41H	MEM_INST_RETIRED.SPLIT_LOADS	Number of load instructions retired with cache-line splits that may impact performance.	PSDLA
D0H	42H	MEM_INST_RETIRED.SPLIT_STORES	Number of store instructions retired with line-split.	PSDLA
D0H	81H	MEM_INST_RETIRED.ALL_LOADS	All retired load instructions.	PSDLA
D0H	82H	MEM_INST_RETIRED.ALL_STORES	All retired store instructions.	PSDLA
D1H	01H	MEM_LOAD_RETIRED.L1_HIT	Retired load instructions with L1 cache hits as data sources.	PSDLA
D1H	02H	MEM_LOAD_RETIRED.L2_HIT	Retired load instructions with L2 cache hits as data sources.	PSDLA
D1H	04H	MEM_LOAD_RETIRED.L3_HIT	Retired load instructions with L3 cache hits as data sources.	PSDLA
D1H	08H	MEM_LOAD_RETIRED.L1_MISS	Retired load instructions missed L1 cache as data sources.	PSDLA
D1H	10H	MEM_LOAD_RETIRED.L2_MISS	Retired load instructions missed L2. Unknown data source excluded.	PSDLA
D1H	20H	MEM_LOAD_RETIRED.L3_MISS	Retired load instructions missed L3. Excludes unknown data source.	PSDLA

Table 19-4. Non-Architectural Performance Events of the Processor Core Supported by Skylake Microarchitecture and Kaby Lake Microarchitecture (Contd.)

Event Num.	Umask Value	Event Mask Mnemonic	Description	Comment
D1H	40H	MEM_LOAD_RETIRED.FB_HIT	Retired load instructions where data sources were load uops missed L1 but hit FB due to preceding miss to the same cache line with data not ready.	PSDLA
D2H	01H	MEM_LOAD_L3_HIT_RETIRED.XSNP_MISS	Retired load instructions where data sources were L3 hit and cross-core snoop missed in on-pkg core cache.	PSDLA
D2H	02H	MEM_LOAD_L3_HIT_RETIRED.XSNP_HIT	Retired load Instructions where data sources were L3 and cross-core snoop hits in on-pkg core cache.	PSDLA
D2H	04H	MEM_LOAD_L3_HIT_RETIRED.XSNP_HITM	Retired load instructions where data sources were HitM responses from shared L3.	PSDLA
D2H	08H	MEM_LOAD_L3_HIT_RETIRED.XSNP_NONE	Retired load instructions where data sources were hits in L3 without snoops required.	PSDLA
E6H	01H	BACLEARS.ANY	Number of front end re-steers due to BPU misprediction.	
F0H	40H	L2_TRANS.L2_WB	L2 writebacks that access L2 cache.	
F1H	07H	L2_LINES_IN.ALL	L2 cache lines filling L2.	

CMSK1: Counter Mask = 1 required; CMSK4: CounterMask = 4 required; CMSK6: CounterMask = 6 required; CMSK8: CounterMask = 8 required; CMSK10: CounterMask = 10 required; CMSK12: CounterMask = 12 required; CMSK16: CounterMask = 16 required; CMSK20: CounterMask = 20 required.

AnyT: AnyThread = 1 required.

INV: Invert = 1 required.

EDG: EDGE = 1 required.

PSDLA: Also supports PEBS and DataLA.

PS: Also supports PEBS.

Table 19-10 lists performance events supporting Intel TSX (see Section 18.3.6.5) and the events are applicable to processors based on Skylake microarchitecture. Where Skylake microarchitecture implements TSX-related event semantics that differ from Table 19-10, they are listed in Table 19-5.

Table 19-5. Intel® TSX Performance Event Addendum in Processors based on Skylake Microarchitecture

Event Num.	Umask Value	Event Mask Mnemonic	Description	Comment
54H	02H	TX_MEM.ABORT_CAPACITY	Number of times a transactional abort was signaled due to a data capacity limitation for transactional reads or writes.	

19.4 PERFORMANCE MONITORING EVENTS FOR INTEL® XEON PHI™ PROCESSOR 3200, 5200, 7200 SERIES

Intel® Xeon Phi™ processors 3200/5200/7200 series are based on the Knights Landing microarchitecture. Non-architectural performance-monitoring events in the processor core are listed in Table 19-6. The events in Table 19-6 apply to processors with CPUID signature of DisplayFamily_DisplayModel encoding with the following value 06_57H.

Table 19-6. Non-Architectural Performance Events of the Processor Core Supported by Knights Landing Microarchitecture

Event Num.	Umask Value	Event Mask Mnemonic	Description	Comment
03H	01H	RECYCLEQ.LD_BLOCK_ST_FORWARD	Counts the number of occurrences a retired load gets blocked because its address partially overlaps with a store.	PSDLA
03H	02H	RECYCLEQ.LD_BLOCK_STD_NOT READY	Counts the number of occurrences a retired load gets blocked because its address overlaps with a store whose data is not ready.	
03H	04H	RECYCLEQ.ST_SPLITS	Counts the number of occurrences a retired store that is a cache line split. Each split should be counted only once.	
03H	08H	RECYCLEQ.LD_SPLITS	Counts the number of occurrences a retired load that is a cache line split. Each split should be counted only once.	PSDLA
03H	10H	RECYCLEQ.LOCK	Counts all the retired locked loads. It does not include stores because we would double count if we count stores.	
03H	20H	RECYCLEQ.STA_FULL	Counts the store micro-ops retired that were pushed in the recycle queue because the store address buffer is full.	
03H	40H	RECYCLEQ.ANY_LD	Counts any retired load that was pushed into the recycle queue for any reason.	
03H	80H	RECYCLEQ.ANY_ST	Counts any retired store that was pushed into the recycle queue for any reason.	
04H	01H	MEM_UOPS_RETIRED.L1_MISS_ LOADS	Counts the number of load micro-ops retired that miss in L1 D cache.	
04H	02H	MEM_UOPS_RETIRED.L2_HIT_L OADS	Counts the number of load micro-ops retired that hit in the L2.	PSDLA
04H	04H	MEM_UOPS_RETIRED.L2_MISS_ LOADS	Counts the number of load micro-ops retired that miss in the L2.	PSDLA
04H	08H	MEM_UOPS_RETIRED.DTLB_MIS S_LOADS	Counts the number of load micro-ops retired that cause a DTLB miss.	PSDLA
04H	10H	MEM_UOPS_RETIRED.UTLB_MIS S_LOADS	Counts the number of load micro-ops retired that caused micro TLB miss.	
04H	20H	MEM_UOPS_RETIRED.HITM	Counts the loads retired that get the data from the other core in the same tile in M state.	
04H	40H	MEM_UOPS_RETIRED.ALL_LOAD S	Counts all the load micro-ops retired.	
04H	80H	MEM_UOPS_RETIRED.ALL_STOR ES	Counts all the store micro-ops retired.	
05H	01H	PAGE_WALKS.D_SIDE_WALKS	Counts the total D-side page walks that are completed or started. The page walks started in the speculative path will also be counted.	EdgeDetect=1
05H	01H	PAGE_WALKS.D_SIDE_CYCLES	Counts the total number of core cycles for all the D-side page walks. The cycles for page walks started in speculative path will also be included.	
05H	02H	PAGE_WALKS.I_SIDE_WALKS	Counts the total I-side page walks that are completed.	EdgeDetect=1

Table 19-6. Non-Architectural Performance Events of the Processor Core Supported by Knights Landing Microarchitecture

Event Num.	Umask Value	Event Mask Mnemonic	Description	Comment
05H	02H	PAGE_WALKS.I_SIDE_CYCLES	Counts the total number of core cycles for all the I-side page walks. The cycles for page walks started in speculative path will also be included.	
05H	03H	PAGE_WALKS.WALKS	Counts the total page walks that are completed (I-side and D-side).	EdgeDetect=1
05H	03H	PAGE_WALKS.CYCLES	Counts the total number of core cycles for all the page walks. The cycles for page walks started in speculative path will also be included.	
2EH	41H	LONGEST_LAT_CACHE.MISS	Counts the number of L2 cache misses. Also called L2_REQUESTS_MISS.	
2EH	4FH	LONGEST_LAT_CACHE.REFERENCE	Counts the total number of L2 cache references. Also called L2_REQUESTS_REFERENCE.	
30H	00H	L2_REQUESTS_REJECT.ALL	Counts the number of MEC requests from the L2Q that reference a cache line (cacheable requests) excluding SW prefetches filling only to L2 cache and L1 evictions (automatically excludes L2HWP, UC, WC) that were rejected - Multiple repeated rejects should be counted multiple times.	
31H	00H	CORE_REJECT_L2Q.ALL	Counts the number of MEC requests that were not accepted into the L2Q because of any L2 queue reject condition. There is no concept of at-ret here. It might include requests due to instructions in the speculative path.	
3CH	00H	CPU_CLK_UNHALTED.THREAD_P	Counts the number of unhalted core clock cycles.	
3CH	01H	CPU_CLK_UNHALTED.REF	Counts the number of unhalted reference clock cycles.	
3EH	04H	L2_PREFETCHER.ALLOC_XQ	Counts the number of L2HWP allocated into XQ GP.	
80H	01H	ICACHE.HIT	Counts all instruction fetches that hit the instruction cache.	
80H	02H	ICACHE.MISSES	Counts all instruction fetches that miss the instruction cache or produce memory requests. An instruction fetch miss is counted only once and not once for every cycle it is outstanding.	
80H	03H	ICACHE.ACCESSES	Counts all instruction fetches, including uncacheable fetches.	
86H	04H	FETCH_STALL.ICACHE_FILL_PENDING_CYCLES	Counts the number of core cycles the fetch stalls because of an icache miss. This is a cumulative count of core cycles the fetch stalled for all icache misses.	
B7H	01H	OFFCORE_RESPONSE_0	See Section 18.4.1.1.2.	Requires MSR_OFFCORE_RESP 0 to specify request type and response.
B7H	02H	OFFCORE_RESPONSE_1	See Section 18.4.1.1.2.	Requires MSR_OFFCORE_RESP 1 to specify request type and response.
C0H	00H	INST_RETIRED.ANY_P	Counts the total number of instructions retired.	PS

Table 19-6. Non-Architectural Performance Events of the Processor Core Supported by Knights Landing Microarchitecture

Event Num.	Umask Value	Event Mask Mnemonic	Description	Comment
C2H	01H	UOPS_RETIRED.MS	Counts the number of micro-ops retired that are from the complex flows issued by the micro-sequencer (MS).	
C2H	10H	UOPS_RETIRED.ALL	Counts the number of micro-ops retired.	
C2H	20H	UOPS_RETIRED.SCALAR_SIMD	Counts the number of scalar SSE, AVX, AVX2, and AVX-512 micro-ops except for loads (memory-to-register mov-type micro ops), division and sqrt.	
C2H	40H	UOPS_RETIRED.PACKED_SIMD	Counts the number of packed SSE, AVX, AVX2, and AVX-512 micro-ops (both floating point and integer) except for loads (memory-to-register mov-type micro-ops), packed byte and word multiplies.	
C3H	01H	MACHINE_CLEARS.SMC	Counts the number of times that the machine clears due to program modifying data within 1K of a recently fetched code page.	
C3H	02H	MACHINE_CLEARS.MEMORY_ORDERING	Counts the number of times the machine clears due to memory ordering hazards.	
C3H	04H	MACHINE_CLEARS.FP_ASSIST	Counts the number of floating operations retired that required microcode assists.	
C3H	08H	MACHINE_CLEARS.ALL	Counts all machine clears.	
C4H	00H	BR_INST_RETIRED.ALL_BRANCHES	Counts the number of branch instructions retired.	PS
C4H	7EH	BR_INST_RETIRED.JCC	Counts the number of JCC branch instructions retired.	PS
C4H	BFH	BR_INST_RETIRED.FAR_BRANCH	Counts the number of far branch instructions retired.	PS
C4H	EBH	BR_INST_RETIRED.NON_RETURN_IND	Counts the number of branch instructions retired that were near indirect CALL or near indirect JMP.	PS
C4H	F7H	BR_INST_RETIRED.RETURN	Counts the number of near RET branch instructions retired.	PS
C4H	F9H	BR_INST_RETIRED.CALL	Counts the number of near CALL branch instructions retired.	PS
C4H	FBH	BR_INST_RETIRED.IND_CALL	Counts the number of near indirect CALL branch instructions retired.	PS
C4H	FDH	BR_INST_RETIRED.REL_CALL	Counts the number of near relative CALL branch instructions retired.	PS
C4H	FEH	BR_INST_RETIRED.TAKEN_JCC	Counts the number of branch instructions retired that were taken conditional jumps.	PS
C5H	00H	BR_MISP_RETIRED.ALL_BRANCHES	Counts the number of mispredicted branch instructions retired.	PS
C5H	7EH	BR_MISP_RETIRED.JCC	Counts the number of mispredicted JCC branch instructions retired.	PS
C5H	BFH	BR_MISP_RETIRED.FAR_BRANCH	Counts the number of mispredicted far branch instructions retired.	PS
C5H	EBH	BR_MISP_RETIRED.NON_RETURN_IND	Counts the number of mispredicted branch instructions retired that were near indirect CALL or near indirect JMP.	PS
C5H	F7H	BR_MISP_RETIRED.RETURN	Counts the number of mispredicted near RET branch instructions retired.	PS

Table 19-6. Non-Architectural Performance Events of the Processor Core Supported by Knights Landing Microarchitecture

Event Num.	Umask Value	Event Mask Mnemonic	Description	Comment
C5H	F9H	BR_MISP_RETIRED.CALL	Counts the number of mispredicted near CALL branch instructions retired.	PS
C5H	FBH	BR_MISP_RETIRED.IND_CALL	Counts the number of mispredicted near indirect CALL branch instructions retired.	PS
C5H	FDH	BR_MISP_RETIRED.REL_CALL	Counts the number of mispredicted near relative CALL branch instructions retired.	PS
C5H	FEH	BR_MISP_RETIRED.TAKEN_JCC	Counts the number of mispredicted branch instructions retired that were taken conditional jumps.	PS
CAH	01H	NO_ALLOC_CYCLES.ROB_FULL	Counts the number of core cycles when no micro-ops are allocated and the ROB is full.	
CAH	04H	NO_ALLOC_CYCLES.MISPREDICTS	Counts the number of core cycles when no micro-ops are allocated and the alloc pipe is stalled waiting for a mispredicted branch to retire.	
CAH	20H	NO_ALLOC_CYCLES.RAT_STALL	Counts the number of core cycles when no micro-ops are allocated and a RATstall (caused by reservation station full) is asserted.	
CAH	90H	NO_ALLOC_CYCLES.NOT_DELIVERED	Counts the number of core cycles when no micro-ops are allocated, the IQ is empty, and no other condition is blocking allocation.	
CAH	7FH	NO_ALLOC_CYCLES.ALL	Counts the total number of core cycles when no micro-ops are allocated for any reason.	
CBH	01H	RS_FULL_STALL.MEC	Counts the number of core cycles when allocation pipeline is stalled and is waiting for a free MEC reservation station entry.	
CBH	1FH	RS_FULL_STALL.ALL	Counts the total number of core cycles the allocation pipeline is stalled when any one of the reservation stations is full.	
CDH	01H	CYCLES_DIV_BUSY.ALL	Cycles the number of core cycles when divider is busy. Does not imply a stall waiting for the divider.	
E6H	01H	BACLEARS.ALL	Counts the number of times the front end resteers for any branch as a result of another branch handling mechanism in the front end.	
E6H	08H	BACLEARS.RETURN	Counts the number of times the front end resteers for RET branches as a result of another branch handling mechanism in the front end.	
E6H	10H	BACLEARS.COND	Counts the number of times the front end resteers for conditional branches as a result of another branch handling mechanism in the front end.	
E7H	01H	MS_DECODED.MS_ENTRY	Counts the number of times the MSROM starts a flow of uops.	

PS: Also supports PEBS.

PSDLA: Also supports PEBS and DataLA.

19.5 PERFORMANCE MONITORING EVENTS FOR THE INTEL® CORE™ M AND 5TH GENERATION INTEL® CORE™ PROCESSORS

The Intel® Core™ M processors, the 5th generation Intel® Core™ processors and the Intel Xeon processor E3 1200 v4 product family are based on the Broadwell microarchitecture. They support the architectural performance-monitoring events listed in Table 19-1. Non-architectural performance-monitoring events in the processor core are listed in Table 19-7. The events in Table 19-7 apply to processors with CPUID signature of DisplayFamily_DisplayModel encoding with the following values: 06_3DH and 06_47H. Table 19-10 lists performance events supporting Intel TSX (see Section 18.3.6.5) and the events are available on processors based on Broadwell microarchitecture. Fixed counters in the core PMU support the architecture events defined in Table 19-2.

Non-architectural performance monitoring events that are located in the uncore sub-system are implementation specific between different platforms using processors based on Broadwell microarchitecture and with different DisplayFamily_DisplayModel signatures. Processors with CPUID signature of DisplayFamily_DisplayModel 06_3DH and 06_47H support uncore performance events listed in Table 19-11.

Table 19-7. Non-Architectural Performance Events of the Processor Core Supported by Broadwell Microarchitecture

Event Num.	Umask Value	Event Mask Mnemonic	Description	Comment
03H	02H	LD_BLOCKS.STORE_FORWARD	Loads blocked by overlapping with store buffer that cannot be forwarded.	
03H	08H	LD_BLOCKS.NO_SR	The number of times that split load operations are temporarily blocked because all resources for handling the split accesses are in use.	
05H	01H	MISALIGN_MEM_REF.LOADS	Speculative cache-line split load uops dispatched to L1D.	
05H	02H	MISALIGN_MEM_REF.STORES	Speculative cache-line split store-address uops dispatched to L1D.	
07H	01H	LD_BLOCKS_PARTIAL.ADDRESS_ALIAS	False dependencies in MOB due to partial compare on address.	
08H	01H	DTLB_LOAD_MISSES.MISS_CAUSES_A_WALK	Load misses in all TLB levels that cause a page walk of any page size.	
08H	02H	DTLB_LOAD_MISSES.WALK_COMPLETED_4K	Completed page walks due to demand load misses that caused 4K page walks in any TLB levels.	
08H	10H	DTLB_LOAD_MISSES.WALK_DURATION	Cycle PMH is busy with a walk.	
08H	20H	DTLB_LOAD_MISSES.STLB_HIT_4K	Load misses that missed DTLB but hit STLB (4K).	
0DH	03H	INT_MISC.RECOVERY_CYCLES	Cycles waiting to recover after Machine Clears except JEClear. Set Cmask= 1.	Set Edge to count occurrences.
0EH	01H	UOPS_ISSUED.ANY	Increments each cycle the # of uops issued by the RAT to RS. Set Cmask = 1, Inv = 1, Any= 1to count stalled cycles of this core.	Set Cmask = 1, Inv = 1to count stalled cycles.
0EH	10H	UOPS_ISSUED.FLAGS_MERGE	Number of flags-merge uops allocated. Such uops add delay.	
0EH	20H	UOPS_ISSUED.SLOW_LEA	Number of slow LEA or similar uops allocated. Such uop has 3 sources (for example, 2 sources + immediate) regardless of whether it is a result of LEA instruction or not.	
0EH	40H	UOPS_ISSUED.SINGLE_MUL	Number of multiply packed/scalar single precision uops allocated.	

Table 19-7. Non-Architectural Performance Events of the Processor Core Supported by Broadwell Microarchitecture (Contd.)

Event Num.	Umask Value	Event Mask Mnemonic	Description	Comment
14H	01H	ARITH.FPU_DIV_ACTIVE	Cycles when divider is busy executing divide operations.	
24H	21H	L2_RQSTS.DEMAND_DATA_RD_MISS	Demand data read requests that missed L2, no rejects.	
24H	41H	L2_RQSTS.DEMAND_DATA_RD_HIT	Demand data read requests that hit L2 cache.	
24H	50H	L2_RQSTS.L2_PF_HIT	Counts all L2 HW prefetcher requests that hit L2.	
24H	30H	L2_RQSTS.L2_PF_MISS	Counts all L2 HW prefetcher requests that missed L2.	
24H	E1H	L2_RQSTS.ALL_DEMAND_DATA_RD	Counts any demand and L1 HW prefetch data load requests to L2.	
24H	E2H	L2_RQSTS.ALL_RFO	Counts all L2 store RFO requests.	
24H	E4H	L2_RQSTS.ALL_CODE_RD	Counts all L2 code requests.	
24H	F8H	L2_RQSTS.ALL_PF	Counts all L2 HW prefetcher requests.	
27H	50H	L2_DEMAND_RQSTS.WB_HIT	Not rejected writebacks that hit L2 cache.	
2EH	4FH	LONGEST_LAT_CACHE.REFERENCE	This event counts requests originating from the core that reference a cache line in the last level cache.	See Table 19-1.
2EH	41H	LONGEST_LAT_CACHE.MISS	This event counts each cache miss condition for references to the last level cache.	See Table 19-1.
3CH	00H	CPU_CLK_UNHALTED.THREAD_P	Counts the number of thread cycles while the thread is not in a halt state. The thread enters the halt state when it is running the HLT instruction. The core frequency may change from time to time due to power or thermal throttling.	See Table 19-1.
3CH	01H	CPU_CLK_THREAD_UNHALTED.REF_XCLK	Increments at the frequency of XCLK (100 MHz) when not halted.	See Table 19-1.
48H	01H	L1D_PEND_MISS.PENDING	Increments the number of outstanding L1D misses every cycle. Set Cmask = 1 and Edge =1 to count occurrences.	Counter 2 only. Set Cmask = 1 to count cycles.
49H	01H	DTLB_STORE_MISSES.MISS_CAUSES_A_WALK	Miss in all TLB levels causes a page walk of any page size (4K/2M/4M/1G).	
49H	02H	DTLB_STORE_MISSES.WALK_COMPLETED_4K	Completed page walks due to store misses in one or more TLB levels of 4K page structure.	
49H	10H	DTLB_STORE_MISSES.WALK_DURATION	Cycles PMH is busy with this walk.	
49H	20H	DTLB_STORE_MISSES.STLB_HIT_4K	Store misses that missed DTLB but hit STLB (4K).	
4CH	02H	LOAD_HIT_PRE.HW_PF	Non-SW-prefetch load dispatches that hit fill buffer allocated for H/W prefetch.	
4FH	10H	EPT.WALK_CYCLES	Cycles of Extended Page Table walks.	
51H	01H	L1D.REPLACEMENT	Counts the number of lines brought into the L1 data cache.	
58H	04H	MOVE_ELIMINATION.INT_NOT_ELIMINATED	Number of integer move elimination candidate uops that were not eliminated.	

Table 19-7. Non-Architectural Performance Events of the Processor Core Supported by Broadwell Microarchitecture (Contd.)

Event Num.	Umask Value	Event Mask Mnemonic	Description	Comment
58H	08H	MOVE_ELIMINATION.SIMD_NOT_ELIMINATED	Number of SIMD move elimination candidate uops that were not eliminated.	
58H	01H	MOVE_ELIMINATION.INT_ELIMINATED	Number of integer move elimination candidate uops that were eliminated.	
58H	02H	MOVE_ELIMINATION.SIMD_ELIMINATED	Number of SIMD move elimination candidate uops that were eliminated.	
5CH	01H	CPL_CYCLES.RING0	Unhalted core cycles when the thread is in ring 0.	Use Edge to count transition.
5CH	02H	CPL_CYCLES.RING123	Unhalted core cycles when the thread is not in ring 0.	
5EH	01H	RS_EVENTS.EMPTY_CYCLES	Cycles the RS is empty for the thread.	
60H	01H	OFFCORE_REQUESTS_OUTSTANDING.DEMAND_DATA_RD	Offcore outstanding demand data read transactions in SQ to uncore. Set Cmask=1 to count cycles.	Use only when HTT is off.
60H	02H	OFFCORE_REQUESTS_OUTSTANDING.DEMAND_CODE_RD	Offcore outstanding demand code read transactions in SQ to uncore. Set Cmask=1 to count cycles.	Use only when HTT is off.
60H	04H	OFFCORE_REQUESTS_OUTSTANDING.DEMAND_RFO	Offcore outstanding RFO store transactions in SQ to uncore. Set Cmask=1 to count cycles.	Use only when HTT is off.
60H	08H	OFFCORE_REQUESTS_OUTSTANDING.ALL_DATA_RD	Offcore outstanding cacheable data read transactions in SQ to uncore. Set Cmask=1 to count cycles.	Use only when HTT is off.
63H	01H	LOCK_CYCLES.SPLIT_LOCK_UC_LOCK_DURATION	Cycles in which the L1D and L2 are locked, due to a UC lock or split lock.	
63H	02H	LOCK_CYCLES.CACHE_LOCK_DURATION	Cycles in which the L1D is locked.	
79H	02H	IDQ.EMPTY	Counts cycles the IDQ is empty.	
79H	04H	IDQ.MITE_UOPS	Increment each cycle # of uops delivered to IDQ from MITE path. Set Cmask = 1 to count cycles.	Can combine Umask 04H and 20H.
79H	08H	IDQ.DSB_UOPS	Increment each cycle # of uops delivered to IDQ from DSB path. Set Cmask = 1 to count cycles.	Can combine Umask 08H and 10H.
79H	10H	IDQ.MS_DSB_UOPS	Increment each cycle # of uops delivered to IDQ when MS_busy by DSB. Set Cmask = 1 to count cycles. Add Edge=1 to count # of delivery.	Can combine Umask 04H, 08H.
79H	20H	IDQ.MS_MITE_UOPS	Increment each cycle # of uops delivered to IDQ when MS_busy by MITE. Set Cmask = 1 to count cycles.	Can combine Umask 04H, 08H.
79H	30H	IDQ.MS_UOPS	Increment each cycle # of uops delivered to IDQ from MS by either DSB or MITE. Set Cmask = 1 to count cycles.	Can combine Umask 04H, 08H.
79H	18H	IDQ.ALL_DSB_CYCLES_ANY_UOPS	Counts cycles DSB is delivered at least one uops. Set Cmask = 1.	
79H	18H	IDQ.ALL_DSB_CYCLES_4_UOPS	Counts cycles DSB is delivered four uops. Set Cmask = 4.	
79H	24H	IDQ.ALL_MITE_CYCLES_ANY_UOPS	Counts cycles MITE is delivered at least one uop. Set Cmask = 1.	
79H	24H	IDQ.ALL_MITE_CYCLES_4_UOPS	Counts cycles MITE is delivered four uops. Set Cmask = 4.	
79H	3CH	IDQ.MITE_ALL_UOPS	Number of uops delivered to IDQ from any path.	

Table 19-7. Non-Architectural Performance Events of the Processor Core Supported by Broadwell Microarchitecture (Contd.)

Event Num.	Umask Value	Event Mask Mnemonic	Description	Comment
80H	02H	ICACHE.MISSES	Number of Instruction Cache, Streaming Buffer and Victim Cache Misses. Includes UC accesses.	
85H	01H	ITLB_MISSES.MISS_CAUSES_A_WALK	Misses in ITLB that cause a page walk of any page size.	
85H	02H	ITLB_MISSES.WALK_COMPLETED_4K	Completed page walks due to misses in ITLB 4K page entries.	
85H	10H	ITLB_MISSES.WALK_DURATION	Cycle PMH is busy with a walk.	
85H	20H	ITLB_MISSES.STLB_HIT_4K	ITLB misses that hit STLB (4K).	
87H	01H	ILD_STALL.LCP	Stalls caused by changing prefix length of the instruction.	
88H	01H	BR_INST_EXEC.COND	Qualify conditional near branch instructions executed, but not necessarily retired.	Must combine with umask 40H, 80H.
88H	02H	BR_INST_EXEC.DIRECT_JMP	Qualify all unconditional near branch instructions excluding calls and indirect branches.	Must combine with umask 80H.
88H	04H	BR_INST_EXEC.INDIRECT_JMP_NON_CALL_RET	Qualify executed indirect near branch instructions that are not calls or returns.	Must combine with umask 80H.
88H	08H	BR_INST_EXEC.RETURN_NEAR	Qualify indirect near branches that have a return mnemonic.	Must combine with umask 80H.
88H	10H	BR_INST_EXEC.DIRECT_NEAR_CALL	Qualify unconditional near call branch instructions, excluding non-call branch, executed.	Must combine with umask 80H.
88H	20H	BR_INST_EXEC.INDIRECT_NEAR_CALL	Qualify indirect near calls, including both register and memory indirect, executed.	Must combine with umask 80H.
88H	40H	BR_INST_EXEC.NONTAKEN	Qualify non-taken near branches executed.	Applicable to umask 01H only.
88H	80H	BR_INST_EXEC.TAKEN	Qualify taken near branches executed. Must combine with 01H,02H, 04H, 08H, 10H, 20H.	
88H	FFH	BR_INST_EXEC.ALL_BRANCHES	Counts all near executed branches (not necessarily retired).	
89H	01H	BR_MISP_EXEC.COND	Qualify conditional near branch instructions mispredicted.	Must combine with umask 40H, 80H.
89H	04H	BR_MISP_EXEC.INDIRECT_JMP_NON_CALL_RET	Qualify mispredicted indirect near branch instructions that are not calls or returns.	Must combine with umask 80H.
89H	08H	BR_MISP_EXEC.RETURN_NEAR	Qualify mispredicted indirect near branches that have a return mnemonic.	Must combine with umask 80H.
89H	10H	BR_MISP_EXEC.DIRECT_NEAR_CALL	Qualify mispredicted unconditional near call branch instructions, excluding non-call branch, executed.	Must combine with umask 80H.
89H	20H	BR_MISP_EXEC.INDIRECT_NEAR_CALL	Qualify mispredicted indirect near calls, including both register and memory indirect, executed.	Must combine with umask 80H.
89H	40H	BR_MISP_EXEC.NONTAKEN	Qualify mispredicted non-taken near branches executed.	Applicable to umask 01H only.
89H	80H	BR_MISP_EXEC.TAKEN	Qualify mispredicted taken near branches executed. Must combine with 01H,02H, 04H, 08H, 10H, 20H.	
89H	FFH	BR_MISP_EXEC.ALL_BRANCHES	Counts all near executed branches (not necessarily retired).	

Table 19-7. Non-Architectural Performance Events of the Processor Core Supported by Broadwell Microarchitecture (Contd.)

Event Num.	Umask Value	Event Mask Mnemonic	Description	Comment
9CH	01H	IDQ_UOPS_NOT_DELIVERED.CORE	Count issue pipeline slots where no uop was delivered from the front end to the back end when there is no back end stall.	Use Cmask to qualify uop b/w.
A1H	01H	UOPS_DISPATCHED_PORT.PORT_0	Counts the number of cycles in which a uop is dispatched to port 0.	Set AnyThread to count per core.
A1H	02H	UOPS_DISPATCHED_PORT.PORT_1	Counts the number of cycles in which a uop is dispatched to port 1.	Set AnyThread to count per core.
A1H	04H	UOPS_DISPATCHED_PORT.PORT_2	Counts the number of cycles in which a uop is dispatched to port 2.	Set AnyThread to count per core.
A1H	08H	UOPS_DISPATCHED_PORT.PORT_3	Counts the number of cycles in which a uop is dispatched to port 3.	Set AnyThread to count per core.
A1H	10H	UOPS_DISPATCHED_PORT.PORT_4	Counts the number of cycles in which a uop is dispatched to port 4.	Set AnyThread to count per core.
A1H	20H	UOPS_DISPATCHED_PORT.PORT_5	Counts the number of cycles in which a uop is dispatched to port 5.	Set AnyThread to count per core.
A1H	40H	UOPS_DISPATCHED_PORT.PORT_6	Counts the number of cycles in which a uop is dispatched to port 6.	Set AnyThread to count per core.
A1H	80H	UOPS_DISPATCHED_PORT.PORT_7	Counts the number of cycles in which a uop is dispatched to port 7.	Set AnyThread to count per core.
A2H	01H	RESOURCE_STALLS.ANY	Cycles Allocation is stalled due to resource related reason.	
A2H	04H	RESOURCE_STALLS.RS	Cycles stalled due to no eligible RS entry available.	
A2H	08H	RESOURCE_STALLS.SB	Cycles stalled due to no store buffers available (not including draining form sync).	
A2H	10H	RESOURCE_STALLS.ROB	Cycles stalled due to re-order buffer full.	
A8H	01H	LSD.UOPS	Number of uops delivered by the LSD.	
ABH	02H	DSB2MITE_SWITCHES.PENALTY_CYCLES	Cycles of delay due to Decode Stream Buffer to MITE switches.	
AEH	01H	ITLB.ITLB_FLUSH	Counts the number of ITLB flushes; includes 4k/2M/4M pages.	
B0H	01H	OFFCORE_REQUESTS.DEMAND_DATA_RD	Demand data read requests sent to uncore.	Use only when HTT is off.
B0H	02H	OFFCORE_REQUESTS.DEMAND_CODE_RD	Demand code read requests sent to uncore.	Use only when HTT is off.
B0H	04H	OFFCORE_REQUESTS.DEMAND_RFO	Demand RFO read requests sent to uncore, including regular RFOs, locks, ItoM.	Use only when HTT is off.
B0H	08H	OFFCORE_REQUESTS.ALL_DATA_RD	Data read requests sent to uncore (demand and prefetch).	Use only when HTT is off.
B1H	01H	UOPS_EXECUTED.THREAD	Counts total number of uops to be executed per-logical-processor each cycle.	Use Cmask to count stall cycles.
B1H	02H	UOPS_EXECUTED.CORE	Counts total number of uops to be executed per-core each cycle.	Do not need to set ANY.
B7H	01H	OFF_CORE_RESPONSE_0	See Section 18.3.4.5, “Off-core Response Performance Monitoring”.	Requires MSR 01A6H.

Table 19-7. Non-Architectural Performance Events of the Processor Core Supported by Broadwell Microarchitecture (Contd.)

Event Num.	Umask Value	Event Mask Mnemonic	Description	Comment
BBH	01H	OFF_CORE_RESPONSE_1	See Section 18.3.4.5, "Off-core Response Performance Monitoring".	Requires MSR 01A7H.
BCH	11H	PAGE_WALKER_LOADS.DTLB_L1	Number of DTLB page walker loads that hit in the L1+FB.	
BCH	21H	PAGE_WALKER_LOADS.ITLB_L1	Number of ITLB page walker loads that hit in the L1+FB.	
BCH	12H	PAGE_WALKER_LOADS.DTLB_L2	Number of DTLB page walker loads that hit in the L2.	
BCH	22H	PAGE_WALKER_LOADS.ITLB_L2	Number of ITLB page walker loads that hit in the L2.	
BCH	14H	PAGE_WALKER_LOADS.DTLB_L3	Number of DTLB page walker loads that hit in the L3.	
BCH	24H	PAGE_WALKER_LOADS.ITLB_L3	Number of ITLB page walker loads that hit in the L3.	
BCH	18H	PAGE_WALKER_LOADS.DTLB_MEMORY	Number of DTLB page walker loads from memory.	
C0H	00H	INST_RETIRED.ANY_P	Number of instructions at retirement.	See Table 19-1.
C0H	01H	INST_RETIRED.PREC_DIST	Precise instruction retired event with HW to reduce effect of PEBS shadow in IP distribution.	PMC1 only.
C0H	02H	INST_RETIRED.X87	FP operations retired. X87 FP operations that have no exceptions.	
C1H	08H	OTHER_ASSISTS.AVX_TO_SSE	Number of transitions from AVX-256 to legacy SSE when penalty applicable.	
C1H	10H	OTHER_ASSISTS.SSE_TO_AVX	Number of transitions from SSE to AVX-256 when penalty applicable.	
C1H	40H	OTHER_ASSISTS.ANY_WB_ASSIST	Number of microcode assists invoked by HW upon uop writeback.	
C2H	01H	UOPS_RETIRED.ALL	Counts the number of micro-ops retired. Use cmask=1 and invert to count active cycles or stalled cycles.	Supports PEBS and DataLA, use Any=1 for core granular.
C2H	02H	UOPS_RETIRED.RETIRE_SLOTS	Counts the number of retirement slots used each cycle.	Supports PEBS.
C3H	01H	MACHINE_CLEARS.CYCLES	Counts cycles while a machine clears stalled forward progress of a logical processor or a processor core.	
C3H	02H	MACHINE_CLEARS.MEMORY_ORDERING	Counts the number of machine clears due to memory order conflicts.	
C3H	04H	MACHINE_CLEARS.SMC	Number of self-modifying-code machine clears detected.	
C3H	20H	MACHINE_CLEARS.MASKMOV	Counts the number of executed AVX masked load operations that refer to an illegal address range with the mask bits set to 0.	
C4H	00H	BR_INST_RETIRED.ALL_BRANCHES	Branch instructions at retirement.	See Table 19-1.
C4H	01H	BR_INST_RETIRED.CONDITIONAL	Counts the number of conditional branch instructions retired.	Supports PEBS.
C4H	02H	BR_INST_RETIRED.NEAR_CALL	Direct and indirect near call instructions retired.	Supports PEBS.
C4H	04H	BR_INST_RETIRED.ALL_BRANCHES	Counts the number of branch instructions retired.	Supports PEBS.

Table 19-7. Non-Architectural Performance Events of the Processor Core Supported by Broadwell Microarchitecture (Contd.)

Event Num.	Umask Value	Event Mask Mnemonic	Description	Comment
C4H	08H	BR_INST_RETIRED.NEAR_RETURN	Counts the number of near return instructions retired.	Supports PEBS.
C4H	10H	BR_INST_RETIRED.NOT_TAKEN	Counts the number of not taken branch instructions retired.	
C4H	20H	BR_INST_RETIRED.NEAR_TAKEN	Number of near taken branches retired.	Supports PEBS.
C4H	40H	BR_INST_RETIRED.FAR_BRANCH	Number of far branches retired.	
C5H	00H	BR_MISP_RETIRED.ALL_BRANCHES	Mispredicted branch instructions at retirement.	See Table 19-1.
C5H	01H	BR_MISP_RETIRED.CONDITIONAL	Mispredicted conditional branch instructions retired.	Supports PEBS.
C5H	04H	BR_MISP_RETIRED.ALL_BRANCHES	Mispredicted macro branch instructions retired.	Supports PEBS.
CAH	02H	FP_ASSIST.X87_OUTPUT	Number of X87 FP assists due to output values.	
CAH	04H	FP_ASSIST.X87_INPUT	Number of X87 FP assists due to input values.	
CAH	08H	FP_ASSIST.SIMD_OUTPUT	Number of SIMD FP assists due to output values.	
CAH	10H	FP_ASSIST.SIMD_INPUT	Number of SIMD FP assists due to input values.	
CAH	1EH	FP_ASSIST.ANY	Cycles with any input/output SSE* or FP assists.	
CCH	20H	ROB_MISC_EVENTS.LBR_INSERTS	Count cases of saving new LBR records by hardware.	
CDH	01H	MEM_TRANS_RETIRED.LOAD_LATENCY	Randomly sampled loads whose latency is above a user defined threshold. A small fraction of the overall loads are sampled due to randomization.	Specify threshold in MSR 3F6H.
D0H	11H	MEM_UOPS_RETIRED.STLB_MISS_LOADS	Retired load uops that miss the STLB.	Supports PEBS and DataLA.
D0H	12H	MEM_UOPS_RETIRED.STLB_MISS_STORES	Retired store uops that miss the STLB.	Supports PEBS and DataLA.
D0H	21H	MEM_UOPS_RETIRED.LOCK_LOADS	Retired load uops with locked access.	Supports PEBS and DataLA.
D0H	41H	MEM_UOPS_RETIRED.SPLIT_LOADS	Retired load uops that split across a cacheline boundary.	Supports PEBS and DataLA.
D0H	42H	MEM_UOPS_RETIRED.SPLIT_STORES	Retired store uops that split across a cacheline boundary.	Supports PEBS and DataLA.
D0H	81H	MEM_UOPS_RETIRED.ALL_LOADS	All retired load uops.	Supports PEBS and DataLA.
D0H	82H	MEM_UOPS_RETIRED.ALL_STORES	All retired store uops.	Supports PEBS and DataLA.
D1H	01H	MEM_LOAD_UOPS_RETIRED.L1_HIT	Retired load uops with L1 cache hits as data sources.	Supports PEBS and DataLA.
D1H	02H	MEM_LOAD_UOPS_RETIRED.L2_HIT	Retired load uops with L2 cache hits as data sources.	Supports PEBS and DataLA.
D1H	04H	MEM_LOAD_UOPS_RETIRED.L3_HIT	Retired load uops with L3 cache hits as data sources.	Supports PEBS and DataLA.

Table 19-7. Non-Architectural Performance Events of the Processor Core Supported by Broadwell Microarchitecture (Contd.)

Event Num.	Umask Value	Event Mask Mnemonic	Description	Comment
D1H	08H	MEM_LOAD_UOPS_RETIRED.L1_MISS	Retired load uops missed L1 cache as data sources.	Supports PEBS and DataLA.
D1H	10H	MEM_LOAD_UOPS_RETIRED.L2_MISS	Retired load uops missed L2. Unknown data source excluded.	Supports PEBS and DataLA.
D1H	20H	MEM_LOAD_UOPS_RETIRED.L3_MISS	Retired load uops missed L3. Excludes unknown data source.	Supports PEBS and DataLA.
D1H	40H	MEM_LOAD_UOPS_RETIRED.HIT_LFB	Retired load uops where data sources were load uops missed L1 but hit FB due to preceding miss to the same cache line with data not ready.	Supports PEBS and DataLA.
D2H	01H	MEM_LOAD_UOPS_L3_HIT_RETIRED.XSNP_MISS	Retired load uops where data sources were L3 hit and cross-core snoop missed in on-pkg core cache.	Supports PEBS and DataLA.
D2H	02H	MEM_LOAD_UOPS_L3_HIT_RETIRED.XSNP_HIT	Retired load uops where data sources were L3 and cross-core snoop hits in on-pkg core cache.	Supports PEBS and DataLA.
D2H	04H	MEM_LOAD_UOPS_L3_HIT_RETIRED.XSNP_HITM	Retired load uops where data sources were HitM responses from shared L3.	Supports PEBS and DataLA.
D2H	08H	MEM_LOAD_UOPS_L3_HIT_RETIRED.XSNP_NONE	Retired load uops where data sources were hits in L3 without snoops required.	Supports PEBS and DataLA.
D3H	01H	MEM_LOAD_UOPS_L3_MISS_RETIRED.LOCAL_DRAM	Retired load uops where data sources missed L3 but serviced from local dram.	Supports PEBS and DataLA.
F0H	01H	L2_TRANS.DEMAND_DATA_RD	Demand data read requests that access L2 cache.	
F0H	02H	L2_TRANS.RFO	RFO requests that access L2 cache.	
F0H	04H	L2_TRANS.CODE_RD	L2 cache accesses when fetching instructions.	
F0H	08H	L2_TRANS.ALL_PF	Any MLC or L3 HW prefetch accessing L2, including rejects.	
F0H	10H	L2_TRANS.L1D_WB	L1D writebacks that access L2 cache.	
F0H	20H	L2_TRANS.L2_FILL	L2 fill requests that access L2 cache.	
F0H	40H	L2_TRANS.L2_WB	L2 writebacks that access L2 cache.	
F0H	80H	L2_TRANS.ALL_REQUESTS	Transactions accessing L2 pipe.	
F1H	01H	L2_LINES_IN.I	L2 cache lines in I state filling L2.	Counting does not cover rejects.
F1H	02H	L2_LINES_IN.S	L2 cache lines in S state filling L2.	Counting does not cover rejects.
F1H	04H	L2_LINES_IN.E	L2 cache lines in E state filling L2.	Counting does not cover rejects.
F1H	07H	L2_LINES_IN.ALL	L2 cache lines filling L2.	Counting does not cover rejects.
F2H	05H	L2_LINES_OUT.DEMAND_CLEAN	Clean L2 cache lines evicted by demand.	

Table 19-10 lists performance events supporting Intel TSX (see Section 18.3.6.5) and the events are applicable to processors based on Broadwell microarchitecture. Where Broadwell microarchitecture implements TSX-related event semantics that differ from Table 19-10, they are listed in Table 19-8.

Table 19-8. Intel® TSX Performance Event Addendum in Processors Based on Broadwell Microarchitecture

Event Num.	Umask Value	Event Mask Mnemonic	Description	Comment
54H	02H	TX_MEM.ABORT_CAPACITY	Number of times a transactional abort was signaled due to a data capacity limitation for transactional reads or writes.	

19.6 PERFORMANCE MONITORING EVENTS FOR THE 4TH GENERATION INTEL® CORE™ PROCESSORS

4th generation Intel® Core™ processors and Intel Xeon processor E3-1200 v3 product family are based on the Haswell microarchitecture. They support the architectural performance-monitoring events listed in Table 19-1. Non-architectural performance-monitoring events in the processor core are listed in Table 19-9. The events in Table 19-9 apply to processors with CPUID signature of DisplayFamily_DisplayModel encoding with the following values: 06_3CH, 06_45H and 06_46H. Table 19-10 lists performance events focused on supporting Intel TSX (see Section 18.3.6.5). Fixed counters in the core PMU support the architecture events defined in Table 19-2.

Additional information on event specifics (e.g., derivative events using specific IA32_PERFEVTSELx modifiers, limitations, special notes and recommendations) can be found at http://software.intel.com/en-us/forums/software-tuning-performance-optimization-platform-monitoring.

Table 19-9. Non-Architectural Performance Events in the Processor Core of 4th Generation Intel® Core™ Processors

Event Num.	Umask Value	Event Mask Mnemonic	Description	Comment
03H	02H	LD_BLOCKS.STORE_FORWARD	Loads blocked by overlapping with store buffer that cannot be forwarded.	
03H	08H	LD_BLOCKS.NO_SR	The number of times that split load operations are temporarily blocked because all resources for handling the split accesses are in use.	
05H	01H	MISALIGN_MEM_REF.LOADS	Speculative cache-line split load uops dispatched to L1D.	
05H	02H	MISALIGN_MEM_REF.STORES	Speculative cache-line split store-address uops dispatched to L1D.	
07H	01H	LD_BLOCKS_PARTIAL.ADDRESS_ALIAS	False dependencies in MOB due to partial compare on address.	
08H	01H	DTLB_LOAD_MISSES.MISS_CAUSES_A_WALK	Misses in all TLB levels that cause a page walk of any page size.	
08H	02H	DTLB_LOAD_MISSES.WALK_COMPLETED_4K	Completed page walks due to demand load misses that caused 4K page walks in any TLB levels.	
08H	04H	DTLB_LOAD_MISSES.WALK_COMPLETED_2M_4M	Completed page walks due to demand load misses that caused 2M/4M page walks in any TLB levels.	
08H	0EH	DTLB_LOAD_MISSES.WALK_COMPLETED	Completed page walks in any TLB of any page size due to demand load misses.	
08H	10H	DTLB_LOAD_MISSES.WALK_DURATION	Cycle PMH is busy with a walk.	
08H	20H	DTLB_LOAD_MISSES.STLB_HIT_4K	Load misses that missed DTLB but hit STLB (4K).	
08H	40H	DTLB_LOAD_MISSES.STLB_HIT_2M	Load misses that missed DTLB but hit STLB (2M).	

Table 19-9. Non-Architectural Performance Events in the Processor Core of 4th Generation Intel® Core™ Processors (Contd.)

Event Num.	Umask Value	Event Mask Mnemonic	Description	Comment
08H	60H	DTLB_LOAD_MISSES.STLB_HIT	Number of cache load STLB hits. No page walk.	
08H	80H	DTLB_LOAD_MISSES.PDE_CACHE_MISS	DTLB demand load misses with low part of linear-to-physical address translation missed.	
0DH	03H	INT_MISC.RECOVERY_CYCLES	Cycles waiting to recover after Machine Clears except JEClear. Set Cmask= 1.	Set Edge to count occurrences.
0EH	01H	UOPS_ISSUED.ANY	Increments each cycle the # of uops issued by the RAT to RS. Set Cmask = 1, Inv = 1, Any= 1 to count stalled cycles of this core.	Set Cmask = 1, Inv = 1to count stalled cycles.
0EH	10H	UOPS_ISSUED.FLAGS_MERGE	Number of flags-merge uops allocated. Such uops add delay.	
0EH	20H	UOPS_ISSUED.SLOW_LEA	Number of slow LEA or similar uops allocated. Such uop has 3 sources (for example, 2 sources + immediate) regardless of whether it is a result of LEA instruction or not.	
0EH	40H	UOPS_ISSUED.SINGLE_MUL	Number of multiply packed/scalar single precision uops allocated.	
24H	21H	L2_RQSTS.DEMAND_DATA_RD_MISS	Demand data read requests that missed L2, no rejects.	
24H	41H	L2_RQSTS.DEMAND_DATA_RD_HIT	Demand data read requests that hit L2 cache.	
24H	E1H	L2_RQSTS.ALL_DEMAND_DATA_RD	Counts any demand and L1 HW prefetch data load requests to L2.	
24H	42H	L2_RQSTS.RFO_HIT	Counts the number of store RFO requests that hit the L2 cache.	
24H	22H	L2_RQSTS.RFO_MISS	Counts the number of store RFO requests that miss the L2 cache.	
24H	E2H	L2_RQSTS.ALL_RFO	Counts all L2 store RFO requests.	
24H	44H	L2_RQSTS.CODE_RD_HIT	Number of instruction fetches that hit the L2 cache.	
24H	24H	L2_RQSTS.CODE_RD_MISS	Number of instruction fetches that missed the L2 cache.	
24H	27H	L2_RQSTS.ALL_DEMAND_MISS	Demand requests that miss L2 cache.	
24H	E7H	L2_RQSTS.ALL_DEMAND_REFERENCES	Demand requests to L2 cache.	
24H	E4H	L2_RQSTS.ALL_CODE_RD	Counts all L2 code requests.	
24H	50H	L2_RQSTS.L2_PF_HIT	Counts all L2 HW prefetcher requests that hit L2.	
24H	30H	L2_RQSTS.L2_PF_MISS	Counts all L2 HW prefetcher requests that missed L2.	
24H	F8H	L2_RQSTS.ALL_PF	Counts all L2 HW prefetcher requests.	
24H	3FH	L2_RQSTS.MISS	All requests that missed L2.	
24H	FFH	L2_RQSTS.REFERENCES	All requests to L2 cache.	
27H	50H	L2_DEMAND_RQSTS.WB_HIT	Not rejected writebacks that hit L2 cache.	
2EH	4FH	LONGEST_LAT_CACHE.REFERENCE	This event counts requests originating from the core that reference a cache line in the last level cache.	See Table 19-1.
2EH	41H	LONGEST_LAT_CACHE.MISS	This event counts each cache miss condition for references to the last level cache.	See Table 19-1.

Table 19-9. Non-Architectural Performance Events in the Processor Core of 4th Generation Intel® Core™ Processors (Contd.)

Event Num.	Umask Value	Event Mask Mnemonic	Description	Comment
3CH	00H	CPU_CLK_UNHALTED.THREAD_P	Counts the number of thread cycles while the thread is not in a halt state. The thread enters the halt state when it is running the HLT instruction. The core frequency may change from time to time due to power or thermal throttling.	See Table 19-1.
3CH	01H	CPU_CLK_THREAD_UNHALTED.REF_XCLK	Increments at the frequency of XCLK (100 MHz) when not halted.	See Table 19-1.
48H	01H	L1D_PEND_MISS.PENDING	Increments the number of outstanding L1D misses every cycle. Set Cmask = 1 and Edge =1 to count occurrences.	Counter 2 only. Set Cmask = 1 to count cycles.
49H	01H	DTLB_STORE_MISSES.MISS_CAUSES_A_WALK	Miss in all TLB levels causes a page walk of any page size (4K/2M/4M/1G).	
49H	02H	DTLB_STORE_MISSES.WALK_COMPLETED_4K	Completed page walks due to store misses in one or more TLB levels of 4K page structure.	
49H	04H	DTLB_STORE_MISSES.WALK_COMPLETED_2M_4M	Completed page walks due to store misses in one or more TLB levels of 2M/4M page structure.	
49H	0EH	DTLB_STORE_MISSES.WALK_COMPLETED	Completed page walks due to store miss in any TLB levels of any page size (4K/2M/4M/1G).	
49H	10H	DTLB_STORE_MISSES.WALK_DURATION	Cycles PMH is busy with this walk.	
49H	20H	DTLB_STORE_MISSES.STLB_HIT_4K	Store misses that missed DTLB but hit STLB (4K).	
49H	40H	DTLB_STORE_MISSES.STLB_HIT_2M	Store misses that missed DTLB but hit STLB (2M).	
49H	60H	DTLB_STORE_MISSES.STLB_HIT	Store operations that miss the first TLB level but hit the second and do not cause page walks.	
49H	80H	DTLB_STORE_MISSES.PDE_CACHE_MISS	DTLB store misses with low part of linear-to-physical address translation missed.	
4CH	01H	LOAD_HIT_PRE.SW_PF	Non-SW-prefetch load dispatches that hit fill buffer allocated for S/W prefetch.	
4CH	02H	LOAD_HIT_PRE.HW_PF	Non-SW-prefetch load dispatches that hit fill buffer allocated for H/W prefetch.	
51H	01H	L1D.REPLACEMENT	Counts the number of lines brought into the L1 data cache.	
58H	04H	MOVE_ELIMINATION.INT_NOT_ELIMINATED	Number of integer move elimination candidate uops that were not eliminated.	
58H	08H	MOVE_ELIMINATION.SIMD_NOT_ELIMINATED	Number of SIMD move elimination candidate uops that were not eliminated.	
58H	01H	MOVE_ELIMINATION.INT_ELIMINATED	Number of integer move elimination candidate uops that were eliminated.	
58H	02H	MOVE_ELIMINATION.SIMD_ELIMINATED	Number of SIMD move elimination candidate uops that were eliminated.	
5CH	01H	CPL_CYCLES.RING0	Unhalted core cycles when the thread is in ring 0.	Use Edge to count transition.
5CH	02H	CPL_CYCLES.RING123	Unhalted core cycles when the thread is not in ring 0.	
5EH	01H	RS_EVENTS.EMPTY_CYCLES	Cycles the RS is empty for the thread.	

Table 19-9. Non-Architectural Performance Events in the Processor Core of 4th Generation Intel® Core™ Processors (Contd.)

Event Num.	Umask Value	Event Mask Mnemonic	Description	Comment
60H	01H	OFFCORE_REQUESTS_OUTSTANDING.DEMAND_DATA_RD	Offcore outstanding demand data read transactions in SQ to uncore. Set Cmask=1 to count cycles.	Use only when HTT is off.
60H	02H	OFFCORE_REQUESTS_OUTSTANDING.DEMAND_CODE_RD	Offcore outstanding Demand code Read transactions in SQ to uncore. Set Cmask=1 to count cycles.	Use only when HTT is off.
60H	04H	OFFCORE_REQUESTS_OUTSTANDING.DEMAND_RFO	Offcore outstanding RFO store transactions in SQ to uncore. Set Cmask=1 to count cycles.	Use only when HTT is off.
60H	08H	OFFCORE_REQUESTS_OUTSTANDING.ALL_DATA_RD	Offcore outstanding cacheable data read transactions in SQ to uncore. Set Cmask=1 to count cycles.	Use only when HTT is off.
63H	01H	LOCK_CYCLES.SPLIT_LOCK_UC_LOCK_DURATION	Cycles in which the L1D and L2 are locked, due to a UC lock or split lock.	
63H	02H	LOCK_CYCLES.CACHE_LOCK_DURATION	Cycles in which the L1D is locked.	
79H	02H	IDQ.EMPTY	Counts cycles the IDQ is empty.	
79H	04H	IDQ.MITE_UOPS	Increment each cycle # of uops delivered to IDQ from MITE path. Set Cmask = 1 to count cycles.	Can combine Umask 04H and 20H.
79H	08H	IDQ.DSB_UOPS	Increment each cycle. # of uops delivered to IDQ from DSB path. Set Cmask = 1 to count cycles.	Can combine Umask 08H and 10H.
79H	10H	IDQ.MS_DSB_UOPS	Increment each cycle # of uops delivered to IDQ when MS_busy by DSB. Set Cmask = 1 to count cycles. Add Edge=1 to count # of delivery.	Can combine Umask 04H, 08H.
79H	20H	IDQ.MS_MITE_UOPS	Increment each cycle # of uops delivered to IDQ when MS_busy by MITE. Set Cmask = 1 to count cycles.	Can combine Umask 04H, 08H.
79H	30H	IDQ.MS_UOPS	Increment each cycle # of uops delivered to IDQ from MS by either DSB or MITE. Set Cmask = 1 to count cycles.	Can combine Umask 04H, 08H.
79H	18H	IDQ.ALL_DSB_CYCLES_ANY_UOPS	Counts cycles DSB is delivered at least one uops. Set Cmask = 1.	
79H	18H	IDQ.ALL_DSB_CYCLES_4_UOPS	Counts cycles DSB is delivered four uops. Set Cmask = 4.	
79H	24H	IDQ.ALL_MITE_CYCLES_ANY_UOPS	Counts cycles MITE is delivered at least one uop. Set Cmask = 1.	
79H	24H	IDQ.ALL_MITE_CYCLES_4_UOPS	Counts cycles MITE is delivered four uops. Set Cmask = 4.	
79H	3CH	IDQ.MITE_ALL_UOPS	# of uops delivered to IDQ from any path.	
80H	02H	ICACHE.MISSES	Number of Instruction Cache, Streaming Buffer and Victim Cache Misses. Includes UC accesses.	
85H	01H	ITLB_MISSES.MISS_CAUSES_A_WALK	Misses in ITLB that causes a page walk of any page size.	
85H	02H	ITLB_MISSES.WALK_COMPLETED_4K	Completed page walks due to misses in ITLB 4K page entries.	
85H	04H	ITLB_MISSES.WALK_COMPLETED_2M_4M	Completed page walks due to misses in ITLB 2M/4M page entries.	
85H	0EH	ITLB_MISSES.WALK_COMPLETED	Completed page walks in ITLB of any page size.	

Table 19-9. Non-Architectural Performance Events in the Processor Core of 4th Generation Intel® Core™ Processors (Contd.)

Event Num.	Umask Value	Event Mask Mnemonic	Description	Comment
85H	10H	ITLB_MISSES.WALK_DURATION	Cycle PMH is busy with a walk.	
85H	20H	ITLB_MISSES.STLB_HIT_4K	ITLB misses that hit STLB (4K).	
85H	40H	ITLB_MISSES.STLB_HIT_2M	ITLB misses that hit STLB (2M).	
85H	60H	ITLB_MISSES.STLB_HIT	ITLB misses that hit STLB. No page walk.	
87H	01H	ILD_STALL.LCP	Stalls caused by changing prefix length of the instruction.	
87H	04H	ILD_STALL.IQ_FULL	Stall cycles due to IQ is full.	
88H	01H	BR_INST_EXEC.COND	Qualify conditional near branch instructions executed, but not necessarily retired.	Must combine with umask 40H, 80H.
88H	02H	BR_INST_EXEC.DIRECT_JMP	Qualify all unconditional near branch instructions excluding calls and indirect branches.	Must combine with umask 80H.
88H	04H	BR_INST_EXEC.INDIRECT_JMP_ NON_CALL_RET	Qualify executed indirect near branch instructions that are not calls or returns.	Must combine with umask 80H.
88H	08H	BR_INST_EXEC.RETURN_NEAR	Qualify indirect near branches that have a return mnemonic.	Must combine with umask 80H.
88H	10H	BR_INST_EXEC.DIRECT_NEAR_C ALL	Qualify unconditional near call branch instructions, excluding non-call branch, executed.	Must combine with umask 80H.
88H	20H	BR_INST_EXEC.INDIRECT_NEAR _CALL	Qualify indirect near calls, including both register and memory indirect, executed.	Must combine with umask 80H.
88H	40H	BR_INST_EXEC.NONTAKEN	Qualify non-taken near branches executed.	Applicable to umask 01H only.
88H	80H	BR_INST_EXEC.TAKEN	Qualify taken near branches executed. Must combine with 01H,02H, 04H, 08H, 10H, 20H.	
88H	FFH	BR_INST_EXEC.ALL_BRANCHES	Counts all near executed branches (not necessarily retired).	
89H	01H	BR_MISP_EXEC.COND	Qualify conditional near branch instructions mispredicted.	Must combine with umask 40H, 80H.
89H	04H	BR_MISP_EXEC.INDIRECT_JMP_ NON_CALL_RET	Qualify mispredicted indirect near branch instructions that are not calls or returns.	Must combine with umask 80H.
89H	08H	BR_MISP_EXEC.RETURN_NEAR	Qualify mispredicted indirect near branches that have a return mnemonic.	Must combine with umask 80H.
89H	10H	BR_MISP_EXEC.DIRECT_NEAR_C ALL	Qualify mispredicted unconditional near call branch instructions, excluding non-call branch, executed.	Must combine with umask 80H.
89H	20H	BR_MISP_EXEC.INDIRECT_NEAR _CALL	Qualify mispredicted indirect near calls, including both register and memory indirect, executed.	Must combine with umask 80H.
89H	40H	BR_MISP_EXEC.NONTAKEN	Qualify mispredicted non-taken near branches executed.	Applicable to umask 01H only.
89H	80H	BR_MISP_EXEC.TAKEN	Qualify mispredicted taken near branches executed. Must combine with 01H,02H, 04H, 08H, 10H, 20H.	
89H	FFH	BR_MISP_EXEC.ALL_BRANCHES	Counts all near executed branches (not necessarily retired).	
9CH	01H	IDQ_UOPS_NOT_DELIVERED.CO RE	Count issue pipeline slots where no uop was delivered from the front end to the back end when there is no back-end stall.	Use Cmask to qualify uop b/w.

Table 19-9. Non-Architectural Performance Events in the Processor Core of 4th Generation Intel® Core™ Processors (Contd.)

Event Num.	Umask Value	Event Mask Mnemonic	Description	Comment
A1H	01H	UOPS_EXECUTED_PORT.PORT_0	Cycles which a uop is dispatched on port 0 in this thread.	Set AnyThread to count per core.
A1H	02H	UOPS_EXECUTED_PORT.PORT_1	Cycles which a uop is dispatched on port 1 in this thread.	Set AnyThread to count per core.
A1H	04H	UOPS_EXECUTED_PORT.PORT_2	Cycles which a uop is dispatched on port 2 in this thread.	Set AnyThread to count per core.
A1H	08H	UOPS_EXECUTED_PORT.PORT_3	Cycles which a uop is dispatched on port 3 in this thread.	Set AnyThread to count per core.
A1H	10H	UOPS_EXECUTED_PORT.PORT_4	Cycles which a uop is dispatched on port 4 in this thread.	Set AnyThread to count per core.
A1H	20H	UOPS_EXECUTED_PORT.PORT_5	Cycles which a uop is dispatched on port 5 in this thread.	Set AnyThread to count per core.
A1H	40H	UOPS_EXECUTED_PORT.PORT_6	Cycles which a uop is dispatched on port 6 in this thread.	Set AnyThread to count per core.
A1H	80H	UOPS_EXECUTED_PORT.PORT_7	Cycles which a uop is dispatched on port 7 in this thread	Set AnyThread to count per core.
A2H	01H	RESOURCE_STALLS.ANY	Cycles allocation is stalled due to resource related reason.	
A2H	04H	RESOURCE_STALLS.RS	Cycles stalled due to no eligible RS entry available.	
A2H	08H	RESOURCE_STALLS.SB	Cycles stalled due to no store buffers available (not including draining form sync).	
A2H	10H	RESOURCE_STALLS.ROB	Cycles stalled due to re-order buffer full.	
A3H	01H	CYCLE_ACTIVITY.CYCLES_L2_PENDING	Cycles with pending L2 miss loads. Set Cmask=2 to count cycle.	Use only when HTT is off.
A3H	02H	CYCLE_ACTIVITY.CYCLES_LDM_PENDING	Cycles with pending memory loads. Set Cmask=2 to count cycle.	
A3H	05H	CYCLE_ACTIVITY.STALLS_L2_PENDING	Number of loads missed L2.	Use only when HTT is off.
A3H	08H	CYCLE_ACTIVITY.CYCLES_L1D_PENDING	Cycles with pending L1 data cache miss loads. Set Cmask=8 to count cycle.	PMC2 only.
A3H	0CH	CYCLE_ACTIVITY.STALLS_L1D_PENDING	Execution stalls due to L1 data cache miss loads. Set Cmask=0CH.	PMC2 only.
A8H	01H	LSD.UOPS	Number of uops delivered by the LSD.	
AEH	01H	ITLB.ITLB_FLUSH	Counts the number of ITLB flushes, includes 4k/2M/4M pages.	
B0H	01H	OFFCORE_REQUESTS.DEMAND_DATA_RD	Demand data read requests sent to uncore.	Use only when HTT is off.
B0H	02H	OFFCORE_REQUESTS.DEMAND_CODE_RD	Demand code read requests sent to uncore.	Use only when HTT is off.
B0H	04H	OFFCORE_REQUESTS.DEMAND_RFO	Demand RFO read requests sent to uncore, including regular RFOs, locks, ItoM.	Use only when HTT is off.
B0H	08H	OFFCORE_REQUESTS.ALL_DATA_RD	Data read requests sent to uncore (demand and prefetch).	Use only when HTT is off.
B1H	02H	UOPS_EXECUTED.CORE	Counts total number of uops to be executed per-core each cycle.	Do not need to set ANY.

Table 19-9. Non-Architectural Performance Events in the Processor Core of 4th Generation Intel® Core™ Processors (Contd.)

Event Num.	Umask Value	Event Mask Mnemonic	Description	Comment
B7H	01H	OFF_CORE_RESPONSE_0	See Table 18-28 or Table 18-29.	Requires MSR 01A6H.
BBH	01H	OFF_CORE_RESPONSE_1	See Table 18-28 or Table 18-29.	Requires MSR 01A7H.
BCH	11H	PAGE_WALKER_LOADS.DTLB_L1	Number of DTLB page walker loads that hit in the L1+FB.	
BCH	21H	PAGE_WALKER_LOADS.ITLB_L1	Number of ITLB page walker loads that hit in the L1+FB.	
BCH	12H	PAGE_WALKER_LOADS.DTLB_L2	Number of DTLB page walker loads that hit in the L2.	
BCH	22H	PAGE_WALKER_LOADS.ITLB_L2	Number of ITLB page walker loads that hit in the L2.	
BCH	14H	PAGE_WALKER_LOADS.DTLB_L3	Number of DTLB page walker loads that hit in the L3.	
BCH	24H	PAGE_WALKER_LOADS.ITLB_L3	Number of ITLB page walker loads that hit in the L3.	
BCH	18H	PAGE_WALKER_LOADS.DTLB_MEMORY	Number of DTLB page walker loads from memory.	
BCH	28H	PAGE_WALKER_LOADS.ITLB_MEMORY	Number of ITLB page walker loads from memory.	
BDH	01H	TLB_FLUSH.DTLB_THREAD	DTLB flush attempts of the thread-specific entries.	
BDH	20H	TLB_FLUSH.STLB_ANY	Count number of STLB flush attempts.	
C0H	00H	INST_RETIRED.ANY_P	Number of instructions at retirement.	See Table 19-1.
C0H	01H	INST_RETIRED.PREC_DIST	Precise instruction retired event with HW to reduce effect of PEBS shadow in IP distribution.	PMC1 only.
C1H	08H	OTHER_ASSISTS.AVX_TO_SSE	Number of transitions from AVX-256 to legacy SSE when penalty applicable.	
C1H	10H	OTHER_ASSISTS.SSE_TO_AVX	Number of transitions from SSE to AVX-256 when penalty applicable.	
C1H	40H	OTHER_ASSISTS.ANY_WB_ASSIST	Number of microcode assists invoked by HW upon uop writeback.	
C2H	01H	UOPS_RETIRED.ALL	Counts the number of micro-ops retired. Use Cmask=1 and invert to count active cycles or stalled cycles.	Supports PEBS and DataLA; use Any=1 for core granular.
C2H	02H	UOPS_RETIRED.RETIRE_SLOTS	Counts the number of retirement slots used each cycle.	Supports PEBS.
C3H	02H	MACHINE_CLEARS.MEMORY_ORDERING	Counts the number of machine clears due to memory order conflicts.	
C3H	04H	MACHINE_CLEARS.SMC	Number of self-modifying-code machine clears detected.	
C3H	20H	MACHINE_CLEARS.MASKMOV	Counts the number of executed AVX masked load operations that refer to an illegal address range with the mask bits set to 0.	
C4H	00H	BR_INST_RETIRED.ALL_BRANCHES	Branch instructions at retirement.	See Table 19-1.
C4H	01H	BR_INST_RETIRED.CONDITIONAL	Counts the number of conditional branch instructions retired.	Supports PEBS.
C4H	02H	BR_INST_RETIRED.NEAR_CALL	Direct and indirect near call instructions retired.	Supports PEBS.
C4H	04H	BR_INST_RETIRED.ALL_BRANCHES	Counts the number of branch instructions retired.	Supports PEBS.

Table 19-9. Non-Architectural Performance Events in the Processor Core of 4th Generation Intel® Core™ Processors (Contd.)

Event Num.	Umask Value	Event Mask Mnemonic	Description	Comment
C4H	08H	BR_INST_RETIRED.NEAR_RETURN	Counts the number of near return instructions retired.	Supports PEBS.
C4H	10H	BR_INST_RETIRED.NOT_TAKEN	Counts the number of not taken branch instructions retired.	
C4H	20H	BR_INST_RETIRED.NEAR_TAKEN	Number of near taken branches retired.	Supports PEBS.
C4H	40H	BR_INST_RETIRED.FAR_BRANCH	Number of far branches retired.	
C5H	00H	BR_MISP_RETIRED.ALL_BRANCHES	Mispredicted branch instructions at retirement.	See Table 19-1.
C5H	01H	BR_MISP_RETIRED.CONDITIONAL	Mispredicted conditional branch instructions retired.	Supports PEBS.
C5H	04H	BR_MISP_RETIRED.ALL_BRANCHES	Mispredicted macro branch instructions retired.	Supports PEBS.
C5H	20H	BR_MISP_RETIRED.NEAR_TAKEN	Number of near branch instructions retired that were taken but mispredicted.	
CAH	02H	FP_ASSIST.X87_OUTPUT	Number of X87 FP assists due to output values.	
CAH	04H	FP_ASSIST.X87_INPUT	Number of X87 FP assists due to input values.	
CAH	08H	FP_ASSIST.SIMD_OUTPUT	Number of SIMD FP assists due to output values.	
CAH	10H	FP_ASSIST.SIMD_INPUT	Number of SIMD FP assists due to input values.	
CAH	1EH	FP_ASSIST.ANY	Cycles with any input/output SSE* or FP assists.	
CCH	20H	ROB_MISC_EVENTS.LBR_INSERTS	Count cases of saving new LBR records by hardware.	
CDH	01H	MEM_TRANS_RETIRED.LOAD_LATENCY	Randomly sampled loads whose latency is above a user defined threshold. A small fraction of the overall loads are sampled due to randomization.	Specify threshold in MSR 3F6H.
D0H	11H	MEM_UOPS_RETIRED.STLB_MISS_LOADS	Retired load uops that miss the STLB.	Supports PEBS and DataLA.
D0H	12H	MEM_UOPS_RETIRED.STLB_MISS_STORES	Retired store uops that miss the STLB.	Supports PEBS and DataLA.
D0H	21H	MEM_UOPS_RETIRED.LOCK_LOADS	Retired load uops with locked access.	Supports PEBS and DataLA.
D0H	41H	MEM_UOPS_RETIRED.SPLIT_LOADS	Retired load uops that split across a cacheline boundary.	Supports PEBS and DataLA.
D0H	42H	MEM_UOPS_RETIRED.SPLIT_STORES	Retired store uops that split across a cacheline boundary.	Supports PEBS and DataLA.
D0H	81H	MEM_UOPS_RETIRED.ALL_LOADS	All retired load uops.	Supports PEBS and DataLA.
D0H	82H	MEM_UOPS_RETIRED.ALL_STORES	All retired store uops.	Supports PEBS and DataLA.
D1H	01H	MEM_LOAD_UOPS_RETIRED.L1_HIT	Retired load uops with L1 cache hits as data sources.	Supports PEBS and DataLA.
D1H	02H	MEM_LOAD_UOPS_RETIRED.L2_HIT	Retired load uops with L2 cache hits as data sources.	Supports PEBS and DataLA.

Table 19-9. Non-Architectural Performance Events in the Processor Core of 4th Generation Intel® Core™ Processors (Contd.)

Event Num.	Umask Value	Event Mask Mnemonic	Description	Comment
D1H	04H	MEM_LOAD_UOPS_RETIRED.L3_HIT	Retired load uops with L3 cache hits as data sources.	Supports PEBS and DataLA.
D1H	08H	MEM_LOAD_UOPS_RETIRED.L1_MISS	Retired load uops missed L1 cache as data sources.	Supports PEBS and DataLA.
D1H	10H	MEM_LOAD_UOPS_RETIRED.L2_MISS	Retired load uops missed L2. Unknown data source excluded.	Supports PEBS and DataLA.
D1H	20H	MEM_LOAD_UOPS_RETIRED.L3_MISS	Retired load uops missed L3. Excludes unknown data source .	Supports PEBS and DataLA.
D1H	40H	MEM_LOAD_UOPS_RETIRED.HIT_LFB	Retired load uops which data sources were load uops missed L1 but hit FB due to preceding miss to the same cache line with data not ready.	Supports PEBS and DataLA.
D2H	01H	MEM_LOAD_UOPS_L3_HIT_RETIRED.XSNP_MISS	Retired load uops which data sources were L3 hit and cross-core snoop missed in on-pkg core cache.	Supports PEBS and DataLA.
D2H	02H	MEM_LOAD_UOPS_L3_HIT_RETIRED.XSNP_HIT	Retired load uops which data sources were L3 and cross-core snoop hits in on-pkg core cache.	Supports PEBS and DataLA.
D2H	04H	MEM_LOAD_UOPS_L3_HIT_RETIRED.XSNP_HITM	Retired load uops which data sources were HitM responses from shared L3.	Supports PEBS and DataLA.
D2H	08H	MEM_LOAD_UOPS_L3_HIT_RETIRED.XSNP_NONE	Retired load uops which data sources were hits in L3 without snoops required.	Supports PEBS and DataLA.
D3H	01H	MEM_LOAD_UOPS_L3_MISS_RETIRED.LOCAL_DRAM	Retired load uops which data sources missed L3 but serviced from local dram.	Supports PEBS and DataLA.
E6H	1FH	BACLEARS.ANY	Number of front end re-steers due to BPU misprediction.	
F0H	01H	L2_TRANS.DEMAND_DATA_RD	Demand data read requests that access L2 cache.	
F0H	02H	L2_TRANS.RFO	RFO requests that access L2 cache.	
F0H	04H	L2_TRANS.CODE_RD	L2 cache accesses when fetching instructions.	
F0H	08H	L2_TRANS.ALL_PF	Any MLC or L3 HW prefetch accessing L2, including rejects.	
F0H	10H	L2_TRANS.L1D_WB	L1D writebacks that access L2 cache.	
F0H	20H	L2_TRANS.L2_FILL	L2 fill requests that access L2 cache.	
F0H	40H	L2_TRANS.L2_WB	L2 writebacks that access L2 cache.	
F0H	80H	L2_TRANS.ALL_REQUESTS	Transactions accessing L2 pipe.	
F1H	01H	L2_LINES_IN.I	L2 cache lines in I state filling L2.	Counting does not cover rejects.
F1H	02H	L2_LINES_IN.S	L2 cache lines in S state filling L2.	Counting does not cover rejects.
F1H	04H	L2_LINES_IN.E	L2 cache lines in E state filling L2.	Counting does not cover rejects.
F1H	07H	L2_LINES_IN.ALL	L2 cache lines filling L2.	Counting does not cover rejects.
F2H	05H	L2_LINES_OUT.DEMAND_CLEAN	Clean L2 cache lines evicted by demand.	
F2H	06H	L2_LINES_OUT.DEMAND_DIRTY	Dirty L2 cache lines evicted by demand.	

Table 19-10. Intel TSX Performance Events in Processors Based on Haswell Microarchitecture

Event Num.	Umask Value	Event Mask Mnemonic	Description	Comment
54H	01H	TX_MEM.ABORT_CONFLICT	Number of times a transactional abort was signaled due to a data conflict on a transactionally accessed address.	
54H	02H	TX_MEM.ABORT_CAPACITY_WRITE	Number of times a transactional abort was signaled due to a data capacity limitation for transactional writes.	
54H	04H	TX_MEM.ABORT_HLE_STORE_TO_ELIDED_LOCK	Number of times a HLE transactional region aborted due to a non XRELEASE prefixed instruction writing to an elided lock in the elision buffer.	
54H	08H	TX_MEM.ABORT_HLE_ELISION_BUFFER_NOT_EMPTY	Number of times an HLE transactional execution aborted due to NoAllocatedElisionBuffer being non-zero.	
54H	10H	TX_MEM.ABORT_HLE_ELISION_BUFFER_MISMATCH	Number of times an HLE transactional execution aborted due to XRELEASE lock not satisfying the address and value requirements in the elision buffer.	
54H	20H	TX_MEM.ABORT_HLE_ELISION_BUFFER_UNSUPPORTED_ALIGNMENT	Number of times an HLE transactional execution aborted due to an unsupported read alignment from the elision buffer.	
54H	40H	TX_MEM.HLE_ELISION_BUFFER_FULL	Number of times HLE lock could not be elided due to ElisionBufferAvailable being zero.	
5DH	01H	TX_EXEC.MISC1	Counts the number of times a class of instructions that may cause a transactional abort was executed. Since this is the count of execution, it may not always cause a transactional abort.	
5DH	02H	TX_EXEC.MISC2	Counts the number of times a class of instructions (for example, vzeroupper) that may cause a transactional abort was executed inside a transactional region.	
5DH	04H	TX_EXEC.MISC3	Counts the number of times an instruction execution caused the transactional nest count supported to be exceeded.	
5DH	08H	TX_EXEC.MISC4	Counts the number of times an XBEGIN instruction was executed inside an HLE transactional region.	
5DH	10H	TX_EXEC.MISC5	Counts the number of times an instruction with HLE-XACQUIRE semantic was executed inside an RTM transactional region.	

Table 19-10. Intel TSX Performance Events in Processors Based on Haswell Microarchitecture

Event Num.	Umask Value	Event Mask Mnemonic	Description	Comment
C8H	01H	HLE_RETIRED.START	Number of times an HLE execution started.	IF HLE is supported.
C8H	02H	HLE_RETIRED.COMMIT	Number of times an HLE execution successfully committed.	
C8H	04H	HLE_RETIRED.ABORTED	Number of times an HLE execution aborted due to any reasons (multiple categories may count as one). Supports PEBS.	
C8H	08H	HLE_RETIRED.ABORTED_MEM	Number of times an HLE execution aborted due to various memory events (for example, read/write capacity and conflicts).	
C8H	10H	HLE_RETIRED.ABORTED_TIMER	Number of times an HLE execution aborted due to uncommon conditions.	
C8H	20H	HLE_RETIRED.ABORTED_UNFRIENDLY	Number of times an HLE execution aborted due to HLE-unfriendly instructions.	
C8H	40H	HLE_RETIRED.ABORTED_MEMTYPE	Number of times an HLE execution aborted due to incompatible memory type.	
C8H	80H	HLE_RETIRED.ABORTED_EVENTS	Number of times an HLE execution aborted due to none of the previous 4 categories (for example, interrupts).	
C9H	01H	RTM_RETIRED.START	Number of times an RTM execution started.	IF RTM is supported.
C9H	02H	RTM_RETIRED.COMMIT	Number of times an RTM execution successfully committed.	
C9H	04H	RTM_RETIRED.ABORTED	Number of times an RTM execution aborted due to any reasons (multiple categories may count as one). Supports PEBS.	
C9H	08H	RTM_RETIRED.ABORTED_MEM	Number of times an RTM execution aborted due to various memory events (for example, read/write capacity and conflicts).	IF RTM is supported.
C9H	10H	RTM_RETIRED.ABORTED_TIMER	Number of times an RTM execution aborted due to uncommon conditions.	
C9H	20H	RTM_RETIRED.ABORTED_UNFRIENDLY	Number of times an RTM execution aborted due to HLE-unfriendly instructions.	
C9H	40H	RTM_RETIRED.ABORTED_MEMTYPE	Number of times an RTM execution aborted due to incompatible memory type.	
C9H	80H	RTM_RETIRED.ABORTED_EVENTS	Number of times an RTM execution aborted due to none of the previous 4 categories (for example, interrupt).	

Non-architectural performance monitoring events that are located in the uncore sub-system are implementation specific between different platforms using processors based on Haswell microarchitecture and with different DisplayFamily_DisplayModel signatures. Processors with CPUID signature of DisplayFamily_DisplayModel 06_3CH and 06_45H support performance events listed in Table 19-11.

Table 19-11. Non-Architectural Uncore Performance Events in the 4th Generation Intel® Core™ Processors

Event Num.[1]	Umask Value	Event Mask Mnemonic	Description	Comment
22H	01H	UNC_CBO_XSNP_RESPONSE.MISS	A snoop misses in some processor core.	Must combine with one of the umask values of 20H, 40H, 80H.
22H	02H	UNC_CBO_XSNP_RESPONSE.INVAL	A snoop invalidates a non-modified line in some processor core.	
22H	04H	UNC_CBO_XSNP_RESPONSE.HIT	A snoop hits a non-modified line in some processor core.	
22H	08H	UNC_CBO_XSNP_RESPONSE.HITM	A snoop hits a modified line in some processor core.	
22H	10H	UNC_CBO_XSNP_RESPONSE.INVAL_M	A snoop invalidates a modified line in some processor core.	
22H	20H	UNC_CBO_XSNP_RESPONSE.EXTERNAL_FILTER	Filter on cross-core snoops initiated by this Cbox due to external snoop request.	Must combine with at least one of 01H, 02H, 04H, 08H, 10H.
22H	40H	UNC_CBO_XSNP_RESPONSE.XCORE_FILTER	Filter on cross-core snoops initiated by this Cbox due to processor core memory request.	
22H	80H	UNC_CBO_XSNP_RESPONSE.EVICTION_FILTER	Filter on cross-core snoops initiated by this Cbox due to L3 eviction.	
34H	01H	UNC_CBO_CACHE_LOOKUP.M	L3 lookup request that access cache and found line in M-state.	Must combine with one of the umask values of 10H, 20H, 40H, 80H.
34H	06H	UNC_CBO_CACHE_LOOKUP.ES	L3 lookup request that access cache and found line in E or S state.	
34H	08H	UNC_CBO_CACHE_LOOKUP.I	L3 lookup request that access cache and found line in I-state.	
34H	10H	UNC_CBO_CACHE_LOOKUP.READ_FILTER	Filter on processor core initiated cacheable read requests. Must combine with at least one of 01H, 02H, 04H, 08H.	
34H	20H	UNC_CBO_CACHE_LOOKUP.WRITE_FILTER	Filter on processor core initiated cacheable write requests. Must combine with at least one of 01H, 02H, 04H, 08H.	
34H	40H	UNC_CBO_CACHE_LOOKUP.EXTSNP_FILTER	Filter on external snoop requests. Must combine with at least one of 01H, 02H, 04H, 08H.	
34H	80H	UNC_CBO_CACHE_LOOKUP.ANY_REQUEST_FILTER	Filter on any IRQ or IPQ initiated requests including uncacheable, non-coherent requests. Must combine with at least one of 01H, 02H, 04H, 08H.	
80H	01H	UNC_ARB_TRK_OCCUPANCY.ALL	Counts cycles weighted by the number of requests waiting for data returning from the memory controller. Accounts for coherent and non-coherent requests initiated by IA cores, processor graphic units, or L3.	Counter 0 only.
81H	01H	UNC_ARB_TRK_REQUEST.ALL	Counts the number of coherent and in-coherent requests initiated by IA cores, processor graphic units, or L3.	
81H	20H	UNC_ARB_TRK_REQUEST.WRITES	Counts the number of allocated write entries, include full, partial, and L3 evictions.	
81H	80H	UNC_ARB_TRK_REQUEST.EVICTIONS	Counts the number of L3 evictions allocated.	

Table 19-11. Non-Architectural Uncore Performance Events in the 4th Generation Intel® Core™ Processors (Contd.)

Event Num.[1]	Umask Value	Event Mask Mnemonic	Description	Comment
83H	01H	UNC_ARB_COH_TRK_OCCUPANCY.ALL	Cycles weighted by number of requests pending in Coherency Tracker.	Counter 0 only.
84H	01H	UNC_ARB_COH_TRK_REQUEST.ALL	Number of requests allocated in Coherency Tracker.	

NOTES:

1. The uncore events must be programmed using MSRs located in specific performance monitoring units in the uncore. UNC_CBO* events are supported using MSR_UNC_CBO* MSRs; UNC_ARB* events are supported using MSR_UNC_ARB*MSRs.

19.6.1 Performance Monitoring Events in the Processor Core of Intel Xeon Processor E5 v3 Family

Non-architectural performance monitoring events in the processor core that are applicable only to Intel Xeon processor E5 v3 family based on the Haswell-E microarchitecture, with CPUID signature of DisplayFamily_DisplayModel 06_3FH, are listed in Table 19-12. The performance events listed in Table 19-9 and Table 19-10 also apply Intel Xeon processor E5 v3 family, except that the OFF_CORE_RESPONSE_x event listed in Table 19-9 should reference Table 18-30.

Uncore performance monitoring events for Intel Xeon Processor E5 v3 families are described in "Intel® Xeon® Processor E5 v3 Uncore Performance Monitoring Programming Reference Manual".

Table 19-12. Non-Architectural Performance Events Applicable only to the Processor Core of Intel® Xeon® Processor E5 v3 Family

Event Num.	Umask Value	Event Mask Mnemonic	Description	Comment
D3H	04H	MEM_LOAD_UOPS_L3_MISS_RETIRED.REMOTE_DRAM	Retired load uops whose data sources were remote DRAM (snoop not needed, Snoop Miss).	Supports PEBS.
D3H	10H	MEM_LOAD_UOPS_L3_MISS_RETIRED.REMOTE_HITM	Retired load uops whose data sources were remote cache HITM.	Supports PEBS.
D3H	20H	MEM_LOAD_UOPS_L3_MISS_RETIRED.REMOTE_FWD	Retired load uops whose data sources were forwards from a remote cache.	Supports PEBS.

19.7 PERFORMANCE MONITORING EVENTS FOR 3RD GENERATION INTEL® CORE™ PROCESSORS

3rd generation Intel® Core™ processors and Intel Xeon processor E3-1200 v2 product family are based on Intel microarchitecture code name Ivy Bridge. They support architectural performance-monitoring events listed in Table 19-1. Non-architectural performance-monitoring events in the processor core are listed in Table 19-13. The events in Table 19-13 apply to processors with CPUID signature of DisplayFamily_DisplayModel encoding with the following values: 06_3AH. Fixed counters in the core PMU support the architecture events defined in Table 19-24.

Additional information on event specifics (e.g. derivative events using specific IA32_PERFEVTSELx modifiers, limitations, special notes and recommendations) can be found at http://software.intel.com/en-us/forums/software-tuning-performance-optimization-platform-monitoring.

Table 19-13. Non-Architectural Performance Events In the Processor Core of 3rd Generation Intel® Core™ i7, i5, i3 Processors

Event Num.	Umask Value	Event Mask Mnemonic	Description	Comment
03H	02H	LD_BLOCKS.STORE_FORWARD	Loads blocked by overlapping with store buffer that cannot be forwarded.	
03H	08H	LD_BLOCKS.NO_SR	The number of times that split load operations are temporarily blocked because all resources for handling the split accesses are in use.	
05H	01H	MISALIGN_MEM_REF.LOADS	Speculative cache-line split load uops dispatched to L1D.	
05H	02H	MISALIGN_MEM_REF.STORES	Speculative cache-line split Store-address uops dispatched to L1D.	
07H	01H	LD_BLOCKS_PARTIAL.ADDRESS_ALIAS	False dependencies in MOB due to partial compare on address.	
08H	81H	DTLB_LOAD_MISSES.MISS_CAUSES_A_WALK	Misses in all TLB levels that cause a page walk of any page size from demand loads.	
08H	82H	DTLB_LOAD_MISSES.WALK_COMPLETED	Misses in all TLB levels that caused page walk completed of any size by demand loads.	
08H	84H	DTLB_LOAD_MISSES.WALK_DURATION	Cycle PMH is busy with a walk due to demand loads.	
08H	88H	DTLB_LOAD_MISSES.LARGE_PAGE_WALK_DURATION	Page walk for a large page completed for Demand load.	
0EH	01H	UOPS_ISSUED.ANY	Increments each cycle the # of Uops issued by the RAT to RS. Set Cmask = 1, Inv = 1, Any= 1to count stalled cycles of this core.	Set Cmask = 1, Inv = 1to count stalled cycles.
0EH	10H	UOPS_ISSUED.FLAGS_MERGE	Number of flags-merge uops allocated. Such uops adds delay.	
0EH	20H	UOPS_ISSUED.SLOW_LEA	Number of slow LEA or similar uops allocated. Such uop has 3 sources (e.g. 2 sources + immediate) regardless if as a result of LEA instruction or not.	
0EH	40H	UOPS_ISSUED.SINGLE_MUL	Number of multiply packed/scalar single precision uops allocated.	
10H	01H	FP_COMP_OPS_EXE.X87	Counts number of X87 uops executed.	
10H	10H	FP_COMP_OPS_EXE.SSE_FP_PACKED_DOUBLE	Counts number of SSE* or AVX-128 double precision FP packed uops executed.	
10H	20H	FP_COMP_OPS_EXE.SSE_FP_SCALAR_SINGLE	Counts number of SSE* or AVX-128 single precision FP scalar uops executed.	
10H	40H	FP_COMP_OPS_EXE.SSE_PACKED SINGLE	Counts number of SSE* or AVX-128 single precision FP packed uops executed.	
10H	80H	FP_COMP_OPS_EXE.SSE_SCALAR_DOUBLE	Counts number of SSE* or AVX-128 double precision FP scalar uops executed.	
11H	01H	SIMD_FP_256.PACKED_SINGLE	Counts 256-bit packed single-precision floating-point instructions.	
11H	02H	SIMD_FP_256.PACKED_DOUBLE	Counts 256-bit packed double-precision floating-point instructions.	
14H	01H	ARITH.FPU_DIV_ACTIVE	Cycles that the divider is active, includes INT and FP. Set 'edge =1, cmask=1' to count the number of divides.	

Table 19-13. Non-Architectural Performance Events In the Processor Core of 3rd Generation Intel® Core™ i7, i5, i3 Processors (Contd.)

Event Num.	Umask Value	Event Mask Mnemonic	Description	Comment
24H	01H	L2_RQSTS.DEMAND_DATA_RD_HIT	Demand Data Read requests that hit L2 cache.	
24H	03H	L2_RQSTS.ALL_DEMAND_DATA_RD	Counts any demand and L1 HW prefetch data load requests to L2.	
24H	04H	L2_RQSTS.RFO_HITS	Counts the number of store RFO requests that hit the L2 cache.	
24H	08H	L2_RQSTS.RFO_MISS	Counts the number of store RFO requests that miss the L2 cache.	
24H	0CH	L2_RQSTS.ALL_RFO	Counts all L2 store RFO requests.	
24H	10H	L2_RQSTS.CODE_RD_HIT	Number of instruction fetches that hit the L2 cache.	
24H	20H	L2_RQSTS.CODE_RD_MISS	Number of instruction fetches that missed the L2 cache.	
24H	30H	L2_RQSTS.ALL_CODE_RD	Counts all L2 code requests.	
24H	40H	L2_RQSTS.PF_HIT	Counts all L2 HW prefetcher requests that hit L2.	
24H	80H	L2_RQSTS.PF_MISS	Counts all L2 HW prefetcher requests that missed L2.	
24H	C0H	L2_RQSTS.ALL_PF	Counts all L2 HW prefetcher requests.	
27H	01H	L2_STORE_LOCK_RQSTS.MISS	RFOs that miss cache lines.	
27H	08H	L2_STORE_LOCK_RQSTS.HIT_M	RFOs that hit cache lines in M state.	
27H	0FH	L2_STORE_LOCK_RQSTS.ALL	RFOs that access cache lines in any state.	
28H	01H	L2_L1D_WB_RQSTS.MISS	Not rejected writebacks that missed LLC.	
28H	04H	L2_L1D_WB_RQSTS.HIT_E	Not rejected writebacks from L1D to L2 cache lines in E state.	
28H	08H	L2_L1D_WB_RQSTS.HIT_M	Not rejected writebacks from L1D to L2 cache lines in M state.	
28H	0FH	L2_L1D_WB_RQSTS.ALL	Not rejected writebacks from L1D to L2 cache lines in any state.	
2EH	4FH	LONGEST_LAT_CACHE.REFERENCE	This event counts requests originating from the core that reference a cache line in the last level cache.	See Table 19-1
2EH	41H	LONGEST_LAT_CACHE.MISS	This event counts each cache miss condition for references to the last level cache.	See Table 19-1
3CH	00H	CPU_CLK_UNHALTED.THREAD_P	Counts the number of thread cycles while the thread is not in a halt state. The thread enters the halt state when it is running the HLT instruction. The core frequency may change from time to time due to power or thermal throttling.	See Table 19-1.
3CH	01H	CPU_CLK_THREAD_UNHALTED.REF_XCLK	Increments at the frequency of XCLK (100 MHz) when not halted.	See Table 19-1.
48H	01H	L1D_PEND_MISS.PENDING	Increments the number of outstanding L1D misses every cycle. Set Cmask = 1 and Edge =1 to count occurrences.	PMC2 only; Set Cmask = 1 to count cycles.
49H	01H	DTLB_STORE_MISSES.MISS_CAUSES_A_WALK	Miss in all TLB levels causes a page walk of any page size (4K/2M/4M/1G).	

Table 19-13. Non-Architectural Performance Events In the Processor Core of 3rd Generation Intel® Core™ i7, i5, i3 Processors (Contd.)

Event Num.	Umask Value	Event Mask Mnemonic	Description	Comment
49H	02H	DTLB_STORE_MISSES.WALK_COMPLETED	Miss in all TLB levels causes a page walk that completes of any page size (4K/2M/4M/1G).	
49H	04H	DTLB_STORE_MISSES.WALK_DURATION	Cycles PMH is busy with this walk.	
49H	10H	DTLB_STORE_MISSES.STLB_HIT	Store operations that miss the first TLB level but hit the second and do not cause page walks.	
4CH	01H	LOAD_HIT_PRE.SW_PF	Non-SW-prefetch load dispatches that hit fill buffer allocated for S/W prefetch.	
4CH	02H	LOAD_HIT_PRE.HW_PF	Non-SW-prefetch load dispatches that hit fill buffer allocated for H/W prefetch.	
51H	01H	L1D.REPLACEMENT	Counts the number of lines brought into the L1 data cache.	
58H	04H	MOVE_ELIMINATION.INT_NOT_ELIMINATED	Number of integer Move Elimination candidate uops that were not eliminated.	
58H	08H	MOVE_ELIMINATION.SIMD_NOT_ELIMINATED	Number of SIMD Move Elimination candidate uops that were not eliminated.	
58H	01H	MOVE_ELIMINATION.INT_ELIMINATED	Number of integer Move Elimination candidate uops that were eliminated.	
58H	02H	MOVE_ELIMINATION.SIMD_ELIMINATED	Number of SIMD Move Elimination candidate uops that were eliminated.	
5CH	01H	CPL_CYCLES.RING0	Unhalted core cycles when the thread is in ring 0.	Use Edge to count transition.
5CH	02H	CPL_CYCLES.RING123	Unhalted core cycles when the thread is not in ring 0.	
5EH	01H	RS_EVENTS.EMPTY_CYCLES	Cycles the RS is empty for the thread.	
5FH	04H	DTLB_LOAD_MISSES.STLB_HIT	Counts load operations that missed 1st level DTLB but hit the 2nd level.	
60H	01H	OFFCORE_REQUESTS_OUTSTANDING.DEMAND_DATA_RD	Offcore outstanding Demand Data Read transactions in SQ to uncore. Set Cmask=1 to count cycles.	
60H	02H	OFFCORE_REQUESTS_OUTSTANDING.DEMAND_CODE_RD	Offcore outstanding Demand Code Read transactions in SQ to uncore. Set Cmask=1 to count cycles.	
60H	04H	OFFCORE_REQUESTS_OUTSTANDING.DEMAND_RFO	Offcore outstanding RFO store transactions in SQ to uncore. Set Cmask=1 to count cycles.	
60H	08H	OFFCORE_REQUESTS_OUTSTANDING.ALL_DATA_RD	Offcore outstanding cacheable data read transactions in SQ to uncore. Set Cmask=1 to count cycles.	
63H	01H	LOCK_CYCLES.SPLIT_LOCK_UC_LOCK_DURATION	Cycles in which the L1D and L2 are locked, due to a UC lock or split lock.	
63H	02H	LOCK_CYCLES.CACHE_LOCK_DURATION	Cycles in which the L1D is locked.	
79H	02H	IDQ.EMPTY	Counts cycles the IDQ is empty.	
79H	04H	IDQ.MITE_UOPS	Increment each cycle # of uops delivered to IDQ from MITE path. Set Cmask = 1 to count cycles.	Can combine Umask 04H and 20H.

Table 19-13. Non-Architectural Performance Events In the Processor Core of 3rd Generation Intel® Core™ i7, i5, i3 Processors (Contd.)

Event Num.	Umask Value	Event Mask Mnemonic	Description	Comment
79H	08H	IDQ.DSB_UOPS	Increment each cycle. # of uops delivered to IDQ from DSB path. Set Cmask = 1 to count cycles.	Can combine Umask 08H and 10H.
79H	10H	IDQ.MS_DSB_UOPS	Increment each cycle # of uops delivered to IDQ when MS_busy by DSB. Set Cmask = 1 to count cycles. Add Edge=1 to count # of delivery.	Can combine Umask 04H, 08H.
79H	20H	IDQ.MS_MITE_UOPS	Increment each cycle # of uops delivered to IDQ when MS_busy by MITE. Set Cmask = 1 to count cycles.	Can combine Umask 04H, 08H.
79H	30H	IDQ.MS_UOPS	Increment each cycle # of uops delivered to IDQ from MS by either DSB or MITE. Set Cmask = 1 to count cycles.	Can combine Umask 04H, 08H.
79H	18H	IDQ.ALL_DSB_CYCLES_ANY_UOPS	Counts cycles DSB is delivered at least one uops. Set Cmask = 1.	
79H	18H	IDQ.ALL_DSB_CYCLES_4_UOPS	Counts cycles DSB is delivered four uops. Set Cmask = 4.	
79H	24H	IDQ.ALL_MITE_CYCLES_ANY_UOPS	Counts cycles MITE is delivered at least one uops. Set Cmask = 1.	
79H	24H	IDQ.ALL_MITE_CYCLES_4_UOPS	Counts cycles MITE is delivered four uops. Set Cmask = 4.	
79H	3CH	IDQ.MITE_ALL_UOPS	# of uops delivered to IDQ from any path.	
80H	04H	ICACHE.IFETCH_STALL	Cycles where a code-fetch stalled due to L1 instruction-cache miss or an iTLB miss.	
80H	02H	ICACHE.MISSES	Number of Instruction Cache, Streaming Buffer and Victim Cache Misses. Includes UC accesses.	
85H	01H	ITLB_MISSES.MISS_CAUSES_A_WALK	Misses in all ITLB levels that cause page walks.	
85H	02H	ITLB_MISSES.WALK_COMPLETED	Misses in all ITLB levels that cause completed page walks.	
85H	04H	ITLB_MISSES.WALK_DURATION	Cycle PMH is busy with a walk.	
85H	10H	ITLB_MISSES.STLB_HIT	Number of cache load STLB hits. No page walk.	
87H	01H	ILD_STALL.LCP	Stalls caused by changing prefix length of the instruction.	
87H	04H	ILD_STALL.IQ_FULL	Stall cycles due to IQ is full.	
88H	01H	BR_INST_EXEC.COND	Qualify conditional near branch instructions executed, but not necessarily retired.	Must combine with umask 40H, 80H.
88H	02H	BR_INST_EXEC.DIRECT_JMP	Qualify all unconditional near branch instructions excluding calls and indirect branches.	Must combine with umask 80H.
88H	04H	BR_INST_EXEC.INDIRECT_JMP_NON_CALL_RET	Qualify executed indirect near branch instructions that are not calls or returns.	Must combine with umask 80H.
88H	08H	BR_INST_EXEC.RETURN_NEAR	Qualify indirect near branches that have a return mnemonic.	Must combine with umask 80H.
88H	10H	BR_INST_EXEC.DIRECT_NEAR_CALL	Qualify unconditional near call branch instructions, excluding non-call branch, executed.	Must combine with umask 80H.
88H	20H	BR_INST_EXEC.INDIRECT_NEAR_CALL	Qualify indirect near calls, including both register and memory indirect, executed.	Must combine with umask 80H.

Table 19-13. Non-Architectural Performance Events In the Processor Core of 3rd Generation Intel® Core™ i7, i5, i3 Processors (Contd.)

Event Num.	Umask Value	Event Mask Mnemonic	Description	Comment
88H	40H	BR_INST_EXEC.NONTAKEN	Qualify non-taken near branches executed.	Applicable to umask 01H only.
88H	80H	BR_INST_EXEC.TAKEN	Qualify taken near branches executed. Must combine with 01H,02H, 04H, 08H, 10H, 20H.	
88H	FFH	BR_INST_EXEC.ALL_BRANCHES	Counts all near executed branches (not necessarily retired).	
89H	01H	BR_MISP_EXEC.COND	Qualify conditional near branch instructions mispredicted.	Must combine with umask 40H, 80H.
89H	04H	BR_MISP_EXEC.INDIRECT_JMP_NON_CALL_RET	Qualify mispredicted indirect near branch instructions that are not calls or returns.	Must combine with umask 80H.
89H	08H	BR_MISP_EXEC.RETURN_NEAR	Qualify mispredicted indirect near branches that have a return mnemonic.	Must combine with umask 80H.
89H	10H	BR_MISP_EXEC.DIRECT_NEAR_CALL	Qualify mispredicted unconditional near call branch instructions, excluding non-call branch, executed.	Must combine with umask 80H.
89H	20H	BR_MISP_EXEC.INDIRECT_NEAR_CALL	Qualify mispredicted indirect near calls, including both register and memory indirect, executed.	Must combine with umask 80H.
89H	40H	BR_MISP_EXEC.NONTAKEN	Qualify mispredicted non-taken near branches executed.	Applicable to umask 01H only.
89H	80H	BR_MISP_EXEC.TAKEN	Qualify mispredicted taken near branches executed. Must combine with 01H,02H, 04H, 08H, 10H, 20H.	
89H	FFH	BR_MISP_EXEC.ALL_BRANCHES	Counts all near executed branches (not necessarily retired).	
9CH	01H	IDQ_UOPS_NOT_DELIVERED.CORE	Count issue pipeline slots where no uop was delivered from the front end to the back end when there is no back-end stall.	Use Cmask to qualify uop b/w.
A1H	01H	UOPS_DISPATCHED_PORT.PORT_0	Cycles which a Uop is dispatched on port 0.	
A1H	02H	UOPS_DISPATCHED_PORT.PORT_1	Cycles which a Uop is dispatched on port 1.	
A1H	0CH	UOPS_DISPATCHED_PORT.PORT_2	Cycles which a Uop is dispatched on port 2.	
A1H	30H	UOPS_DISPATCHED_PORT.PORT_3	Cycles which a Uop is dispatched on port 3.	
A1H	40H	UOPS_DISPATCHED_PORT.PORT_4	Cycles which a Uop is dispatched on port 4.	
A1H	80H	UOPS_DISPATCHED_PORT.PORT_5	Cycles which a Uop is dispatched on port 5.	
A2H	01H	RESOURCE_STALLS.ANY	Cycles Allocation is stalled due to Resource Related reason.	
A2H	04H	RESOURCE_STALLS.RS	Cycles stalled due to no eligible RS entry available.	
A2H	08H	RESOURCE_STALLS.SB	Cycles stalled due to no store buffers available (not including draining form sync).	
A2H	10H	RESOURCE_STALLS.ROB	Cycles stalled due to re-order buffer full.	
A3H	01H	CYCLE_ACTIVITY.CYCLES_L2_PENDING	Cycles with pending L2 miss loads. Set AnyThread to count per core.	

Table 19-13. Non-Architectural Performance Events In the Processor Core of 3rd Generation Intel® Core™ i7, i5, i3 Processors (Contd.)

Event Num.	Umask Value	Event Mask Mnemonic	Description	Comment
A3H	02H	CYCLE_ACTIVITY.CYCLES_LDM_PENDING	Cycles with pending memory loads. Set AnyThread to count per core.	Restricted to counters 0-3 when HTT is disabled.
A3H	04H	CYCLE_ACTIVITY.CYCLES_NO_EXECUTE	Cycles of dispatch stalls. Set AnyThread to count per core.	Restricted to counters 0-3 when HTT is disabled.
A3H	05H	CYCLE_ACTIVITY.STALLS_L2_PENDING	Number of loads missed L2.	Restricted to counters 0-3 when HTT is disabled.
A3H	06H	CYCLE_ACTIVITY.STALLS_LDM_PENDING		Restricted to counters 0-3 when HTT is disabled.
A3H	08H	CYCLE_ACTIVITY.CYCLES_L1D_PENDING	Cycles with pending L1 cache miss loads. Set AnyThread to count per core.	PMC2 only.
A3H	0CH	CYCLE_ACTIVITY.STALLS_L1D_PENDING	Execution stalls due to L1 data cache miss loads. Set Cmask=0CH.	PMC2 only.
A8H	01H	LSD.UOPS	Number of Uops delivered by the LSD.	
ABH	01H	DSB2MITE_SWITCHES.COUNT	Number of DSB to MITE switches.	
ABH	02H	DSB2MITE_SWITCHES.PENALTY_CYCLES	Cycles DSB to MITE switches caused delay.	
ACH	08H	DSB_FILL.EXCEED_DSB_LINES	DSB Fill encountered > 3 DSB lines.	
AEH	01H	ITLB.ITLB_FLUSH	Counts the number of ITLB flushes, includes 4k/2M/4M pages.	
B0H	01H	OFFCORE_REQUESTS.DEMAND_DATA_RD	Demand data read requests sent to uncore.	
B0H	02H	OFFCORE_REQUESTS.DEMAND_CODE_RD	Demand code read requests sent to uncore.	
B0H	04H	OFFCORE_REQUESTS.DEMAND_RFO	Demand RFO read requests sent to uncore, including regular RFOs, locks, ItoM.	
B0H	08H	OFFCORE_REQUESTS.ALL_DATA_RD	Data read requests sent to uncore (demand and prefetch).	
B1H	01H	UOPS_EXECUTED.THREAD	Counts total number of uops to be executed per-thread each cycle. Set Cmask = 1, INV =1 to count stall cycles.	
B1H	02H	UOPS_EXECUTED.CORE	Counts total number of uops to be executed per-core each cycle.	Do not need to set ANY.
B7H	01H	OFFCORE_RESPONSE_0	See Section 18.3.4.5, "Off-core Response Performance Monitoring".	Requires MSR 01A6H.
BBH	01H	OFFCORE_RESPONSE_1	See Section 18.3.4.5, "Off-core Response Performance Monitoring".	Requires MSR 01A7H.
BDH	01H	TLB_FLUSH.DTLB_THREAD	DTLB flush attempts of the thread-specific entries.	
BDH	20H	TLB_FLUSH.STLB_ANY	Count number of STLB flush attempts.	
C0H	00H	INST_RETIRED.ANY_P	Number of instructions at retirement.	See Table 19-1.
C0H	01H	INST_RETIRED.PREC_DIST	Precise instruction retired event with HW to reduce effect of PEBS shadow in IP distribution.	PMC1 only.
C1H	08H	OTHER_ASSISTS.AVX_STORE	Number of assists associated with 256-bit AVX store operations.	
C1H	10H	OTHER_ASSISTS.AVX_TO_SSE	Number of transitions from AVX-256 to legacy SSE when penalty applicable.	

Table 19-13. Non-Architectural Performance Events In the Processor Core of 3rd Generation Intel® Core™ i7, i5, i3 Processors (Contd.)

Event Num.	Umask Value	Event Mask Mnemonic	Description	Comment
C1H	20H	OTHER_ASSISTS.SSE_TO_AVX	Number of transitions from SSE to AVX-256 when penalty applicable.	
C1H	80H	OTHER_ASSISTS.WB	Number of times microcode assist is invoked by hardware upon uop writeback.	
C2H	01H	UOPS_RETIRED.ALL	Counts the number of micro-ops retired, Use cmask=1 and invert to count active cycles or stalled cycles.	Supports PEBS, use Any=1 for core granular.
C2H	02H	UOPS_RETIRED.RETIRE_SLOTS	Counts the number of retirement slots used each cycle.	Supports PEBS.
C3H	02H	MACHINE_CLEARS.MEMORY_ORDERING	Counts the number of machine clears due to memory order conflicts.	
C3H	04H	MACHINE_CLEARS.SMC	Number of self-modifying-code machine clears detected.	
C3H	20H	MACHINE_CLEARS.MASKMOV	Counts the number of executed AVX masked load operations that refer to an illegal address range with the mask bits set to 0.	
C4H	00H	BR_INST_RETIRED.ALL_BRANCHES	Branch instructions at retirement.	See Table 19-1.
C4H	01H	BR_INST_RETIRED.CONDITIONAL	Counts the number of conditional branch instructions retired.	Supports PEBS.
C4H	02H	BR_INST_RETIRED.NEAR_CALL	Direct and indirect near call instructions retired.	Supports PEBS.
C4H	04H	BR_INST_RETIRED.ALL_BRANCHES	Counts the number of branch instructions retired.	Supports PEBS.
C4H	08H	BR_INST_RETIRED.NEAR_RETURN	Counts the number of near return instructions retired.	Supports PEBS.
C4H	10H	BR_INST_RETIRED.NOT_TAKEN	Counts the number of not taken branch instructions retired.	Supports PEBS.
C4H	20H	BR_INST_RETIRED.NEAR_TAKEN	Number of near taken branches retired.	Supports PEBS.
C4H	40H	BR_INST_RETIRED.FAR_BRANCH	Number of far branches retired.	Supports PEBS.
C5H	00H	BR_MISP_RETIRED.ALL_BRANCHES	Mispredicted branch instructions at retirement.	See Table 19-1.
C5H	01H	BR_MISP_RETIRED.CONDITIONAL	Mispredicted conditional branch instructions retired.	Supports PEBS.
C5H	04H	BR_MISP_RETIRED.ALL_BRANCHES	Mispredicted macro branch instructions retired.	Supports PEBS.
C5H	20H	BR_MISP_RETIRED.NEAR_TAKEN	Mispredicted taken branch instructions retired.	Supports PEBS.
CAH	02H	FP_ASSIST.X87_OUTPUT	Number of X87 FP assists due to output values.	Supports PEBS.
CAH	04H	FP_ASSIST.X87_INPUT	Number of X87 FP assists due to input values.	Supports PEBS.
CAH	08H	FP_ASSIST.SIMD_OUTPUT	Number of SIMD FP assists due to output values.	Supports PEBS.
CAH	10H	FP_ASSIST.SIMD_INPUT	Number of SIMD FP assists due to input values.	
CAH	1EH	FP_ASSIST.ANY	Cycles with any input/output SSE* or FP assists.	
CCH	20H	ROB_MISC_EVENTS.LBR_INSERTS	Count cases of saving new LBR records by hardware.	
CDH	01H	MEM_TRANS_RETIRED.LOAD_LATENCY	Randomly sampled loads whose latency is above a user defined threshold. A small fraction of the overall loads are sampled due to randomization.	Specify threshold in MSR 3F6H. PMC 3 only.

Table 19-13. Non-Architectural Performance Events In the Processor Core of 3rd Generation Intel® Core™ i7, i5, i3 Processors (Contd.)

Event Num.	Umask Value	Event Mask Mnemonic	Description	Comment
CDH	02H	MEM_TRANS_RETIRED.PRECISE_STORE	Sample stores and collect precise store operation via PEBS record. PMC3 only.	See Section 18.3.4.4.3.
D0H	11H	MEM_UOPS_RETIRED.STLB_MISS_LOADS	Retired load uops that miss the STLB.	Supports PEBS.
D0H	12H	MEM_UOPS_RETIRED.STLB_MISS_STORES	Retired store uops that miss the STLB.	Supports PEBS.
D0H	21H	MEM_UOPS_RETIRED.LOCK_LOADS	Retired load uops with locked access.	Supports PEBS.
D0H	41H	MEM_UOPS_RETIRED.SPLIT_LOADS	Retired load uops that split across a cacheline boundary.	Supports PEBS.
D0H	42H	MEM_UOPS_RETIRED.SPLIT_STORES	Retired store uops that split across a cacheline boundary.	Supports PEBS.
D0H	81H	MEM_UOPS_RETIRED.ALL_LOADS	All retired load uops.	Supports PEBS.
D0H	82H	MEM_UOPS_RETIRED.ALL_STORES	All retired store uops.	Supports PEBS.
D1H	01H	MEM_LOAD_UOPS_RETIRED.L1_HIT	Retired load uops with L1 cache hits as data sources.	Supports PEBS.
D1H	02H	MEM_LOAD_UOPS_RETIRED.L2_HIT	Retired load uops with L2 cache hits as data sources.	Supports PEBS.
D1H	04H	MEM_LOAD_UOPS_RETIRED.LLC_HIT	Retired load uops whose data source was LLC hit with no snoop required.	Supports PEBS.
D1H	08H	MEM_LOAD_UOPS_RETIRED.L1_MISS	Retired load uops whose data source followed an L1 miss.	Supports PEBS.
D1H	10H	MEM_LOAD_UOPS_RETIRED.L2_MISS	Retired load uops that missed L2, excluding unknown sources.	Supports PEBS.
D1H	20H	MEM_LOAD_UOPS_RETIRED.LLC_MISS	Retired load uops whose data source is LLC miss.	Supports PEBS. Restricted to counters 0-3 when HTT is disabled.
D1H	40H	MEM_LOAD_UOPS_RETIRED.HIT_LFB	Retired load uops which data sources were load uops missed L1 but hit FB due to preceding miss to the same cache line with data not ready.	Supports PEBS.
D2H	01H	MEM_LOAD_UOPS_LLC_HIT_RETIRED.XSNP_MISS	Retired load uops whose data source was an on-package core cache LLC hit and cross-core snoop missed.	Supports PEBS.
D2H	02H	MEM_LOAD_UOPS_LLC_HIT_RETIRED.XSNP_HIT	Retired load uops whose data source was an on-package LLC hit and cross-core snoop hits.	Supports PEBS.
D2H	04H	MEM_LOAD_UOPS_LLC_HIT_RETIRED.XSNP_HITM	Retired load uops whose data source was an on-package core cache with HitM responses.	Supports PEBS.
D2H	08H	MEM_LOAD_UOPS_LLC_HIT_RETIRED.XSNP_NONE	Retired load uops whose data source was LLC hit with no snoop required.	Supports PEBS.
D3H	01H	MEM_LOAD_UOPS_LLC_MISS_RETIRED.LOCAL_DRAM	Retired load uops whose data source was local memory (cross-socket snoop not needed or missed).	Supports PEBS.
E6H	1FH	BACLEARS.ANY	Number of front end re-steers due to BPU misprediction.	
F0H	01H	L2_TRANS.DEMAND_DATA_RD	Demand Data Read requests that access L2 cache.	
F0H	02H	L2_TRANS.RFO	RFO requests that access L2 cache.	

Table 19-13. Non-Architectural Performance Events In the Processor Core of 3rd Generation Intel® Core™ i7, i5, i3 Processors (Contd.)

Event Num.	Umask Value	Event Mask Mnemonic	Description	Comment
F0H	04H	L2_TRANS.CODE_RD	L2 cache accesses when fetching instructions.	
F0H	08H	L2_TRANS.ALL_PF	Any MLC or LLC HW prefetch accessing L2, including rejects.	
F0H	10H	L2_TRANS.L1D_WB	L1D writebacks that access L2 cache.	
F0H	20H	L2_TRANS.L2_FILL	L2 fill requests that access L2 cache.	
F0H	40H	L2_TRANS.L2_WB	L2 writebacks that access L2 cache.	
F0H	80H	L2_TRANS.ALL_REQUESTS	Transactions accessing L2 pipe.	
F1H	01H	L2_LINES_IN.I	L2 cache lines in I state filling L2.	Counting does not cover rejects.
F1H	02H	L2_LINES_IN.S	L2 cache lines in S state filling L2.	Counting does not cover rejects.
F1H	04H	L2_LINES_IN.E	L2 cache lines in E state filling L2.	Counting does not cover rejects.
F1H	07H	L2_LINES_IN.ALL	L2 cache lines filling L2.	Counting does not cover rejects.
F2H	01H	L2_LINES_OUT.DEMAND_CLEAN	Clean L2 cache lines evicted by demand.	
F2H	02H	L2_LINES_OUT.DEMAND_DIRTY	Dirty L2 cache lines evicted by demand.	
F2H	04H	L2_LINES_OUT.PF_CLEAN	Clean L2 cache lines evicted by the MLC prefetcher.	
F2H	08H	L2_LINES_OUT.PF_DIRTY	Dirty L2 cache lines evicted by the MLC prefetcher.	
F2H	0AH	L2_LINES_OUT.DIRTY_ALL	Dirty L2 cache lines filling the L2.	Counting does not cover rejects.

19.7.1 Performance Monitoring Events in the Processor Core of Intel Xeon Processor E5 v2 Family and Intel Xeon Processor E7 v2 Family

Non-architectural performance monitoring events in the processor core that are applicable only to Intel Xeon processor E5 v2 family and Intel Xeon processor E7 v2 family based on the Ivy Bridge-E microarchitecture, with CPUID signature of DisplayFamily_DisplayModel 06_3EH, are listed in Table 19-14.

Table 19-14. Non-Architectural Performance Events Applicable Only to the Processor Core of Intel® Xeon® Processor E5 v2 Family and Intel® Xeon® Processor E7 v2 Family

Event Num.	Umask Value	Event Mask Mnemonic	Description	Comment
D3H	03H	MEM_LOAD_UOPS_LLC_MISS_RETIRED.LOCAL_DRAM	Retired load uops whose data sources were local DRAM (snoop not needed, Snoop Miss, or Snoop Hit data not forwarded).	Supports PEBS.
D3H	0CH	MEM_LOAD_UOPS_LLC_MISS_RETIRED.REMOTE_DRAM	Retired load uops whose data source was remote DRAM (snoop not needed, Snoop Miss, or Snoop Hit data not forwarded).	Supports PEBS.
D3H	10H	MEM_LOAD_UOPS_LLC_MISS_RETIRED.REMOTE_HITM	Retired load uops whose data sources were remote HITM.	Supports PEBS.
D3H	20H	MEM_LOAD_UOPS_LLC_MISS_RETIRED.REMOTE_FWD	Retired load uops whose data sources were forwards from a remote cache.	Supports PEBS.

19.8 PERFORMANCE MONITORING EVENTS FOR 2ND GENERATION INTEL® CORE™ I7-2XXX, INTEL® CORE™ I5-2XXX, INTEL® CORE™ I3-2XXX PROCESSOR SERIES

2nd generation Intel® Core™ i7-2xxx, Intel® Core™ i5-2xxx, Intel® Core™ i3-2xxx processor series, and Intel Xeon processor E3-1200 product family are based on the Intel microarchitecture code name Sandy Bridge. They support architectural performance-monitoring events listed in Table 19-1. Non-architectural performance-monitoring events in the processor core are listed in Table 19-15, Table 19-16, and Table 19-17. The events in Table 19-15 apply to processors with CPUID signature of DisplayFamily_DisplayModel encoding with the following values: 06_2AH and 06_2DH. The events in Table 19-16 apply to processors with CPUID signature 06_2AH. The events in Table 19-17 apply to processors with CPUID signature 06_2DH. Fixed counters in the core PMU support the architecture events defined in Table 19-2.

Additional information on event specifics (e.g. derivative events using specific IA32_PERFEVTSELx modifiers, limitations, special notes and recommendations) can be found at http://software.intel.com/en-us/forums/software-tuning-performance-optimization-platform-monitoring.

Table 19-15. Non-Architectural Performance Events In the Processor Core Common to 2nd Generation Intel® Core™ i7-2xxx, Intel® Core™ i5-2xxx, Intel® Core™ i3-2xxx Processor Series and Intel® Xeon® Processors E3 and E5 Family

Event Num.	Umask Value	Event Mask Mnemonic	Description	Comment
03H	01H	LD_BLOCKS.DATA_UNKNOWN	Blocked loads due to store buffer blocks with unknown data.	
03H	02H	LD_BLOCKS.STORE_FORWARD	Loads blocked by overlapping with store buffer that cannot be forwarded.	
03H	08H	LD_BLOCKS.NO_SR	# of Split loads blocked due to resource not available.	
03H	10H	LD_BLOCKS.ALL_BLOCK	Number of cases where any load is blocked but has no DCU miss.	
05H	01H	MISALIGN_MEM_REF.LOADS	Speculative cache-line split load uops dispatched to L1D.	
05H	02H	MISALIGN_MEM_REF.STORES	Speculative cache-line split Store-address uops dispatched to L1D.	
07H	01H	LD_BLOCKS_PARTIAL.ADDRESS_ALIAS	False dependencies in MOB due to partial compare on address.	
07H	08H	LD_BLOCKS_PARTIAL.ALL_STA_BLOCK	The number of times that load operations are temporarily blocked because of older stores, with addresses that are not yet known. A load operation may incur more than one block of this type.	
08H	01H	DTLB_LOAD_MISSES.MISS_CAUSES_A_WALK	Misses in all TLB levels that cause a page walk of any page size.	
08H	02H	DTLB_LOAD_MISSES.WALK_COMPLETED	Misses in all TLB levels that caused page walk completed of any size.	
08H	04H	DTLB_LOAD_MISSES.WALK_DURATION	Cycle PMH is busy with a walk.	
08H	10H	DTLB_LOAD_MISSES.STLB_HIT	Number of cache load STLB hits. No page walk.	
0DH	03H	INT_MISC.RECOVERY_CYCLES	Cycles waiting to recover after Machine Clears or JEClear. Set Cmask= 1.	Set Edge to count occurrences.
0DH	40H	INT_MISC.RAT_STALL_CYCLES	Cycles RAT external stall is sent to IDQ for this thread.	

Table 19-15. Non-Architectural Performance Events In the Processor Core Common to 2nd Generation Intel® Core™ i7-2xxx, Intel® Core™ i5-2xxx, Intel® Core™ i3-2xxx Processor Series and Intel® Xeon® Processors E3 and E5 Family

Event Num.	Umask Value	Event Mask Mnemonic	Description	Comment
0EH	01H	UOPS_ISSUED.ANY	Increments each cycle the # of Uops issued by the RAT to RS. Set Cmask = 1, Inv = 1, Any= 1to count stalled cycles of this core.	Set Cmask = 1, Inv = 1to count stalled cycles.
10H	01H	FP_COMP_OPS_EXE.X87	Counts number of X87 uops executed.	
10H	10H	FP_COMP_OPS_EXE.SSE_FP_PACKED_DOUBLE	Counts number of SSE* double precision FP packed uops executed.	
10H	20H	FP_COMP_OPS_EXE.SSE_FP_SCALAR_SINGLE	Counts number of SSE* single precision FP scalar uops executed.	
10H	40H	FP_COMP_OPS_EXE.SSE_PACKED SINGLE	Counts number of SSE* single precision FP packed uops executed.	
10H	80H	FP_COMP_OPS_EXE.SSE_SCALAR_DOUBLE	Counts number of SSE* double precision FP scalar uops executed.	
11H	01H	SIMD_FP_256.PACKED_SINGLE	Counts 256-bit packed single-precision floating-point instructions.	
11H	02H	SIMD_FP_256.PACKED_DOUBLE	Counts 256-bit packed double-precision floating-point instructions.	
14H	01H	ARITH.FPU_DIV_ACTIVE	Cycles that the divider is active, includes INT and FP. Set 'edge =1, cmask=1' to count the number of divides.	
17H	01H	INSTS_WRITTEN_TO_IQ.INSTS	Counts the number of instructions written into the IQ every cycle.	
24H	01H	L2_RQSTS.DEMAND_DATA_RD_HIT	Demand Data Read requests that hit L2 cache.	
24H	03H	L2_RQSTS.ALL_DEMAND_DATA_RD	Counts any demand and L1 HW prefetch data load requests to L2.	
24H	04H	L2_RQSTS.RFO_HITS	Counts the number of store RFO requests that hit the L2 cache.	
24H	08H	L2_RQSTS.RFO_MISS	Counts the number of store RFO requests that miss the L2 cache.	
24H	0CH	L2_RQSTS.ALL_RFO	Counts all L2 store RFO requests.	
24H	10H	L2_RQSTS.CODE_RD_HIT	Number of instruction fetches that hit the L2 cache.	
24H	20H	L2_RQSTS.CODE_RD_MISS	Number of instruction fetches that missed the L2 cache.	
24H	30H	L2_RQSTS.ALL_CODE_RD	Counts all L2 code requests.	
24H	40H	L2_RQSTS.PF_HIT	Requests from L2 Hardware prefetcher that hit L2.	
24H	80H	L2_RQSTS.PF_MISS	Requests from L2 Hardware prefetcher that missed L2.	
24H	C0H	L2_RQSTS.ALL_PF	Any requests from L2 Hardware prefetchers.	
27H	01H	L2_STORE_LOCK_RQSTS.MISS	RFOs that miss cache lines.	
27H	04H	L2_STORE_LOCK_RQSTS.HIT_E	RFOs that hit cache lines in E state.	
27H	08H	L2_STORE_LOCK_RQSTS.HIT_M	RFOs that hit cache lines in M state.	
27H	0FH	L2_STORE_LOCK_RQSTS.ALL	RFOs that access cache lines in any state.	

Table 19-15. Non-Architectural Performance Events In the Processor Core Common to 2nd Generation Intel® Core™ i7-2xxx, Intel® Core™ i5-2xxx, Intel® Core™ i3-2xxx Processor Series and Intel® Xeon® Processors E3 and E5 Family

Event Num.	Umask Value	Event Mask Mnemonic	Description	Comment
28H	01H	L2_L1D_WB_RQSTS.MISS	Not rejected writebacks from L1D to L2 cache lines that missed L2.	
28H	02H	L2_L1D_WB_RQSTS.HIT_S	Not rejected writebacks from L1D to L2 cache lines in S state.	
28H	04H	L2_L1D_WB_RQSTS.HIT_E	Not rejected writebacks from L1D to L2 cache lines in E state.	
28H	08H	L2_L1D_WB_RQSTS.HIT_M	Not rejected writebacks from L1D to L2 cache lines in M state.	
28H	0FH	L2_L1D_WB_RQSTS.ALL	Not rejected writebacks from L1D to L2 cache.	
2EH	4FH	LONGEST_LAT_CACHE.REFERENCE	This event counts requests originating from the core that reference a cache line in the last level cache.	See Table 19-1.
2EH	41H	LONGEST_LAT_CACHE.MISS	This event counts each cache miss condition for references to the last level cache.	See Table 19-1.
3CH	00H	CPU_CLK_UNHALTED.THREAD_P	Counts the number of thread cycles while the thread is not in a halt state. The thread enters the halt state when it is running the HLT instruction. The core frequency may change from time to time due to power or thermal throttling.	See Table 19-1.
3CH	01H	CPU_CLK_THREAD_UNHALTED.REF_XCLK	Increments at the frequency of XCLK (100 MHz) when not halted.	See Table 19-1.
48H	01H	L1D_PEND_MISS.PENDING	Increments the number of outstanding L1D misses every cycle. Set Cmask = 1 and Edge =1 to count occurrences.	PMC2 only; Set Cmask = 1 to count cycles.
49H	01H	DTLB_STORE_MISSES.MISS_CAUSES_A_WALK	Miss in all TLB levels causes a page walk of any page size (4K/2M/4M/1G).	
49H	02H	DTLB_STORE_MISSES.WALK_COMPLETED	Miss in all TLB levels causes a page walk that completes of any page size (4K/2M/4M/1G).	
49H	04H	DTLB_STORE_MISSES.WALK_DURATION	Cycles PMH is busy with this walk.	
49H	10H	DTLB_STORE_MISSES.STLB_HIT	Store operations that miss the first TLB level but hit the second and do not cause page walks.	
4CH	01H	LOAD_HIT_PRE.SW_PF	Not SW-prefetch load dispatches that hit fill buffer allocated for S/W prefetch.	
4CH	02H	LOAD_HIT_PRE.HW_PF	Not SW-prefetch load dispatches that hit fill buffer allocated for H/W prefetch.	
4EH	02H	HW_PRE_REQ.DL1_MISS	Hardware Prefetch requests that miss the L1D cache. A request is being counted each time it access the cache & miss it, including if a block is applicable or if hit the Fill Buffer for example.	This accounts for both L1 streamer and IP-based (IPP) HW prefetchers.
51H	01H	L1D.REPLACEMENT	Counts the number of lines brought into the L1 data cache.	
51H	02H	L1D.ALLOCATED_IN_M	Counts the number of allocations of modified L1D cache lines.	
51H	04H	L1D.EVICTION	Counts the number of modified lines evicted from the L1 data cache due to replacement.	

Table 19-15. Non-Architectural Performance Events In the Processor Core Common to 2nd Generation Intel® Core™ i7-2xxx, Intel® Core™ i5-2xxx, Intel® Core™ i3-2xxx Processor Series and Intel® Xeon® Processors E3 and E5 Family

Event Num.	Umask Value	Event Mask Mnemonic	Description	Comment
51H	08H	L1D.ALL_M_REPLACEMENT	Cache lines in M state evicted out of L1D due to Snoop HitM or dirty line replacement.	
59H	20H	PARTIAL_RAT_STALLS.FLAGS_MERGE_UOP	Increments the number of flags-merge uops in flight each cycle. Set Cmask = 1 to count cycles.	
59H	40H	PARTIAL_RAT_STALLS.SLOW_LEA_WINDOW	Cycles with at least one slow LEA uop allocated.	
59H	80H	PARTIAL_RAT_STALLS.MUL_SINGLE_UOP	Number of Multiply packed/scalar single precision uops allocated.	
5BH	0CH	RESOURCE_STALLS2.ALL_FL_EMPTY	Cycles stalled due to free list empty.	PMC0-3 only regardless HTT.
5BH	0FH	RESOURCE_STALLS2.ALL_PRF_CONTROL	Cycles stalled due to control structures full for physical registers.	
5BH	40H	RESOURCE_STALLS2.BOB_FULL	Cycles Allocator is stalled due Branch Order Buffer.	
5BH	4FH	RESOURCE_STALLS2.OOO_RSRC	Cycles stalled due to out of order resources full.	
5CH	01H	CPL_CYCLES.RING0	Unhalted core cycles when the thread is in ring 0.	Use Edge to count transition.
5CH	02H	CPL_CYCLES.RING123	Unhalted core cycles when the thread is not in ring 0.	
5EH	01H	RS_EVENTS.EMPTY_CYCLES	Cycles the RS is empty for the thread.	
60H	01H	OFFCORE_REQUESTS_OUTSTANDING.DEMAND_DATA_RD	Offcore outstanding Demand Data Read transactions in SQ to uncore. Set Cmask=1 to count cycles.	
60H	04H	OFFCORE_REQUESTS_OUTSTANDING.DEMAND_RFO	Offcore outstanding RFO store transactions in SQ to uncore. Set Cmask=1 to count cycles.	
60H	08H	OFFCORE_REQUESTS_OUTSTANDING.ALL_DATA_RD	Offcore outstanding cacheable data read transactions in SQ to uncore. Set Cmask=1 to count cycles.	
63H	01H	LOCK_CYCLES.SPLIT_LOCK_UC_LOCK_DURATION	Cycles in which the L1D and L2 are locked, due to a UC lock or split lock.	
63H	02H	LOCK_CYCLES.CACHE_LOCK_DURATION	Cycles in which the L1D is locked.	
79H	02H	IDQ.EMPTY	Counts cycles the IDQ is empty.	
79H	04H	IDQ.MITE_UOPS	Increment each cycle # of uops delivered to IDQ from MITE path. Set Cmask = 1 to count cycles.	Can combine Umask 04H and 20H.
79H	08H	IDQ.DSB_UOPS	Increment each cycle. # of uops delivered to IDQ from DSB path. Set Cmask = 1 to count cycles.	Can combine Umask 08H and 10H.
79H	10H	IDQ.MS_DSB_UOPS	Increment each cycle # of uops delivered to IDQ when MS busy by DSB. Set Cmask = 1 to count cycles MS is busy. Set Cmask=1 and Edge =1 to count MS activations.	Can combine Umask 08H and 10H.
79H	20H	IDQ.MS_MITE_UOPS	Increment each cycle # of uops delivered to IDQ when MS is busy by MITE. Set Cmask = 1 to count cycles.	Can combine Umask 04H and 20H.

Table 19-15. Non-Architectural Performance Events In the Processor Core Common to 2nd Generation Intel® Core™ i7-2xxx, Intel® Core™ i5-2xxx, Intel® Core™ i3-2xxx Processor Series and Intel® Xeon® Processors E3 and E5 Family

Event Num.	Umask Value	Event Mask Mnemonic	Description	Comment
79H	30H	IDQ.MS_UOPS	Increment each cycle # of uops delivered to IDQ from MS by either DSB or MITE. Set Cmask = 1 to count cycles.	Can combine Umask 04H, 08H and 30H.
80H	02H	ICACHE.MISSES	Number of Instruction Cache, Streaming Buffer and Victim Cache Misses. Includes UC accesses.	
85H	01H	ITLB_MISSES.MISS_CAUSES_A_WALK	Misses in all ITLB levels that cause page walks.	
85H	02H	ITLB_MISSES.WALK_COMPLETED	Misses in all ITLB levels that cause completed page walks.	
85H	04H	ITLB_MISSES.WALK_DURATION	Cycle PMH is busy with a walk.	
85H	10H	ITLB_MISSES.STLB_HIT	Number of cache load STLB hits. No page walk.	
87H	01H	ILD_STALL.LCP	Stalls caused by changing prefix length of the instruction.	
87H	04H	ILD_STALL.IQ_FULL	Stall cycles due to IQ is full.	
88H	41H	BR_INST_EXEC.NONTAKEN_CONDITIONAL	Not-taken macro conditional branches.	
88H	81H	BR_INST_EXEC.TAKEN_CONDITIONAL	Taken speculative and retired conditional branches.	
88H	82H	BR_INST_EXEC.TAKEN_DIRECT_JUMP	Taken speculative and retired conditional branches excluding calls and indirects.	
88H	84H	BR_INST_EXEC.TAKEN_INDIRECT_JUMP_NON_CALL_RET	Taken speculative and retired indirect branches excluding calls and returns.	
88H	88H	BR_INST_EXEC.TAKEN_INDIRECT_NEAR_RETURN	Taken speculative and retired indirect branches that are returns.	
88H	90H	BR_INST_EXEC.TAKEN_DIRECT_NEAR_CALL	Taken speculative and retired direct near calls.	
88H	A0H	BR_INST_EXEC.TAKEN_INDIRECT_NEAR_CALL	Taken speculative and retired indirect near calls.	
88H	C1H	BR_INST_EXEC.ALL_CONDITIONAL	Speculative and retired conditional branches.	
88H	C2H	BR_INST_EXEC.ALL_DIRECT_JUMP	Speculative and retired conditional branches excluding calls and indirects.	
88H	C4H	BR_INST_EXEC.ALL_INDIRECT_JUMP_NON_CALL_RET	Speculative and retired indirect branches excluding calls and returns.	
88H	C8H	BR_INST_EXEC.ALL_INDIRECT_NEAR_RETURN	Speculative and retired indirect branches that are returns.	
88H	D0H	BR_INST_EXEC.ALL_NEAR_CALL	Speculative and retired direct near calls.	
88H	FFH	BR_INST_EXEC.ALL_BRANCHES	Speculative and retired branches.	
89H	41H	BR_MISP_EXEC.NONTAKEN_CONDITIONAL	Not-taken mispredicted macro conditional branches.	
89H	81H	BR_MISP_EXEC.TAKEN_CONDITIONAL	Taken speculative and retired mispredicted conditional branches.	

Table 19-15. Non-Architectural Performance Events In the Processor Core Common to 2nd Generation Intel® Core™ i7-2xxx, Intel® Core™ i5-2xxx, Intel® Core™ i3-2xxx Processor Series and Intel® Xeon® Processors E3 and E5 Family

Event Num.	Umask Value	Event Mask Mnemonic	Description	Comment
89H	84H	BR_MISP_EXEC.TAKEN_INDIRECT_JUMP_NON_CALL_RET	Taken speculative and retired mispredicted indirect branches excluding calls and returns.	
89H	88H	BR_MISP_EXEC.TAKEN_RETURN_NEAR	Taken speculative and retired mispredicted indirect branches that are returns.	
89H	90H	BR_MISP_EXEC.TAKEN_DIRECT_NEAR_CALL	Taken speculative and retired mispredicted direct near calls.	
89H	A0H	BR_MISP_EXEC.TAKEN_INDIRECT_NEAR_CALL	Taken speculative and retired mispredicted indirect near calls.	
89H	C1H	BR_MISP_EXEC.ALL_CONDITIONAL	Speculative and retired mispredicted conditional branches.	
89H	C4H	BR_MISP_EXEC.ALL_INDIRECT_JUMP_NON_CALL_RET	Speculative and retired mispredicted indirect branches excluding calls and returns.	
89H	D0H	BR_MISP_EXEC.ALL_NEAR_CALL	Speculative and retired mispredicted direct near calls.	
89H	FFH	BR_MISP_EXEC.ALL_BRANCHES	Speculative and retired mispredicted branches.	
9CH	01H	IDQ_UOPS_NOT_DELIVERED.CORE	Count issue pipeline slots where no uop was delivered from the front end to the back end when there is no back-end stall.	Use Cmask to qualify uop b/w.
A1H	01H	UOPS_DISPATCHED_PORT.PORT_0	Cycles which a Uop is dispatched on port 0.	
A1H	02H	UOPS_DISPATCHED_PORT.PORT_1	Cycles which a Uop is dispatched on port 1.	
A1H	0CH	UOPS_DISPATCHED_PORT.PORT_2	Cycles which a Uop is dispatched on port 2.	
A1H	30H	UOPS_DISPATCHED_PORT.PORT_3	Cycles which a Uop is dispatched on port 3.	
A1H	40H	UOPS_DISPATCHED_PORT.PORT_4	Cycles which a Uop is dispatched on port 4.	
A1H	80H	UOPS_DISPATCHED_PORT.PORT_5	Cycles which a Uop is dispatched on port 5.	
A2H	01H	RESOURCE_STALLS.ANY	Cycles Allocation is stalled due to Resource Related reason.	
A2H	02H	RESOURCE_STALLS.LB	Counts the cycles of stall due to lack of load buffers.	
A2H	04H	RESOURCE_STALLS.RS	Cycles stalled due to no eligible RS entry available.	
A2H	08H	RESOURCE_STALLS.SB	Cycles stalled due to no store buffers available (not including draining form sync).	
A2H	10H	RESOURCE_STALLS.ROB	Cycles stalled due to re-order buffer full.	
A2H	20H	RESOURCE_STALLS.FCSW	Cycles stalled due to writing the FPU control word.	
A3H	01H	CYCLE_ACTIVITY.CYCLES_L2_PENDING	Cycles with pending L2 miss loads. Set AnyThread to count per core.	
A3H	02H	CYCLE_ACTIVITY.CYCLES_L1D_PENDING	Cycles with pending L1 cache miss loads. Set AnyThread to count per core.	PMC2 only.
A3H	04H	CYCLE_ACTIVITY.CYCLES_NO_DISPATCH	Cycles of dispatch stalls. Set AnyThread to count per core.	PMC0-3 only.

Table 19-15. Non-Architectural Performance Events In the Processor Core Common to 2nd Generation Intel® Core™ i7-2xxx, Intel® Core™ i5-2xxx, Intel® Core™ i3-2xxx Processor Series and Intel® Xeon® Processors E3 and E5 Family

Event Num.	Umask Value	Event Mask Mnemonic	Description	Comment
A3H	05H	CYCLE_ACTIVITY.STALL_CYCLES_L2_PENDING		PMC0-3 only.
A3H	06H	CYCLE_ACTIVITY.STALL_CYCLES_L1D_PENDING		PMC2 only.
A8H	01H	LSD.UOPS	Number of Uops delivered by the LSD.	
ABH	01H	DSB2MITE_SWITCHES.COUNT	Number of DSB to MITE switches.	
ABH	02H	DSB2MITE_SWITCHES.PENALTY_CYCLES	Cycles DSB to MITE switches caused delay.	
ACH	02H	DSB_FILL.OTHER_CANCEL	Cases of cancelling valid DSB fill not because of exceeding way limit.	
ACH	08H	DSB_FILL.EXCEED_DSB_LINES	DSB Fill encountered > 3 DSB lines.	
AEH	01H	ITLB.ITLB_FLUSH	Counts the number of ITLB flushes; includes 4k/2M/4M pages.	
B0H	01H	OFFCORE_REQUESTS.DEMAND_DATA_RD	Demand data read requests sent to uncore.	
B0H	04H	OFFCORE_REQUESTS.DEMAND_RFO	Demand RFO read requests sent to uncore, including regular RFOs, locks, ItoM.	
B0H	08H	OFFCORE_REQUESTS.ALL_DATA_RD	Data read requests sent to uncore (demand and prefetch).	
B1H	01H	UOPS_DISPATCHED.THREAD	Counts total number of uops to be dispatched per-thread each cycle. Set Cmask = 1, INV =1 to count stall cycles.	PMC0-3 only regardless HTT.
B1H	02H	UOPS_DISPATCHED.CORE	Counts total number of uops to be dispatched per-core each cycle.	Do not need to set ANY.
B2H	01H	OFFCORE_REQUESTS_BUFFER.SQ_FULL	Offcore requests buffer cannot take more entries for this thread core.	
B6H	01H	AGU_BYPASS_CANCEL.COUNT	Counts executed load operations with all the following traits: 1. Addressing of the format [base + offset], 2. The offset is between 1 and 2047, 3. The address specified in the base register is in one page and the address [base+offset] is in another page.	
B7H	01H	OFF_CORE_RESPONSE_0	See Section 18.3.4.5, "Off-core Response Performance Monitoring".	Requires MSR 01A6H.
BBH	01H	OFF_CORE_RESPONSE_1	See Section 18.3.4.5, "Off-core Response Performance Monitoring".	Requires MSR 01A7H.
BDH	01H	TLB_FLUSH.DTLB_THREAD	DTLB flush attempts of the thread-specific entries.	
BDH	20H	TLB_FLUSH.STLB_ANY	Count number of STLB flush attempts.	
BFH	05H	L1D_BLOCKS.BANK_CONFLICT_CYCLES	Cycles when dispatched loads are cancelled due to L1D bank conflicts with other load ports.	Cmask=1.
C0H	00H	INST_RETIRED.ANY_P	Number of instructions at retirement.	See Table 19-1.
C0H	01H	INST_RETIRED.PREC_DIST	Precise instruction retired event with HW to reduce effect of PEBS shadow in IP distribution.	PMC1 only; must quiesce other PMCs.
C1H	02H	OTHER_ASSISTS.ITLB_MISS_RETIRED	Instructions that experienced an ITLB miss.	

Table 19-15. Non-Architectural Performance Events In the Processor Core Common to 2nd Generation Intel® Core™ i7-2xxx, Intel® Core™ i5-2xxx, Intel® Core™ i3-2xxx Processor Series and Intel® Xeon® Processors E3 and E5 Family

Event Num.	Umask Value	Event Mask Mnemonic	Description	Comment
C1H	08H	OTHER_ASSISTS.AVX_STORE	Number of assists associated with 256-bit AVX store operations.	
C1H	10H	OTHER_ASSISTS.AVX_TO_SSE	Number of transitions from AVX-256 to legacy SSE when penalty applicable.	
C1H	20H	OTHER_ASSISTS.SSE_TO_AVX	Number of transitions from SSE to AVX-256 when penalty applicable.	
C2H	01H	UOPS_RETIRED.ALL	Counts the number of micro-ops retired, Use cmask=1 and invert to count active cycles or stalled cycles.	Supports PEBS.
C2H	02H	UOPS_RETIRED.RETIRE_SLOTS	Counts the number of retirement slots used each cycle.	Supports PEBS.
C3H	02H	MACHINE_CLEARS.MEMORY_ORDERING	Counts the number of machine clears due to memory order conflicts.	
C3H	04H	MACHINE_CLEARS.SMC	Counts the number of times that a program writes to a code section.	
C3H	20H	MACHINE_CLEARS.MASKMOV	Counts the number of executed AVX masked load operations that refer to an illegal address range with the mask bits set to 0.	
C4H	00H	BR_INST_RETIRED.ALL_BRANCHES	Branch instructions at retirement.	See Table 19-1.
C4H	01H	BR_INST_RETIRED.CONDITIONAL	Counts the number of conditional branch instructions retired.	Supports PEBS.
C4H	02H	BR_INST_RETIRED.NEAR_CALL	Direct and indirect near call instructions retired.	Supports PEBS.
C4H	04H	BR_INST_RETIRED.ALL_BRANCHES	Counts the number of branch instructions retired.	Supports PEBS.
C4H	08H	BR_INST_RETIRED.NEAR_RETURN	Counts the number of near return instructions retired.	Supports PEBS.
C4H	10H	BR_INST_RETIRED.NOT_TAKEN	Counts the number of not taken branch instructions retired.	
C4H	20H	BR_INST_RETIRED.NEAR_TAKEN	Number of near taken branches retired.	Supports PEBS.
C4H	40H	BR_INST_RETIRED.FAR_BRANCH	Number of far branches retired.	
C5H	00H	BR_MISP_RETIRED.ALL_BRANCHES	Mispredicted branch instructions at retirement.	See Table 19-1.
C5H	01H	BR_MISP_RETIRED.CONDITIONAL	Mispredicted conditional branch instructions retired.	Supports PEBS.
C5H	02H	BR_MISP_RETIRED.NEAR_CALL	Direct and indirect mispredicted near call instructions retired.	Supports PEBS.
C5H	04H	BR_MISP_RETIRED.ALL_BRANCHES	Mispredicted macro branch instructions retired.	Supports PEBS.
C5H	10H	BR_MISP_RETIRED.NOT_TAKEN	Mispredicted not taken branch instructions retired.	Supports PEBS.
C5H	20H	BR_MISP_RETIRED.TAKEN	Mispredicted taken branch instructions retired.	Supports PEBS.
CAH	02H	FP_ASSIST.X87_OUTPUT	Number of X87 assists due to output value.	

Table 19-15. Non-Architectural Performance Events In the Processor Core Common to 2nd Generation Intel® Core™ i7-2xxx, Intel® Core™ i5-2xxx, Intel® Core™ i3-2xxx Processor Series and Intel® Xeon® Processors E3 and E5 Family

Event Num.	Umask Value	Event Mask Mnemonic	Description	Comment
CAH	04H	FP_ASSIST.X87_INPUT	Number of X87 assists due to input value.	
CAH	08H	FP_ASSIST.SIMD_OUTPUT	Number of SIMD FP assists due to output values.	
CAH	10H	FP_ASSIST.SIMD_INPUT	Number of SIMD FP assists due to input values.	
CAH	1EH	FP_ASSIST.ANY	Cycles with any input/output SSE* or FP assists.	
CCH	20H	ROB_MISC_EVENTS.LBR_INSERTS	Count cases of saving new LBR records by hardware.	
CDH	01H	MEM_TRANS_RETIRED.LOAD_LATENCY	Randomly sampled loads whose latency is above a user defined threshold. A small fraction of the overall loads are sampled due to randomization. PMC3 only.	Specify threshold in MSR 3F6H.
CDH	02H	MEM_TRANS_RETIRED.PRECISE_STORE	Sample stores and collect precise store operation via PEBS record. PMC3 only.	See Section 18.3.4.4.3.
D0H	11H	MEM_UOPS_RETIRED.STLB_MISS_LOADS	Retired load uops that miss the STLB.	Supports PEBS. PMC0-3 only regardless HTT.
D0H	12H	MEM_UOPS_RETIRED.STLB_MISS_STORES	Retired store uops that miss the STLB.	Supports PEBS. PMC0-3 only regardless HTT.
D0H	21H	MEM_UOPS_RETIRED.LOCK_LOADS	Retired load uops with locked access.	Supports PEBS. PMC0-3 only regardless HTT.
D0H	41H	MEM_UOPS_RETIRED.SPLIT_LOADS	Retired load uops that split across a cacheline boundary.	Supports PEBS. PMC0-3 only regardless HTT.
D0H	42H	MEM_UOPS_RETIRED.SPLIT_STORES	Retired store uops that split across a cacheline boundary.	Supports PEBS. PMC0-3 only regardless HTT.
D0H	81H	MEM_UOPS_RETIRED.ALL_LOADS	All retired load uops.	Supports PEBS. PMC0-3 only regardless HTT.
D0H	82H	MEM_UOPS_RETIRED.ALL_STORES	All retired store uops.	Supports PEBS. PMC0-3 only regardless HTT.
D1H	01H	MEM_LOAD_UOPS_RETIRED.L1_HIT	Retired load uops with L1 cache hits as data sources.	Supports PEBS. PMC0-3 only regardless HTT.
D1H	02H	MEM_LOAD_UOPS_RETIRED.L2_HIT	Retired load uops with L2 cache hits as data sources.	Supports PEBS.
D1H	04H	MEM_LOAD_UOPS_RETIRED.LLC_HIT	Retired load uops which data sources were data hits in LLC without snoops required.	Supports PEBS.
D1H	20H	MEM_LOAD_UOPS_RETIRED.LLC_MISS	Retired load uops which data sources were data missed LLC (excluding unknown data source).	Supports PEBS.
D1H	40H	MEM_LOAD_UOPS_RETIRED.HIT_LFB	Retired load uops which data sources were load uops missed L1 but hit FB due to preceding miss to the same cache line with data not ready.	Supports PEBS.
D2H	01H	MEM_LOAD_UOPS_LLC_HIT_RETIRED.XSNP_MISS	Retired load uops whose data source was an on-package core cache LLC hit and cross-core snoop missed.	Supports PEBS.
D2H	02H	MEM_LOAD_UOPS_LLC_HIT_RETIRED.XSNP_HIT	Retired load uops whose data source was an on-package LLC hit and cross-core snoop hits.	Supports PEBS.
D2H	04H	MEM_LOAD_UOPS_LLC_HIT_RETIRED.XSNP_HITM	Retired load uops whose data source was an on-package core cache with HitM responses.	Supports PEBS.
D2H	08H	MEM_LOAD_UOPS_LLC_HIT_RETIRED.XSNP_NONE	Retired load uops whose data source was LLC hit with no snoop required.	Supports PEBS.

Table 19-15. Non-Architectural Performance Events In the Processor Core Common to 2nd Generation Intel® Core™ i7-2xxx, Intel® Core™ i5-2xxx, Intel® Core™ i3-2xxx Processor Series and Intel® Xeon® Processors E3 and E5 Family

Event Num.	Umask Value	Event Mask Mnemonic	Description	Comment
E6H	01H	BACLEARS.ANY	Counts the number of times the front end is re-steered, mainly when the BPU cannot provide a correct prediction and this is corrected by other branch handling mechanisms at the front end.	
F0H	01H	L2_TRANS.DEMAND_DATA_RD	Demand Data Read requests that access L2 cache.	
F0H	02H	L2_TRANS.RFO	RFO requests that access L2 cache.	
F0H	04H	L2_TRANS.CODE_RD	L2 cache accesses when fetching instructions.	
F0H	08H	L2_TRANS.ALL_PF	L2 or LLC HW prefetches that access L2 cache.	Including rejects.
F0H	10H	L2_TRANS.L1D_WB	L1D writebacks that access L2 cache.	
F0H	20H	L2_TRANS.L2_FILL	L2 fill requests that access L2 cache.	
F0H	40H	L2_TRANS.L2_WB	L2 writebacks that access L2 cache.	
F0H	80H	L2_TRANS.ALL_REQUESTS	Transactions accessing L2 pipe.	
F1H	01H	L2_LINES_IN.I	L2 cache lines in I state filling L2.	Counting does not cover rejects.
F1H	02H	L2_LINES_IN.S	L2 cache lines in S state filling L2.	Counting does not cover rejects.
F1H	04H	L2_LINES_IN.E	L2 cache lines in E state filling L2.	Counting does not cover rejects.
F1H	07H	L2_LINES_IN.ALL	L2 cache lines filling L2.	Counting does not cover rejects.
F2H	01H	L2_LINES_OUT.DEMAND_CLEAN	Clean L2 cache lines evicted by demand.	
F2H	02H	L2_LINES_OUT.DEMAND_DIRTY	Dirty L2 cache lines evicted by demand.	
F2H	04H	L2_LINES_OUT.PF_CLEAN	Clean L2 cache lines evicted by L2 prefetch.	
F2H	08H	L2_LINES_OUT.PF_DIRTY	Dirty L2 cache lines evicted by L2 prefetch.	
F2H	0AH	L2_LINES_OUT.DIRTY_ALL	Dirty L2 cache lines filling the L2.	Counting does not cover rejects.
F4H	10H	SQ_MISC.SPLIT_LOCK	Split locks in SQ.	

Non-architecture performance monitoring events in the processor core that are applicable only to Intel processors with CPUID signature of DisplayFamily_DisplayModel 06_2AH are listed in Table 19-16.

Table 19-16. Non-Architectural Performance Events applicable only to the Processor core for 2nd Generation Intel® Core™ i7-2xxx, Intel® Core™ i5-2xxx, Intel® Core™ i3-2xxx Processor Series

Event Num.	Umask Value	Event Mask Mnemonic	Description	Comment
D2H	01H	MEM_LOAD_UOPS_LLC_HIT_RETIRED.XSNP_MISS	Retired load uops which data sources were LLC hit and cross-core snoop missed in on-pkg core cache.	Supports PEBS. PMC0-3 only regardless HTT.
D2H	02H	MEM_LOAD_UOPS_LLC_HIT_RETIRED.XSNP_HIT	Retired load uops which data sources were LLC and cross-core snoop hits in on-pkg core cache.	Supports PEBS.
D2H	04H	MEM_LOAD_UOPS_LLC_HIT_RETIRED.XSNP_HITM	Retired load uops which data sources were HitM responses from shared LLC.	Supports PEBS.
D2H	08H	MEM_LOAD_UOPS_LLC_HIT_RETIRED.XSNP_NONE	Retired load uops which data sources were hits in LLC without snoops required.	Supports PEBS.

Table 19-16. Non-Architectural Performance Events applicable only to the Processor core for 2nd Generation Intel® Core™ i7-2xxx, Intel® Core™ i5-2xxx, Intel® Core™ i3-2xxx Processor Series (Contd.)

Event Num.	Umask Value	Event Mask Mnemonic	Description	Comment
D4H	02H	MEM_LOAD_UOPS_MISC_RETIRED.LLC_MISS	Retired load uops with unknown information as data source in cache serviced the load.	Supports PEBS. PMC0-3 only regardless HTT.
B7H/BBH	01H	OFF_CORE_RESPONSE_N	Sub-events of OFF_CORE_RESPONSE_N (suffix N = 0, 1) programmed using MSR 01A6H/01A7H with values shown in the comment column.	
		OFFCORE_RESPONSE.ALL_CODE_RD.LLC_HIT_N		10003C0244H
		OFFCORE_RESPONSE.ALL_CODE_RD.LLC_HIT.NO_SNOOP_NEEDED_N		1003C0244H
		OFFCORE_RESPONSE.ALL_CODE_RD.LLC_HIT.SNOOP_MISS_N		2003C0244H
		OFFCORE_RESPONSE.ALL_CODE_RD.LLC_HIT.MISS_DRAM_N		300400244H
		OFFCORE_RESPONSE.ALL_DATA_RD.LLC_HIT.ANY_RESPONSE_N		3F803C0091H
		OFFCORE_RESPONSE.ALL_DATA_RD.LLC_MISS.DRAM_N		300400091H
		OFFCORE_RESPONSE.ALL_PF_CODE_RD.LLC_HIT.ANY_RESPONSE_N		3F803C0240H
		OFFCORE_RESPONSE.ALL_PF_CODE_RD.LLC_HIT.HIT_OTHER_CORE_NO_FWD_N		4003C0240H
		OFFCORE_RESPONSE.ALL_PF_CODE_RD.LLC_HIT.HITM_OTHER_CORE_N		10003C0240H
		OFFCORE_RESPONSE.ALL_PF_CODE_RD.LLC_HIT.NO_SNOOP_NEEDED_N		1003C0240H
		OFFCORE_RESPONSE.ALL_PF_CODE_RD.LLC_HIT.SNOOP_MISS_N		2003C0240H
		OFFCORE_RESPONSE.ALL_PF_CODE_RD.LLC_MISS.DRAM_N		300400240H
		OFFCORE_RESPONSE.ALL_PF_DATA_RD.LLC_MISS.DRAM_N		300400090H
		OFFCORE_RESPONSE.ALL_PF_RFO.LLC_HIT.ANY_RESPONSE_N		3F803C0120H
		OFFCORE_RESPONSE.ALL_PF_RFO.LLC_HIT.HIT_OTHER_CORE_NO_FWD_N		4003C0120H
		OFFCORE_RESPONSE.ALL_PF_RFO.LLC_HIT.HITM_OTHER_CORE_N		10003C0120H
		OFFCORE_RESPONSE.ALL_PF_RfO.LLC_HIT.NO_SNOOP_NEEDED_N		1003C0120H
		OFFCORE_RESPONSE.ALL_PF_RFO.LLC_HIT.SNOOP_MISS_N		2003C0120H
		OFFCORE_RESPONSE.ALL_PF_RFO.LLC_MISS.DRAM_N		300400120H
		OFFCORE_RESPONSE.ALL_READS.LLC_MISS.DRAM_N		3004003F7H
		OFFCORE_RESPONSE.ALL_RFO.LLC_HIT.ANY_RESPONSE_N		3F803C0122H
		OFFCORE_RESPONSE.ALL_RFO.LLC_HIT.HIT_OTHER_CORE_NO_FWD_N		4003C0122H
		OFFCORE_RESPONSE.ALL_RFO.LLC_HIT.HITM_OTHER_CORE_N		10003C0122H
		OFFCORE_RESPONSE.ALL_RFO.LLC_HIT.NO_SNOOP_NEEDED_N		1003C0122H
		OFFCORE_RESPONSE.ALL_RFO.LLC_HIT.SNOOP_MISS_N		2003C0122H
		OFFCORE_RESPONSE.ALL_RFO.LLC_MISS.DRAM_N		300400122H
		OFFCORE_RESPONSE.DEMAND_CODE_RD.LLC_HIT.HIT_OTHER_CORE_NO_FWD_N		4003C0004H
		OFFCORE_RESPONSE.DEMAND_CODE_RD.LLC_HIT.HITM_OTHER_CORE_N		10003C0004H
		OFFCORE_RESPONSE.DEMAND_CODE_RD.LLC_HIT.NO_SNOOP_NEEDED_N		1003C0004H
		OFFCORE_RESPONSE.DEMAND_CODE_RD.LLC_HIT.SNOOP_MISS_N		2003C0004H
		OFFCORE_RESPONSE.DEMAND_CODE_RD.LLC_MISS.DRAM_N		300400004H
		OFFCORE_RESPONSE.DEMAND_DATA_RD.LLC_MISS.DRAM_N		300400001H
		OFFCORE_RESPONSE.DEMAND_RFO.LLC_HIT.ANY_RESPONSE_N		3F803C0002H
		OFFCORE_RESPONSE.DEMAND_RFO.LLC_HIT.HIT_OTHER_CORE_NO_FWD_N		4003C0002H
		OFFCORE_RESPONSE.DEMAND_RFO.LLC_HIT.HITM_OTHER_CORE_N		10003C0002H

Table 19-16. Non-Architectural Performance Events applicable only to the Processor core for 2nd Generation Intel® Core™ i7-2xxx, Intel® Core™ i5-2xxx, Intel® Core™ i3-2xxx Processor Series (Contd.)

Event Num.	Umask Value	Event Mask Mnemonic	Description	Comment
		OFFCORE_RESPONSE.DEMAND_RFO.LLC_HIT.NO_SNOOP_NEEDED_N		1003C0002H
		OFFCORE_RESPONSE.DEMAND_RFO.LLC_HIT.SNOOP_MISS_N		2003C0002H
		OFFCORE_RESPONSE.DEMAND_RFO.LLC_MISS.DRAM_N		300400002H
		OFFCORE_RESPONSE.OTHER.ANY_RESPONSE_N		18000H
		OFFCORE_RESPONSE.PF_L2_CODE_RD.LLC_HIT.HIT_OTHER_CORE_NO_FWD_N		4003C0040H
		OFFCORE_RESPONSE.PF_L2_CODE_RD.LLC_HIT.HITM_OTHER_CORE_N		10003C0040H
		OFFCORE_RESPONSE.PF_L2_CODE_RD.LLC_HIT.NO_SNOOP_NEEDED_N		1003C0040H
		OFFCORE_RESPONSE.PF_L2_CODE_RD.LLC_HIT.SNOOP_MISS_N		2003C0040H
		OFFCORE_RESPONSE.PF_L2_CODE_RD.LLC_MISS.DRAM_N		300400040H
		OFFCORE_RESPONSE.PF_L2_DATA_RD.LLC_MISS.DRAM_N		300400010H
		OFFCORE_RESPONSE.PF_L2_RFO.LLC_HIT.ANY_RESPONSE_N		3F803C0020H
		OFFCORE_RESPONSE.PF_L2_RFO.LLC_HIT.HIT_OTHER_CORE_NO_FWD_N		4003C0020H
		OFFCORE_RESPONSE.PF_L2_RFO.LLC_HIT.HITM_OTHER_CORE_N		10003C0020H
		OFFCORE_RESPONSE.PF_L2_RFO.LLC_HIT.NO_SNOOP_NEEDED_N		1003C0020H
		OFFCORE_RESPONSE.PF_L2_RFO.LLC_HIT.SNOOP_MISS_N		2003C0020H
		OFFCORE_RESPONSE.PF_L2_RFO.LLC_MISS.DRAM_N		300400020H
		OFFCORE_RESPONSE.PF_LLC_CODE_RD.LLC_HIT.HIT_OTHER_CORE_NO_FWD_N		4003C0200H
		OFFCORE_RESPONSE.PF_LLC_CODE_RD.LLC_HIT.HITM_OTHER_CORE_N		10003C0200H
		OFFCORE_RESPONSE.PF_LLC_CODE_RD.LLC_HIT.NO_SNOOP_NEEDED_N		1003C0200H
		OFFCORE_RESPONSE.PF_LLC_CODE_RD.LLC_HIT.SNOOP_MISS_N		2003C0200H
		OFFCORE_RESPONSE.PF_LLC_CODE_RD.LLC_MISS.DRAM_N		300400200H
		OFFCORE_RESPONSE.PF_LLC_DATA_RD.LLC_MISS.DRAM_N		300400080H
		OFFCORE_RESPONSE.PF_LLC_RFO.LLC_HIT.ANY_RESPONSE_N		3F803C0100H
		OFFCORE_RESPONSE.PF_LLC_RFO.LLC_HIT.HIT_OTHER_CORE_NO_FWD_N		4003C0100H
		OFFCORE_RESPONSE.PF_LLC_RFO.LLC_HIT.HITM_OTHER_CORE_N		10003C0100H
		OFFCORE_RESPONSE.PF_LLC_RFO.LLC_HIT.NO_SNOOP_NEEDED_N		1003C0100H
		OFFCORE_RESPONSE.PF_LLC_RFO.LLC_HIT.SNOOP_MISS_N		2003C0100H
		OFFCORE_RESPONSE.PF_LLC_RFO.LLC_MISS.DRAM_N		300400100H

Non-architecture performance monitoring events in the processor core that are applicable only to Intel Xeon processor E5 family (and Intel Core i7-3930 processor) based on Intel microarchitecture code name Sandy Bridge, with CPUID signature of DisplayFamily_DisplayModel 06_2DH, are listed in Table 19-17.

Table 19-17. Non-Architectural Performance Events Applicable only to the Processor Core of Intel® Xeon® Processor E5 Family

Event Num.	Umask Value	Event Mask Mnemonic	Description	Comment
CDH	01H	MEM_TRANS_RETIRED.LOAD_LATENCY	Additional Configuration: Disable BL bypass and direct2core, and if the memory is remotely homed. The count is not reliable If the memory is locally homed.	
D1H	04H	MEM_LOAD_UOPS_RETIRED.LLC_HIT	Additional Configuration: Disable BL bypass. Supports PEBS.	

Table 19-17. Non-Architectural Performance Events Applicable only to the Processor Core of Intel® Xeon® Processor E5 Family

Event Num.	Umask Value	Event Mask Mnemonic	Description	Comment
D1H	20H	MEM_LOAD_UOPS_RETIRED.LLC_MISS	Additional Configuration: Disable BL bypass and direct2core. Supports PEBS.	
D2H	01H	MEM_LOAD_UOPS_LLC_HIT_RETIRED.XSNP_MISS	Additional Configuration: Disable bypass. Supports PEBS.	
D2H	02H	MEM_LOAD_UOPS_LLC_HIT_RETIRED.XSNP_HIT	Additional Configuration: Disable bypass. Supports PEBS.	
D2H	04H	MEM_LOAD_UOPS_LLC_HIT_RETIRED.XSNP_HITM	Additional Configuration: Disable bypass. Supports PEBS.	
D2H	08H	MEM_LOAD_UOPS_LLC_HIT_RETIRED.XSNP_NONE	Additional Configuration: Disable bypass. Supports PEBS.	
D3H	01H	MEM_LOAD_UOPS_LLC_MISS_RETIRED.LOCAL_DRAM	Retired load uops which data sources were data missed LLC but serviced by local DRAM. Supports PEBS.	Disable BL bypass and direct2core (see MSR 3C9H).
D3H	04H	MEM_LOAD_UOPS_LLC_MISS_RETIRED.REMOTE_DRAM	Retired load uops which data sources were data missed LLC but serviced by remote DRAM. Supports PEBS.	Disable BL bypass and direct2core (see MSR 3C9H).
B7H/BBH	01H	OFF_CORE_RESPONSE_N	Sub-events of OFF_CORE_RESPONSE_N (suffix N = 0, 1) programmed using MSR 01A6H/01A7H with values shown in the comment column.	
		OFFCORE_RESPONSE.DEMAND_CODE_RD.LLC_MISS.ANY_RESPONSE_N		3FFFC00004H
		OFFCORE_RESPONSE.DEMAND_CODE_RD.LLC_MISS.LOCAL_DRAM_N		600400004H
		OFFCORE_RESPONSE.DEMAND_CODE_RD.LLC_MISS.REMOTE_DRAM_N		67F800004H
		OFFCORE_RESPONSE.DEMAND_CODE_RD.LLC_MISS.REMOTE_HIT_FWD_N		87F800004H
		OFFCORE_RESPONSE.DEMAND_CODE_RD.LLC_MISS.REMOTE_HITM_N		107FC00004H
		OFFCORE_RESPONSE.DEMAND_DATA_RD.LLC_MISS.ANY_DRAM_N		67FC00001H
		OFFCORE_RESPONSE.DEMAND_DATA_RD.LLC_MISS.ANY_RESPONSE_N		3F803C0001H
		OFFCORE_RESPONSE.DEMAND_DATA_RD.LLC_MISS.LOCAL_DRAM_N		600400001H
		OFFCORE_RESPONSE.DEMAND_DATA_RD.LLC_MISS.REMOTE_DRAM_N		67F800001H
		OFFCORE_RESPONSE.DEMAND_DATA_RD.LLC_MISS.REMOTE_HIT_FWD_N		87F800001H
		OFFCORE_RESPONSE.DEMAND_DATA_RD.LLC_MISS.REMOTE_HITM_N		107FC00001H
		OFFCORE_RESPONSE.PF_L2_CODE_RD.LLC_MISS.ANY_RESPONSE_N		3F803C0040H
		OFFCORE_RESPONSE.PF_L2_DATA_RD.LLC_MISS.ANY_DRAM_N		67FC00010H
		OFFCORE_RESPONSE.PF_L2_DATA_RD.LLC_MISS.ANY_RESPONSE_N		3F803C0010H
		OFFCORE_RESPONSE.PF_L2_DATA_RD.LLC_MISS.LOCAL_DRAM_N		600400010H
		OFFCORE_RESPONSE.PF_L2_DATA_RD.LLC_MISS.REMOTE_DRAM_N		67F800010H
		OFFCORE_RESPONSE.PF_L2_DATA_RD.LLC_MISS.REMOTE_HIT_FWD_N		87F800010H
		OFFCORE_RESPONSE.PF_L2_DATA_RD.LLC_MISS.REMOTE_HITM_N		107FC00010H
		OFFCORE_RESPONSE.PF_LLC_CODE_RD.LLC_MISS.ANY_RESPONSE_N		3FFFC00200H
		OFFCORE_RESPONSE.PF_LLC_DATA_RD.LLC_MISS.ANY_RESPONSE_N		3FFFC00080H

Non-architectural Performance monitoring events that are located in the uncore sub-system are implementation specific between different platforms using processors based on Intel microarchitecture code name Sandy Bridge. Processors with CPUID signature of DisplayFamily_DisplayModel 06_2AH support performance events listed in Table 19-18.

Table 19-18. Non-Architectural Performance Events In the Processor Uncore for 2nd Generation Intel® Core™ i7-2xxx, Intel® Core™ i5-2xxx, Intel® Core™ i3-2xxx Processor Series

Event Num.[1]	Umask Value	Event Mask Mnemonic	Description	Comment
22H	01H	UNC_CBO_XSNP_RESPONSE.MISS	A snoop misses in some processor core.	Must combine with one of the umask values of 20H, 40H, 80H.
22H	02H	UNC_CBO_XSNP_RESPONSE.INVAL	A snoop invalidates a non-modified line in some processor core.	
22H	04H	UNC_CBO_XSNP_RESPONSE.HIT	A snoop hits a non-modified line in some processor core.	
22H	08H	UNC_CBO_XSNP_RESPONSE.HITM	A snoop hits a modified line in some processor core.	
22H	10H	UNC_CBO_XSNP_RESPONSE.INVAL_M	A snoop invalidates a modified line in some processor core.	
22H	20H	UNC_CBO_XSNP_RESPONSE.EXTERNAL_FILTER	Filter on cross-core snoops initiated by this Cbox due to external snoop request.	Must combine with at least one of 01H, 02H, 04H, 08H, 10H.
22H	40H	UNC_CBO_XSNP_RESPONSE.XCORE_FILTER	Filter on cross-core snoops initiated by this Cbox due to processor core memory request.	
22H	80H	UNC_CBO_XSNP_RESPONSE.EVICTION_FILTER	Filter on cross-core snoops initiated by this Cbox due to LLC eviction.	
34H	01H	UNC_CBO_CACHE_LOOKUP.M	LLC lookup request that access cache and found line in M-state.	Must combine with one of the umask values of 10H, 20H, 40H, 80H.
34H	02H	UNC_CBO_CACHE_LOOKUP.E	LLC lookup request that access cache and found line in E-state.	
34H	04H	UNC_CBO_CACHE_LOOKUP.S	LLC lookup request that access cache and found line in S-state.	
34H	08H	UNC_CBO_CACHE_LOOKUP.I	LLC lookup request that access cache and found line in I-state.	
34H	10H	UNC_CBO_CACHE_LOOKUP.READ_FILTER	Filter on processor core initiated cacheable read requests. Must combine with at least one of 01H, 02H, 04H, 08H.	
34H	20H	UNC_CBO_CACHE_LOOKUP.WRITE_FILTER	Filter on processor core initiated cacheable write requests. Must combine with at least one of 01H, 02H, 04H, 08H.	
34H	40H	UNC_CBO_CACHE_LOOKUP.EXTSNP_FILTER	Filter on external snoop requests. Must combine with at least one of 01H, 02H, 04H, 08H.	
34H	80H	UNC_CBO_CACHE_LOOKUP.ANY_REQUEST_FILTER	Filter on any IRQ or IPQ initiated requests including uncacheable, non-coherent requests. Must combine with at least one of 01H, 02H, 04H, 08H.	
80H	01H	UNC_ARB_TRK_OCCUPANCY.ALL	Counts cycles weighted by the number of requests waiting for data returning from the memory controller. Accounts for coherent and non-coherent requests initiated by IA cores, processor graphic units, or LLC.	Counter 0 only.
81H	01H	UNC_ARB_TRK_REQUEST.ALL	Counts the number of coherent and in-coherent requests initiated by IA cores, processor graphic units, or LLC.	
81H	20H	UNC_ARB_TRK_REQUEST.WRITES	Counts the number of allocated write entries, include full, partial, and LLC evictions.	
81H	80H	UNC_ARB_TRK_REQUEST.EVICTIONS	Counts the number of LLC evictions allocated.	

Table 19-18. Non-Architectural Performance Events In the Processor Uncore for 2nd Generation Intel® Core™ i7-2xxx, Intel® Core™ i5-2xxx, Intel® Core™ i3-2xxx Processor Series (Contd.)

Event Num.[1]	Umask Value	Event Mask Mnemonic	Description	Comment
83H	01H	UNC_ARB_COH_TRK_OCCUPANCY.ALL	Cycles weighted by number of requests pending in Coherency Tracker.	Counter 0 only.
84H	01H	UNC_ARB_COH_TRK_REQUEST.ALL	Number of requests allocated in Coherency Tracker.	

NOTES:

1. The uncore events must be programmed using MSRs located in specific performance monitoring units in the uncore. UNC_CBO* events are supported using MSR_UNC_CBO* MSRs; UNC_ARB* events are supported using MSR_UNC_ARB*MSRs.

19.9 PERFORMANCE MONITORING EVENTS FOR INTEL® CORE™ I7 PROCESSOR FAMILY AND INTEL® XEON® PROCESSOR FAMILY

Processors based on the Intel microarchitecture code name Nehalem support the architectural and non-architectural performance-monitoring events listed in Table 19-1 and Table 19-19. The events in Table 19-19 generally applies to processors with CPUID signature of DisplayFamily_DisplayModel encoding with the following values: 06_1AH, 06_1EH, 06_1FH, and 06_2EH. However, Intel Xeon processors with CPUID signature of DisplayFamily_DisplayModel 06_2EH have a small number of events that are not supported in processors with CPUID signature 06_1AH, 06_1EH, and 06_1FH. These events are noted in the comment column.

In addition, these processors (CPUID signature of DisplayFamily_DisplayModel 06_1AH, 06_1EH, 06_1FH) also support the following non-architectural, product-specific uncore performance-monitoring events listed in Table 19-20.

Fixed counters in the core PMU support the architecture events defined in Table 19-2.

Table 19-19. Non-Architectural Performance Events In the Processor Core for Intel® Core™ i7 Processor and Intel® Xeon® Processor 5500 Series

Event Num.	Umask Value	Event Mask Mnemonic	Description	Comment
04H	07H	SB_DRAIN.ANY	Counts the number of store buffer drains.	
06H	04H	STORE_BLOCKS.AT_RET	Counts number of loads delayed with at-Retirement block code. The following loads need to be executed at retirement and wait for all senior stores on the same thread to be drained: load splitting across 4K boundary (page split), load accessing uncacheable (UC or WC) memory, load lock, and load with page table in UC or WC memory region.	
06H	08H	STORE_BLOCKS.L1D_BLOCK	Cacheable loads delayed with L1D block code.	
07H	01H	PARTIAL_ADDRESS_ALIAS	Counts false dependency due to partial address aliasing.	
08H	01H	DTLB_LOAD_MISSES.ANY	Counts all load misses that cause a page walk.	
08H	02H	DTLB_LOAD_MISSES.WALK_COMPLETED	Counts number of completed page walks due to load miss in the STLB.	
08H	10H	DTLB_LOAD_MISSES.STLB_HIT	Number of cache load STLB hits.	
08H	20H	DTLB_LOAD_MISSES.PDE_MISS	Number of DTLB cache load misses where the low part of the linear to physical address translation was missed.	
08H	80H	DTLB_LOAD_MISSES.LARGE_WALK_COMPLETED	Counts number of completed large page walks due to load miss in the STLB.	

Table 19-19. Non-Architectural Performance Events In the Processor Core for Intel® Core™ i7 Processor and Intel® Xeon® Processor 5500 Series (Contd.)

Event Num.	Umask Value	Event Mask Mnemonic	Description	Comment
0BH	01H	MEM_INST_RETIRED.LOADS	Counts the number of instructions with an architecturally-visible load retired on the architected path.	
0BH	02H	MEM_INST_RETIRED.STORES	Counts the number of instructions with an architecturally-visible store retired on the architected path.	
0BH	10H	MEM_INST_RETIRED.LATENCY_ABOVE_THRESHOLD	Counts the number of instructions exceeding the latency specified with ld_lat facility.	In conjunction with ld_lat facility.
0CH	01H	MEM_STORE_RETIRED.DTLB_MISS	The event counts the number of retired stores that missed the DTLB. The DTLB miss is not counted if the store operation causes a fault. Does not counter prefetches. Counts both primary and secondary misses to the TLB.	
0EH	01H	UOPS_ISSUED.ANY	Counts the number of Uops issued by the Register Allocation Table to the Reservation Station, i.e. the UOPs issued from the front end to the back end.	
0EH	01H	UOPS_ISSUED.STALLED_CYCLES	Counts the number of cycles no Uops issued by the Register Allocation Table to the Reservation Station, i.e. the UOPs issued from the front end to the back end.	Set "invert=1, cmask = 1".
0EH	02H	UOPS_ISSUED.FUSED	Counts the number of fused Uops that were issued from the Register Allocation Table to the Reservation Station.	
0FH	01H	MEM_UNCORE_RETIRED.L3_DATA_MISS_UNKNOWN	Counts number of memory load instructions retired where the memory reference missed L3 and data source is unknown.	Available only for CPUID signature 06_2EH.
0FH	02H	MEM_UNCORE_RETIRED.OTHER_CORE_L2_HITM	Counts number of memory load instructions retired where the memory reference hit modified data in a sibling core residing on the same socket.	
0FH	08H	MEM_UNCORE_RETIRED.REMOTE_CACHE_LOCAL_HOME_HIT	Counts number of memory load instructions retired where the memory reference missed the L1, L2 and L3 caches and HIT in a remote socket's cache. Only counts locally homed lines.	
0FH	10H	MEM_UNCORE_RETIRED.REMOTE_DRAM	Counts number of memory load instructions retired where the memory reference missed the L1, L2 and L3 caches and was remotely homed. This includes both DRAM access and HITM in a remote socket's cache for remotely homed lines.	
0FH	20H	MEM_UNCORE_RETIRED.LOCAL_DRAM	Counts number of memory load instructions retired where the memory reference missed the L1, L2 and L3 caches and required a local socket memory reference. This includes locally homed cachelines that were in a modified state in another socket.	
0FH	80H	MEM_UNCORE_RETIRED.UNCACHEABLE	Counts number of memory load instructions retired where the memory reference missed the L1, L2 and L3 caches and to perform I/O.	Available only for CPUID signature 06_2EH.

Table 19-19. Non-Architectural Performance Events In the Processor Core for Intel® Core™ i7 Processor and Intel® Xeon® Processor 5500 Series (Contd.)

Event Num.	Umask Value	Event Mask Mnemonic	Description	Comment
10H	01H	FP_COMP_OPS_EXE.X87	Counts the number of FP Computational Uops Executed. The number of FADD, FSUB, FCOM, FMULs, integer MULs and IMULs, FDIVs, FPREMs, FSQRTS, integer DIVs, and IDIVs. This event does not distinguish an FADD used in the middle of a transcendental flow from a separate FADD instruction.	
10H	02H	FP_COMP_OPS_EXE.MMX	Counts number of MMX Uops executed.	
10H	04H	FP_COMP_OPS_EXE.SSE_FP	Counts number of SSE and SSE2 FP uops executed.	
10H	08H	FP_COMP_OPS_EXE.SSE2_INTEGER	Counts number of SSE2 integer uops executed.	
10H	10H	FP_COMP_OPS_EXE.SSE_FP_PACKED	Counts number of SSE FP packed uops executed.	
10H	20H	FP_COMP_OPS_EXE.SSE_FP_SCALAR	Counts number of SSE FP scalar uops executed.	
10H	40H	FP_COMP_OPS_EXE.SSE_SINGLE_PRECISION	Counts number of SSE* FP single precision uops executed.	
10H	80H	FP_COMP_OPS_EXE.SSE_DOUBLE_PRECISION	Counts number of SSE* FP double precision uops executed.	
12H	01H	SIMD_INT_128.PACKED_MPY	Counts number of 128 bit SIMD integer multiply operations.	
12H	02H	SIMD_INT_128.PACKED_SHIFT	Counts number of 128 bit SIMD integer shift operations.	
12H	04H	SIMD_INT_128.PACK	Counts number of 128 bit SIMD integer pack operations.	
12H	08H	SIMD_INT_128.UNPACK	Counts number of 128 bit SIMD integer unpack operations.	
12H	10H	SIMD_INT_128.PACKED_LOGICAL	Counts number of 128 bit SIMD integer logical operations.	
12H	20H	SIMD_INT_128.PACKED_ARITH	Counts number of 128 bit SIMD integer arithmetic operations.	
12H	40H	SIMD_INT_128.SHUFFLE_MOVE	Counts number of 128 bit SIMD integer shuffle and move operations.	
13H	01H	LOAD_DISPATCH.RS	Counts number of loads dispatched from the Reservation Station that bypass the Memory Order Buffer.	
13H	02H	LOAD_DISPATCH.RS_DELAYED	Counts the number of delayed RS dispatches at the stage latch. If an RS dispatch cannot bypass to LB, it has another chance to dispatch from the one-cycle delayed staging latch before it is written into the LB.	
13H	04H	LOAD_DISPATCH.MOB	Counts the number of loads dispatched from the Reservation Station to the Memory Order Buffer.	
13H	07H	LOAD_DISPATCH.ANY	Counts all loads dispatched from the Reservation Station.	

Table 19-19. Non-Architectural Performance Events In the Processor Core for Intel® Core™ i7 Processor and Intel® Xeon® Processor 5500 Series (Contd.)

Event Num.	Umask Value	Event Mask Mnemonic	Description	Comment
14H	01H	ARITH.CYCLES_DIV_BUSY	Counts the number of cycles the divider is busy executing divide or square root operations. The divide can be integer, X87 or Streaming SIMD Extensions (SSE). The square root operation can be either X87 or SSE. Set 'edge =1, invert=1, cmask=1' to count the number of divides.	Count may be incorrect When SMT is on.
14H	02H	ARITH.MUL	Counts the number of multiply operations executed. This includes integer as well as floating point multiply operations but excludes DPPS mul and MPSAD.	Count may be incorrect When SMT is on.
17H	01H	INST_QUEUE_WRITES	Counts the number of instructions written into the instruction queue every cycle.	
18H	01H	INST_DECODED.DEC0	Counts number of instructions that require decoder 0 to be decoded. Usually, this means that the instruction maps to more than 1 uop.	
19H	01H	TWO_UOP_INSTS_DECODED	An instruction that generates two uops was decoded.	
1EH	01H	INST_QUEUE_WRITE_CYCLES	This event counts the number of cycles during which instructions are written to the instruction queue. Dividing this counter by the number of instructions written to the instruction queue (INST_QUEUE_WRITES) yields the average number of instructions decoded each cycle. If this number is less than four and the pipe stalls, this indicates that the decoder is failing to decode enough instructions per cycle to sustain the 4-wide pipeline.	If SSE* instructions that are 6 bytes or longer arrive one after another, then front end throughput may limit execution speed.
20H	01H	LSD_OVERFLOW	Counts number of loops that can't stream from the instruction queue.	
24H	01H	L2_RQSTS.LD_HIT	Counts number of loads that hit the L2 cache. L2 loads include both L1D demand misses as well as L1D prefetches. L2 loads can be rejected for various reasons. Only non rejected loads are counted.	
24H	02H	L2_RQSTS.LD_MISS	Counts the number of loads that miss the L2 cache. L2 loads include both L1D demand misses as well as L1D prefetches.	
24H	03H	L2_RQSTS.LOADS	Counts all L2 load requests. L2 loads include both L1D demand misses as well as L1D prefetches.	
24H	04H	L2_RQSTS.RFO_HIT	Counts the number of store RFO requests that hit the L2 cache. L2 RFO requests include both L1D demand RFO misses as well as L1D RFO prefetches. Count includes WC memory requests, where the data is not fetched but the permission to write the line is required.	
24H	08H	L2_RQSTS.RFO_MISS	Counts the number of store RFO requests that miss the L2 cache. L2 RFO requests include both L1D demand RFO misses as well as L1D RFO prefetches.	

Table 19-19. Non-Architectural Performance Events In the Processor Core for Intel® Core™ i7 Processor and Intel® Xeon® Processor 5500 Series (Contd.)

Event Num.	Umask Value	Event Mask Mnemonic	Description	Comment
24H	0CH	L2_RQSTS.RFOS	Counts all L2 store RFO requests. L2 RFO requests include both L1D demand RFO misses as well as L1D RFO prefetches.	
24H	10H	L2_RQSTS.IFETCH_HIT	Counts number of instruction fetches that hit the L2 cache. L2 instruction fetches include both L1I demand misses as well as L1I instruction prefetches.	
24H	20H	L2_RQSTS.IFETCH_MISS	Counts number of instruction fetches that miss the L2 cache. L2 instruction fetches include both L1I demand misses as well as L1I instruction prefetches.	
24H	30H	L2_RQSTS.IFETCHES	Counts all instruction fetches. L2 instruction fetches include both L1I demand misses as well as L1I instruction prefetches.	
24H	40H	L2_RQSTS.PREFETCH_HIT	Counts L2 prefetch hits for both code and data.	
24H	80H	L2_RQSTS.PREFETCH_MISS	Counts L2 prefetch misses for both code and data.	
24H	C0H	L2_RQSTS.PREFETCHES	Counts all L2 prefetches for both code and data.	
24H	AAH	L2_RQSTS.MISS	Counts all L2 misses for both code and data.	
24H	FFH	L2_RQSTS.REFERENCES	Counts all L2 requests for both code and data.	
26H	01H	L2_DATA_RQSTS.DEMAND.I_STATE	Counts number of L2 data demand loads where the cache line to be loaded is in the I (invalid) state, i.e., a cache miss. L2 demand loads are both L1D demand misses and L1D prefetches.	
26H	02H	L2_DATA_RQSTS.DEMAND.S_STATE	Counts number of L2 data demand loads where the cache line to be loaded is in the S (shared) state. L2 demand loads are both L1D demand misses and L1D prefetches.	
26H	04H	L2_DATA_RQSTS.DEMAND.E_STATE	Counts number of L2 data demand loads where the cache line to be loaded is in the E (exclusive) state. L2 demand loads are both L1D demand misses and L1D prefetches.	
26H	08H	L2_DATA_RQSTS.DEMAND.M_STATE	Counts number of L2 data demand loads where the cache line to be loaded is in the M (modified) state. L2 demand loads are both L1D demand misses and L1D prefetches.	
26H	0FH	L2_DATA_RQSTS.DEMAND.MESI	Counts all L2 data demand requests. L2 demand loads are both L1D demand misses and L1D prefetches.	
26H	10H	L2_DATA_RQSTS.PREFETCH.I_STATE	Counts number of L2 prefetch data loads where the cache line to be loaded is in the I (invalid) state, i.e., a cache miss.	
26H	20H	L2_DATA_RQSTS.PREFETCH.S_STATE	Counts number of L2 prefetch data loads where the cache line to be loaded is in the S (shared) state. A prefetch RFO will miss on an S state line, while a prefetch read will hit on an S state line.	
26H	40H	L2_DATA_RQSTS.PREFETCH.E_STATE	Counts number of L2 prefetch data loads where the cache line to be loaded is in the E (exclusive) state.	

Table 19-19. Non-Architectural Performance Events In the Processor Core for Intel® Core™ i7 Processor and Intel® Xeon® Processor 5500 Series (Contd.)

Event Num.	Umask Value	Event Mask Mnemonic	Description	Comment
26H	80H	L2_DATA_RQSTS.PREFETCH.M_STATE	Counts number of L2 prefetch data loads where the cache line to be loaded is in the M (modified) state.	
26H	F0H	L2_DATA_RQSTS.PREFETCH.MESI	Counts all L2 prefetch requests.	
26H	FFH	L2_DATA_RQSTS.ANY	Counts all L2 data requests.	
27H	01H	L2_WRITE.RFO.I_STATE	Counts number of L2 demand store RFO requests where the cache line to be loaded is in the I (invalid) state, i.e., a cache miss. The L1D prefetcher does not issue a RFO prefetch.	This is a demand RFO request.
27H	02H	L2_WRITE.RFO.S_STATE	Counts number of L2 store RFO requests where the cache line to be loaded is in the S (shared) state. The L1D prefetcher does not issue a RFO prefetch.	This is a demand RFO request.
27H	08H	L2_WRITE.RFO.M_STATE	Counts number of L2 store RFO requests where the cache line to be loaded is in the M (modified) state. The L1D prefetcher does not issue a RFO prefetch.	This is a demand RFO request.
27H	0EH	L2_WRITE.RFO.HIT	Counts number of L2 store RFO requests where the cache line to be loaded is in either the S, E or M states. The L1D prefetcher does not issue a RFO prefetch.	This is a demand RFO request.
27H	0FH	L2_WRITE.RFO.MESI	Counts all L2 store RFO requests. The L1D prefetcher does not issue a RFO prefetch.	This is a demand RFO request.
27H	10H	L2_WRITE.LOCK.I_STATE	Counts number of L2 demand lock RFO requests where the cache line to be loaded is in the I (invalid) state, for example, a cache miss.	
27H	20H	L2_WRITE.LOCK.S_STATE	Counts number of L2 lock RFO requests where the cache line to be loaded is in the S (shared) state.	
27H	40H	L2_WRITE.LOCK.E_STATE	Counts number of L2 demand lock RFO requests where the cache line to be loaded is in the E (exclusive) state.	
27H	80H	L2_WRITE.LOCK.M_STATE	Counts number of L2 demand lock RFO requests where the cache line to be loaded is in the M (modified) state.	
27H	E0H	L2_WRITE.LOCK.HIT	Counts number of L2 demand lock RFO requests where the cache line to be loaded is in either the S, E, or M state.	
27H	F0H	L2_WRITE.LOCK.MESI	Counts all L2 demand lock RFO requests.	
28H	01H	L1D_WB_L2.I_STATE	Counts number of L1 writebacks to the L2 where the cache line to be written is in the I (invalid) state, i.e., a cache miss.	
28H	02H	L1D_WB_L2.S_STATE	Counts number of L1 writebacks to the L2 where the cache line to be written is in the S state.	
28H	04H	L1D_WB_L2.E_STATE	Counts number of L1 writebacks to the L2 where the cache line to be written is in the E (exclusive) state.	
28H	08H	L1D_WB_L2.M_STATE	Counts number of L1 writebacks to the L2 where the cache line to be written is in the M (modified) state.	

Table 19-19. Non-Architectural Performance Events In the Processor Core for Intel® Core™ i7 Processor and Intel® Xeon® Processor 5500 Series (Contd.)

Event Num.	Umask Value	Event Mask Mnemonic	Description	Comment
28H	0FH	L1D_WB_L2.MESI	Counts all L1 writebacks to the L2 .	
2EH	4FH	L3_LAT_CACHE.REFERENCE	This event counts requests originating from the core that reference a cache line in the last level cache. The event count includes speculative traffic but excludes cache line fills due to a L2 hardware-prefetch. Because cache hierarchy, cache sizes and other implementation-specific characteristics; value comparison to estimate performance differences is not recommended.	See Table 19-1.
2EH	41H	L3_LAT_CACHE.MISS	This event counts each cache miss condition for references to the last level cache. The event count may include speculative traffic but excludes cache line fills due to L2 hardware-prefetches. Because cache hierarchy, cache sizes and other implementation-specific characteristics; value comparison to estimate performance differences is not recommended.	See Table 19-1.
3CH	00H	CPU_CLK_UNHALTED.THREAD_P	Counts the number of thread cycles while the thread is not in a halt state. The thread enters the halt state when it is running the HLT instruction. The core frequency may change from time to time due to power or thermal throttling.	See Table 19-1.
3CH	01H	CPU_CLK_UNHALTED.REF_P	Increments at the frequency of TSC when not halted.	See Table 19-1.
40H	01H	L1D_CACHE_LD.I_STATE	Counts L1 data cache read requests where the cache line to be loaded is in the I (invalid) state, i.e. the read request missed the cache.	Counter 0, 1 only.
40H	02H	L1D_CACHE_LD.S_STATE	Counts L1 data cache read requests where the cache line to be loaded is in the S (shared) state.	Counter 0, 1 only.
40H	04H	L1D_CACHE_LD.E_STATE	Counts L1 data cache read requests where the cache line to be loaded is in the E (exclusive) state.	Counter 0, 1 only.
40H	08H	L1D_CACHE_LD.M_STATE	Counts L1 data cache read requests where the cache line to be loaded is in the M (modified) state.	Counter 0, 1 only.
40H	0FH	L1D_CACHE_LD.MESI	Counts L1 data cache read requests.	Counter 0, 1 only.
41H	02H	L1D_CACHE_ST.S_STATE	Counts L1 data cache store RFO requests where the cache line to be loaded is in the S (shared) state.	Counter 0, 1 only.
41H	04H	L1D_CACHE_ST.E_STATE	Counts L1 data cache store RFO requests where the cache line to be loaded is in the E (exclusive) state.	Counter 0, 1 only.
41H	08H	L1D_CACHE_ST.M_STATE	Counts L1 data cache store RFO requests where cache line to be loaded is in the M (modified) state.	Counter 0, 1 only.
42H	01H	L1D_CACHE_LOCK.HIT	Counts retired load locks that hit in the L1 data cache or hit in an already allocated fill buffer. The lock portion of the load lock transaction must hit in the L1D.	The initial load will pull the lock into the L1 data cache. Counter 0, 1 only.
42H	02H	L1D_CACHE_LOCK.S_STATE	Counts L1 data cache retired load locks that hit the target cache line in the shared state.	Counter 0, 1 only.
42H	04H	L1D_CACHE_LOCK.E_STATE	Counts L1 data cache retired load locks that hit the target cache line in the exclusive state.	Counter 0, 1 only.

Table 19-19. Non-Architectural Performance Events In the Processor Core for Intel® Core™ i7 Processor and Intel® Xeon® Processor 5500 Series (Contd.)

Event Num.	Umask Value	Event Mask Mnemonic	Description	Comment
42H	08H	L1D_CACHE_LOCK.M_STATE	Counts L1 data cache retired load locks that hit the target cache line in the modified state.	Counter 0, 1 only.
43H	01H	L1D_ALL_REF.ANY	Counts all references (uncached, speculated and retired) to the L1 data cache, including all loads and stores with any memory types. The event counts memory accesses only when they are actually performed. For example, a load blocked by unknown store address and later performed is only counted once.	The event does not include non-memory accesses, such as I/O accesses. Counter 0, 1 only.
43H	02H	L1D_ALL_REF.CACHEABLE	Counts all data reads and writes (speculated and retired) from cacheable memory, including locked operations.	Counter 0, 1 only.
49H	01H	DTLB_MISSES.ANY	Counts the number of misses in the STLB which causes a page walk.	
49H	02H	DTLB_MISSES.WALK_COMPLETED	Counts number of misses in the STLB which resulted in a completed page walk.	
49H	10H	DTLB_MISSES.STLB_HIT	Counts the number of DTLB first level misses that hit in the second level TLB. This event is only relevant if the core contains multiple DTLB levels.	
49H	20H	DTLB_MISSES.PDE_MISS	Number of DTLB misses caused by low part of address, includes references to 2M pages because 2M pages do not use the PDE.	
49H	80H	DTLB_MISSES.LARGE_WALK_COMPLETED	Counts number of misses in the STLB which resulted in a completed page walk for large pages.	
4CH	01H	LOAD_HIT_PRE	Counts load operations sent to the L1 data cache while a previous SSE prefetch instruction to the same cache line has started prefetching but has not yet finished.	
4EH	01H	L1D_PREFETCH.REQUESTS	Counts number of hardware prefetch requests dispatched out of the prefetch FIFO.	
4EH	02H	L1D_PREFETCH.MISS	Counts number of hardware prefetch requests that miss the L1D. There are two prefetchers in the L1D. A streamer, which predicts lines sequentially after this one should be fetched, and the IP prefetcher that remembers access patterns for the current instruction. The streamer prefetcher stops on an L1D hit, while the IP prefetcher does not.	
4EH	04H	L1D_PREFETCH.TRIGGERS	Counts number of prefetch requests triggered by the Finite State Machine and pushed into the prefetch FIFO. Some of the prefetch requests are dropped due to overwrites or competition between the IP index prefetcher and streamer prefetcher. The prefetch FIFO contains 4 entries.	
51H	01H	L1D.REPL	Counts the number of lines brought into the L1 data cache.	Counter 0, 1 only.
51H	02H	L1D.M_REPL	Counts the number of modified lines brought into the L1 data cache.	Counter 0, 1 only.
51H	04H	L1D.M_EVICT	Counts the number of modified lines evicted from the L1 data cache due to replacement.	Counter 0, 1 only.

Table 19-19. Non-Architectural Performance Events In the Processor Core for Intel® Core™ i7 Processor and Intel® Xeon® Processor 5500 Series (Contd.)

Event Num.	Umask Value	Event Mask Mnemonic	Description	Comment
51H	08H	L1D.M_SNOOP_EVICT	Counts the number of modified lines evicted from the L1 data cache due to snoop HITM intervention.	Counter 0, 1 only.
52H	01H	L1D_CACHE_PREFETCH_LOCK_FB_HIT	Counts the number of cacheable load lock speculated instructions accepted into the fill buffer.	
53H	01H	L1D_CACHE_LOCK_FB_HIT	Counts the number of cacheable load lock speculated or retired instructions accepted into the fill buffer.	
63H	01H	CACHE_LOCK_CYCLES.L1D_L2	Cycle count during which the L1D and L2 are locked. A lock is asserted when there is a locked memory access, due to uncacheable memory, a locked operation that spans two cache lines, or a page walk from an uncacheable page table.	Counter 0, 1 only. L1D and L2 locks have a very high performance penalty and it is highly recommended to avoid such accesses.
63H	02H	CACHE_LOCK_CYCLES.L1D	Counts the number of cycles that cacheline in the L1 data cache unit is locked.	Counter 0, 1 only.
6CH	01H	IO_TRANSACTIONS	Counts the number of completed I/O transactions.	
80H	01H	L1I.HITS	Counts all instruction fetches that hit the L1 instruction cache.	
80H	02H	L1I.MISSES	Counts all instruction fetches that miss the L1I cache. This includes instruction cache misses, streaming buffer misses, victim cache misses and uncacheable fetches. An instruction fetch miss is counted only once and not once for every cycle it is outstanding.	
80H	03H	L1I.READS	Counts all instruction fetches, including uncacheable fetches that bypass the L1I.	
80H	04H	L1I.CYCLES_STALLED	Cycle counts for which an instruction fetch stalls due to a L1I cache miss, ITLB miss or ITLB fault.	
82H	01H	LARGE_ITLB.HIT	Counts number of large ITLB hits.	
85H	01H	ITLB_MISSES.ANY	Counts the number of misses in all levels of the ITLB which causes a page walk.	
85H	02H	ITLB_MISSES.WALK_COMPLETED	Counts number of misses in all levels of the ITLB which resulted in a completed page walk.	
87H	01H	ILD_STALL.LCP	Cycles Instruction Length Decoder stalls due to length changing prefixes: 66, 67 or REX.W (for Intel 64) instructions which change the length of the decoded instruction.	
87H	02H	ILD_STALL.MRU	Instruction Length Decoder stall cycles due to Brand Prediction Unit (PBU) Most Recently Used (MRU) bypass.	
87H	04H	ILD_STALL.IQ_FULL	Stall cycles due to a full instruction queue.	
87H	08H	ILD_STALL.REGEN	Counts the number of regen stalls.	
87H	0FH	ILD_STALL.ANY	Counts any cycles the Instruction Length Decoder is stalled.	
88H	01H	BR_INST_EXEC.COND	Counts the number of conditional near branch instructions executed, but not necessarily retired.	

Table 19-19. Non-Architectural Performance Events In the Processor Core for Intel® Core™ i7 Processor and Intel® Xeon® Processor 5500 Series (Contd.)

Event Num.	Umask Value	Event Mask Mnemonic	Description	Comment
88H	02H	BR_INST_EXEC.DIRECT	Counts all unconditional near branch instructions excluding calls and indirect branches.	
88H	04H	BR_INST_EXEC.INDIRECT_NON_CALL	Counts the number of executed indirect near branch instructions that are not calls.	
88H	07H	BR_INST_EXEC.NON_CALLS	Counts all non-call near branch instructions executed, but not necessarily retired.	
88H	08H	BR_INST_EXEC.RETURN_NEAR	Counts indirect near branches that have a return mnemonic.	
88H	10H	BR_INST_EXEC.DIRECT_NEAR_CALL	Counts unconditional near call branch instructions, excluding non-call branch, executed.	
88H	20H	BR_INST_EXEC.INDIRECT_NEAR_CALL	Counts indirect near calls, including both register and memory indirect, executed.	
88H	30H	BR_INST_EXEC.NEAR_CALLS	Counts all near call branches executed, but not necessarily retired.	
88H	40H	BR_INST_EXEC.TAKEN	Counts taken near branches executed, but not necessarily retired.	
88H	7FH	BR_INST_EXEC.ANY	Counts all near executed branches (not necessarily retired). This includes only instructions and not micro-op branches. Frequent branching is not necessarily a major performance issue. However frequent branch mispredictions may be a problem.	
89H	01H	BR_MISP_EXEC.COND	Counts the number of mispredicted conditional near branch instructions executed, but not necessarily retired.	
89H	02H	BR_MISP_EXEC.DIRECT	Counts mispredicted macro unconditional near branch instructions, excluding calls and indirect branches (should always be 0).	
89H	04H	BR_MISP_EXEC.INDIRECT_NON_CALL	Counts the number of executed mispredicted indirect near branch instructions that are not calls.	
89H	07H	BR_MISP_EXEC.NON_CALLS	Counts mispredicted non-call near branches executed, but not necessarily retired.	
89H	08H	BR_MISP_EXEC.RETURN_NEAR	Counts mispredicted indirect branches that have a rear return mnemonic.	
89H	10H	BR_MISP_EXEC.DIRECT_NEAR_CALL	Counts mispredicted non-indirect near calls executed, (should always be 0).	
89H	20H	BR_MISP_EXEC.INDIRECT_NEAR_CALL	Counts mispredicted indirect near calls executed, including both register and memory indirect.	
89H	30H	BR_MISP_EXEC.NEAR_CALLS	Counts all mispredicted near call branches executed, but not necessarily retired.	
89H	40H	BR_MISP_EXEC.TAKEN	Counts executed mispredicted near branches that are taken, but not necessarily retired.	
89H	7FH	BR_MISP_EXEC.ANY	Counts the number of mispredicted near branch instructions that were executed, but not necessarily retired.	

Table 19-19. Non-Architectural Performance Events In the Processor Core for Intel® Core™ i7 Processor and Intel® Xeon® Processor 5500 Series (Contd.)

Event Num.	Umask Value	Event Mask Mnemonic	Description	Comment
A2H	01H	RESOURCE_STALLS.ANY	Counts the number of Allocator resource related stalls. Includes register renaming buffer entries, memory buffer entries. In addition to resource related stalls, this event counts some other events. Includes stalls arising during branch misprediction recovery, such as if retirement of the mispredicted branch is delayed and stalls arising while store buffer is draining from synchronizing operations.	Does not include stalls due to SuperQ (off core) queue full, too many cache misses, etc.
A2H	02H	RESOURCE_STALLS.LOAD	Counts the cycles of stall due to lack of load buffer for load operation.	
A2H	04H	RESOURCE_STALLS.RS_FULL	This event counts the number of cycles when the number of instructions in the pipeline waiting for execution reaches the limit the processor can handle. A high count of this event indicates that there are long latency operations in the pipe (possibly load and store operations that miss the L2 cache, or instructions dependent upon instructions further down the pipeline that have yet to retire.	When RS is full, new instructions cannot enter the reservation station and start execution.
A2H	08H	RESOURCE_STALLS.STORE	This event counts the number of cycles that a resource related stall will occur due to the number of store instructions reaching the limit of the pipeline, (i.e. all store buffers are used). The stall ends when a store instruction commits its data to the cache or memory.	
A2H	10H	RESOURCE_STALLS.ROB_FULL	Counts the cycles of stall due to re-order buffer full.	
A2H	20H	RESOURCE_STALLS.FPCW	Counts the number of cycles while execution was stalled due to writing the floating-point unit (FPU) control word.	
A2H	40H	RESOURCE_STALLS.MXCSR	Stalls due to the MXCSR register rename occurring to close to a previous MXCSR rename. The MXCSR provides control and status for the MMX registers.	
A2H	80H	RESOURCE_STALLS.OTHER	Counts the number of cycles while execution was stalled due to other resource issues.	
A6H	01H	MACRO_INSTS.FUSIONS_DECODED	Counts the number of instructions decoded that are macro-fused but not necessarily executed or retired.	
A7H	01H	BACLEAR_FORCE_IQ	Counts number of times a BACLEAR was forced by the Instruction Queue. The IQ is also responsible for providing conditional branch prediction direction based on a static scheme and dynamic data provided by the L2 Branch Prediction Unit. If the conditional branch target is not found in the Target Array and the IQ predicts that the branch is taken, then the IQ will force the Branch Address Calculator to issue a BACLEAR. Each BACLEAR asserted by the BAC generates approximately an 8 cycle bubble in the instruction fetch pipeline.	
A8H	01H	LSD.UOPS	Counts the number of micro-ops delivered by loop stream detector.	Use cmask=1 and invert to count cycles.
AEH	01H	ITLB_FLUSH	Counts the number of ITLB flushes.	

Table 19-19. Non-Architectural Performance Events In the Processor Core for Intel® Core™ i7 Processor and Intel® Xeon® Processor 5500 Series (Contd.)

Event Num.	Umask Value	Event Mask Mnemonic	Description	Comment
B0H	40H	OFFCORE_REQUESTS.L1D_WRITEBACK	Counts number of L1D writebacks to the uncore.	
B1H	01H	UOPS_EXECUTED.PORT0	Counts number of uops executed that were issued on port 0. Port 0 handles integer arithmetic, SIMD and FP add uops.	
B1H	02H	UOPS_EXECUTED.PORT1	Counts number of uops executed that were issued on port 1. Port 1 handles integer arithmetic, SIMD, integer shift, FP multiply and FP divide uops.	
B1H	04H	UOPS_EXECUTED.PORT2_CORE	Counts number of uops executed that were issued on port 2. Port 2 handles the load uops. This is a core count only and cannot be collected per thread.	
B1H	08H	UOPS_EXECUTED.PORT3_CORE	Counts number of uops executed that were issued on port 3. Port 3 handles store uops. This is a core count only and cannot be collected per thread.	
B1H	10H	UOPS_EXECUTED.PORT4_CORE	Counts number of uops executed that where issued on port 4. Port 4 handles the value to be stored for the store uops issued on port 3. This is a core count only and cannot be collected per thread.	
B1H	1FH	UOPS_EXECUTED.CORE_ACTIVE_CYCLES_NO_PORT5	Counts cycles when the uops executed were issued from any ports except port 5. Use Cmask=1 for active cycles; Cmask=0 for weighted cycles. Use CMask=1, Invert=1 to count P0-4 stalled cycles. Use Cmask=1, Edge=1, Invert=1 to count P0-4 stalls.	
B1H	20H	UOPS_EXECUTED.PORT5	Counts number of uops executed that where issued on port 5.	
B1H	3FH	UOPS_EXECUTED.CORE_ACTIVE_CYCLES	Counts cycles when the uops are executing. Use Cmask=1 for active cycles; Cmask=0 for weighted cycles. Use CMask=1, Invert=1 to count P0-4 stalled cycles. Use Cmask=1, Edge=1, Invert=1 to count P0-4 stalls.	
B1H	40H	UOPS_EXECUTED.PORT015	Counts number of uops executed that where issued on port 0, 1, or 5.	Use cmask=1, invert=1 to count stall cycles.
B1H	80H	UOPS_EXECUTED.PORT234	Counts number of uops executed that where issued on port 2, 3, or 4.	
B2H	01H	OFFCORE_REQUESTS_SQ_FULL	Counts number of cycles the SQ is full to handle off-core requests.	
B7H	01H	OFF_CORE_RESPONSE_0	See Section 18.3.1.1.3, "Off-core Response Performance Monitoring in the Processor Core".	Requires programming MSR 01A6H.
B8H	01H	SNOOP_RESPONSE.HIT	Counts HIT snoop response sent by this thread in response to a snoop request.	
B8H	02H	SNOOP_RESPONSE.HITE	Counts HIT E snoop response sent by this thread in response to a snoop request.	
B8H	04H	SNOOP_RESPONSE.HITM	Counts HIT M snoop response sent by this thread in response to a snoop request.	
BBH	01H	OFF_CORE_RESPONSE_1	See Section 18.6.3, "Performance Monitoring (Processors Based on Intel NetBurst® Microarchitecture)".	Requires programming MSR 01A7H.

Table 19-19. Non-Architectural Performance Events In the Processor Core for Intel® Core™ i7 Processor and Intel® Xeon® Processor 5500 Series (Contd.)

Event Num.	Umask Value	Event Mask Mnemonic	Description	Comment
C0H	00H	INST_RETIRED.ANY_P	See Table 19-1. Notes: INST_RETIRED.ANY is counted by a designated fixed counter. INST_RETIRED.ANY_P is counted by a programmable counter and is an architectural performance event. Event is supported if CPUID.A.EBX[1] = 0.	Counting: Faulting executions of GETSEC/VM entry/VM Exit/MWait will not count as retired instructions.
C0H	02H	INST_RETIRED.X87	Counts the number of MMX instructions retired.	
C0H	04H	INST_RETIRED.MMX	Counts the number of floating point computational operations retired: floating point computational operations executed by the assist handler and sub-operations of complex floating point instructions like transcendental instructions.	
C2H	01H	UOPS_RETIRED.ANY	Counts the number of micro-ops retired, (macro-fused=1, micro-fused=2, others=1; maximum count of 8 per cycle). Most instructions are composed of one or two micro-ops. Some instructions are decoded into longer sequences such as repeat instructions, floating point transcendental instructions, and assists.	Use cmask=1 and invert to count active cycles or stalled cycles.
C2H	02H	UOPS_RETIRED.RETIRE_SLOTS	Counts the number of retirement slots used each cycle.	
C2H	04H	UOPS_RETIRED.MACRO_FUSED	Counts number of macro-fused uops retired.	
C3H	01H	MACHINE_CLEARS.CYCLES	Counts the cycles machine clear is asserted.	
C3H	02H	MACHINE_CLEARS.MEM_ORDER	Counts the number of machine clears due to memory order conflicts.	
C3H	04H	MACHINE_CLEARS.SMC	Counts the number of times that a program writes to a code section. Self-modifying code causes a severe penalty in all Intel 64 and IA-32 processors. The modified cache line is written back to the L2 and L3caches.	
C4H	00H	BR_INST_RETIRED.ALL_BRANCHES	Branch instructions at retirement.	See Table 19-1.
C4H	01H	BR_INST_RETIRED.CONDITIONAL	Counts the number of conditional branch instructions retired.	
C4H	02H	BR_INST_RETIRED.NEAR_CALL	Counts the number of direct & indirect near unconditional calls retired.	
C5H	00H	BR_MISP_RETIRED.ALL_BRANCHES	Mispredicted branch instructions at retirement.	See Table 19-1.
C5H	02H	BR_MISP_RETIRED.NEAR_CALL	Counts mispredicted direct & indirect near unconditional retired calls.	
C7H	01H	SSEX_UOPS_RETIRED.PACKED_SINGLE	Counts SIMD packed single-precision floating point Uops retired.	
C7H	02H	SSEX_UOPS_RETIRED.SCALAR_SINGLE	Counts SIMD scalar single-precision floating point Uops retired.	
C7H	04H	SSEX_UOPS_RETIRED.PACKED_DOUBLE	Counts SIMD packed double-precision floating point Uops retired.	

Table 19-19. Non-Architectural Performance Events In the Processor Core for Intel® Core™ i7 Processor and Intel® Xeon® Processor 5500 Series (Contd.)

Event Num.	Umask Value	Event Mask Mnemonic	Description	Comment
C7H	08H	SSEX_UOPS_RETIRED.SCALAR_DOUBLE	Counts SIMD scalar double-precision floating point Uops retired.	
C7H	10H	SSEX_UOPS_RETIRED.VECTOR_INTEGER	Counts 128-bit SIMD vector integer Uops retired.	
C8H	20H	ITLB_MISS_RETIRED	Counts the number of retired instructions that missed the ITLB when the instruction was fetched.	
CBH	01H	MEM_LOAD_RETIRED.L1D_HIT	Counts number of retired loads that hit the L1 data cache.	
CBH	02H	MEM_LOAD_RETIRED.L2_HIT	Counts number of retired loads that hit the L2 data cache.	
CBH	04H	MEM_LOAD_RETIRED.L3_UNSHARED_HIT	Counts number of retired loads that hit their own, unshared lines in the L3 cache.	
CBH	08H	MEM_LOAD_RETIRED.OTHER_CORE_L2_HIT_HITM	Counts number of retired loads that hit in a sibling core's L2 (on die core). Since the L3 is inclusive of all cores on the package, this is an L3 hit. This counts both clean and modified hits.	
CBH	10H	MEM_LOAD_RETIRED.L3_MISS	Counts number of retired loads that miss the L3 cache. The load was satisfied by a remote socket, local memory or an IOH.	
CBH	40H	MEM_LOAD_RETIRED.HIT_LFB	Counts number of retired loads that miss the L1D and the address is located in an allocated line fill buffer and will soon be committed to cache. This is counting secondary L1D misses.	
CBH	80H	MEM_LOAD_RETIRED.DTLB_MISS	Counts the number of retired loads that missed the DTLB. The DTLB miss is not counted if the load operation causes a fault. This event counts loads from cacheable memory only. The event does not count loads by software prefetches. Counts both primary and secondary misses to the TLB.	
CCH	01H	FP_MMX_TRANS.TO_FP	Counts the first floating-point instruction following any MMX instruction. You can use this event to estimate the penalties for the transitions between floating-point and MMX technology states.	
CCH	02H	FP_MMX_TRANS.TO_MMX	Counts the first MMX instruction following a floating-point instruction. You can use this event to estimate the penalties for the transitions between floating-point and MMX technology states.	
CCH	03H	FP_MMX_TRANS.ANY	Counts all transitions from floating point to MMX instructions and from MMX instructions to floating point instructions. You can use this event to estimate the penalties for the transitions between floating-point and MMX technology states.	
D0H	01H	MACRO_INSTS.DECODED	Counts the number of instructions decoded, (but not necessarily executed or retired).	
D1H	02H	UOPS_DECODED.MS	Counts the number of Uops decoded by the Microcode Sequencer, MS. The MS delivers uops when the instruction is more than 4 uops long or a microcode assist is occurring.	

Table 19-19. Non-Architectural Performance Events In the Processor Core for Intel® Core™ i7 Processor and Intel® Xeon® Processor 5500 Series (Contd.)

Event Num.	Umask Value	Event Mask Mnemonic	Description	Comment
D1H	04H	UOPS_DECODED.ESP_FOLDING	Counts number of stack pointer (ESP) instructions decoded: push, pop, call, ret, etc. ESP instructions do not generate a Uop to increment or decrement ESP. Instead, they update an ESP_Offset register that keeps track of the delta to the current value of the ESP register.	
D1H	08H	UOPS_DECODED.ESP_SYNC	Counts number of stack pointer (ESP) sync operations where an ESP instruction is corrected by adding the ESP offset register to the current value of the ESP register.	
D2H	01H	RAT_STALLS.FLAGS	Counts the number of cycles during which execution stalled due to several reasons, one of which is a partial flag register stall. A partial register stall may occur when two conditions are met: 1) an instruction modifies some, but not all, of the flags in the flag register and 2) the next instruction, which depends on flags, depends on flags that were not modified by this instruction.	
D2H	02H	RAT_STALLS.REGISTERS	This event counts the number of cycles instruction execution latency became longer than the defined latency because the instruction used a register that was partially written by previous instruction.	
D2H	04H	RAT_STALLS.ROB_READ_PORT	Counts the number of cycles when ROB read port stalls occurred, which did not allow new micro-ops to enter the out-of-order pipeline. Note that, at this stage in the pipeline, additional stalls may occur at the same cycle and prevent the stalled micro-ops from entering the pipe. In such a case, micro-ops retry entering the execution pipe in the next cycle and the ROB-read port stall is counted again.	
D2H	08H	RAT_STALLS.SCOREBOARD	Counts the cycles where we stall due to microarchitecturally required serialization. Microcode scoreboarding stalls.	
D2H	0FH	RAT_STALLS.ANY	Counts all Register Allocation Table stall cycles due to: Cycles when ROB read port stalls occurred, which did not allow new micro-ops to enter the execution pipe. Cycles when partial register stalls occurred. Cycles when flag stalls occurred. Cycles floating-point unit (FPU) status word stalls occurred. To count each of these conditions separately use the events: RAT_STALLS.ROB_READ_PORT, RAT_STALLS.PARTIAL, RAT_STALLS.FLAGS, and RAT_STALLS.FPSW.	
D4H	01H	SEG_RENAME_STALLS	Counts the number of stall cycles due to the lack of renaming resources for the ES, DS, FS, and GS segment registers. If a segment is renamed but not retired and a second update to the same segment occurs, a stall occurs in the front end of the pipeline until the renamed segment retires.	
D5H	01H	ES_REG_RENAMES	Counts the number of times the ES segment register is renamed.	

Table 19-19. Non-Architectural Performance Events In the Processor Core for Intel® Core™ i7 Processor and Intel® Xeon® Processor 5500 Series (Contd.)

Event Num.	Umask Value	Event Mask Mnemonic	Description	Comment
DBH	01H	UOP_UNFUSION	Counts unfusion events due to floating-point exception to a fused uop.	
E0H	01H	BR_INST_DECODED	Counts the number of branch instructions decoded.	
E5H	01H	BPU_MISSED_CALL_RET	Counts number of times the Branch Prediction Unit missed predicting a call or return branch.	
E6H	01H	BACLEAR.CLEAR	Counts the number of times the front end is resteered, mainly when the Branch Prediction Unit cannot provide a correct prediction and this is corrected by the Branch Address Calculator at the front end. This can occur if the code has many branches such that they cannot be consumed by the BPU. Each BACLEAR asserted by the BAC generates approximately an 8 cycle bubble in the instruction fetch pipeline. The effect on total execution time depends on the surrounding code.	
E6H	02H	BACLEAR.BAD_TARGET	Counts number of Branch Address Calculator clears (BACLEAR) asserted due to conditional branch instructions in which there was a target hit but the direction was wrong. Each BACLEAR asserted by the BAC generates approximately an 8 cycle bubble in the instruction fetch pipeline.	
E8H	01H	BPU_CLEARS.EARLY	Counts early (normal) Branch Prediction Unit clears: BPU predicted a taken branch after incorrectly assuming that it was not taken.	The BPU clear leads to 2 cycle bubble in the front end.
E8H	02H	BPU_CLEARS.LATE	Counts late Branch Prediction Unit clears due to Most Recently Used conflicts. The PBU clear leads to a 3 cycle bubble in the front end.	
F0H	01H	L2_TRANSACTIONS.LOAD	Counts L2 load operations due to HW prefetch or demand loads.	
F0H	02H	L2_TRANSACTIONS.RFO	Counts L2 RFO operations due to HW prefetch or demand RFOs.	
F0H	04H	L2_TRANSACTIONS.IFETCH	Counts L2 instruction fetch operations due to HW prefetch or demand ifetch.	
F0H	08H	L2_TRANSACTIONS.PREFETCH	Counts L2 prefetch operations.	
F0H	10H	L2_TRANSACTIONS.L1D_WB	Counts L1D writeback operations to the L2.	
F0H	20H	L2_TRANSACTIONS.FILL	Counts L2 cache line fill operations due to load, RFO, L1D writeback or prefetch.	
F0H	40H	L2_TRANSACTIONS.WB	Counts L2 writeback operations to the L3.	
F0H	80H	L2_TRANSACTIONS.ANY	Counts all L2 cache operations.	
F1H	02H	L2_LINES_IN.S_STATE	Counts the number of cache lines allocated in the L2 cache in the S (shared) state.	
F1H	04H	L2_LINES_IN.E_STATE	Counts the number of cache lines allocated in the L2 cache in the E (exclusive) state.	
F1H	07H	L2_LINES_IN.ANY	Counts the number of cache lines allocated in the L2 cache.	
F2H	01H	L2_LINES_OUT.DEMAND_CLEAN	Counts L2 clean cache lines evicted by a demand request.	

Table 19-19. Non-Architectural Performance Events In the Processor Core for Intel® Core™ i7 Processor and Intel® Xeon® Processor 5500 Series (Contd.)

Event Num.	Umask Value	Event Mask Mnemonic	Description	Comment
F2H	02H	L2_LINES_OUT.DEMAND_DIRTY	Counts L2 dirty (modified) cache lines evicted by a demand request.	
F2H	04H	L2_LINES_OUT.PREFETCH_CLEAN	Counts L2 clean cache line evicted by a prefetch request.	
F2H	08H	L2_LINES_OUT.PREFETCH_DIRTY	Counts L2 modified cache line evicted by a prefetch request.	
F2H	0FH	L2_LINES_OUT.ANY	Counts all L2 cache lines evicted for any reason.	
F4H	10H	SQ_MISC.SPLIT_LOCK	Counts the number of SQ lock splits across a cache line.	
F6H	01H	SQ_FULL_STALL_CYCLES	Counts cycles the Super Queue is full. Neither of the threads on this core will be able to access the uncore.	
F7H	01H	FP_ASSIST.ALL	Counts the number of floating point operations executed that required micro-code assist intervention. Assists are required in the following cases: SSE instructions (denormal input when the DAZ flag is off or underflow result when the FTZ flag is off); x87 instructions (NaN or denormal are loaded to a register or used as input from memory, division by 0 or underflow output).	
F7H	02H	FP_ASSIST.OUTPUT	Counts number of floating point micro-code assist when the output value (destination register) is invalid.	
F7H	04H	FP_ASSIST.INPUT	Counts number of floating point micro-code assist when the input value (one of the source operands to an FP instruction) is invalid.	
FDH	01H	SIMD_INT_64.PACKED_MPY	Counts number of SID integer 64 bit packed multiply operations.	
FDH	02H	SIMD_INT_64.PACKED_SHIFT	Counts number of SID integer 64 bit packed shift operations.	
FDH	04H	SIMD_INT_64.PACK	Counts number of SID integer 64 bit pack operations.	
FDH	08H	SIMD_INT_64.UNPACK	Counts number of SID integer 64 bit unpack operations.	
FDH	10H	SIMD_INT_64.PACKED_LOGICAL	Counts number of SID integer 64 bit logical operations.	
FDH	20H	SIMD_INT_64.PACKED_ARITH	Counts number of SID integer 64 bit arithmetic operations.	
FDH	40H	SIMD_INT_64.SHUFFLE_MOVE	Counts number of SID integer 64 bit shift or move operations.	

Non-architectural performance monitoring events that are located in the uncore sub-system are implementation specific between different platforms using processors based on Intel microarchitecture code name Nehalem. Processors with CPUID signature of DisplayFamily_DisplayModel 06_1AH, 06_1EH, and 06_1FH support performance events listed in Table 19-20.

Table 19-20. Non-Architectural Performance Events In the Processor Uncore for Intel® Core™ i7 Processor and Intel® Xeon® Processor 5500 Series

Event Num.	Umask Value	Event Mask Mnemonic	Description	Comment
00H	01H	UNC_GQ_CYCLES_FULL.READ_TRACKER	Uncore cycles Global Queue read tracker is full.	
00H	02H	UNC_GQ_CYCLES_FULL.WRITE_TRACKER	Uncore cycles Global Queue write tracker is full.	
00H	04H	UNC_GQ_CYCLES_FULL.PEER_PROBE_TRACKER	Uncore cycles Global Queue peer probe tracker is full. The peer probe tracker queue tracks snoops from the IOH and remote sockets.	
01H	01H	UNC_GQ_CYCLES_NOT_EMPTY.READ_TRACKER	Uncore cycles were Global Queue read tracker has at least one valid entry.	
01H	02H	UNC_GQ_CYCLES_NOT_EMPTY.WRITE_TRACKER	Uncore cycles were Global Queue write tracker has at least one valid entry.	
01H	04H	UNC_GQ_CYCLES_NOT_EMPTY.PEER_PROBE_TRACKER	Uncore cycles were Global Queue peer probe tracker has at least one valid entry. The peer probe tracker queue tracks IOH and remote socket snoops.	
03H	01H	UNC_GQ_ALLOC.READ_TRACKER	Counts the number of tread tracker allocate to deallocate entries. The GQ read tracker allocate to deallocate occupancy count is divided by the count to obtain the average read tracker latency.	
03H	02H	UNC_GQ_ALLOC.RT_L3_MISS	Counts the number GQ read tracker entries for which a full cache line read has missed the L3. The GQ read tracker L3 miss to fill occupancy count is divided by this count to obtain the average cache line read L3 miss latency. The latency represents the time after which the L3 has determined that the cache line has missed. The time between a GQ read tracker allocation and the L3 determining that the cache line has missed is the average L3 hit latency. The total L3 cache line read miss latency is the hit latency + L3 miss latency.	
03H	04H	UNC_GQ_ALLOC.RT_TO_L3_RESP	Counts the number of GQ read tracker entries that are allocated in the read tracker queue that hit or miss the L3. The GQ read tracker L3 hit occupancy count is divided by this count to obtain the average L3 hit latency.	
03H	08H	UNC_GQ_ALLOC.RT_TO_RTID_ACQUIRED	Counts the number of GQ read tracker entries that are allocated in the read tracker, have missed in the L3 and have not acquired a Request Transaction ID. The GQ read tracker L3 miss to RTID acquired occupancy count is divided by this count to obtain the average latency for a read L3 miss to acquire an RTID.	
03H	10H	UNC_GQ_ALLOC.WT_TO_RTID_ACQUIRED	Counts the number of GQ write tracker entries that are allocated in the write tracker, have missed in the L3 and have not acquired a Request Transaction ID. The GQ write tracker L3 miss to RTID occupancy count is divided by this count to obtain the average latency for a write L3 miss to acquire an RTID.	
03H	20H	UNC_GQ_ALLOC.WRITE_TRACKER	Counts the number of GQ write tracker entries that are allocated in the write tracker queue that miss the L3. The GQ write tracker occupancy count is divided by this count to obtain the average L3 write miss latency.	

Table 19-20. Non-Architectural Performance Events In the Processor Uncore for Intel® Core™ i7 Processor and Intel® Xeon® Processor 5500 Series (Contd.)

Event Num.	Umask Value	Event Mask Mnemonic	Description	Comment
03H	40H	UNC_GQ_ALLOC.PEER_PROBE_TRACKER	Counts the number of GQ peer probe tracker (snoop) entries that are allocated in the peer probe tracker queue that miss the L3. The GQ peer probe occupancy count is divided by this count to obtain the average L3 peer probe miss latency.	
04H	01H	UNC_GQ_DATA.FROM_QPI	Cycles Global Queue Quickpath Interface input data port is busy importing data from the Quickpath Interface. Each cycle the input port can transfer 8 or 16 bytes of data.	
04H	02H	UNC_GQ_DATA.FROM_QMC	Cycles Global Queue Quickpath Memory Interface input data port is busy importing data from the Quickpath Memory Interface. Each cycle the input port can transfer 8 or 16 bytes of data.	
04H	04H	UNC_GQ_DATA.FROM_L3	Cycles GQ L3 input data port is busy importing data from the Last Level Cache. Each cycle the input port can transfer 32 bytes of data.	
04H	08H	UNC_GQ_DATA.FROM_CORES_02	Cycles GQ Core 0 and 2 input data port is busy importing data from processor cores 0 and 2. Each cycle the input port can transfer 32 bytes of data.	
04H	10H	UNC_GQ_DATA.FROM_CORES_13	Cycles GQ Core 1 and 3 input data port is busy importing data from processor cores 1 and 3. Each cycle the input port can transfer 32 bytes of data.	
05H	01H	UNC_GQ_DATA.TO_QPI_QMC	Cycles GQ QPI and QMC output data port is busy sending data to the Quickpath Interface or Quickpath Memory Interface. Each cycle the output port can transfer 32 bytes of data.	
05H	02H	UNC_GQ_DATA.TO_L3	Cycles GQ L3 output data port is busy sending data to the Last Level Cache. Each cycle the output port can transfer 32 bytes of data.	
05H	04H	UNC_GQ_DATA.TO_CORES	Cycles GQ Core output data port is busy sending data to the Cores. Each cycle the output port can transfer 32 bytes of data.	
06H	01H	UNC_SNP_RESP_TO_LOCAL_HOME.I_STATE	Number of snoop responses to the local home that L3 does not have the referenced cache line.	
06H	02H	UNC_SNP_RESP_TO_LOCAL_HOME.S_STATE	Number of snoop responses to the local home that L3 has the referenced line cached in the S state.	
06H	04H	UNC_SNP_RESP_TO_LOCAL_HOME.FWD_S_STATE	Number of responses to code or data read snoops to the local home that the L3 has the referenced cache line in the E state. The L3 cache line state is changed to the S state and the line is forwarded to the local home in the S state.	
06H	08H	UNC_SNP_RESP_TO_LOCAL_HOME.FWD_I_STATE	Number of responses to read invalidate snoops to the local home that the L3 has the referenced cache line in the M state. The L3 cache line state is invalidated and the line is forwarded to the local home in the M state.	
06H	10H	UNC_SNP_RESP_TO_LOCAL_HOME.CONFLICT	Number of conflict snoop responses sent to the local home.	

Table 19-20. Non-Architectural Performance Events In the Processor Uncore for Intel® Core™ i7 Processor and Intel® Xeon® Processor 5500 Series (Contd.)

Event Num.	Umask Value	Event Mask Mnemonic	Description	Comment
06H	20H	UNC_SNP_RESP_TO_LOCAL_HOME.WB	Number of responses to code or data read snoops to the local home that the L3 has the referenced line cached in the M state.	
07H	01H	UNC_SNP_RESP_TO_REMOTE_HOME.I_STATE	Number of snoop responses to a remote home that L3 does not have the referenced cache line.	
07H	02H	UNC_SNP_RESP_TO_REMOTE_HOME.S_STATE	Number of snoop responses to a remote home that L3 has the referenced line cached in the S state.	
07H	04H	UNC_SNP_RESP_TO_REMOTE_HOME.FWD_S_STATE	Number of responses to code or data read snoops to a remote home that the L3 has the referenced cache line in the E state. The L3 cache line state is changed to the S state and the line is forwarded to the remote home in the S state.	
07H	08H	UNC_SNP_RESP_TO_REMOTE_HOME.FWD_I_STATE	Number of responses to read invalidate snoops to a remote home that the L3 has the referenced cache line in the M state. The L3 cache line state is invalidated and the line is forwarded to the remote home in the M state.	
07H	10H	UNC_SNP_RESP_TO_REMOTE_HOME.CONFLICT	Number of conflict snoop responses sent to the local home.	
07H	20H	UNC_SNP_RESP_TO_REMOTE_HOME.WB	Number of responses to code or data read snoops to a remote home that the L3 has the referenced line cached in the M state.	
07H	24H	UNC_SNP_RESP_TO_REMOTE_HOME.HITM	Number of HITM snoop responses to a remote home.	
08H	01H	UNC_L3_HITS.READ	Number of code read, data read and RFO requests that hit in the L3.	
08H	02H	UNC_L3_HITS.WRITE	Number of writeback requests that hit in the L3. Writebacks from the cores will always result in L3 hits due to the inclusive property of the L3.	
08H	04H	UNC_L3_HITS.PROBE	Number of snoops from IOH or remote sockets that hit in the L3.	
08H	03H	UNC_L3_HITS.ANY	Number of reads and writes that hit the L3.	
09H	01H	UNC_L3_MISS.READ	Number of code read, data read and RFO requests that miss the L3.	
09H	02H	UNC_L3_MISS.WRITE	Number of writeback requests that miss the L3. Should always be zero as writebacks from the cores will always result in L3 hits due to the inclusive property of the L3.	
09H	04H	UNC_L3_MISS.PROBE	Number of snoops from IOH or remote sockets that miss the L3.	
09H	03H	UNC_L3_MISS.ANY	Number of reads and writes that miss the L3.	
0AH	01H	UNC_L3_LINES_IN.M_STATE	Counts the number of L3 lines allocated in M state. The only time a cache line is allocated in the M state is when the line was forwarded in M state is forwarded due to a Snoop Read Invalidate Own request.	
0AH	02H	UNC_L3_LINES_IN.E_STATE	Counts the number of L3 lines allocated in E state.	
0AH	04H	UNC_L3_LINES_IN.S_STATE	Counts the number of L3 lines allocated in S state.	

Table 19-20. Non-Architectural Performance Events In the Processor Uncore for Intel® Core™ i7 Processor and Intel® Xeon® Processor 5500 Series (Contd.)

Event Num.	Umask Value	Event Mask Mnemonic	Description	Comment
0AH	08H	UNC_L3_LINES_IN.F_STATE	Counts the number of L3 lines allocated in F state.	
0AH	0FH	UNC_L3_LINES_IN.ANY	Counts the number of L3 lines allocated in any state.	
0BH	01H	UNC_L3_LINES_OUT.M_STATE	Counts the number of L3 lines victimized that were in the M state. When the victim cache line is in M state, the line is written to its home cache agent which can be either local or remote.	
0BH	02H	UNC_L3_LINES_OUT.E_STATE	Counts the number of L3 lines victimized that were in the E state.	
0BH	04H	UNC_L3_LINES_OUT.S_STATE	Counts the number of L3 lines victimized that were in the S state.	
0BH	08H	UNC_L3_LINES_OUT.I_STATE	Counts the number of L3 lines victimized that were in the I state.	
0BH	10H	UNC_L3_LINES_OUT.F_STATE	Counts the number of L3 lines victimized that were in the F state.	
0BH	1FH	UNC_L3_LINES_OUT.ANY	Counts the number of L3 lines victimized in any state.	
20H	01H	UNC_QHL_REQUESTS.IOH_READS	Counts number of Quickpath Home Logic read requests from the IOH.	
20H	02H	UNC_QHL_REQUESTS.IOH_WRITES	Counts number of Quickpath Home Logic write requests from the IOH.	
20H	04H	UNC_QHL_REQUESTS.REMOTE_READS	Counts number of Quickpath Home Logic read requests from a remote socket.	
20H	08H	UNC_QHL_REQUESTS.REMOTE_WRITES	Counts number of Quickpath Home Logic write requests from a remote socket.	
20H	10H	UNC_QHL_REQUESTS.LOCAL_READS	Counts number of Quickpath Home Logic read requests from the local socket.	
20H	20H	UNC_QHL_REQUESTS.LOCAL_WRITES	Counts number of Quickpath Home Logic write requests from the local socket.	
21H	01H	UNC_QHL_CYCLES_FULL.IOH	Counts uclk cycles all entries in the Quickpath Home Logic IOH are full.	
21H	02H	UNC_QHL_CYCLES_FULL.REMOTE	Counts uclk cycles all entries in the Quickpath Home Logic remote tracker are full.	
21H	04H	UNC_QHL_CYCLES_FULL.LOCAL	Counts uclk cycles all entries in the Quickpath Home Logic local tracker are full.	
22H	01H	UNC_QHL_CYCLES_NOT_EMPTY.IOH	Counts uclk cycles all entries in the Quickpath Home Logic IOH is busy.	
22H	02H	UNC_QHL_CYCLES_NOT_EMPTY.REMOTE	Counts uclk cycles all entries in the Quickpath Home Logic remote tracker is busy.	
22H	04H	UNC_QHL_CYCLES_NOT_EMPTY.LOCAL	Counts uclk cycles all entries in the Quickpath Home Logic local tracker is busy.	
23H	01H	UNC_QHL_OCCUPANCY.IOH	QHL IOH tracker allocate to deallocate read occupancy.	
23H	02H	UNC_QHL_OCCUPANCY.REMOTE	QHL remote tracker allocate to deallocate read occupancy.	
23H	04H	UNC_QHL_OCCUPANCY.LOCAL	QHL local tracker allocate to deallocate read occupancy.	

Table 19-20. Non-Architectural Performance Events In the Processor Uncore for Intel® Core™ i7 Processor and Intel® Xeon® Processor 5500 Series (Contd.)

Event Num.	Umask Value	Event Mask Mnemonic	Description	Comment
24H	02H	UNC_QHL_ADDRESS_CONFLICTS.2WAY	Counts number of QHL Active Address Table (AAT) entries that saw a max of 2 conflicts. The AAT is a structure that tracks requests that are in conflict. The requests themselves are in the home tracker entries. The count is reported when an AAT entry deallocates.	
24H	04H	UNC_QHL_ADDRESS_CONFLICTS.3WAY	Counts number of QHL Active Address Table (AAT) entries that saw a max of 3 conflicts. The AAT is a structure that tracks requests that are in conflict. The requests themselves are in the home tracker entries. The count is reported when an AAT entry deallocates.	
25H	01H	UNC_QHL_CONFLICT_CYCLES.IOH	Counts cycles the Quickpath Home Logic IOH Tracker contains two or more requests with an address conflict. A max of 3 requests can be in conflict.	
25H	02H	UNC_QHL_CONFLICT_CYCLES.REMOTE	Counts cycles the Quickpath Home Logic Remote Tracker contains two or more requests with an address conflict. A max of 3 requests can be in conflict.	
25H	04H	UNC_QHL_CONFLICT_CYCLES.LOCAL	Counts cycles the Quickpath Home Logic Local Tracker contains two or more requests with an address conflict. A max of 3 requests can be in conflict.	
26H	01H	UNC_QHL_TO_QMC_BYPASS	Counts number or requests to the Quickpath Memory Controller that bypass the Quickpath Home Logic. All local accesses can be bypassed. For remote requests, only read requests can be bypassed.	
27H	01H	UNC_QMC_NORMAL_FULL.READ.CH0	Uncore cycles all the entries in the DRAM channel 0 medium or low priority queue are occupied with read requests.	
27H	02H	UNC_QMC_NORMAL_FULL.READ.CH1	Uncore cycles all the entries in the DRAM channel 1 medium or low priority queue are occupied with read requests.	
27H	04H	UNC_QMC_NORMAL_FULL.READ.CH2	Uncore cycles all the entries in the DRAM channel 2 medium or low priority queue are occupied with read requests.	
27H	08H	UNC_QMC_NORMAL_FULL.WRITE.CH0	Uncore cycles all the entries in the DRAM channel 0 medium or low priority queue are occupied with write requests.	
27H	10H	UNC_QMC_NORMAL_FULL.WRITE.CH1	Counts cycles all the entries in the DRAM channel 1 medium or low priority queue are occupied with write requests.	
27H	20H	UNC_QMC_NORMAL_FULL.WRITE.CH2	Uncore cycles all the entries in the DRAM channel 2 medium or low priority queue are occupied with write requests.	
28H	01H	UNC_QMC_ISOC_FULL.READ.CH0	Counts cycles all the entries in the DRAM channel 0 high priority queue are occupied with isochronous read requests.	
28H	02H	UNC_QMC_ISOC_FULL.READ.CH1	Counts cycles all the entries in the DRAM channel 1high priority queue are occupied with isochronous read requests.	

Table 19-20. Non-Architectural Performance Events In the Processor Uncore for Intel® Core™ i7 Processor and Intel® Xeon® Processor 5500 Series (Contd.)

Event Num.	Umask Value	Event Mask Mnemonic	Description	Comment
28H	04H	UNC_QMC_ISOC_FULL.READ.CH2	Counts cycles all the entries in the DRAM channel 2 high priority queue are occupied with isochronous read requests.	
28H	08H	UNC_QMC_ISOC_FULL.WRITE.CH0	Counts cycles all the entries in the DRAM channel 0 high priority queue are occupied with isochronous write requests.	
28H	10H	UNC_QMC_ISOC_FULL.WRITE.CH1	Counts cycles all the entries in the DRAM channel 1 high priority queue are occupied with isochronous write requests.	
28H	20H	UNC_QMC_ISOC_FULL.WRITE.CH2	Counts cycles all the entries in the DRAM channel 2 high priority queue are occupied with isochronous write requests.	
29H	01H	UNC_QMC_BUSY.READ.CH0	Counts cycles where Quickpath Memory Controller has at least 1 outstanding read request to DRAM channel 0.	
29H	02H	UNC_QMC_BUSY.READ.CH1	Counts cycles where Quickpath Memory Controller has at least 1 outstanding read request to DRAM channel 1.	
29H	04H	UNC_QMC_BUSY.READ.CH2	Counts cycles where Quickpath Memory Controller has at least 1 outstanding read request to DRAM channel 2.	
29H	08H	UNC_QMC_BUSY.WRITE.CH0	Counts cycles where Quickpath Memory Controller has at least 1 outstanding write request to DRAM channel 0.	
29H	10H	UNC_QMC_BUSY.WRITE.CH1	Counts cycles where Quickpath Memory Controller has at least 1 outstanding write request to DRAM channel 1.	
29H	20H	UNC_QMC_BUSY.WRITE.CH2	Counts cycles where Quickpath Memory Controller has at least 1 outstanding write request to DRAM channel 2.	
2AH	01H	UNC_QMC_OCCUPANCY.CH0	IMC channel 0 normal read request occupancy.	
2AH	02H	UNC_QMC_OCCUPANCY.CH1	IMC channel 1 normal read request occupancy.	
2AH	04H	UNC_QMC_OCCUPANCY.CH2	IMC channel 2 normal read request occupancy.	
2BH	01H	UNC_QMC_ISSOC_OCCUPANCY.CH0	IMC channel 0 issoc read request occupancy.	
2BH	02H	UNC_QMC_ISSOC_OCCUPANCY.CH1	IMC channel 1 issoc read request occupancy.	
2BH	04H	UNC_QMC_ISSOC_OCCUPANCY.CH2	IMC channel 2 issoc read request occupancy.	
2BH	07H	UNC_QMC_ISSOC_READS.ANY	IMC issoc read request occupancy.	
2CH	01H	UNC_QMC_NORMAL_READS.CH0	Counts the number of Quickpath Memory Controller channel 0 medium and low priority read requests. The QMC channel 0 normal read occupancy divided by this count provides the average QMC channel 0 read latency.	

Table 19-20. Non-Architectural Performance Events In the Processor Uncore for Intel® Core™ i7 Processor and Intel® Xeon® Processor 5500 Series (Contd.)

Event Num.	Umask Value	Event Mask Mnemonic	Description	Comment
2CH	02H	UNC_QMC_NORMAL_READS.CH1	Counts the number of Quickpath Memory Controller channel 1 medium and low priority read requests. The QMC channel 1 normal read occupancy divided by this count provides the average QMC channel 1 read latency.	
2CH	04H	UNC_QMC_NORMAL_READS.CH2	Counts the number of Quickpath Memory Controller channel 2 medium and low priority read requests. The QMC channel 2 normal read occupancy divided by this count provides the average QMC channel 2 read latency.	
2CH	07H	UNC_QMC_NORMAL_READS.ANY	Counts the number of Quickpath Memory Controller medium and low priority read requests. The QMC normal read occupancy divided by this count provides the average QMC read latency.	
2DH	01H	UNC_QMC_HIGH_PRIORITY_READS.CH0	Counts the number of Quickpath Memory Controller channel 0 high priority isochronous read requests.	
2DH	02H	UNC_QMC_HIGH_PRIORITY_READS.CH1	Counts the number of Quickpath Memory Controller channel 1 high priority isochronous read requests.	
2DH	04H	UNC_QMC_HIGH_PRIORITY_READS.CH2	Counts the number of Quickpath Memory Controller channel 2 high priority isochronous read requests.	
2DH	07H	UNC_QMC_HIGH_PRIORITY_READS.ANY	Counts the number of Quickpath Memory Controller high priority isochronous read requests.	
2EH	01H	UNC_QMC_CRITICAL_PRIORITY_READS.CH0	Counts the number of Quickpath Memory Controller channel 0 critical priority isochronous read requests.	
2EH	02H	UNC_QMC_CRITICAL_PRIORITY_READS.CH1	Counts the number of Quickpath Memory Controller channel 1 critical priority isochronous read requests.	
2EH	04H	UNC_QMC_CRITICAL_PRIORITY_READS.CH2	Counts the number of Quickpath Memory Controller channel 2 critical priority isochronous read requests.	
2EH	07H	UNC_QMC_CRITICAL_PRIORITY_READS.ANY	Counts the number of Quickpath Memory Controller critical priority isochronous read requests.	
2FH	01H	UNC_QMC_WRITES.FULL.CH0	Counts number of full cache line writes to DRAM channel 0.	
2FH	02H	UNC_QMC_WRITES.FULL.CH1	Counts number of full cache line writes to DRAM channel 1.	
2FH	04H	UNC_QMC_WRITES.FULL.CH2	Counts number of full cache line writes to DRAM channel 2.	
2FH	07H	UNC_QMC_WRITES.FULL.ANY	Counts number of full cache line writes to DRAM.	
2FH	08H	UNC_QMC_WRITES.PARTIAL.CH0	Counts number of partial cache line writes to DRAM channel 0.	
2FH	10H	UNC_QMC_WRITES.PARTIAL.CH1	Counts number of partial cache line writes to DRAM channel 1.	
2FH	20H	UNC_QMC_WRITES.PARTIAL.CH2	Counts number of partial cache line writes to DRAM channel 2.	
2FH	38H	UNC_QMC_WRITES.PARTIAL.ANY	Counts number of partial cache line writes to DRAM.	
30H	01H	UNC_QMC_CANCEL.CH0	Counts number of DRAM channel 0 cancel requests.	
30H	02H	UNC_QMC_CANCEL.CH1	Counts number of DRAM channel 1 cancel requests.	

Table 19-20. Non-Architectural Performance Events In the Processor Uncore for Intel® Core™ i7 Processor and Intel® Xeon® Processor 5500 Series (Contd.)

Event Num.	Umask Value	Event Mask Mnemonic	Description	Comment
30H	04H	UNC_QMC_CANCEL.CH2	Counts number of DRAM channel 2 cancel requests.	
30H	07H	UNC_QMC_CANCEL.ANY	Counts number of DRAM cancel requests.	
31H	01H	UNC_QMC_PRIORITY_UPDATES.CH0	Counts number of DRAM channel 0 priority updates. A priority update occurs when an ISOC high or critical request is received by the QHL and there is a matching request with normal priority that has already been issued to the QMC. In this instance, the QHL will send a priority update to QMC to expedite the request.	
31H	02H	UNC_QMC_PRIORITY_UPDATES.CH1	Counts number of DRAM channel 1 priority updates. A priority update occurs when an ISOC high or critical request is received by the QHL and there is a matching request with normal priority that has already been issued to the QMC. In this instance, the QHL will send a priority update to QMC to expedite the request.	
31H	04H	UNC_QMC_PRIORITY_UPDATES.CH2	Counts number of DRAM channel 2 priority updates. A priority update occurs when an ISOC high or critical request is received by the QHL and there is a matching request with normal priority that has already been issued to the QMC. In this instance, the QHL will send a priority update to QMC to expedite the request.	
31H	07H	UNC_QMC_PRIORITY_UPDATES.ANY	Counts number of DRAM priority updates. A priority update occurs when an ISOC high or critical request is received by the QHL and there is a matching request with normal priority that has already been issued to the QMC. In this instance, the QHL will send a priority update to QMC to expedite the request.	
33H	04H	UNC_QHL_FRC_ACK_CNFLTS.LOCAL	Counts number of Force Acknowledge Conflict messages sent by the Quickpath Home Logic to the local home.	
40H	01H	UNC_QPI_TX_STALLED_SINGLE_FLIT.HOME.LINK_0	Counts cycles the Quickpath outbound link 0 HOME virtual channel is stalled due to lack of a VNA and VN0 credit. Note that this event does not filter out when a flit would not have been selected for arbitration because another virtual channel is getting arbitrated.	
40H	02H	UNC_QPI_TX_STALLED_SINGLE_FLIT.SNOOP.LINK_0	Counts cycles the Quickpath outbound link 0 SNOOP virtual channel is stalled due to lack of a VNA and VN0 credit. Note that this event does not filter out when a flit would not have been selected for arbitration because another virtual channel is getting arbitrated.	
40H	04H	UNC_QPI_TX_STALLED_SINGLE_FLIT.NDR.LINK_0	Counts cycles the Quickpath outbound link 0 non-data response virtual channel is stalled due to lack of a VNA and VN0 credit. Note that this event does not filter out when a flit would not have been selected for arbitration because another virtual channel is getting arbitrated.	
40H	08H	UNC_QPI_TX_STALLED_SINGLE_FLIT.HOME.LINK_1	Counts cycles the Quickpath outbound link 1 HOME virtual channel is stalled due to lack of a VNA and VN0 credit. Note that this event does not filter out when a flit would not have been selected for arbitration because another virtual channel is getting arbitrated.	

Table 19-20. Non-Architectural Performance Events In the Processor Uncore for Intel® Core™ i7 Processor and Intel® Xeon® Processor 5500 Series (Contd.)

Event Num.	Umask Value	Event Mask Mnemonic	Description	Comment
40H	10H	UNC_QPI_TX_STALLED_SINGLE_FLIT.SNOOP.LINK_1	Counts cycles the Quickpath outbound link 1 SNOOP virtual channel is stalled due to lack of a VNA and VN0 credit. Note that this event does not filter out when a flit would not have been selected for arbitration because another virtual channel is getting arbitrated.	
40H	20H	UNC_QPI_TX_STALLED_SINGLE_FLIT.NDR.LINK_1	Counts cycles the Quickpath outbound link 1 non-data response virtual channel is stalled due to lack of a VNA and VN0 credit. Note that this event does not filter out when a flit would not have been selected for arbitration because another virtual channel is getting arbitrated.	
40H	07H	UNC_QPI_TX_STALLED_SINGLE_FLIT.LINK_0	Counts cycles the Quickpath outbound link 0 virtual channels are stalled due to lack of a VNA and VN0 credit. Note that this event does not filter out when a flit would not have been selected for arbitration because another virtual channel is getting arbitrated.	
40H	38H	UNC_QPI_TX_STALLED_SINGLE_FLIT.LINK_1	Counts cycles the Quickpath outbound link 1 virtual channels are stalled due to lack of a VNA and VN0 credit. Note that this event does not filter out when a flit would not have been selected for arbitration because another virtual channel is getting arbitrated.	
41H	01H	UNC_QPI_TX_STALLED_MULTI_FLIT.DRS.LINK_0	Counts cycles the Quickpath outbound link 0 Data Response virtual channel is stalled due to lack of VNA and VN0 credits. Note that this event does not filter out when a flit would not have been selected for arbitration because another virtual channel is getting arbitrated.	
41H	02H	UNC_QPI_TX_STALLED_MULTI_FLIT.NCB.LINK_0	Counts cycles the Quickpath outbound link 0 Non-Coherent Bypass virtual channel is stalled due to lack of VNA and VN0 credits. Note that this event does not filter out when a flit would not have been selected for arbitration because another virtual channel is getting arbitrated.	
41H	04H	UNC_QPI_TX_STALLED_MULTI_FLIT.NCS.LINK_0	Counts cycles the Quickpath outbound link 0 Non-Coherent Standard virtual channel is stalled due to lack of VNA and VN0 credits. Note that this event does not filter out when a flit would not have been selected for arbitration because another virtual channel is getting arbitrated.	
41H	08H	UNC_QPI_TX_STALLED_MULTI_FLIT.DRS.LINK_1	Counts cycles the Quickpath outbound link 1 Data Response virtual channel is stalled due to lack of VNA and VN0 credits. Note that this event does not filter out when a flit would not have been selected for arbitration because another virtual channel is getting arbitrated.	
41H	10H	UNC_QPI_TX_STALLED_MULTI_FLIT.NCB.LINK_1	Counts cycles the Quickpath outbound link 1 Non-Coherent Bypass virtual channel is stalled due to lack of VNA and VN0 credits. Note that this event does not filter out when a flit would not have been selected for arbitration because another virtual channel is getting arbitrated.	

Table 19-20. Non-Architectural Performance Events In the Processor Uncore for Intel® Core™ i7 Processor and Intel® Xeon® Processor 5500 Series (Contd.)

Event Num.	Umask Value	Event Mask Mnemonic	Description	Comment
41H	20H	UNC_QPI_TX_STALLED_MULTI_FLIT.NCS.LINK_1	Counts cycles the Quickpath outbound link 1 Non-Coherent Standard virtual channel is stalled due to lack of VNA and VN0 credits. Note that this event does not filter out when a flit would not have been selected for arbitration because another virtual channel is getting arbitrated.	
41H	07H	UNC_QPI_TX_STALLED_MULTI_FLIT.LINK_0	Counts cycles the Quickpath outbound link 0 virtual channels are stalled due to lack of VNA and VN0 credits. Note that this event does not filter out when a flit would not have been selected for arbitration because another virtual channel is getting arbitrated.	
41H	38H	UNC_QPI_TX_STALLED_MULTI_FLIT.LINK_1	Counts cycles the Quickpath outbound link 1 virtual channels are stalled due to lack of VNA and VN0 credits. Note that this event does not filter out when a flit would not have been selected for arbitration because another virtual channel is getting arbitrated.	
42H	02H	UNC_QPI_TX_HEADER.BUSY.LINK_0	Number of cycles that the header buffer in the Quickpath Interface outbound link 0 is busy.	
42H	08H	UNC_QPI_TX_HEADER.BUSY.LINK_1	Number of cycles that the header buffer in the Quickpath Interface outbound link 1 is busy.	
43H	01H	UNC_QPI_RX_NO_PPT_CREDIT.STALLS.LINK_0	Number of cycles that snoop packets incoming to the Quickpath Interface link 0 are stalled and not sent to the GQ because the GQ Peer Probe Tracker (PPT) does not have any available entries.	
43H	02H	UNC_QPI_RX_NO_PPT_CREDIT.STALLS.LINK_1	Number of cycles that snoop packets incoming to the Quickpath Interface link 1 are stalled and not sent to the GQ because the GQ Peer Probe Tracker (PPT) does not have any available entries.	
60H	01H	UNC_DRAM_OPEN.CH0	Counts number of DRAM Channel 0 open commands issued either for read or write. To read or write data, the referenced DRAM page must first be opened.	
60H	02H	UNC_DRAM_OPEN.CH1	Counts number of DRAM Channel 1 open commands issued either for read or write. To read or write data, the referenced DRAM page must first be opened.	
60H	04H	UNC_DRAM_OPEN.CH2	Counts number of DRAM Channel 2 open commands issued either for read or write. To read or write data, the referenced DRAM page must first be opened.	
61H	01H	UNC_DRAM_PAGE_CLOSE.CH0	DRAM channel 0 command issued to CLOSE a page due to page idle timer expiration. Closing a page is done by issuing a precharge.	
61H	02H	UNC_DRAM_PAGE_CLOSE.CH1	DRAM channel 1 command issued to CLOSE a page due to page idle timer expiration. Closing a page is done by issuing a precharge.	
61H	04H	UNC_DRAM_PAGE_CLOSE.CH2	DRAM channel 2 command issued to CLOSE a page due to page idle timer expiration. Closing a page is done by issuing a precharge.	

Table 19-20. Non-Architectural Performance Events In the Processor Uncore for Intel® Core™ i7 Processor and Intel® Xeon® Processor 5500 Series (Contd.)

Event Num.	Umask Value	Event Mask Mnemonic	Description	Comment
62H	01H	UNC_DRAM_PAGE_MISS.CH0	Counts the number of precharges (PRE) that were issued to DRAM channel 0 because there was a page miss. A page miss refers to a situation in which a page is currently open and another page from the same bank needs to be opened. The new page experiences a page miss. Closing of the old page is done by issuing a precharge.	
62H	02H	UNC_DRAM_PAGE_MISS.CH1	Counts the number of precharges (PRE) that were issued to DRAM channel 1 because there was a page miss. A page miss refers to a situation in which a page is currently open and another page from the same bank needs to be opened. The new page experiences a page miss. Closing of the old page is done by issuing a precharge.	
62H	04H	UNC_DRAM_PAGE_MISS.CH2	Counts the number of precharges (PRE) that were issued to DRAM channel 2 because there was a page miss. A page miss refers to a situation in which a page is currently open and another page from the same bank needs to be opened. The new page experiences a page miss. Closing of the old page is done by issuing a precharge.	
63H	01H	UNC_DRAM_READ_CAS.CH0	Counts the number of times a read CAS command was issued on DRAM channel 0.	
63H	02H	UNC_DRAM_READ_CAS.AUTO PRE_CH0	Counts the number of times a read CAS command was issued on DRAM channel 0 where the command issued used the auto-precharge (auto page close) mode.	
63H	04H	UNC_DRAM_READ_CAS.CH1	Counts the number of times a read CAS command was issued on DRAM channel 1.	
63H	08H	UNC_DRAM_READ_CAS.AUTO PRE_CH1	Counts the number of times a read CAS command was issued on DRAM channel 1 where the command issued used the auto-precharge (auto page close) mode.	
63H	10H	UNC_DRAM_READ_CAS.CH2	Counts the number of times a read CAS command was issued on DRAM channel 2.	
63H	20H	UNC_DRAM_READ_CAS.AUTO PRE_CH2	Counts the number of times a read CAS command was issued on DRAM channel 2 where the command issued used the auto-precharge (auto page close) mode.	
64H	01H	UNC_DRAM_WRITE_CAS.CH0	Counts the number of times a write CAS command was issued on DRAM channel 0.	
64H	02H	UNC_DRAM_WRITE_CAS.AUTO PRE_CH0	Counts the number of times a write CAS command was issued on DRAM channel 0 where the command issued used the auto-precharge (auto page close) mode.	
64H	04H	UNC_DRAM_WRITE_CAS.CH1	Counts the number of times a write CAS command was issued on DRAM channel 1.	
64H	08H	UNC_DRAM_WRITE_CAS.AUTO PRE_CH1	Counts the number of times a write CAS command was issued on DRAM channel 1 where the command issued used the auto-precharge (auto page close) mode.	
64H	10H	UNC_DRAM_WRITE_CAS.CH2	Counts the number of times a write CAS command was issued on DRAM channel 2.	

Table 19-20. Non-Architectural Performance Events In the Processor Uncore for Intel® Core™ i7 Processor and Intel® Xeon® Processor 5500 Series (Contd.)

Event Num.	Umask Value	Event Mask Mnemonic	Description	Comment
64H	20H	UNC_DRAM_WRITE_CAS.AUTOPRE_CH2	Counts the number of times a write CAS command was issued on DRAM channel 2 where the command issued used the auto-precharge (auto page close) mode.	
65H	01H	UNC_DRAM_REFRESH.CH0	Counts number of DRAM channel 0 refresh commands. DRAM loses data content over time. In order to keep correct data content, the data values have to be refreshed periodically.	
65H	02H	UNC_DRAM_REFRESH.CH1	Counts number of DRAM channel 1 refresh commands. DRAM loses data content over time. In order to keep correct data content, the data values have to be refreshed periodically.	
65H	04H	UNC_DRAM_REFRESH.CH2	Counts number of DRAM channel 2 refresh commands. DRAM loses data content over time. In order to keep correct data content, the data values have to be refreshed periodically.	
66H	01H	UNC_DRAM_PRE_ALL.CH0	Counts number of DRAM Channel 0 precharge-all (PREALL) commands that close all open pages in a rank. PREALL is issued when the DRAM needs to be refreshed or needs to go into a power down mode.	
66H	02H	UNC_DRAM_PRE_ALL.CH1	Counts number of DRAM Channel 1 precharge-all (PREALL) commands that close all open pages in a rank. PREALL is issued when the DRAM needs to be refreshed or needs to go into a power down mode.	
66H	04H	UNC_DRAM_PRE_ALL.CH2	Counts number of DRAM Channel 2 precharge-all (PREALL) commands that close all open pages in a rank. PREALL is issued when the DRAM needs to be refreshed or needs to go into a power down mode.	

Intel Xeon processors with CPUID signature of DisplayFamily_DisplayModel 06_2EH have a distinct uncore sub-system that is significantly different from the uncore found in processors with CPUID signature 06_1AH, 06_1EH, and 06_1FH. Non-architectural Performance monitoring events for its uncore will be available in future documentation.

19.10 PERFORMANCE MONITORING EVENTS FOR PROCESSORS BASED ON INTEL® MICROARCHITECTURE CODE NAME WESTMERE

Intel 64 processors based on Intel® microarchitecture code name Westmere support the architectural and non-architectural performance-monitoring events listed in Table 19-1 and Table 19-21. Table 19-21 applies to processors with CPUID signature of DisplayFamily_DisplayModel encoding with the following values: 06_25H, 06_2CH. In addition, these processors (CPUID signature of DisplayFamily_DisplayModel 06_25H, 06_2CH) also support the following non-architectural, product-specific uncore performance-monitoring events listed in Table 19-22. Fixed counters support the architecture events defined in Table 19-2.

Table 19-21. Non-Architectural Performance Events In the Processor Core for Processors Based on Intel® Microarchitecture Code Name Westmere

Event Num.	Umask Value	Event Mask Mnemonic	Description	Comment
03H	02H	LOAD_BLOCK.OVERLAP_STORE	Loads that partially overlap an earlier store.	
04H	07H	SB_DRAIN.ANY	All Store buffer stall cycles.	
05H	02H	MISALIGN_MEMORY.STORE	All store referenced with misaligned address.	
06H	04H	STORE_BLOCKS.AT_RET	Counts number of loads delayed with at-Retirement block code. The following loads need to be executed at retirement and wait for all senior stores on the same thread to be drained: load splitting across 4K boundary (page split), load accessing uncacheable (UC or WC) memory, load lock, and load with page table in UC or WC memory region.	
06H	08H	STORE_BLOCKS.L1D_BLOCK	Cacheable loads delayed with L1D block code.	
07H	01H	PARTIAL_ADDRESS_ALIAS	Counts false dependency due to partial address aliasing.	
08H	01H	DTLB_LOAD_MISSES.ANY	Counts all load misses that cause a page walk.	
08H	02H	DTLB_LOAD_MISSES.WALK_COMPLETED	Counts number of completed page walks due to load miss in the STLB.	
08H	04H	DTLB_LOAD_MISSES.WALK_CYCLES	Cycles PMH is busy with a page walk due to a load miss in the STLB.	
08H	10H	DTLB_LOAD_MISSES.STLB_HIT	Number of cache load STLB hits.	
08H	20H	DTLB_LOAD_MISSES.PDE_MISS	Number of DTLB cache load misses where the low part of the linear to physical address translation was missed.	
0BH	01H	MEM_INST_RETIRED.LOADS	Counts the number of instructions with an architecturally-visible load retired on the architected path.	
0BH	02H	MEM_INST_RETIRED.STORES	Counts the number of instructions with an architecturally-visible store retired on the architected path.	
0BH	10H	MEM_INST_RETIRED.LATENCY_ABOVE_THRESHOLD	Counts the number of instructions exceeding the latency specified with ld_lat facility.	In conjunction with ld_lat facility.
0CH	01H	MEM_STORE_RETIRED.DTLB_MISS	The event counts the number of retired stores that missed the DTLB. The DTLB miss is not counted if the store operation causes a fault. Does not counter prefetches. Counts both primary and secondary misses to the TLB.	
0EH	01H	UOPS_ISSUED.ANY	Counts the number of Uops issued by the Register Allocation Table to the Reservation Station, i.e. the UOPs issued from the front end to the back end.	
0EH	01H	UOPS_ISSUED.STALLED_CYCLES	Counts the number of cycles no uops issued by the Register Allocation Table to the Reservation Station, i.e. the UOPs issued from the front end to the back end.	Set “invert=1, cmask = 1“.
0EH	02H	UOPS_ISSUED.FUSED	Counts the number of fused Uops that were issued from the Register Allocation Table to the Reservation Station.	

Table 19-21. Non-Architectural Performance Events In the Processor Core for Processors Based on Intel® Microarchitecture Code Name Westmere (Contd.)

Event Num.	Umask Value	Event Mask Mnemonic	Description	Comment
0FH	01H	MEM_UNCORE_RETIRED.UNKNOWN_SOURCE	Load instructions retired with unknown LLC miss (Precise Event).	Applicable to one and two sockets.
0FH	02H	MEM_UNCORE_RETIRED.OHTER_CORE_L2_HIT	Load instructions retired that HIT modified data in sibling core (Precise Event).	Applicable to one and two sockets.
0FH	04H	MEM_UNCORE_RETIRED.REMOTE_HITM	Load instructions retired that HIT modified data in remote socket (Precise Event).	Applicable to two sockets only.
0FH	08H	MEM_UNCORE_RETIRED.LOCAL_DRAM_AND_REMOTE_CACHE_HIT	Load instructions retired local dram and remote cache HIT data sources (Precise Event).	Applicable to one and two sockets.
0FH	10H	MEM_UNCORE_RETIRED.REMOTE_DRAM	Load instructions retired remote DRAM and remote home-remote cache HITM (Precise Event).	Applicable to two sockets only.
0FH	20H	MEM_UNCORE_RETIRED.OTHER_LLC_MISS	Load instructions retired other LLC miss (Precise Event).	Applicable to two sockets only.
0FH	80H	MEM_UNCORE_RETIRED.UNCACHEABLE	Load instructions retired I/O (Precise Event).	Applicable to one and two sockets.
10H	01H	FP_COMP_OPS_EXE.X87	Counts the number of FP Computational Uops Executed. The number of FADD, FSUB, FCOM, FMULs, integer MULs and IMULs, FDIVs, FPREMs, FSQRTS, integer DIVs, and IDIVs. This event does not distinguish an FADD used in the middle of a transcendental flow from a separate FADD instruction.	
10H	02H	FP_COMP_OPS_EXE.MMX	Counts number of MMX Uops executed.	
10H	04H	FP_COMP_OPS_EXE.SSE_FP	Counts number of SSE and SSE2 FP uops executed.	
10H	08H	FP_COMP_OPS_EXE.SSE2_INTEGER	Counts number of SSE2 integer uops executed.	
10H	10H	FP_COMP_OPS_EXE.SSE_FP_PACKED	Counts number of SSE FP packed uops executed.	
10H	20H	FP_COMP_OPS_EXE.SSE_FP_SCALAR	Counts number of SSE FP scalar uops executed.	
10H	40H	FP_COMP_OPS_EXE.SSE_SINGLE_PRECISION	Counts number of SSE* FP single precision uops executed.	
10H	80H	FP_COMP_OPS_EXE.SSE_DOUBLE_PRECISION	Counts number of SSE* FP double precision uops executed.	
12H	01H	SIMD_INT_128.PACKED_MPY	Counts number of 128 bit SIMD integer multiply operations.	
12H	02H	SIMD_INT_128.PACKED_SHIFT	Counts number of 128 bit SIMD integer shift operations.	
12H	04H	SIMD_INT_128.PACK	Counts number of 128 bit SIMD integer pack operations.	
12H	08H	SIMD_INT_128.UNPACK	Counts number of 128 bit SIMD integer unpack operations.	
12H	10H	SIMD_INT_128.PACKED_LOGICAL	Counts number of 128 bit SIMD integer logical operations.	
12H	20H	SIMD_INT_128.PACKED_ARITH	Counts number of 128 bit SIMD integer arithmetic operations.	

Table 19-21. Non-Architectural Performance Events In the Processor Core for Processors Based on Intel® Microarchitecture Code Name Westmere (Contd.)

Event Num.	Umask Value	Event Mask Mnemonic	Description	Comment
12H	40H	SIMD_INT_128.SHUFFLE_MOVE	Counts number of 128 bit SIMD integer shuffle and move operations.	
13H	01H	LOAD_DISPATCH.RS	Counts number of loads dispatched from the Reservation Station that bypass the Memory Order Buffer.	
13H	02H	LOAD_DISPATCH.RS_DELAYED	Counts the number of delayed RS dispatches at the stage latch. If an RS dispatch cannot bypass to LB, it has another chance to dispatch from the one-cycle delayed staging latch before it is written into the LB.	
13H	04H	LOAD_DISPATCH.MOB	Counts the number of loads dispatched from the Reservation Station to the Memory Order Buffer.	
13H	07H	LOAD_DISPATCH.ANY	Counts all loads dispatched from the Reservation Station.	
14H	01H	ARITH.CYCLES_DIV_BUSY	Counts the number of cycles the divider is busy executing divide or square root operations. The divide can be integer, X87 or Streaming SIMD Extensions (SSE). The square root operation can be either X87 or SSE. Set 'edge =1, invert=1, cmask=1' to count the number of divides.	Count may be incorrect When SMT is on.
14H	02H	ARITH.MUL	Counts the number of multiply operations executed. This includes integer as well as floating point multiply operations but excludes DPPS mul and MPSAD.	Count may be incorrect When SMT is on.
17H	01H	INST_QUEUE_WRITES	Counts the number of instructions written into the instruction queue every cycle.	
18H	01H	INST_DECODED.DEC0	Counts number of instructions that require decoder 0 to be decoded. Usually, this means that the instruction maps to more than 1 uop.	
19H	01H	TWO_UOP_INSTS_DECODED	An instruction that generates two uops was decoded.	
1EH	01H	INST_QUEUE_WRITE_CYCLES	This event counts the number of cycles during which instructions are written to the instruction queue. Dividing this counter by the number of instructions written to the instruction queue (INST_QUEUE_WRITES) yields the average number of instructions decoded each cycle. If this number is less than four and the pipe stalls, this indicates that the decoder is failing to decode enough instructions per cycle to sustain the 4-wide pipeline.	If SSE* instructions that are 6 bytes or longer arrive one after another, then front end throughput may limit execution speed.
20H	01H	LSD_OVERFLOW	Number of loops that cannot stream from the instruction queue.	
24H	01H	L2_RQSTS.LD_HIT	Counts number of loads that hit the L2 cache. L2 loads include both L1D demand misses as well as L1D prefetches. L2 loads can be rejected for various reasons. Only non rejected loads are counted.	
24H	02H	L2_RQSTS.LD_MISS	Counts the number of loads that miss the L2 cache. L2 loads include both L1D demand misses as well as L1D prefetches.	

Table 19-21. Non-Architectural Performance Events In the Processor Core for Processors Based on Intel® Microarchitecture Code Name Westmere (Contd.)

Event Num.	Umask Value	Event Mask Mnemonic	Description	Comment
24H	03H	L2_RQSTS.LOADS	Counts all L2 load requests. L2 loads include both L1D demand misses as well as L1D prefetches.	
24H	04H	L2_RQSTS.RFO_HIT	Counts the number of store RFO requests that hit the L2 cache. L2 RFO requests include both L1D demand RFO misses as well as L1D RFO prefetches. Count includes WC memory requests, where the data is not fetched but the permission to write the line is required.	
24H	08H	L2_RQSTS.RFO_MISS	Counts the number of store RFO requests that miss the L2 cache. L2 RFO requests include both L1D demand RFO misses as well as L1D RFO prefetches.	
24H	0CH	L2_RQSTS.RFOS	Counts all L2 store RFO requests. L2 RFO requests include both L1D demand RFO misses as well as L1D RFO prefetches.	
24H	10H	L2_RQSTS.IFETCH_HIT	Counts number of instruction fetches that hit the L2 cache. L2 instruction fetches include both L1I demand misses as well as L1I instruction prefetches.	
24H	20H	L2_RQSTS.IFETCH_MISS	Counts number of instruction fetches that miss the L2 cache. L2 instruction fetches include both L1I demand misses as well as L1I instruction prefetches.	
24H	30H	L2_RQSTS.IFETCHES	Counts all instruction fetches. L2 instruction fetches include both L1I demand misses as well as L1I instruction prefetches.	
24H	40H	L2_RQSTS.PREFETCH_HIT	Counts L2 prefetch hits for both code and data.	
24H	80H	L2_RQSTS.PREFETCH_MISS	Counts L2 prefetch misses for both code and data.	
24H	C0H	L2_RQSTS.PREFETCHES	Counts all L2 prefetches for both code and data.	
24H	AAH	L2_RQSTS.MISS	Counts all L2 misses for both code and data.	
24H	FFH	L2_RQSTS.REFERENCES	Counts all L2 requests for both code and data.	
26H	01H	L2_DATA_RQSTS.DEMAND.I_STATE	Counts number of L2 data demand loads where the cache line to be loaded is in the I (invalid) state, i.e., a cache miss. L2 demand loads are both L1D demand misses and L1D prefetches.	
26H	02H	L2_DATA_RQSTS.DEMAND.S_STATE	Counts number of L2 data demand loads where the cache line to be loaded is in the S (shared) state. L2 demand loads are both L1D demand misses and L1D prefetches.	
26H	04H	L2_DATA_RQSTS.DEMAND.E_STATE	Counts number of L2 data demand loads where the cache line to be loaded is in the E (exclusive) state. L2 demand loads are both L1D demand misses and L1D prefetches.	
26H	08H	L2_DATA_RQSTS.DEMAND.M_STATE	Counts number of L2 data demand loads where the cache line to be loaded is in the M (modified) state. L2 demand loads are both L1D demand misses and L1D prefetches.	

Table 19-21. Non-Architectural Performance Events In the Processor Core for Processors Based on Intel® Microarchitecture Code Name Westmere (Contd.)

Event Num.	Umask Value	Event Mask Mnemonic	Description	Comment
26H	0FH	L2_DATA_RQSTS.DEMAND.MESI	Counts all L2 data demand requests. L2 demand loads are both L1D demand misses and L1D prefetches.	
26H	10H	L2_DATA_RQSTS.PREFETCH.I_STATE	Counts number of L2 prefetch data loads where the cache line to be loaded is in the I (invalid) state, i.e., a cache miss.	
26H	20H	L2_DATA_RQSTS.PREFETCH.S_STATE	Counts number of L2 prefetch data loads where the cache line to be loaded is in the S (shared) state. A prefetch RFO will miss on an S state line, while a prefetch read will hit on an S state line.	
26H	40H	L2_DATA_RQSTS.PREFETCH.E_STATE	Counts number of L2 prefetch data loads where the cache line to be loaded is in the E (exclusive) state.	
26H	80H	L2_DATA_RQSTS.PREFETCH.M_STATE	Counts number of L2 prefetch data loads where the cache line to be loaded is in the M (modified) state.	
26H	F0H	L2_DATA_RQSTS.PREFETCH.MESI	Counts all L2 prefetch requests.	
26H	FFH	L2_DATA_RQSTS.ANY	Counts all L2 data requests.	
27H	01H	L2_WRITE.RFO.I_STATE	Counts number of L2 demand store RFO requests where the cache line to be loaded is in the I (invalid) state, i.e., a cache miss. The L1D prefetcher does not issue a RFO prefetch.	This is a demand RFO request.
27H	02H	L2_WRITE.RFO.S_STATE	Counts number of L2 store RFO requests where the cache line to be loaded is in the S (shared) state. The L1D prefetcher does not issue a RFO prefetch.	This is a demand RFO request.
27H	08H	L2_WRITE.RFO.M_STATE	Counts number of L2 store RFO requests where the cache line to be loaded is in the M (modified) state. The L1D prefetcher does not issue a RFO prefetch.	This is a demand RFO request.
27H	0EH	L2_WRITE.RFO.HIT	Counts number of L2 store RFO requests where the cache line to be loaded is in either the S, E or M states. The L1D prefetcher does not issue a RFO prefetch.	This is a demand RFO request.
27H	0FH	L2_WRITE.RFO.MESI	Counts all L2 store RFO requests. The L1D prefetcher does not issue a RFO prefetch.	This is a demand RFO request.
27H	10H	L2_WRITE.LOCK.I_STATE	Counts number of L2 demand lock RFO requests where the cache line to be loaded is in the I (invalid) state, i.e., a cache miss.	
27H	20H	L2_WRITE.LOCK.S_STATE	Counts number of L2 lock RFO requests where the cache line to be loaded is in the S (shared) state.	
27H	40H	L2_WRITE.LOCK.E_STATE	Counts number of L2 demand lock RFO requests where the cache line to be loaded is in the E (exclusive) state.	
27H	80H	L2_WRITE.LOCK.M_STATE	Counts number of L2 demand lock RFO requests where the cache line to be loaded is in the M (modified) state.	
27H	E0H	L2_WRITE.LOCK.HIT	Counts number of L2 demand lock RFO requests where the cache line to be loaded is in either the S, E, or M state.	

Table 19-21. Non-Architectural Performance Events In the Processor Core for Processors Based on Intel® Microarchitecture Code Name Westmere (Contd.)

Event Num.	Umask Value	Event Mask Mnemonic	Description	Comment
27H	F0H	L2_WRITE.LOCK.MESI	Counts all L2 demand lock RFO requests.	
28H	01H	L1D_WB_L2.I_STATE	Counts number of L1 writebacks to the L2 where the cache line to be written is in the I (invalid) state, i.e., a cache miss.	
28H	02H	L1D_WB_L2.S_STATE	Counts number of L1 writebacks to the L2 where the cache line to be written is in the S state.	
28H	04H	L1D_WB_L2.E_STATE	Counts number of L1 writebacks to the L2 where the cache line to be written is in the E (exclusive) state.	
28H	08H	L1D_WB_L2.M_STATE	Counts number of L1 writebacks to the L2 where the cache line to be written is in the M (modified) state.	
28H	0FH	L1D_WB_L2.MESI	Counts all L1 writebacks to the L2 .	
2EH	41H	L3_LAT_CACHE.MISS	Counts uncore Last Level Cache misses. Because cache hierarchy, cache sizes and other implementation-specific characteristics; value comparison to estimate performance differences is not recommended.	See Table 19-1.
2EH	4FH	L3_LAT_CACHE.REFERENCE	Counts uncore Last Level Cache references. Because cache hierarchy, cache sizes and other implementation-specific characteristics; value comparison to estimate performance differences is not recommended.	See Table 19-1.
3CH	00H	CPU_CLK_UNHALTED.THREAD_P	Counts the number of thread cycles while the thread is not in a halt state. The thread enters the halt state when it is running the HLT instruction. The core frequency may change from time to time due to power or thermal throttling.	See Table 19-1.
3CH	01H	CPU_CLK_UNHALTED.REF_P	Increments at the frequency of TSC when not halted.	See Table 19-1.
49H	01H	DTLB_MISSES.ANY	Counts the number of misses in the STLB which causes a page walk.	
49H	02H	DTLB_MISSES.WALK_COMPLETED	Counts number of misses in the STLB which resulted in a completed page walk.	
49H	04H	DTLB_MISSES.WALK_CYCLES	Counts cycles of page walk due to misses in the STLB.	
49H	10H	DTLB_MISSES.STLB_HIT	Counts the number of DTLB first level misses that hit in the second level TLB. This event is only relevant if the core contains multiple DTLB levels.	
49H	20H	DTLB_MISSES.PDE_MISS	Number of DTLB misses caused by low part of address, includes references to 2M pages because 2M pages do not use the PDE.	
49H	80H	DTLB_MISSES.LARGE_WALK_COMPLETED	Counts number of completed large page walks due to misses in the STLB.	
4CH	01H	LOAD_HIT_PRE	Counts load operations sent to the L1 data cache while a previous SSE prefetch instruction to the same cache line has started prefetching but has not yet finished.	Counter 0, 1 only.

Table 19-21. Non-Architectural Performance Events In the Processor Core for Processors Based on Intel® Microarchitecture Code Name Westmere (Contd.)

Event Num.	Umask Value	Event Mask Mnemonic	Description	Comment
4EH	01H	L1D_PREFETCH.REQUESTS	Counts number of hardware prefetch requests dispatched out of the prefetch FIFO.	Counter 0, 1 only.
4EH	02H	L1D_PREFETCH.MISS	Counts number of hardware prefetch requests that miss the L1D. There are two prefetchers in the L1D. A streamer, which predicts lines sequentially after this one should be fetched, and the IP prefetcher that remembers access patterns for the current instruction. The streamer prefetcher stops on an L1D hit, while the IP prefetcher does not.	Counter 0, 1 only.
4EH	04H	L1D_PREFETCH.TRIGGERS	Counts number of prefetch requests triggered by the Finite State Machine and pushed into the prefetch FIFO. Some of the prefetch requests are dropped due to overwrites or competition between the IP index prefetcher and streamer prefetcher. The prefetch FIFO contains 4 entries.	Counter 0, 1 only.
4FH	10H	EPT.WALK_CYCLES	Counts Extended Page walk cycles.	
51H	01H	L1D.REPL	Counts the number of lines brought into the L1 data cache.	Counter 0, 1 only.
51H	02H	L1D.M_REPL	Counts the number of modified lines brought into the L1 data cache.	Counter 0, 1 only.
51H	04H	L1D.M_EVICT	Counts the number of modified lines evicted from the L1 data cache due to replacement.	Counter 0, 1 only.
51H	08H	L1D.M_SNOOP_EVICT	Counts the number of modified lines evicted from the L1 data cache due to snoop HITM intervention.	Counter 0, 1 only.
52H	01H	L1D_CACHE_PREFETCH_LOCK_FB_HIT	Counts the number of cacheable load lock speculated instructions accepted into the fill buffer.	
60H	01H	OFFCORE_REQUESTS_OUTSTANDING.DEMAND.READ_DATA	Counts weighted cycles of offcore demand data read requests. Does not include L2 prefetch requests.	Counter 0.
60H	02H	OFFCORE_REQUESTS_OUTSTANDING.DEMAND.READ_CODE	Counts weighted cycles of offcore demand code read requests. Does not include L2 prefetch requests.	Counter 0.
60H	04H	OFFCORE_REQUESTS_OUTSTANDING.DEMAND.RFO	Counts weighted cycles of offcore demand RFO requests. Does not include L2 prefetch requests.	Counter 0.
60H	08H	OFFCORE_REQUESTS_OUTSTANDING.ANY.READ	Counts weighted cycles of offcore read requests of any kind. Include L2 prefetch requests.	Counter 0.
63H	01H	CACHE_LOCK_CYCLES.L1D_L2	Cycle count during which the L1D and L2 are locked. A lock is asserted when there is a locked memory access, due to uncacheable memory, a locked operation that spans two cache lines, or a page walk from an uncacheable page table. This event does not cause locks, it merely detects them.	Counter 0, 1 only. L1D and L2 locks have a very high performance penalty and it is highly recommended to avoid such accesses.
63H	02H	CACHE_LOCK_CYCLES.L1D	Counts the number of cycles that cacheline in the L1 data cache unit is locked.	Counter 0, 1 only.
6CH	01H	IO_TRANSACTIONS	Counts the number of completed I/O transactions.	
80H	01H	L1I.HITS	Counts all instruction fetches that hit the L1 instruction cache.	

Table 19-21. Non-Architectural Performance Events In the Processor Core for Processors Based on Intel® Microarchitecture Code Name Westmere (Contd.)

Event Num.	Umask Value	Event Mask Mnemonic	Description	Comment
80H	02H	L1I.MISSES	Counts all instruction fetches that miss the L1I cache. This includes instruction cache misses, streaming buffer misses, victim cache misses and uncacheable fetches. An instruction fetch miss is counted only once and not once for every cycle it is outstanding.	
80H	03H	L1I.READS	Counts all instruction fetches, including uncacheable fetches that bypass the L1I.	
80H	04H	L1I.CYCLES_STALLED	Cycle counts for which an instruction fetch stalls due to a L1I cache miss, ITLB miss or ITLB fault.	
82H	01H	LARGE_ITLB.HIT	Counts number of large ITLB hits.	
85H	01H	ITLB_MISSES.ANY	Counts the number of misses in all levels of the ITLB which causes a page walk.	
85H	02H	ITLB_MISSES.WALK_COMPLETED	Counts number of misses in all levels of the ITLB which resulted in a completed page walk.	
85H	04H	ITLB_MISSES.WALK_CYCLES	Counts ITLB miss page walk cycles.	
85H	10H	ITLB_MISSES.STLB_HIT	Counts number of ITLB first level miss but second level hits.	
85H	80H	ITLB_MISSES.LARGE_WALK_COMPLETED	Counts number of completed large page walks due to misses in the STLB.	
87H	01H	ILD_STALL.LCP	Cycles Instruction Length Decoder stalls due to length changing prefixes: 66, 67 or REX.W (for Intel 64) instructions which change the length of the decoded instruction.	
87H	02H	ILD_STALL.MRU	Instruction Length Decoder stall cycles due to Brand Prediction Unit (PBU) Most Recently Used (MRU) bypass.	
87H	04H	ILD_STALL.IQ_FULL	Stall cycles due to a full instruction queue.	
87H	08H	ILD_STALL.REGEN	Counts the number of regen stalls.	
87H	0FH	ILD_STALL.ANY	Counts any cycles the Instruction Length Decoder is stalled.	
88H	01H	BR_INST_EXEC.COND	Counts the number of conditional near branch instructions executed, but not necessarily retired.	
88H	02H	BR_INST_EXEC.DIRECT	Counts all unconditional near branch instructions excluding calls and indirect branches.	
88H	04H	BR_INST_EXEC.INDIRECT_NON_CALL	Counts the number of executed indirect near branch instructions that are not calls.	
88H	07H	BR_INST_EXEC.NON_CALLS	Counts all non-call near branch instructions executed, but not necessarily retired.	
88H	08H	BR_INST_EXEC.RETURN_NEAR	Counts indirect near branches that have a return mnemonic.	
88H	10H	BR_INST_EXEC.DIRECT_NEAR_CALL	Counts unconditional near call branch instructions, excluding non-call branch, executed.	
88H	20H	BR_INST_EXEC.INDIRECT_NEAR_CALL	Counts indirect near calls, including both register and memory indirect, executed.	

Table 19-21. Non-Architectural Performance Events In the Processor Core for Processors Based on Intel® Microarchitecture Code Name Westmere (Contd.)

Event Num.	Umask Value	Event Mask Mnemonic	Description	Comment
88H	30H	BR_INST_EXEC.NEAR_CALLS	Counts all near call branches executed, but not necessarily retired.	
88H	40H	BR_INST_EXEC.TAKEN	Counts taken near branches executed, but not necessarily retired.	
88H	7FH	BR_INST_EXEC.ANY	Counts all near executed branches (not necessarily retired). This includes only instructions and not micro-op branches. Frequent branching is not necessarily a major performance issue. However frequent branch mispredictions may be a problem.	
89H	01H	BR_MISP_EXEC.COND	Counts the number of mispredicted conditional near branch instructions executed, but not necessarily retired.	
89H	02H	BR_MISP_EXEC.DIRECT	Counts mispredicted macro unconditional near branch instructions, excluding calls and indirect branches (should always be 0).	
89H	04H	BR_MISP_EXEC.INDIRECT_NON_CALL	Counts the number of executed mispredicted indirect near branch instructions that are not calls.	
89H	07H	BR_MISP_EXEC.NON_CALLS	Counts mispredicted non-call near branches executed, but not necessarily retired.	
89H	08H	BR_MISP_EXEC.RETURN_NEAR	Counts mispredicted indirect branches that have a rear return mnemonic.	
89H	10H	BR_MISP_EXEC.DIRECT_NEAR_CALL	Counts mispredicted non-indirect near calls executed, (should always be 0).	
89H	20H	BR_MISP_EXEC.INDIRECT_NEAR_CALL	Counts mispredicted indirect near calls executed, including both register and memory indirect.	
89H	30H	BR_MISP_EXEC.NEAR_CALLS	Counts all mispredicted near call branches executed, but not necessarily retired.	
89H	40H	BR_MISP_EXEC.TAKEN	Counts executed mispredicted near branches that are taken, but not necessarily retired.	
89H	7FH	BR_MISP_EXEC.ANY	Counts the number of mispredicted near branch instructions that were executed, but not necessarily retired.	
A2H	01H	RESOURCE_STALLS.ANY	Counts the number of Allocator resource related stalls. Includes register renaming buffer entries, memory buffer entries. In addition to resource related stalls, this event counts some other events. Includes stalls arising during branch misprediction recovery, such as if retirement of the mispredicted branch is delayed and stalls arising while store buffer is draining from synchronizing operations.	Does not include stalls due to SuperQ (off core) queue full, too many cache misses, etc.
A2H	02H	RESOURCE_STALLS.LOAD	Counts the cycles of stall due to lack of load buffer for load operation.	

Table 19-21. Non-Architectural Performance Events In the Processor Core for Processors Based on Intel® Microarchitecture Code Name Westmere (Contd.)

Event Num.	Umask Value	Event Mask Mnemonic	Description	Comment
A2H	04H	RESOURCE_STALLS.RS_FULL	This event counts the number of cycles when the number of instructions in the pipeline waiting for execution reaches the limit the processor can handle. A high count of this event indicates that there are long latency operations in the pipe (possibly load and store operations that miss the L2 cache, or instructions dependent upon instructions further down the pipeline that have yet to retire.	When RS is full, new instructions cannot enter the reservation station and start execution.
A2H	08H	RESOURCE_STALLS.STORE	This event counts the number of cycles that a resource related stall will occur due to the number of store instructions reaching the limit of the pipeline, (i.e. all store buffers are used). The stall ends when a store instruction commits its data to the cache or memory.	
A2H	10H	RESOURCE_STALLS.ROB_FULL	Counts the cycles of stall due to re-order buffer full.	
A2H	20H	RESOURCE_STALLS.FPCW	Counts the number of cycles while execution was stalled due to writing the floating-point unit (FPU) control word.	
A2H	40H	RESOURCE_STALLS.MXCSR	Stalls due to the MXCSR register rename occurring to close to a previous MXCSR rename. The MXCSR provides control and status for the MMX registers.	
A2H	80H	RESOURCE_STALLS.OTHER	Counts the number of cycles while execution was stalled due to other resource issues.	
A6H	01H	MACRO_INSTS.FUSIONS_DECODED	Counts the number of instructions decoded that are macro-fused but not necessarily executed or retired.	
A7H	01H	BACLEAR_FORCE_IQ	Counts number of times a BACLEAR was forced by the Instruction Queue. The IQ is also responsible for providing conditional branch prediction direction based on a static scheme and dynamic data provided by the L2 Branch Prediction Unit. If the conditional branch target is not found in the Target Array and the IQ predicts that the branch is taken, then the IQ will force the Branch Address Calculator to issue a BACLEAR. Each BACLEAR asserted by the BAC generates approximately an 8 cycle bubble in the instruction fetch pipeline.	
A8H	01H	LSD.UOPS	Counts the number of micro-ops delivered by loop stream detector.	Use cmask=1 and invert to count cycles.
AEH	01H	ITLB_FLUSH	Counts the number of ITLB flushes.	
B0H	01H	OFFCORE_REQUESTS.DEMAND.READ_DATA	Counts number of offcore demand data read requests. Does not count L2 prefetch requests.	
B0H	02H	OFFCORE_REQUESTS.DEMAND.READ_CODE	Counts number of offcore demand code read requests. Does not count L2 prefetch requests.	
B0H	04H	OFFCORE_REQUESTS.DEMAND.RFO	Counts number of offcore demand RFO requests. Does not count L2 prefetch requests.	
B0H	08H	OFFCORE_REQUESTS.ANY.READ	Counts number of offcore read requests. Includes L2 prefetch requests.	

Table 19-21. Non-Architectural Performance Events In the Processor Core for Processors Based on Intel® Microarchitecture Code Name Westmere (Contd.)

Event Num.	Umask Value	Event Mask Mnemonic	Description	Comment
B0H	10H	OFFCORE_REQUESTS.ANY.RFO	Counts number of offcore RFO requests. Includes L2 prefetch requests.	
B0H	40H	OFFCORE_REQUESTS.L1D_WRITEBACK	Counts number of L1D writebacks to the uncore.	
B0H	80H	OFFCORE_REQUESTS.ANY	Counts all offcore requests.	
B1H	01H	UOPS_EXECUTED.PORT0	Counts number of uops executed that were issued on port 0. Port 0 handles integer arithmetic, SIMD and FP add uops.	
B1H	02H	UOPS_EXECUTED.PORT1	Counts number of uops executed that were issued on port 1. Port 1 handles integer arithmetic, SIMD, integer shift, FP multiply and FP divide uops.	
B1H	04H	UOPS_EXECUTED.PORT2_CORE	Counts number of uops executed that were issued on port 2. Port 2 handles the load uops. This is a core count only and cannot be collected per thread.	
B1H	08H	UOPS_EXECUTED.PORT3_CORE	Counts number of uops executed that were issued on port 3. Port 3 handles store uops. This is a core count only and cannot be collected per thread.	
B1H	10H	UOPS_EXECUTED.PORT4_CORE	Counts number of uops executed that where issued on port 4. Port 4 handles the value to be stored for the store uops issued on port 3. This is a core count only and cannot be collected per thread.	
B1H	1FH	UOPS_EXECUTED.CORE_ACTIVE_CYCLES_NO_PORT5	Counts number of cycles there are one or more uops being executed and were issued on ports 0-4. This is a core count only and cannot be collected per thread.	
B1H	20H	UOPS_EXECUTED.PORT5	Counts number of uops executed that where issued on port 5.	
B1H	3FH	UOPS_EXECUTED.CORE_ACTIVE_CYCLES	Counts number of cycles there are one or more uops being executed on any ports. This is a core count only and cannot be collected per thread.	
B1H	40H	UOPS_EXECUTED.PORT015	Counts number of uops executed that where issued on port 0, 1, or 5.	Use cmask=1, invert=1 to count stall cycles.
B1H	80H	UOPS_EXECUTED.PORT234	Counts number of uops executed that where issued on port 2, 3, or 4.	
B2H	01H	OFFCORE_REQUESTS_SQ_FULL	Counts number of cycles the SQ is full to handle off-core requests.	
B3H	01H	SNOOPQ_REQUESTS_OUTSTANDING.DATA	Counts weighted cycles of snoopq requests for data. Counter 0 only.	Use cmask=1 to count cycles not empty.
B3H	02H	SNOOPQ_REQUESTS_OUTSTANDING.INVALIDATE	Counts weighted cycles of snoopq invalidate requests. Counter 0 only.	Use cmask=1 to count cycles not empty.
B3H	04H	SNOOPQ_REQUESTS_OUTSTANDING.CODE	Counts weighted cycles of snoopq requests for code. Counter 0 only.	Use cmask=1 to count cycles not empty.
B4H	01H	SNOOPQ_REQUESTS.CODE	Counts the number of snoop code requests.	
B4H	02H	SNOOPQ_REQUESTS.DATA	Counts the number of snoop data requests.	
B4H	04H	SNOOPQ_REQUESTS.INVALIDATE	Counts the number of snoop invalidate requests.	

Table 19-21. Non-Architectural Performance Events In the Processor Core for Processors Based on Intel® Microarchitecture Code Name Westmere (Contd.)

Event Num.	Umask Value	Event Mask Mnemonic	Description	Comment
B7H	01H	OFF_CORE_RESPONSE_0	See Section 18.3.1.1.3, "Off-core Response Performance Monitoring in the Processor Core".	Requires programming MSR 01A6H.
B8H	01H	SNOOP_RESPONSE.HIT	Counts HIT snoop response sent by this thread in response to a snoop request.	
B8H	02H	SNOOP_RESPONSE.HITE	Counts HIT E snoop response sent by this thread in response to a snoop request.	
B8H	04H	SNOOP_RESPONSE.HITM	Counts HIT M snoop response sent by this thread in response to a snoop request.	
BBH	01H	OFF_CORE_RESPONSE_1	See Section 18.3.1.1.3, "Off-core Response Performance Monitoring in the Processor Core".	Use MSR 01A7H.
C0H	00H	INST_RETIRED.ANY_P	See Table 19-1. Notes: INST_RETIRED.ANY is counted by a designated fixed counter. INST_RETIRED.ANY_P is counted by a programmable counter and is an architectural performance event. Event is supported if CPUID.A.EBX[1] = 0.	Counting: Faulting executions of GETSEC/VM entry/VM Exit/MWait will not count as retired instructions.
C0H	02H	INST_RETIRED.X87	Counts the number of floating point computational operations retired: floating point computational operations executed by the assist handler and sub-operations of complex floating point instructions like transcendental instructions.	
C0H	04H	INST_RETIRED.MMX	Counts the number of retired: MMX instructions.	
C2H	01H	UOPS_RETIRED.ANY	Counts the number of micro-ops retired, (macro-fused=1, micro-fused=2, others=1; maximum count of 8 per cycle). Most instructions are composed of one or two micro-ops. Some instructions are decoded into longer sequences such as repeat instructions, floating point transcendental instructions, and assists.	Use cmask=1 and invert to count active cycles or stalled cycles.
C2H	02H	UOPS_RETIRED.RETIRE_SLOTS	Counts the number of retirement slots used each cycle.	
C2H	04H	UOPS_RETIRED.MACRO_FUSED	Counts number of macro-fused uops retired.	
C3H	01H	MACHINE_CLEARS.CYCLES	Counts the cycles machine clear is asserted.	
C3H	02H	MACHINE_CLEARS.MEM_ORDER	Counts the number of machine clears due to memory order conflicts.	
C3H	04H	MACHINE_CLEARS.SMC	Counts the number of times that a program writes to a code section. Self-modifying code causes a severe penalty in all Intel 64 and IA-32 processors. The modified cache line is written back to the L2 and L3caches.	
C4H	00H	BR_INST_RETIRED.ALL_BRANCHES	Branch instructions at retirement.	See Table 19-1.
C4H	01H	BR_INST_RETIRED.CONDITIONAL	Counts the number of conditional branch instructions retired.	
C4H	02H	BR_INST_RETIRED.NEAR_CALL	Counts the number of direct & indirect near unconditional calls retired.	

Table 19-21. Non-Architectural Performance Events In the Processor Core for Processors Based on Intel® Microarchitecture Code Name Westmere (Contd.)

Event Num.	Umask Value	Event Mask Mnemonic	Description	Comment
C5H	00H	BR_MISP_RETIRED.ALL_BRANCHES	Mispredicted branch instructions at retirement.	See Table 19-1.
C5H	01H	BR_MISP_RETIRED.CONDITIONAL	Counts mispredicted conditional retired calls.	
C5H	02H	BR_MISP_RETIRED.NEAR_CALL	Counts mispredicted direct & indirect near unconditional retired calls.	
C5H	04H	BR_MISP_RETIRED.ALL_BRANCHES	Counts all mispredicted retired calls.	
C7H	01H	SSEX_UOPS_RETIRED.PACKED_SINGLE	Counts SIMD packed single-precision floating-point uops retired.	
C7H	02H	SSEX_UOPS_RETIRED.SCALAR_SINGLE	Counts SIMD scalar single-precision floating-point uops retired.	
C7H	04H	SSEX_UOPS_RETIRED.PACKED_DOUBLE	Counts SIMD packed double-precision floating-point uops retired.	
C7H	08H	SSEX_UOPS_RETIRED.SCALAR_DOUBLE	Counts SIMD scalar double-precision floating-point uops retired.	
C7H	10H	SSEX_UOPS_RETIRED.VECTOR_INTEGER	Counts 128-bit SIMD vector integer uops retired.	
C8H	20H	ITLB_MISS_RETIRED	Counts the number of retired instructions that missed the ITLB when the instruction was fetched.	
CBH	01H	MEM_LOAD_RETIRED.L1D_HIT	Counts number of retired loads that hit the L1 data cache.	
CBH	02H	MEM_LOAD_RETIRED.L2_HIT	Counts number of retired loads that hit the L2 data cache.	
CBH	04H	MEM_LOAD_RETIRED.L3_UNSHARED_HIT	Counts number of retired loads that hit their own, unshared lines in the L3 cache.	
CBH	08H	MEM_LOAD_RETIRED.OTHER_CORE_L2_HIT_HITM	Counts number of retired loads that hit in a sibling core's L2 (on die core). Since the L3 is inclusive of all cores on the package, this is an L3 hit. This counts both clean and modified hits.	
CBH	10H	MEM_LOAD_RETIRED.L3_MISS	Counts number of retired loads that miss the L3 cache. The load was satisfied by a remote socket, local memory or an IOH.	
CBH	40H	MEM_LOAD_RETIRED.HIT_LFB	Counts number of retired loads that miss the L1D and the address is located in an allocated line fill buffer and will soon be committed to cache. This is counting secondary L1D misses.	
CBH	80H	MEM_LOAD_RETIRED.DTLB_MISS	Counts the number of retired loads that missed the DTLB. The DTLB miss is not counted if the load operation causes a fault. This event counts loads from cacheable memory only. The event does not count loads by software prefetches. Counts both primary and secondary misses to the TLB.	
CCH	01H	FP_MMX_TRANS.TO_FP	Counts the first floating-point instruction following any MMX instruction. You can use this event to estimate the penalties for the transitions between floating-point and MMX technology states.	

Table 19-21. Non-Architectural Performance Events In the Processor Core for Processors Based on Intel® Microarchitecture Code Name Westmere (Contd.)

Event Num.	Umask Value	Event Mask Mnemonic	Description	Comment
CCH	02H	FP_MMX_TRANS.TO_MMX	Counts the first MMX instruction following a floating-point instruction. You can use this event to estimate the penalties for the transitions between floating-point and MMX technology states.	
CCH	03H	FP_MMX_TRANS.ANY	Counts all transitions from floating point to MMX instructions and from MMX instructions to floating point instructions. You can use this event to estimate the penalties for the transitions between floating-point and MMX technology states.	
D0H	01H	MACRO_INSTS.DECODED	Counts the number of instructions decoded, (but not necessarily executed or retired).	
D1H	01H	UOPS_DECODED.STALL_CYCLES	Counts the cycles of decoder stalls. INV=1, Cmask=1.	
D1H	02H	UOPS_DECODED.MS	Counts the number of Uops decoded by the Microcode Sequencer, MS. The MS delivers uops when the instruction is more than 4 uops long or a microcode assist is occurring.	
D1H	04H	UOPS_DECODED.ESP_FOLDING	Counts number of stack pointer (ESP) instructions decoded: push, pop, call, ret, etc. ESP instructions do not generate a Uop to increment or decrement ESP. Instead, they update an ESP_Offset register that keeps track of the delta to the current value of the ESP register.	
D1H	08H	UOPS_DECODED.ESP_SYNC	Counts number of stack pointer (ESP) sync operations where an ESP instruction is corrected by adding the ESP offset register to the current value of the ESP register.	
D2H	01H	RAT_STALLS.FLAGS	Counts the number of cycles during which execution stalled due to several reasons, one of which is a partial flag register stall. A partial register stall may occur when two conditions are met: 1) an instruction modifies some, but not all, of the flags in the flag register and 2) the next instruction, which depends on flags, depends on flags that were not modified by this instruction.	
D2H	02H	RAT_STALLS.REGISTERS	This event counts the number of cycles instruction execution latency became longer than the defined latency because the instruction used a register that was partially written by previous instruction.	
D2H	04H	RAT_STALLS.ROB_READ_PORT	Counts the number of cycles when ROB read port stalls occurred, which did not allow new micro-ops to enter the out-of-order pipeline. Note that, at this stage in the pipeline, additional stalls may occur at the same cycle and prevent the stalled micro-ops from entering the pipe. In such a case, micro-ops retry entering the execution pipe in the next cycle and the ROB-read port stall is counted again.	
D2H	08H	RAT_STALLS.SCOREBOARD	Counts the cycles where we stall due to microarchitecturally required serialization. Microcode scoreboarding stalls.	

Table 19-21. Non-Architectural Performance Events In the Processor Core for Processors Based on Intel® Microarchitecture Code Name Westmere (Contd.)

Event Num.	Umask Value	Event Mask Mnemonic	Description	Comment
D2H	0FH	RAT_STALLS.ANY	Counts all Register Allocation Table stall cycles due to: Cycles when ROB read port stalls occurred, which did not allow new micro-ops to enter the execution pipe, Cycles when partial register stalls occurred, Cycles when flag stalls occurred, Cycles floating-point unit (FPU) status word stalls occurred. To count each of these conditions separately use the events: RAT_STALLS.ROB_READ_PORT, RAT_STALLS.PARTIAL, RAT_STALLS.FLAGS, and RAT_STALLS.FPSW.	
D4H	01H	SEG_RENAME_STALLS	Counts the number of stall cycles due to the lack of renaming resources for the ES, DS, FS, and GS segment registers. If a segment is renamed but not retired and a second update to the same segment occurs, a stall occurs in the front end of the pipeline until the renamed segment retires.	
D5H	01H	ES_REG_RENAMES	Counts the number of times the ES segment register is renamed.	
DBH	01H	UOP_UNFUSION	Counts unfusion events due to floating point exception to a fused uop.	
E0H	01H	BR_INST_DECODED	Counts the number of branch instructions decoded.	
E5H	01H	BPU_MISSED_CALL_RET	Counts number of times the Branch Prediction Unit missed predicting a call or return branch.	
E6H	01H	BACLEAR.CLEAR	Counts the number of times the front end is resteered, mainly when the Branch Prediction Unit cannot provide a correct prediction and this is corrected by the Branch Address Calculator at the front end. This can occur if the code has many branches such that they cannot be consumed by the BPU. Each BACLEAR asserted by the BAC generates approximately an 8 cycle bubble in the instruction fetch pipeline. The effect on total execution time depends on the surrounding code.	
E6H	02H	BACLEAR.BAD_TARGET	Counts number of Branch Address Calculator clears (BACLEAR) asserted due to conditional branch instructions in which there was a target hit but the direction was wrong. Each BACLEAR asserted by the BAC generates approximately an 8 cycle bubble in the instruction fetch pipeline.	
E8H	01H	BPU_CLEARS.EARLY	Counts early (normal) Branch Prediction Unit clears: BPU predicted a taken branch after incorrectly assuming that it was not taken.	The BPU clear leads to 2 cycle bubble in the front end.
E8H	02H	BPU_CLEARS.LATE	Counts late Branch Prediction Unit clears due to Most Recently Used conflicts. The PBU clear leads to a 3 cycle bubble in the front end.	
ECH	01H	THREAD_ACTIVE	Counts cycles threads are active.	
F0H	01H	L2_TRANSACTIONS.LOAD	Counts L2 load operations due to HW prefetch or demand loads.	

Table 19-21. Non-Architectural Performance Events In the Processor Core for Processors Based on Intel® Microarchitecture Code Name Westmere (Contd.)

Event Num.	Umask Value	Event Mask Mnemonic	Description	Comment
F0H	02H	L2_TRANSACTIONS.RFO	Counts L2 RFO operations due to HW prefetch or demand RFOs.	
F0H	04H	L2_TRANSACTIONS.IFETCH	Counts L2 instruction fetch operations due to HW prefetch or demand ifetch.	
F0H	08H	L2_TRANSACTIONS.PREFETCH	Counts L2 prefetch operations.	
F0H	10H	L2_TRANSACTIONS.L1D_WB	Counts L1D writeback operations to the L2.	
F0H	20H	L2_TRANSACTIONS.FILL	Counts L2 cache line fill operations due to load, RFO, L1D writeback or prefetch.	
F0H	40H	L2_TRANSACTIONS.WB	Counts L2 writeback operations to the L3.	
F0H	80H	L2_TRANSACTIONS.ANY	Counts all L2 cache operations.	
F1H	02H	L2_LINES_IN.S_STATE	Counts the number of cache lines allocated in the L2 cache in the S (shared) state.	
F1H	04H	L2_LINES_IN.E_STATE	Counts the number of cache lines allocated in the L2 cache in the E (exclusive) state.	
F1H	07H	L2_LINES_IN.ANY	Counts the number of cache lines allocated in the L2 cache.	
F2H	01H	L2_LINES_OUT.DEMAND_CLEAN	Counts L2 clean cache lines evicted by a demand request.	
F2H	02H	L2_LINES_OUT.DEMAND_DIRTY	Counts L2 dirty (modified) cache lines evicted by a demand request.	
F2H	04H	L2_LINES_OUT.PREFETCH_CLEAN	Counts L2 clean cache line evicted by a prefetch request.	
F2H	08H	L2_LINES_OUT.PREFETCH_DIRTY	Counts L2 modified cache line evicted by a prefetch request.	
F2H	0FH	L2_LINES_OUT.ANY	Counts all L2 cache lines evicted for any reason.	
F4H	04H	SQ_MISC.LRU_HINTS	Counts number of Super Queue LRU hints sent to L3.	
F4H	10H	SQ_MISC.SPLIT_LOCK	Counts the number of SQ lock splits across a cache line.	
F6H	01H	SQ_FULL_STALL_CYCLES	Counts cycles the Super Queue is full. Neither of the threads on this core will be able to access the uncore.	
F7H	01H	FP_ASSIST.ALL	Counts the number of floating point operations executed that required micro-code assist intervention. Assists are required in the following cases: SSE instructions, (Denormal input when the DAZ flag is off or Underflow result when the FTZ flag is off): x87 instructions, (NaN or denormal are loaded to a register or used as input from memory, Division by 0 or Underflow output).	
F7H	02H	FP_ASSIST.OUTPUT	Counts number of floating point micro-code assist when the output value (destination register) is invalid.	
F7H	04H	FP_ASSIST.INPUT	Counts number of floating point micro-code assist when the input value (one of the source operands to an FP instruction) is invalid.	

Table 19-21. Non-Architectural Performance Events In the Processor Core for Processors Based on Intel® Microarchitecture Code Name Westmere (Contd.)

Event Num.	Umask Value	Event Mask Mnemonic	Description	Comment
FDH	01H	SIMD_INT_64.PACKED_MPY	Counts number of SID integer 64 bit packed multiply operations.	
FDH	02H	SIMD_INT_64.PACKED_SHIFT	Counts number of SID integer 64 bit packed shift operations.	
FDH	04H	SIMD_INT_64.PACK	Counts number of SID integer 64 bit pack operations.	
FDH	08H	SIMD_INT_64.UNPACK	Counts number of SID integer 64 bit unpack operations.	
FDH	10H	SIMD_INT_64.PACKED_LOGICAL	Counts number of SID integer 64 bit logical operations.	
FDH	20H	SIMD_INT_64.PACKED_ARITH	Counts number of SID integer 64 bit arithmetic operations.	
FDH	40H	SIMD_INT_64.SHUFFLE_MOVE	Counts number of SID integer 64 bit shift or move operations.	

Non-architectural Performance monitoring events of the uncore sub-system for processors with CPUID signature of DisplayFamily_DisplayModel 06_25H, 06_2CH, and 06_1FH support performance events listed in Table 19-22.

Table 19-22. Non-Architectural Performance Events In the Processor Uncore for Processors Based on Intel® Microarchitecture Code Name Westmere

Event Num.	Umask Value	Event Mask Mnemonic	Description	Comment
00H	01H	UNC_GQ_CYCLES_FULL.READ_TRACKER	Uncore cycles Global Queue read tracker is full.	
00H	02H	UNC_GQ_CYCLES_FULL.WRITE_TRACKER	Uncore cycles Global Queue write tracker is full.	
00H	04H	UNC_GQ_CYCLES_FULL.PEER_PROBE_TRACKER	Uncore cycles Global Queue peer probe tracker is full. The peer probe tracker queue tracks snoops from the IOH and remote sockets.	
01H	01H	UNC_GQ_CYCLES_NOT_EMPTY.READ_TRACKER	Uncore cycles were Global Queue read tracker has at least one valid entry.	
01H	02H	UNC_GQ_CYCLES_NOT_EMPTY.WRITE_TRACKER	Uncore cycles were Global Queue write tracker has at least one valid entry.	
01H	04H	UNC_GQ_CYCLES_NOT_EMPTY.PEER_PROBE_TRACKER	Uncore cycles were Global Queue peer probe tracker has at least one valid entry. The peer probe tracker queue tracks IOH and remote socket snoops.	
02H	01H	UNC_GQ_OCCUPANCY.READ_TRACKER	Increments the number of queue entries (code read, data read, and RFOs) in the tread tracker. The GQ read tracker allocate to deallocate occupancy count is divided by the count to obtain the average read tracker latency.	
03H	01H	UNC_GQ_ALLOC.READ_TRACKER	Counts the number of tread tracker allocate to deallocate entries. The GQ read tracker allocate to deallocate occupancy count is divided by the count to obtain the average read tracker latency.	

Table 19-22. Non-Architectural Performance Events In the Processor Uncore for Processors Based on Intel® Microarchitecture Code Name Westmere (Contd.)

Event Num.	Umask Value	Event Mask Mnemonic	Description	Comment
03H	02H	UNC_GQ_ALLOC.RT_L3_MISS	Counts the number GQ read tracker entries for which a full cache line read has missed the L3. The GQ read tracker L3 miss to fill occupancy count is divided by this count to obtain the average cache line read L3 miss latency. The latency represents the time after which the L3 has determined that the cache line has missed. The time between a GQ read tracker allocation and the L3 determining that the cache line has missed is the average L3 hit latency. The total L3 cache line read miss latency is the hit latency + L3 miss latency.	
03H	04H	UNC_GQ_ALLOC.RT_TO_L3_RESP	Counts the number of GQ read tracker entries that are allocated in the read tracker queue that hit or miss the L3. The GQ read tracker L3 hit occupancy count is divided by this count to obtain the average L3 hit latency.	
03H	08H	UNC_GQ_ALLOC.RT_TO_RTID_ACQUIRED	Counts the number of GQ read tracker entries that are allocated in the read tracker, have missed in the L3 and have not acquired a Request Transaction ID. The GQ read tracker L3 miss to RTID acquired occupancy count is divided by this count to obtain the average latency for a read L3 miss to acquire an RTID.	
03H	10H	UNC_GQ_ALLOC.WT_TO_RTID_ACQUIRED	Counts the number of GQ write tracker entries that are allocated in the write tracker, have missed in the L3 and have not acquired a Request Transaction ID. The GQ write tracker L3 miss to RTID occupancy count is divided by this count to obtain the average latency for a write L3 miss to acquire an RTID.	
03H	20H	UNC_GQ_ALLOC.WRITE_TRACKER	Counts the number of GQ write tracker entries that are allocated in the write tracker queue that miss the L3. The GQ write tracker occupancy count is divided by this count to obtain the average L3 write miss latency.	
03H	40H	UNC_GQ_ALLOC.PEER_PROBE_TRACKER	Counts the number of GQ peer probe tracker (snoop) entries that are allocated in the peer probe tracker queue that miss the L3. The GQ peer probe occupancy count is divided by this count to obtain the average L3 peer probe miss latency.	
04H	01H	UNC_GQ_DATA.FROM_QPI	Cycles Global Queue Quickpath Interface input data port is busy importing data from the Quickpath Interface. Each cycle the input port can transfer 8 or 16 bytes of data.	
04H	02H	UNC_GQ_DATA.FROM_QMC	Cycles Global Queue Quickpath Memory Interface input data port is busy importing data from the Quickpath Memory Interface. Each cycle the input port can transfer 8 or 16 bytes of data.	
04H	04H	UNC_GQ_DATA.FROM_L3	Cycles GQ L3 input data port is busy importing data from the Last Level Cache. Each cycle the input port can transfer 32 bytes of data.	
04H	08H	UNC_GQ_DATA.FROM_CORES_02	Cycles GQ Core 0 and 2 input data port is busy importing data from processor cores 0 and 2. Each cycle the input port can transfer 32 bytes of data.	

Table 19-22. Non-Architectural Performance Events In the Processor Uncore for Processors Based on Intel® Microarchitecture Code Name Westmere (Contd.)

Event Num.	Umask Value	Event Mask Mnemonic	Description	Comment
04H	10H	UNC_GQ_DATA.FROM_CORES_13	Cycles GQ Core 1 and 3 input data port is busy importing data from processor cores 1 and 3. Each cycle the input port can transfer 32 bytes of data.	
05H	01H	UNC_GQ_DATA.TO_QPI_QMC	Cycles GQ QPI and QMC output data port is busy sending data to the Quickpath Interface or Quickpath Memory Interface. Each cycle the output port can transfer 32 bytes of data.	
05H	02H	UNC_GQ_DATA.TO_L3	Cycles GQ L3 output data port is busy sending data to the Last Level Cache. Each cycle the output port can transfer 32 bytes of data.	
05H	04H	UNC_GQ_DATA.TO_CORES	Cycles GQ Core output data port is busy sending data to the Cores. Each cycle the output port can transfer 32 bytes of data.	
06H	01H	UNC_SNP_RESP_TO_LOCAL_HOME.I_STATE	Number of snoop responses to the local home that L3 does not have the referenced cache line.	
06H	02H	UNC_SNP_RESP_TO_LOCAL_HOME.S_STATE	Number of snoop responses to the local home that L3 has the referenced line cached in the S state.	
06H	04H	UNC_SNP_RESP_TO_LOCAL_HOME.FWD_S_STATE	Number of responses to code or data read snoops to the local home that the L3 has the referenced cache line in the E state. The L3 cache line state is changed to the S state and the line is forwarded to the local home in the S state.	
06H	08H	UNC_SNP_RESP_TO_LOCAL_HOME.FWD_I_STATE	Number of responses to read invalidate snoops to the local home that the L3 has the referenced cache line in the M state. The L3 cache line state is invalidated and the line is forwarded to the local home in the M state.	
06H	10H	UNC_SNP_RESP_TO_LOCAL_HOME.CONFLICT	Number of conflict snoop responses sent to the local home.	
06H	20H	UNC_SNP_RESP_TO_LOCAL_HOME.WB	Number of responses to code or data read snoops to the local home that the L3 has the referenced line cached in the M state.	
07H	01H	UNC_SNP_RESP_TO_REMOTE_HOME.I_STATE	Number of snoop responses to a remote home that L3 does not have the referenced cache line.	
07H	02H	UNC_SNP_RESP_TO_REMOTE_HOME.S_STATE	Number of snoop responses to a remote home that L3 has the referenced line cached in the S state.	
07H	04H	UNC_SNP_RESP_TO_REMOTE_HOME.FWD_S_STATE	Number of responses to code or data read snoops to a remote home that the L3 has the referenced cache line in the E state. The L3 cache line state is changed to the S state and the line is forwarded to the remote home in the S state.	
07H	08H	UNC_SNP_RESP_TO_REMOTE_HOME.FWD_I_STATE	Number of responses to read invalidate snoops to a remote home that the L3 has the referenced cache line in the M state. The L3 cache line state is invalidated and the line is forwarded to the remote home in the M state.	
07H	10H	UNC_SNP_RESP_TO_REMOTE_HOME.CONFLICT	Number of conflict snoop responses sent to the local home.	

Table 19-22. Non-Architectural Performance Events In the Processor Uncore for Processors Based on Intel® Microarchitecture Code Name Westmere (Contd.)

Event Num.	Umask Value	Event Mask Mnemonic	Description	Comment
07H	20H	UNC_SNP_RESP_TO_REMOTE_HOME.WB	Number of responses to code or data read snoops to a remote home that the L3 has the referenced line cached in the M state.	
07H	24H	UNC_SNP_RESP_TO_REMOTE_HOME.HITM	Number of HITM snoop responses to a remote home.	
08H	01H	UNC_L3_HITS.READ	Number of code read, data read and RFO requests that hit in the L3.	
08H	02H	UNC_L3_HITS.WRITE	Number of writeback requests that hit in the L3. Writebacks from the cores will always result in L3 hits due to the inclusive property of the L3.	
08H	04H	UNC_L3_HITS.PROBE	Number of snoops from IOH or remote sockets that hit in the L3.	
08H	03H	UNC_L3_HITS.ANY	Number of reads and writes that hit the L3.	
09H	01H	UNC_L3_MISS.READ	Number of code read, data read and RFO requests that miss the L3.	
09H	02H	UNC_L3_MISS.WRITE	Number of writeback requests that miss the L3. Should always be zero as writebacks from the cores will always result in L3 hits due to the inclusive property of the L3.	
09H	04H	UNC_L3_MISS.PROBE	Number of snoops from IOH or remote sockets that miss the L3.	
09H	03H	UNC_L3_MISS.ANY	Number of reads and writes that miss the L3.	
0AH	01H	UNC_L3_LINES_IN.M_STATE	Counts the number of L3 lines allocated in M state. The only time a cache line is allocated in the M state is when the line was forwarded in M state is forwarded due to a Snoop Read Invalidate Own request.	
0AH	02H	UNC_L3_LINES_IN.E_STATE	Counts the number of L3 lines allocated in E state.	
0AH	04H	UNC_L3_LINES_IN.S_STATE	Counts the number of L3 lines allocated in S state.	
0AH	08H	UNC_L3_LINES_IN.F_STATE	Counts the number of L3 lines allocated in F state.	
0AH	0FH	UNC_L3_LINES_IN.ANY	Counts the number of L3 lines allocated in any state.	
0BH	01H	UNC_L3_LINES_OUT.M_STATE	Counts the number of L3 lines victimized that were in the M state. When the victim cache line is in M state, the line is written to its home cache agent which can be either local or remote.	
0BH	02H	UNC_L3_LINES_OUT.E_STATE	Counts the number of L3 lines victimized that were in the E state.	
0BH	04H	UNC_L3_LINES_OUT.S_STATE	Counts the number of L3 lines victimized that were in the S state.	
0BH	08H	UNC_L3_LINES_OUT.I_STATE	Counts the number of L3 lines victimized that were in the I state.	
0BH	10H	UNC_L3_LINES_OUT.F_STATE	Counts the number of L3 lines victimized that were in the F state.	
0BH	1FH	UNC_L3_LINES_OUT.ANY	Counts the number of L3 lines victimized in any state.	
0CH	01H	UNC_GQ_SNOOP.GOTO_S	Counts the number of remote snoops that have requested a cache line be set to the S state.	

Table 19-22. Non-Architectural Performance Events In the Processor Uncore for Processors Based on Intel® Microarchitecture Code Name Westmere (Contd.)

Event Num.	Umask Value	Event Mask Mnemonic	Description	Comment
0CH	02H	UNC_GQ_SNOOP.GOTO_I	Counts the number of remote snoops that have requested a cache line be set to the I state.	
0CH	04H	UNC_GQ_SNOOP.GOTO_S_HIT_E	Counts the number of remote snoops that have requested a cache line be set to the S state from E state.	Requires writing MSR 301H with mask = 2H.
0CH	04H	UNC_GQ_SNOOP.GOTO_S_HIT_F	Counts the number of remote snoops that have requested a cache line be set to the S state from F (forward) state.	Requires writing MSR 301H with mask = 8H.
0CH	04H	UNC_GQ_SNOOP.GOTO_S_HIT_M	Counts the number of remote snoops that have requested a cache line be set to the S state from M state.	Requires writing MSR 301H with mask = 1H.
0CH	04H	UNC_GQ_SNOOP.GOTO_S_HIT_S	Counts the number of remote snoops that have requested a cache line be set to the S state from S state.	Requires writing MSR 301H with mask = 4H.
0CH	08H	UNC_GQ_SNOOP.GOTO_I_HIT_E	Counts the number of remote snoops that have requested a cache line be set to the I state from E state.	Requires writing MSR 301H with mask = 2H.
0CH	08H	UNC_GQ_SNOOP.GOTO_I_HIT_F	Counts the number of remote snoops that have requested a cache line be set to the I state from F (forward) state.	Requires writing MSR 301H with mask = 8H.
0CH	08H	UNC_GQ_SNOOP.GOTO_I_HIT_M	Counts the number of remote snoops that have requested a cache line be set to the I state from M state.	Requires writing MSR 301H with mask = 1H.
0CH	08H	UNC_GQ_SNOOP.GOTO_I_HIT_S	Counts the number of remote snoops that have requested a cache line be set to the I state from S state.	Requires writing MSR 301H with mask = 4H.
20H	01H	UNC_QHL_REQUESTS.IOH_READS	Counts number of Quickpath Home Logic read requests from the IOH.	
20H	02H	UNC_QHL_REQUESTS.IOH_WRITES	Counts number of Quickpath Home Logic write requests from the IOH.	
20H	04H	UNC_QHL_REQUESTS.REMOTE_READS	Counts number of Quickpath Home Logic read requests from a remote socket.	
20H	08H	UNC_QHL_REQUESTS.REMOTE_WRITES	Counts number of Quickpath Home Logic write requests from a remote socket.	
20H	10H	UNC_QHL_REQUESTS.LOCAL_READS	Counts number of Quickpath Home Logic read requests from the local socket.	
20H	20H	UNC_QHL_REQUESTS.LOCAL_WRITES	Counts number of Quickpath Home Logic write requests from the local socket.	
21H	01H	UNC_QHL_CYCLES_FULL.IOH	Counts uclk cycles all entries in the Quickpath Home Logic IOH are full.	
21H	02H	UNC_QHL_CYCLES_FULL.REMOTE	Counts uclk cycles all entries in the Quickpath Home Logic remote tracker are full.	
21H	04H	UNC_QHL_CYCLES_FULL.LOCAL	Counts uclk cycles all entries in the Quickpath Home Logic local tracker are full.	
22H	01H	UNC_QHL_CYCLES_NOT_EMPTY.IOH	Counts uclk cycles all entries in the Quickpath Home Logic IOH is busy.	

Table 19-22. Non-Architectural Performance Events In the Processor Uncore for Processors Based on Intel® Microarchitecture Code Name Westmere (Contd.)

Event Num.	Umask Value	Event Mask Mnemonic	Description	Comment
22H	02H	UNC_QHL_CYCLES_NOT_EMPTY.REMOTE	Counts uclk cycles all entries in the Quickpath Home Logic remote tracker is busy.	
22H	04H	UNC_QHL_CYCLES_NOT_EMPTY.LOCAL	Counts uclk cycles all entries in the Quickpath Home Logic local tracker is busy.	
23H	01H	UNC_QHL_OCCUPANCY.IOH	QHL IOH tracker allocate to deallocate read occupancy.	
23H	02H	UNC_QHL_OCCUPANCY.REMOTE	QHL remote tracker allocate to deallocate read occupancy.	
23H	04H	UNC_QHL_OCCUPANCY.LOCAL	QHL local tracker allocate to deallocate read occupancy.	
24H	02H	UNC_QHL_ADDRESS_CONFLICTS.2WAY	Counts number of QHL Active Address Table (AAT) entries that saw a max of 2 conflicts. The AAT is a structure that tracks requests that are in conflict. The requests themselves are in the home tracker entries. The count is reported when an AAT entry deallocates.	
24H	04H	UNC_QHL_ADDRESS_CONFLICTS.3WAY	Counts number of QHL Active Address Table (AAT) entries that saw a max of 3 conflicts. The AAT is a structure that tracks requests that are in conflict. The requests themselves are in the home tracker entries. The count is reported when an AAT entry deallocates.	
25H	01H	UNC_QHL_CONFLICT_CYCLES.IOH	Counts cycles the Quickpath Home Logic IOH Tracker contains two or more requests with an address conflict. A max of 3 requests can be in conflict.	
25H	02H	UNC_QHL_CONFLICT_CYCLES.REMOTE	Counts cycles the Quickpath Home Logic Remote Tracker contains two or more requests with an address conflict. A max of 3 requests can be in conflict.	
25H	04H	UNC_QHL_CONFLICT_CYCLES.LOCAL	Counts cycles the Quickpath Home Logic Local Tracker contains two or more requests with an address conflict. A max of 3 requests can be in conflict.	
26H	01H	UNC_QHL_TO_QMC_BYPASS	Counts number or requests to the Quickpath Memory Controller that bypass the Quickpath Home Logic. All local accesses can be bypassed. For remote requests, only read requests can be bypassed.	
28H	01H	UNC_QMC_ISOC_FULL.READ.CH0	Counts cycles all the entries in the DRAM channel 0 high priority queue are occupied with isochronous read requests.	
28H	02H	UNC_QMC_ISOC_FULL.READ.CH1	Counts cycles all the entries in the DRAM channel 1high priority queue are occupied with isochronous read requests.	
28H	04H	UNC_QMC_ISOC_FULL.READ.CH2	Counts cycles all the entries in the DRAM channel 2 high priority queue are occupied with isochronous read requests.	
28H	08H	UNC_QMC_ISOC_FULL.WRITE.CH0	Counts cycles all the entries in the DRAM channel 0 high priority queue are occupied with isochronous write requests.	
28H	10H	UNC_QMC_ISOC_FULL.WRITE.CH1	Counts cycles all the entries in the DRAM channel 1 high priority queue are occupied with isochronous write requests.	

Table 19-22. Non-Architectural Performance Events In the Processor Uncore for Processors Based on Intel® Microarchitecture Code Name Westmere (Contd.)

Event Num.	Umask Value	Event Mask Mnemonic	Description	Comment
28H	20H	UNC_QMC_ISOC_FULL.WRITE.CH2	Counts cycles all the entries in the DRAM channel 2 high priority queue are occupied with isochronous write requests.	
29H	01H	UNC_QMC_BUSY.READ.CH0	Counts cycles where Quickpath Memory Controller has at least 1 outstanding read request to DRAM channel 0.	
29H	02H	UNC_QMC_BUSY.READ.CH1	Counts cycles where Quickpath Memory Controller has at least 1 outstanding read request to DRAM channel 1.	
29H	04H	UNC_QMC_BUSY.READ.CH2	Counts cycles where Quickpath Memory Controller has at least 1 outstanding read request to DRAM channel 2.	
29H	08H	UNC_QMC_BUSY.WRITE.CH0	Counts cycles where Quickpath Memory Controller has at least 1 outstanding write request to DRAM channel 0.	
29H	10H	UNC_QMC_BUSY.WRITE.CH1	Counts cycles where Quickpath Memory Controller has at least 1 outstanding write request to DRAM channel 1.	
29H	20H	UNC_QMC_BUSY.WRITE.CH2	Counts cycles where Quickpath Memory Controller has at least 1 outstanding write request to DRAM channel 2.	
2AH	01H	UNC_QMC_OCCUPANCY.CH0	IMC channel 0 normal read request occupancy.	
2AH	02H	UNC_QMC_OCCUPANCY.CH1	IMC channel 1 normal read request occupancy.	
2AH	04H	UNC_QMC_OCCUPANCY.CH2	IMC channel 2 normal read request occupancy.	
2AH	07H	UNC_QMC_OCCUPANCY.ANY	Normal read request occupancy for any channel.	
2BH	01H	UNC_QMC_ISSOC_OCCUPANCY.CH0	IMC channel 0 issoc read request occupancy.	
2BH	02H	UNC_QMC_ISSOC_OCCUPANCY.CH1	IMC channel 1 issoc read request occupancy.	
2BH	04H	UNC_QMC_ISSOC_OCCUPANCY.CH2	IMC channel 2 issoc read request occupancy.	
2BH	07H	UNC_QMC_ISSOC_READS.ANY	IMC issoc read request occupancy.	
2CH	01H	UNC_QMC_NORMAL_READS.CH0	Counts the number of Quickpath Memory Controller channel 0 medium and low priority read requests. The QMC channel 0 normal read occupancy divided by this count provides the average QMC channel 0 read latency.	
2CH	02H	UNC_QMC_NORMAL_READS.CH1	Counts the number of Quickpath Memory Controller channel 1 medium and low priority read requests. The QMC channel 1 normal read occupancy divided by this count provides the average QMC channel 1 read latency.	
2CH	04H	UNC_QMC_NORMAL_READS.CH2	Counts the number of Quickpath Memory Controller channel 2 medium and low priority read requests. The QMC channel 2 normal read occupancy divided by this count provides the average QMC channel 2 read latency.	

Table 19-22. Non-Architectural Performance Events In the Processor Uncore for Processors Based on Intel® Microarchitecture Code Name Westmere (Contd.)

Event Num.	Umask Value	Event Mask Mnemonic	Description	Comment
2CH	07H	UNC_QMC_NORMAL_READS.ANY	Counts the number of Quickpath Memory Controller medium and low priority read requests. The QMC normal read occupancy divided by this count provides the average QMC read latency.	
2DH	01H	UNC_QMC_HIGH_PRIORITY_READS.CH0	Counts the number of Quickpath Memory Controller channel 0 high priority isochronous read requests.	
2DH	02H	UNC_QMC_HIGH_PRIORITY_READS.CH1	Counts the number of Quickpath Memory Controller channel 1 high priority isochronous read requests.	
2DH	04H	UNC_QMC_HIGH_PRIORITY_READS.CH2	Counts the number of Quickpath Memory Controller channel 2 high priority isochronous read requests.	
2DH	07H	UNC_QMC_HIGH_PRIORITY_READS.ANY	Counts the number of Quickpath Memory Controller high priority isochronous read requests.	
2EH	01H	UNC_QMC_CRITICAL_PRIORITY_READS.CH0	Counts the number of Quickpath Memory Controller channel 0 critical priority isochronous read requests.	
2EH	02H	UNC_QMC_CRITICAL_PRIORITY_READS.CH1	Counts the number of Quickpath Memory Controller channel 1 critical priority isochronous read requests.	
2EH	04H	UNC_QMC_CRITICAL_PRIORITY_READS.CH2	Counts the number of Quickpath Memory Controller channel 2 critical priority isochronous read requests.	
2EH	07H	UNC_QMC_CRITICAL_PRIORITY_READS.ANY	Counts the number of Quickpath Memory Controller critical priority isochronous read requests.	
2FH	01H	UNC_QMC_WRITES.FULL.CH0	Counts number of full cache line writes to DRAM channel 0.	
2FH	02H	UNC_QMC_WRITES.FULL.CH1	Counts number of full cache line writes to DRAM channel 1.	
2FH	04H	UNC_QMC_WRITES.FULL.CH2	Counts number of full cache line writes to DRAM channel 2.	
2FH	07H	UNC_QMC_WRITES.FULL.ANY	Counts number of full cache line writes to DRAM.	
2FH	08H	UNC_QMC_WRITES.PARTIAL.CH0	Counts number of partial cache line writes to DRAM channel 0.	
2FH	10H	UNC_QMC_WRITES.PARTIAL.CH1	Counts number of partial cache line writes to DRAM channel 1.	
2FH	20H	UNC_QMC_WRITES.PARTIAL.CH2	Counts number of partial cache line writes to DRAM channel 2.	
2FH	38H	UNC_QMC_WRITES.PARTIAL.ANY	Counts number of partial cache line writes to DRAM.	
30H	01H	UNC_QMC_CANCEL.CH0	Counts number of DRAM channel 0 cancel requests.	
30H	02H	UNC_QMC_CANCEL.CH1	Counts number of DRAM channel 1 cancel requests.	
30H	04H	UNC_QMC_CANCEL.CH2	Counts number of DRAM channel 2 cancel requests.	
30H	07H	UNC_QMC_CANCEL.ANY	Counts number of DRAM cancel requests.	
31H	01H	UNC_QMC_PRIORITY_UPDATES.CH0	Counts number of DRAM channel 0 priority updates. A priority update occurs when an ISOC high or critical request is received by the QHL and there is a matching request with normal priority that has already been issued to the QMC. In this instance, the QHL will send a priority update to QMC to expedite the request.	

Table 19-22. Non-Architectural Performance Events In the Processor Uncore for Processors Based on Intel® Microarchitecture Code Name Westmere (Contd.)

Event Num.	Umask Value	Event Mask Mnemonic	Description	Comment
31H	02H	UNC_QMC_PRIORITY_UPDATES.CH1	Counts number of DRAM channel 1 priority updates. A priority update occurs when an ISOC high or critical request is received by the QHL and there is a matching request with normal priority that has already been issued to the QMC. In this instance, the QHL will send a priority update to QMC to expedite the request.	
31H	04H	UNC_QMC_PRIORITY_UPDATES.CH2	Counts number of DRAM channel 2 priority updates. A priority update occurs when an ISOC high or critical request is received by the QHL and there is a matching request with normal priority that has already been issued to the QMC. In this instance, the QHL will send a priority update to QMC to expedite the request.	
31H	07H	UNC_QMC_PRIORITY_UPDATES.ANY	Counts number of DRAM priority updates. A priority update occurs when an ISOC high or critical request is received by the QHL and there is a matching request with normal priority that has already been issued to the QMC. In this instance, the QHL will send a priority update to QMC to expedite the request.	
32H	01H	UNC_IMC_RETRY.CH0	Counts number of IMC DRAM channel 0 retries. DRAM retry only occurs when configured in RAS mode.	
32H	02H	UNC_IMC_RETRY.CH1	Counts number of IMC DRAM channel 1 retries. DRAM retry only occurs when configured in RAS mode.	
32H	04H	UNC_IMC_RETRY.CH2	Counts number of IMC DRAM channel 2 retries. DRAM retry only occurs when configured in RAS mode.	
32H	07H	UNC_IMC_RETRY.ANY	Counts number of IMC DRAM retries from any channel. DRAM retry only occurs when configured in RAS mode.	
33H	01H	UNC_QHL_FRC_ACK_CNFLTS.IOH	Counts number of Force Acknowledge Conflict messages sent by the Quickpath Home Logic to the IOH.	
33H	02H	UNC_QHL_FRC_ACK_CNFLTS.REMOTE	Counts number of Force Acknowledge Conflict messages sent by the Quickpath Home Logic to the remote home.	
33H	04H	UNC_QHL_FRC_ACK_CNFLTS.LOCAL	Counts number of Force Acknowledge Conflict messages sent by the Quickpath Home Logic to the local home.	
33H	07H	UNC_QHL_FRC_ACK_CNFLTS.ANY	Counts number of Force Acknowledge Conflict messages sent by the Quickpath Home Logic.	
34H	01H	UNC_QHL_SLEEPS.IOH_ORDER	Counts number of occurrences a request was put to sleep due to IOH ordering (write after read) conflicts. While in the sleep state, the request is not eligible to be scheduled to the QMC.	
34H	02H	UNC_QHL_SLEEPS.REMOTE_ORDER	Counts number of occurrences a request was put to sleep due to remote socket ordering (write after read) conflicts. While in the sleep state, the request is not eligible to be scheduled to the QMC.	
34H	04H	UNC_QHL_SLEEPS.LOCAL_ORDER	Counts number of occurrences a request was put to sleep due to local socket ordering (write after read) conflicts. While in the sleep state, the request is not eligible to be scheduled to the QMC.	

Table 19-22. Non-Architectural Performance Events In the Processor Uncore for Processors Based on Intel® Microarchitecture Code Name Westmere (Contd.)

Event Num.	Umask Value	Event Mask Mnemonic	Description	Comment
34H	08H	UNC_QHL_SLEEPS.IOH_CONFLICT	Counts number of occurrences a request was put to sleep due to IOH address conflicts. While in the sleep state, the request is not eligible to be scheduled to the QMC.	
34H	10H	UNC_QHL_SLEEPS.REMOTE_CONFLICT	Counts number of occurrences a request was put to sleep due to remote socket address conflicts. While in the sleep state, the request is not eligible to be scheduled to the QMC.	
34H	20H	UNC_QHL_SLEEPS.LOCAL_CONFLICT	Counts number of occurrences a request was put to sleep due to local socket address conflicts. While in the sleep state, the request is not eligible to be scheduled to the QMC.	
35H	01H	UNC_ADDR_OPCODE_MATCH.IOH	Counts number of requests from the IOH, address/opcode of request is qualified by mask value written to MSR 396H. The following mask values are supported: 0: NONE 40000000_00000000H:RSPFWDI 40001A00_00000000H:RSPFWDS 40001D00_00000000H:RSPIWB	Match opcode/address by writing MSR 396H with mask supported mask value.
35H	02H	UNC_ADDR_OPCODE_MATCH.REMOTE	Counts number of requests from the remote socket, address/opcode of request is qualified by mask value written to MSR 396H. The following mask values are supported: 0: NONE 40000000_00000000H:RSPFWDI 40001A00_00000000H:RSPFWDS 40001D00_00000000H:RSPIWB	Match opcode/address by writing MSR 396H with mask supported mask value.
35H	04H	UNC_ADDR_OPCODE_MATCH.LOCAL	Counts number of requests from the local socket, address/opcode of request is qualified by mask value written to MSR 396H. The following mask values are supported: 0: NONE 40000000_00000000H:RSPFWDI 40001A00_00000000H:RSPFWDS 40001D00_00000000H:RSPIWB	Match opcode/address by writing MSR 396H with mask supported mask value.
40H	01H	UNC_QPI_TX_STALLED_SINGLE_FLIT.HOME.LINK_0	Counts cycles the Quickpath outbound link 0 HOME virtual channel is stalled due to lack of a VNA and VN0 credit. Note that this event does not filter out when a flit would not have been selected for arbitration because another virtual channel is getting arbitrated.	
40H	02H	UNC_QPI_TX_STALLED_SINGLE_FLIT.SNOOP.LINK_0	Counts cycles the Quickpath outbound link 0 SNOOP virtual channel is stalled due to lack of a VNA and VN0 credit. Note that this event does not filter out when a flit would not have been selected for arbitration because another virtual channel is getting arbitrated.	

Table 19-22. Non-Architectural Performance Events In the Processor Uncore for Processors Based on Intel® Microarchitecture Code Name Westmere (Contd.)

Event Num.	Umask Value	Event Mask Mnemonic	Description	Comment
40H	04H	UNC_QPI_TX_STALLED_SINGLE_FLIT.NDR.LINK_0	Counts cycles the Quickpath outbound link 0 non-data response virtual channel is stalled due to lack of a VNA and VN0 credit. Note that this event does not filter out when a flit would not have been selected for arbitration because another virtual channel is getting arbitrated.	
40H	08H	UNC_QPI_TX_STALLED_SINGLE_FLIT.HOME.LINK_1	Counts cycles the Quickpath outbound link 1 HOME virtual channel is stalled due to lack of a VNA and VN0 credit. Note that this event does not filter out when a flit would not have been selected for arbitration because another virtual channel is getting arbitrated.	
40H	10H	UNC_QPI_TX_STALLED_SINGLE_FLIT.SNOOP.LINK_1	Counts cycles the Quickpath outbound link 1 SNOOP virtual channel is stalled due to lack of a VNA and VN0 credit. Note that this event does not filter out when a flit would not have been selected for arbitration because another virtual channel is getting arbitrated.	
40H	20H	UNC_QPI_TX_STALLED_SINGLE_FLIT.NDR.LINK_1	Counts cycles the Quickpath outbound link 1 non-data response virtual channel is stalled due to lack of a VNA and VN0 credit. Note that this event does not filter out when a flit would not have been selected for arbitration because another virtual channel is getting arbitrated.	
40H	07H	UNC_QPI_TX_STALLED_SINGLE_FLIT.LINK_0	Counts cycles the Quickpath outbound link 0 virtual channels are stalled due to lack of a VNA and VN0 credit. Note that this event does not filter out when a flit would not have been selected for arbitration because another virtual channel is getting arbitrated.	
40H	38H	UNC_QPI_TX_STALLED_SINGLE_FLIT.LINK_1	Counts cycles the Quickpath outbound link 1 virtual channels are stalled due to lack of a VNA and VN0 credit. Note that this event does not filter out when a flit would not have been selected for arbitration because another virtual channel is getting arbitrated.	
41H	01H	UNC_QPI_TX_STALLED_MULTI_FLIT.DRS.LINK_0	Counts cycles the Quickpath outbound link 0 Data Response virtual channel is stalled due to lack of VNA and VN0 credits. Note that this event does not filter out when a flit would not have been selected for arbitration because another virtual channel is getting arbitrated.	
41H	02H	UNC_QPI_TX_STALLED_MULTI_FLIT.NCB.LINK_0	Counts cycles the Quickpath outbound link 0 Non-Coherent Bypass virtual channel is stalled due to lack of VNA and VN0 credits. Note that this event does not filter out when a flit would not have been selected for arbitration because another virtual channel is getting arbitrated.	
41H	04H	UNC_QPI_TX_STALLED_MULTI_FLIT.NCS.LINK_0	Counts cycles the Quickpath outbound link 0 Non-Coherent Standard virtual channel is stalled due to lack of VNA and VN0 credits. Note that this event does not filter out when a flit would not have been selected for arbitration because another virtual channel is getting arbitrated.	

Table 19-22. Non-Architectural Performance Events In the Processor Uncore for Processors Based on Intel® Microarchitecture Code Name Westmere (Contd.)

Event Num.	Umask Value	Event Mask Mnemonic	Description	Comment
41H	08H	UNC_QPI_TX_STALLED_MULTI_FLIT.DRS.LINK_1	Counts cycles the Quickpath outbound link 1 Data Response virtual channel is stalled due to lack of VNA and VN0 credits. Note that this event does not filter out when a flit would not have been selected for arbitration because another virtual channel is getting arbitrated.	
41H	10H	UNC_QPI_TX_STALLED_MULTI_FLIT.NCB.LINK_1	Counts cycles the Quickpath outbound link 1 Non-Coherent Bypass virtual channel is stalled due to lack of VNA and VN0 credits. Note that this event does not filter out when a flit would not have been selected for arbitration because another virtual channel is getting arbitrated.	
41H	20H	UNC_QPI_TX_STALLED_MULTI_FLIT.NCS.LINK_1	Counts cycles the Quickpath outbound link 1 Non-Coherent Standard virtual channel is stalled due to lack of VNA and VN0 credits. Note that this event does not filter out when a flit would not have been selected for arbitration because another virtual channel is getting arbitrated.	
41H	07H	UNC_QPI_TX_STALLED_MULTI_FLIT.LINK_0	Counts cycles the Quickpath outbound link 0 virtual channels are stalled due to lack of VNA and VN0 credits. Note that this event does not filter out when a flit would not have been selected for arbitration because another virtual channel is getting arbitrated.	
41H	38H	UNC_QPI_TX_STALLED_MULTI_FLIT.LINK_1	Counts cycles the Quickpath outbound link 1 virtual channels are stalled due to lack of VNA and VN0 credits. Note that this event does not filter out when a flit would not have been selected for arbitration because another virtual channel is getting arbitrated.	
42H	01H	UNC_QPI_TX_HEADER.FULL.LINK_0	Number of cycles that the header buffer in the Quickpath Interface outbound link 0 is full.	
42H	02H	UNC_QPI_TX_HEADER.BUSY.LINK_0	Number of cycles that the header buffer in the Quickpath Interface outbound link 0 is busy.	
42H	04H	UNC_QPI_TX_HEADER.FULL.LINK_1	Number of cycles that the header buffer in the Quickpath Interface outbound link 1 is full.	
42H	08H	UNC_QPI_TX_HEADER.BUSY.LINK_1	Number of cycles that the header buffer in the Quickpath Interface outbound link 1 is busy.	
43H	01H	UNC_QPI_RX_NO_PPT_CREDIT.STALLS.LINK_0	Number of cycles that snoop packets incoming to the Quickpath Interface link 0 are stalled and not sent to the GQ because the GQ Peer Probe Tracker (PPT) does not have any available entries.	
43H	02H	UNC_QPI_RX_NO_PPT_CREDIT.STALLS.LINK_1	Number of cycles that snoop packets incoming to the Quickpath Interface link 1 are stalled and not sent to the GQ because the GQ Peer Probe Tracker (PPT) does not have any available entries.	
60H	01H	UNC_DRAM_OPEN.CH0	Counts number of DRAM Channel 0 open commands issued either for read or write. To read or write data, the referenced DRAM page must first be opened.	

Table 19-22. Non-Architectural Performance Events In the Processor Uncore for Processors Based on Intel® Microarchitecture Code Name Westmere (Contd.)

Event Num.	Umask Value	Event Mask Mnemonic	Description	Comment
60H	02H	UNC_DRAM_OPEN.CH1	Counts number of DRAM Channel 1 open commands issued either for read or write. To read or write data, the referenced DRAM page must first be opened.	
60H	04H	UNC_DRAM_OPEN.CH2	Counts number of DRAM Channel 2 open commands issued either for read or write. To read or write data, the referenced DRAM page must first be opened.	
61H	01H	UNC_DRAM_PAGE_CLOSE.CH0	DRAM channel 0 command issued to CLOSE a page due to page idle timer expiration. Closing a page is done by issuing a precharge.	
61H	02H	UNC_DRAM_PAGE_CLOSE.CH1	DRAM channel 1 command issued to CLOSE a page due to page idle timer expiration. Closing a page is done by issuing a precharge.	
61H	04H	UNC_DRAM_PAGE_CLOSE.CH2	DRAM channel 2 command issued to CLOSE a page due to page idle timer expiration. Closing a page is done by issuing a precharge.	
62H	01H	UNC_DRAM_PAGE_MISS.CH0	Counts the number of precharges (PRE) that were issued to DRAM channel 0 because there was a page miss. A page miss refers to a situation in which a page is currently open and another page from the same bank needs to be opened. The new page experiences a page miss. Closing of the old page is done by issuing a precharge.	
62H	02H	UNC_DRAM_PAGE_MISS.CH1	Counts the number of precharges (PRE) that were issued to DRAM channel 1 because there was a page miss. A page miss refers to a situation in which a page is currently open and another page from the same bank needs to be opened. The new page experiences a page miss. Closing of the old page is done by issuing a precharge.	
62H	04H	UNC_DRAM_PAGE_MISS.CH2	Counts the number of precharges (PRE) that were issued to DRAM channel 2 because there was a page miss. A page miss refers to a situation in which a page is currently open and another page from the same bank needs to be opened. The new page experiences a page miss. Closing of the old page is done by issuing a precharge.	
63H	01H	UNC_DRAM_READ_CAS.CH0	Counts the number of times a read CAS command was issued on DRAM channel 0.	
63H	02H	UNC_DRAM_READ_CAS.AUTOPRE_CH0	Counts the number of times a read CAS command was issued on DRAM channel 0 where the command issued used the auto-precharge (auto page close) mode.	
63H	04H	UNC_DRAM_READ_CAS.CH1	Counts the number of times a read CAS command was issued on DRAM channel 1.	
63H	08H	UNC_DRAM_READ_CAS.AUTOPRE_CH1	Counts the number of times a read CAS command was issued on DRAM channel 1 where the command issued used the auto-precharge (auto page close) mode.	
63H	10H	UNC_DRAM_READ_CAS.CH2	Counts the number of times a read CAS command was issued on DRAM channel 2.	

Table 19-22. Non-Architectural Performance Events In the Processor Uncore for Processors Based on Intel® Microarchitecture Code Name Westmere (Contd.)

Event Num.	Umask Value	Event Mask Mnemonic	Description	Comment
63H	20H	UNC_DRAM_READ_CAS.AUTO PRE_CH2	Counts the number of times a read CAS command was issued on DRAM channel 2 where the command issued used the auto-precharge (auto page close) mode.	
64H	01H	UNC_DRAM_WRITE_CAS.CH0	Counts the number of times a write CAS command was issued on DRAM channel 0.	
64H	02H	UNC_DRAM_WRITE_CAS.AUTO PRE_CH0	Counts the number of times a write CAS command was issued on DRAM channel 0 where the command issued used the auto-precharge (auto page close) mode.	
64H	04H	UNC_DRAM_WRITE_CAS.CH1	Counts the number of times a write CAS command was issued on DRAM channel 1.	
64H	08H	UNC_DRAM_WRITE_CAS.AUTO PRE_CH1	Counts the number of times a write CAS command was issued on DRAM channel 1 where the command issued used the auto-precharge (auto page close) mode.	
64H	10H	UNC_DRAM_WRITE_CAS.CH2	Counts the number of times a write CAS command was issued on DRAM channel 2.	
64H	20H	UNC_DRAM_WRITE_CAS.AUTO PRE_CH2	Counts the number of times a write CAS command was issued on DRAM channel 2 where the command issued used the auto-precharge (auto page close) mode.	
65H	01H	UNC_DRAM_REFRESH.CH0	Counts number of DRAM channel 0 refresh commands. DRAM loses data content over time. In order to keep correct data content, the data values have to be refreshed periodically.	
65H	02H	UNC_DRAM_REFRESH.CH1	Counts number of DRAM channel 1 refresh commands. DRAM loses data content over time. In order to keep correct data content, the data values have to be refreshed periodically.	
65H	04H	UNC_DRAM_REFRESH.CH2	Counts number of DRAM channel 2 refresh commands. DRAM loses data content over time. In order to keep correct data content, the data values have to be refreshed periodically.	
66H	01H	UNC_DRAM_PRE_ALL.CH0	Counts number of DRAM Channel 0 precharge-all (PREALL) commands that close all open pages in a rank. PREALL is issued when the DRAM needs to be refreshed or needs to go into a power down mode.	
66H	02H	UNC_DRAM_PRE_ALL.CH1	Counts number of DRAM Channel 1 precharge-all (PREALL) commands that close all open pages in a rank. PREALL is issued when the DRAM needs to be refreshed or needs to go into a power down mode.	
66H	04H	UNC_DRAM_PRE_ALL.CH2	Counts number of DRAM Channel 2 precharge-all (PREALL) commands that close all open pages in a rank. PREALL is issued when the DRAM needs to be refreshed or needs to go into a power down mode.	
67H	01H	UNC_DRAM_THERMAL_THROT TLED	Uncore cycles DRAM was throttled due to its temperature being above the thermal throttling threshold.	
80H	01H	UNC_THERMAL_THROTTLING_ TEMP.CORE_0	Cycles that the PCU records that core 0 is above the thermal throttling threshold temperature.	

Table 19-22. Non-Architectural Performance Events In the Processor Uncore for Processors Based on Intel® Microarchitecture Code Name Westmere (Contd.)

Event Num.	Umask Value	Event Mask Mnemonic	Description	Comment
80H	02H	UNC_THERMAL_THROTTLING_TEMP.CORE_1	Cycles that the PCU records that core 1 is above the thermal throttling threshold temperature.	
80H	04H	UNC_THERMAL_THROTTLING_TEMP.CORE_2	Cycles that the PCU records that core 2 is above the thermal throttling threshold temperature.	
80H	08H	UNC_THERMAL_THROTTLING_TEMP.CORE_3	Cycles that the PCU records that core 3 is above the thermal throttling threshold temperature.	
81H	01H	UNC_THERMAL_THROTTLED_TEMP.CORE_0	Cycles that the PCU records that core 0 is in the power throttled state due to core's temperature being above the thermal throttling threshold.	
81H	02H	UNC_THERMAL_THROTTLED_TEMP.CORE_1	Cycles that the PCU records that core 1 is in the power throttled state due to core's temperature being above the thermal throttling threshold.	
81H	04H	UNC_THERMAL_THROTTLED_TEMP.CORE_2	Cycles that the PCU records that core 2 is in the power throttled state due to core's temperature being above the thermal throttling threshold.	
81H	08H	UNC_THERMAL_THROTTLED_TEMP.CORE_3	Cycles that the PCU records that core 3 is in the power throttled state due to core's temperature being above the thermal throttling threshold.	
82H	01H	UNC_PROCHOT_ASSERTION	Number of system assertions of PROCHOT indicating the entire processor has exceeded the thermal limit.	
83H	01H	UNC_THERMAL_THROTTLING_PROCHOT.CORE_0	Cycles that the PCU records that core 0 is a low power state due to the system asserting PROCHOT the entire processor has exceeded the thermal limit.	
83H	02H	UNC_THERMAL_THROTTLING_PROCHOT.CORE_1	Cycles that the PCU records that core 1 is a low power state due to the system asserting PROCHOT the entire processor has exceeded the thermal limit.	
83H	04H	UNC_THERMAL_THROTTLING_PROCHOT.CORE_2	Cycles that the PCU records that core 2 is a low power state due to the system asserting PROCHOT the entire processor has exceeded the thermal limit.	
83H	08H	UNC_THERMAL_THROTTLING_PROCHOT.CORE_3	Cycles that the PCU records that core 3 is a low power state due to the system asserting PROCHOT the entire processor has exceeded the thermal limit.	
84H	01H	UNC_TURBO_MODE.CORE_0	Uncore cycles that core 0 is operating in turbo mode.	
84H	02H	UNC_TURBO_MODE.CORE_1	Uncore cycles that core 1 is operating in turbo mode.	
84H	04H	UNC_TURBO_MODE.CORE_2	Uncore cycles that core 2 is operating in turbo mode.	
84H	08H	UNC_TURBO_MODE.CORE_3	Uncore cycles that core 3 is operating in turbo mode.	
85H	02H	UNC_CYCLES_UNHALTED_L3_FLL_ENABLE	Uncore cycles that at least one core is unhalted and all L3 ways are enabled.	
86H	01H	UNC_CYCLES_UNHALTED_L3_FLL_DISABLE	Uncore cycles that at least one core is unhalted and all L3 ways are disabled.	

19.11 PERFORMANCE MONITORING EVENTS FOR INTEL® XEON® PROCESSOR 5200, 5400 SERIES AND INTEL® CORE™2 EXTREME PROCESSORS QX 9000 SERIES

Processors based on the Enhanced Intel Core microarchitecture support the architectural and non-architectural performance-monitoring events listed in Table 19-1 and Table 19-25. In addition, they also support the following non-architectural performance-monitoring events listed in Table 19-23. Fixed counters support the architecture events defined in Table 19-24.

Table 19-23. Non-Architectural Performance Events for Processors Based on Enhanced Intel Core Microarchitecture

Event Num.	Umask Value	Event Mask Mnemonic	Description	Comment
C0H	08H	INST_RETIRED.VM_HOST	Instruction retired while in VMX root operations.	
D2H	10H	RAT_STAALS.OTHER_SERIALIZATION_STALLS	This event counts the number of stalls due to other RAT resource serialization not counted by Umask value 0FH.	

19.12 PERFORMANCE MONITORING EVENTS FOR INTEL® XEON® PROCESSOR 3000, 3200, 5100, 5300 SERIES AND INTEL® CORE™2 DUO PROCESSORS

Processors based on the Intel® Core™ microarchitecture support architectural and non-architectural performance-monitoring events.

Fixed-function performance counters are introduced first on processors based on Intel Core microarchitecture. Table 19-24 lists pre-defined performance events that can be counted using fixed-function performance counters.

Table 19-24. Fixed-Function Performance Counter and Pre-defined Performance Events

Fixed-Function Performance Counter	Address	Event Mask Mnemonic	Description
MSR_PERF_FIXED_CTR0/IA32_PERF_FIXED_CTR0	309H	Inst_Retired.Any	This event counts the number of instructions that retire execution. For instructions that consist of multiple micro-ops, this event counts the retirement of the last micro-op of the instruction. The counter continues counting during hardware interrupts, traps, and inside interrupt handlers.
MSR_PERF_FIXED_CTR1/IA32_PERF_FIXED_CTR1	30AH	CPU_CLK_UNHALTED.CORE	This event counts the number of core cycles while the core is not in a halt state. The core enters the halt state when it is running the HLT instruction. This event is a component in many key event ratios. The core frequency may change from time to time due to transitions associated with Enhanced Intel SpeedStep Technology or TM2. For this reason this event may have a changing ratio with regards to time. When the core frequency is constant, this event can approximate elapsed time while the core was not in halt state.
MSR_PERF_FIXED_CTR2/IA32_PERF_FIXED_CTR2	30BH	CPU_CLK_UNHALTED.REF	This event counts the number of reference cycles when the core is not in a halt state and not in a TM stop-clock state. The core enters the halt state when it is running the HLT instruction or the MWAIT instruction.

Table 19-24. Fixed-Function Performance Counter and Pre-defined Performance Events (Contd.)

Fixed-Function Performance Counter	Address	Event Mask Mnemonic	Description
			This event is not affected by core frequency changes (e.g., P states) but counts at the same frequency as the time stamp counter. This event can approximate elapsed time while the core was not in halt state and not in a TM stop-clock state. This event has a constant ratio with the CPU_CLK_UNHALTED.BUS event.

Table 19-25 lists general-purpose non-architectural performance-monitoring events supported in processors based on Intel® Core™ microarchitecture. For convenience, Table 19-25 also includes architectural events and describes minor model-specific behavior where applicable. Software must use a general-purpose performance counter to count events listed in Table 19-25.

Table 19-25. Non-Architectural Performance Events in Processors Based on Intel® Core™ Microarchitecture

Event Num	Umask Value	Event Name	Definition	Description and Comment
03H	02H	LOAD_BLOCK.STA	Loads blocked by a preceding store with unknown address.	This event indicates that loads are blocked by preceding stores. A load is blocked when there is a preceding store to an address that is not yet calculated. The number of events is greater or equal to the number of load operations that were blocked. If the load and the store are always to different addresses, check why the memory disambiguation mechanism is not working. To avoid such blocks, increase the distance between the store and the following load so that the store address is known at the time the load is dispatched.
03H	04H	LOAD_BLOCK.STD	Loads blocked by a preceding store with unknown data.	This event indicates that loads are blocked by preceding stores. A load is blocked when there is a preceding store to the same address and the stored data value is not yet known. The number of events is greater or equal to the number of load operations that were blocked. To avoid such blocks, increase the distance between the store and the dependent load, so that the store data is known at the time the load is dispatched.
03H	08H	LOAD_BLOCK. OVERLAP_STORE	Loads that partially overlap an earlier store, or 4-Kbyte aliased with a previous store.	This event indicates that loads are blocked due to a variety of reasons. Some of the triggers for this event are when a load is blocked by a preceding store, in one of the following: ▪ Some of the loaded byte locations are written by the preceding store and some are not. ▪ The load is from bytes written by the preceding store, the store is aligned to its size and either: ▪ The load's data size is one or two bytes and it is not aligned to the store. ▪ The load's data size is of four or eight bytes and the load is misaligned.

Table 19-25. Non-Architectural Performance Events in Processors Based on Intel® Core™ Microarchitecture (Contd.)

Event Num	Umask Value	Event Name	Definition	Description and Comment
				▪ The load is from bytes written by the preceding store, the store is misaligned and the load is not aligned on the beginning of the store. ▪ The load is split over an eight byte boundary (excluding 16-byte loads). ▪ The load and store have the same offset relative to the beginning of different 4-KByte pages. This case is also called 4-KByte aliasing. ▪ In all these cases the load is blocked until after the blocking store retires and the stored data is committed to the cache hierarchy.
03H	10H	LOAD_BLOCK. UNTIL_RETIRE	Loads blocked until retirement.	This event indicates that load operations were blocked until retirement. The number of events is greater or equal to the number of load operations that were blocked. This includes mainly uncacheable loads and split loads (loads that cross the cache line boundary) but may include other cases where loads are blocked until retirement.
03H	20H	LOAD_BLOCK.L1D	Loads blocked by the L1 data cache.	This event indicates that loads are blocked due to one or more reasons. Some triggers for this event are: ▪ The number of L1 data cache misses exceeds the maximum number of outstanding misses supported by the processor. This includes misses generated as result of demand fetches, software prefetches or hardware prefetches. ▪ Cache line split loads. ▪ Partial reads, such as reads to un-cacheable memory, I/O instructions and more. ▪ A locked load operation is in progress. The number of events is greater or equal to the number of load operations that were blocked.
04H	01H	SB_DRAIN_ CYCLES	Cycles while stores are blocked due to store buffer drain.	This event counts every cycle during which the store buffer is draining. This includes: ▪ Serializing operations such as CPUID ▪ Synchronizing operations such as XCHG ▪ Interrupt acknowledgment ▪ Other conditions, such as cache flushing
04H	02H	STORE_BLOCK. ORDER	Cycles while store is waiting for a preceding store to be globally observed.	This event counts the total duration, in number of cycles, which stores are waiting for a preceding stored cache line to be observed by other cores. This situation happens as a result of the strong store ordering behavior, as defined in "Memory Ordering," Chapter 8, *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*. The stall may occur and be noticeable if there are many cases when a store either misses the L1 data cache or hits a cache line in the Shared state. If the store requires a bus transaction to read the cache line then the stall ends when snoop response for the bus transaction arrives.
04H	08H	STORE_BLOCK. SNOOP	A store is blocked due to a conflict with an external or internal snoop.	This event counts the number of cycles the store port was used for snooping the L1 data cache and a store was stalled by the snoop. The store is typically resubmitted one cycle later.

Table 19-25. Non-Architectural Performance Events in Processors Based on Intel® Core™ Microarchitecture (Contd.)

Event Num	Umask Value	Event Name	Definition	Description and Comment
06H	00H	SEGMENT_REG_ LOADS	Number of segment register loads.	This event counts the number of segment register load operations. Instructions that load new values into segment registers cause a penalty. This event indicates performance issues in 16-bit code. If this event occurs frequently, it may be useful to calculate the number of instructions retired per segment register load. If the resulting calculation is low (on average a small number of instructions are executed between segment register loads), then the code's segment register usage should be optimized. As a result of branch misprediction, this event is speculative and may include segment register loads that do not actually occur. However, most segment register loads are internally serialized and such speculative effects are minimized.
07H	00H	SSE_PRE_EXEC. NTA	Streaming SIMD Extensions (SSE) Prefetch NTA instructions executed.	This event counts the number of times the SSE instruction prefetchNTA is executed. This instruction prefetches the data to the L1 data cache.
07H	01H	SSE_PRE_EXEC.L1	Streaming SIMD Extensions (SSE) PrefetchT0 instructions executed.	This event counts the number of times the SSE instruction prefetchT0 is executed. This instruction prefetches the data to the L1 data cache and L2 cache.
07H	02H	SSE_PRE_EXEC.L2	Streaming SIMD Extensions (SSE) PrefetchT1 and PrefetchT2 instructions executed.	This event counts the number of times the SSE instructions prefetchT1 and prefetchT2 are executed. These instructions prefetch the data to the L2 cache.
07H	03H	SSE_PRE_ EXEC.STORES	Streaming SIMD Extensions (SSE) Weakly-ordered store instructions executed.	This event counts the number of times SSE non-temporal store instructions are executed.
08H	01H	DTLB_MISSES. ANY	Memory accesses that missed the DTLB.	This event counts the number of Data Table Lookaside Buffer (DTLB) misses. The count includes misses detected as a result of speculative accesses. Typically a high count for this event indicates that the code accesses a large number of data pages.
08H	02H	DTLB_MISSES .MISS_LD	DTLB misses due to load operations.	This event counts the number of Data Table Lookaside Buffer (DTLB) misses due to load operations. This count includes misses detected as a result of speculative accesses.
08H	04H	DTLB_MISSES.L0_MISS_LD	L0 DTLB misses due to load operations.	This event counts the number of level 0 Data Table Lookaside Buffer (DTLB0) misses due to load operations. This count includes misses detected as a result of speculative accesses. Loads that miss that DTLB0 and hit the DTLB1 can incur two-cycle penalty.

Table 19-25. Non-Architectural Performance Events in Processors Based on Intel® Core™ Microarchitecture (Contd.)

Event Num	Umask Value	Event Name	Definition	Description and Comment
08H	08H	DTLB_MISSES. MISS_ST	TLB misses due to store operations.	This event counts the number of Data Table Lookaside Buffer (DTLB) misses due to store operations. This count includes misses detected as a result of speculative accesses. Address translation for store operations is performed in the DTLB1.
09H	01H	MEMORY_ DISAMBIGUATION.RESET	Memory disambiguation reset cycles.	This event counts the number of cycles during which memory disambiguation misprediction occurs. As a result the execution pipeline is cleaned and execution of the mispredicted load instruction and all succeeding instructions restarts. This event occurs when the data address accessed by a load instruction, collides infrequently with preceding stores, but usually there is no collision. It happens rarely, and may have a penalty of about 20 cycles.
09H	02H	MEMORY_DISAMBIGUATIO N.SUCCESS	Number of loads successfully disambiguated.	This event counts the number of load operations that were successfully disambiguated. Loads are preceded by a store with an unknown address, but they are not blocked.
0CH	01H	PAGE_WALKS .COUNT	Number of page-walks executed.	This event counts the number of page-walks executed due to either a DTLB or ITLB miss. The page walk duration, PAGE_WALKS.CYCLES, divided by number of page walks is the average duration of a page walk. The average can hint whether most of the page-walks are satisfied by the caches or cause an L2 cache miss.
0CH	02H	PAGE_WALKS. CYCLES	Duration of page-walks in core cycles.	This event counts the duration of page-walks in core cycles. The paging mode in use typically affects the duration of page walks. Page walk duration divided by number of page walks is the average duration of page-walks. The average can hint at whether most of the page-walks are satisfied by the caches or cause an L2 cache miss.
10H	00H	FP_COMP_OPS _EXE	Floating point computational micro-ops executed.	This event counts the number of floating point computational micro-ops executed. Use IA32_PMC0 only.
11H	00H	FP_ASSIST	Floating point assists.	This event counts the number of floating point operations executed that required micro-code assist intervention. Assists are required in the following cases: ▪ Streaming SIMD Extensions (SSE) instructions: ▪ Denormal input when the DAZ (Denormals Are Zeros) flag is off ▪ Underflow result when the FTZ (Flush To Zero) flag is off ▪ X87 instructions: ▪ NaN or denormal are loaded to a register or used as input from memory ▪ Division by 0 ▪ Underflow output Use IA32_PMC1 only.
12H	00H	MUL	Multiply operations executed.	This event counts the number of multiply operations executed. This includes integer as well as floating point multiply operations. Use IA32_PMC1 only.

Table 19-25. Non-Architectural Performance Events in Processors Based on Intel® Core™ Microarchitecture (Contd.)

Event Num	Umask Value	Event Name	Definition	Description and Comment
13H	00H	DIV	Divide operations executed.	This event counts the number of divide operations executed. This includes integer divides, floating point divides and square-root operations executed. Use IA32_PMC1 only.
14H	00H	CYCLES_DIV _BUSY	Cycles the divider busy.	This event counts the number of cycles the divider is busy executing divide or square root operations. The divide can be integer, X87 or Streaming SIMD Extensions (SSE). The square root operation can be either X87 or SSE. Use IA32_PMC0 only.
18H	00H	IDLE_DURING _DIV	Cycles the divider is busy and all other execution units are idle.	This event counts the number of cycles the divider is busy (with a divide or a square root operation) and no other execution unit or load operation is in progress. Load operations are assumed to hit the L1 data cache. This event considers only micro-ops dispatched after the divider started operating. Use IA32_PMC0 only.
19H	00H	DELAYED_ BYPASS.FP	Delayed bypass to FP operation.	This event counts the number of times floating point operations use data immediately after the data was generated by a non-floating point execution unit. Such cases result in one penalty cycle due to data bypass between the units. Use IA32_PMC1 only.
19H	01H	DELAYED_ BYPASS.SIMD	Delayed bypass to SIMD operation.	This event counts the number of times SIMD operations use data immediately after the data was generated by a non-SIMD execution unit. Such cases result in one penalty cycle due to data bypass between the units. Use IA32_PMC1 only.
19H	02H	DELAYED_ BYPASS.LOAD	Delayed bypass to load operation.	This event counts the number of delayed bypass penalty cycles that a load operation incurred. When load operations use data immediately after the data was generated by an integer execution unit, they may (pending on certain dynamic internal conditions) incur one penalty cycle due to delayed data bypass between the units. Use IA32_PMC1 only.
21H	See Table 18-61	L2_ADS.(Core)	Cycles L2 address bus is in use.	This event counts the number of cycles the L2 address bus is being used for accesses to the L2 cache or bus queue. It can count occurrences for this core or both cores.
23H	See Table 18-61	L2_DBUS_BUSY _RD.(Core)	Cycles the L2 transfers data to the core.	This event counts the number of cycles during which the L2 data bus is busy transferring data from the L2 cache to the core. It counts for all L1 cache misses (data and instruction) that hit the L2 cache. This event can count occurrences for this core or both cores.

Table 19-25. Non-Architectural Performance Events in Processors Based on Intel® Core™ Microarchitecture (Contd.)

Event Num	Umask Value	Event Name	Definition	Description and Comment
24H	Combined mask from Table 18-61 and Table 18-63	L2_LINES_IN. (Core, Prefetch)	L2 cache misses.	This event counts the number of cache lines allocated in the L2 cache. Cache lines are allocated in the L2 cache as a result of requests from the L1 data and instruction caches and the L2 hardware prefetchers to cache lines that are missing in the L2 cache. This event can count occurrences for this core or both cores. It can also count demand requests and L2 hardware prefetch requests together or separately.
25H	See Table 18-61	L2_M_LINES_IN. (Core)	L2 cache line modifications.	This event counts whenever a modified cache line is written back from the L1 data cache to the L2 cache. This event can count occurrences for this core or both cores.
26H	See Table 18-61 and Table 18-63	L2_LINES_OUT. (Core, Prefetch)	L2 cache lines evicted.	This event counts the number of L2 cache lines evicted. This event can count occurrences for this core or both cores. It can also count evictions due to demand requests and L2 hardware prefetch requests together or separately.
27H	See Table 18-61 and Table 18-63	L2_M_LINES_OUT.(Core, Prefetch)	Modified lines evicted from the L2 cache.	This event counts the number of L2 modified cache lines evicted. These lines are written back to memory unless they also exist in a modified-state in one of the L1 data caches. This event can count occurrences for this core or both cores. It can also count evictions due to demand requests and L2 hardware prefetch requests together or separately.
28H	Combined mask from Table 18-61 and Table 18-64	L2_IFETCH.(Core, Cache Line State)	L2 cacheable instruction fetch requests.	This event counts the number of instruction cache line requests from the IFU. It does not include fetch requests from uncacheable memory. It does not include ITLB miss accesses. This event can count occurrences for this core or both cores. It can also count accesses to cache lines at different MESI states.
29H	Combined mask from Table 18-61, Table 18-63, and Table 18-64	L2_LD.(Core, Prefetch, Cache Line State)	L2 cache reads.	This event counts L2 cache read requests coming from the L1 data cache and L2 prefetchers. The event can count occurrences: ▪ For this core or both cores. ▪ Due to demand requests and L2 hardware prefetch requests together or separately. ▪ Of accesses to cache lines at different MESI states.
2AH	See Table 18-61 and Table 18-64	L2_ST.(Core, Cache Line State)	L2 store requests.	This event counts all store operations that miss the L1 data cache and request the data from the L2 cache. The event can count occurrences for this core or both cores. It can also count accesses to cache lines at different MESI states.

Table 19-25. Non-Architectural Performance Events in Processors Based on Intel® Core™ Microarchitecture (Contd.)

Event Num	Umask Value	Event Name	Definition	Description and Comment
2BH	See Table 18-61 and Table 18-64	L2_LOCK.(Core, Cache Line State)	L2 locked accesses.	This event counts all locked accesses to cache lines that miss the L1 data cache. The event can count occurrences for this core or both cores. It can also count accesses to cache lines at different MESI states.
2EH	See Table 18-61, Table 18-63, and Table 18-64	L2_RQSTS.(Core, Prefetch, Cache Line State)	L2 cache requests.	This event counts all completed L2 cache requests. This includes L1 data cache reads, writes, and locked accesses, L1 data prefetch requests, instruction fetches, and all L2 hardware prefetch requests. This event can count occurrences: ▪ For this core or both cores. ▪ Due to demand requests and L2 hardware prefetch requests together, or separately. ▪ Of accesses to cache lines at different MESI states.
2EH	41H	L2_RQSTS.SELF. DEMAND.I_STATE	L2 cache demand requests from this core that missed the L2.	This event counts all completed L2 cache demand requests from this core that miss the L2 cache. This includes L1 data cache reads, writes, and locked accesses, L1 data prefetch requests, and instruction fetches. This is an architectural performance event.
2EH	4FH	L2_RQSTS.SELF. DEMAND.MESI	L2 cache demand requests from this core.	This event counts all completed L2 cache demand requests from this core. This includes L1 data cache reads, writes, and locked accesses, L1 data prefetch requests, and instruction fetches. This is an architectural performance event.
30H	See Table 18-61, Table 18-63, and Table 18-64	L2_REJECT_BUSQ.(Core, Prefetch, Cache Line State)	Rejected L2 cache requests.	This event indicates that a pending L2 cache request that requires a bus transaction is delayed from moving to the bus queue. Some of the reasons for this event are: ▪ The bus queue is full. ▪ The bus queue already holds an entry for a cache line in the same set. The number of events is greater or equal to the number of requests that were rejected. ▪ For this core or both cores. ▪ Due to demand requests and L2 hardware prefetch requests together, or separately. ▪ Of accesses to cache lines at different MESI states.
32H	See Table 18-61	L2_NO_REQ.(Core)	Cycles no L2 cache requests are pending.	This event counts the number of cycles that no L2 cache requests were pending from a core. When using the BOTH_CORE modifier, the event counts only if none of the cores have a pending request. The event counts also when one core is halted and the other is not halted. The event can count occurrences for this core or both cores.
3AH	00H	EIST_TRANS	Number of Enhanced Intel SpeedStep Technology (EIST) transitions.	This event counts the number of transitions that include a frequency change, either with or without voltage change. This includes Enhanced Intel SpeedStep Technology (EIST) and TM2 transitions. The event is incremented only while the counting core is in C0 state. Since transitions to higher-numbered CxE states and TM2 transitions include a frequency change or voltage transition, the event is incremented accordingly.

Table 19-25. Non-Architectural Performance Events in Processors Based on Intel® Core™ Microarchitecture (Contd.)

Event Num	Umask Value	Event Name	Definition	Description and Comment
3BH	C0H	THERMAL_TRIP	Number of thermal trips.	This event counts the number of thermal trips. A thermal trip occurs whenever the processor temperature exceeds the thermal trip threshold temperature. Following a thermal trip, the processor automatically reduces frequency and voltage. The processor checks the temperature every millisecond and returns to normal when the temperature falls below the thermal trip threshold temperature.
3CH	00H	CPU_CLK_ UNHALTED. CORE_P	Core cycles when core is not halted.	This event counts the number of core cycles while the core is not in a halt state. The core enters the halt state when it is running the HLT instruction. This event is a component in many key event ratios. The core frequency may change due to transitions associated with Enhanced Intel SpeedStep Technology or TM2. For this reason, this event may have a changing ratio in regard to time. When the core frequency is constant, this event can give approximate elapsed time while the core not in halt state. This is an architectural performance event.
3CH	01H	CPU_CLK_ UNHALTED.BUS	Bus cycles when core is not halted.	This event counts the number of bus cycles while the core is not in the halt state. This event can give a measurement of the elapsed time while the core was not in the halt state. The core enters the halt state when it is running the HLT instruction. The event also has a constant ratio with CPU_CLK_UNHALTED.REF event, which is the maximum bus to processor frequency ratio. Non-halted bus cycles are a component in many key event ratios.
3CH	02H	CPU_CLK_ UNHALTED.NO _OTHER	Bus cycles when core is active and the other is halted.	This event counts the number of bus cycles during which the core remains non-halted and the other core on the processor is halted. This event can be used to determine the amount of parallelism exploited by an application or a system. Divide this event count by the bus frequency to determine the amount of time that only one core was in use.
40H	See Table 18-64	L1D_CACHE_LD. (Cache Line State)	L1 cacheable data reads.	This event counts the number of data reads from cacheable memory. Locked reads are not counted.
41H	See Table 18-64	L1D_CACHE_ST. (Cache Line State)	L1 cacheable data writes.	This event counts the number of data writes to cacheable memory. Locked writes are not counted.
42H	See Table 18-64	L1D_CACHE_ LOCK.(Cache Line State)	L1 data cacheable locked reads.	This event counts the number of locked data reads from cacheable memory.

Table 19-25. Non-Architectural Performance Events in Processors Based on Intel® Core™ Microarchitecture (Contd.)

Event Num	Umask Value	Event Name	Definition	Description and Comment
42H	10H	L1D_CACHE_ LOCK_DURATION	Duration of L1 data cacheable locked operation.	This event counts the number of cycles during which any cache line is locked by any locking instruction. Locking happens at retirement and therefore the event does not occur for instructions that are speculatively executed. Locking duration is shorter than locked instruction execution duration.
43H	01H	L1D_ALL_REF	All references to the L1 data cache.	This event counts all references to the L1 data cache, including all loads and stores with any memory types. The event counts memory accesses only when they are actually performed. For example, a load blocked by unknown store address and later performed is only counted once. The event includes non-cacheable accesses, such as I/O accesses.
43H	02H	L1D_ALL_ CACHE_REF	L1 Data cacheable reads and writes.	This event counts the number of data reads and writes from cacheable memory, including locked operations. This event is a sum of: ▪ L1D_CACHE_LD.MESI ▪ L1D_CACHE_ST.MESI ▪ L1D_CACHE_LOCK.MESI
45H	0FH	L1D_REPL	Cache lines allocated in the L1 data cache.	This event counts the number of lines brought into the L1 data cache.
46H	00H	L1D_M_REPL	Modified cache lines allocated in the L1 data cache.	This event counts the number of modified lines brought into the L1 data cache.
47H	00H	L1D_M_EVICT	Modified cache lines evicted from the L1 data cache.	This event counts the number of modified lines evicted from the L1 data cache, whether due to replacement or by snoop HITM intervention.
48H	00H	L1D_PEND_ MISS	Total number of outstanding L1 data cache misses at any cycle.	This event counts the number of outstanding L1 data cache misses at any cycle. An L1 data cache miss is outstanding from the cycle on which the miss is determined until the first chunk of data is available. This event counts: ▪ All cacheable demand requests. ▪ L1 data cache hardware prefetch requests. ▪ Requests to write through memory. ▪ Requests to write combine memory. Uncacheable requests are not counted. The count of this event divided by the number of L1 data cache misses, L1D_REPL, is the average duration in core cycles of an L1 data cache miss.
49H	01H	L1D_SPLIT.LOADS	Cache line split loads from the L1 data cache.	This event counts the number of load operations that span two cache lines. Such load operations are also called split loads. Split load operations are executed at retirement.
49H	02H	L1D_SPLIT. STORES	Cache line split stores to the L1 data cache.	This event counts the number of store operations that span two cache lines.
4BH	00H	SSE_PRE_ MISS.NTA	Streaming SIMD Extensions (SSE) Prefetch NTA instructions missing all cache levels.	This event counts the number of times the SSE instructions prefetchNTA were executed and missed all cache levels. Due to speculation an executed instruction might not retire. This instruction prefetches the data to the L1 data cache.

Table 19-25. Non-Architectural Performance Events in Processors Based on Intel® Core™ Microarchitecture (Contd.)

Event Num	Umask Value	Event Name	Definition	Description and Comment
4BH	01H	SSE_PRE_ MISS.L1	Streaming SIMD Extensions (SSE) PrefetchT0 instructions missing all cache levels.	This event counts the number of times the SSE instructions prefetchT0 were executed and missed all cache levels. Due to speculation executed instruction might not retire. The prefetchT0 instruction prefetches data to the L2 cache and L1 data cache.
4BH	02H	SSE_PRE_ MISS.L2	Streaming SIMD Extensions (SSE) PrefetchT1 and PrefetchT2 instructions missing all cache levels.	This event counts the number of times the SSE instructions prefetchT1 and prefetchT2 were executed and missed all cache levels. Due to speculation, an executed instruction might not retire. The prefetchT1 and PrefetchNT2 instructions prefetch data to the L2 cache.
4CH	00H	LOAD_HIT_PRE	Load operations conflicting with a software prefetch to the same address.	This event counts load operations sent to the L1 data cache while a previous Streaming SIMD Extensions (SSE) prefetch instruction to the same cache line has started prefetching but has not yet finished.
4EH	10H	L1D_PREFETCH. REQUESTS	L1 data cache prefetch requests.	This event counts the number of times the L1 data cache requested to prefetch a data cache line. Requests can be rejected when the L2 cache is busy and resubmitted later or lost. All requests are counted, including those that are rejected.
60H	See Table 18-61 and Table 18-62.	BUS_REQUEST_ OUTSTANDING. (Core and Bus Agents)	Outstanding cacheable data read bus requests duration.	This event counts the number of pending full cache line read transactions on the bus occurring in each cycle. A read transaction is pending from the cycle it is sent on the bus until the full cache line is received by the processor. The event counts only full-line cacheable read requests from either the L1 data cache or the L2 prefetchers. It does not count Read for Ownership transactions, instruction byte fetch transactions, or any other bus transaction.
61H	See Table 18-62.	BUS_BNR_DRV. (Bus Agents)	Number of Bus Not Ready signals asserted.	This event counts the number of Bus Not Ready (BNR) signals that the processor asserts on the bus to suspend additional bus requests by other bus agents. A bus agent asserts the BNR signal when the number of data and snoop transactions is close to the maximum that the bus can handle. To obtain the number of bus cycles during which the BNR signal is asserted, multiply the event count by two. While this signal is asserted, new transactions cannot be submitted on the bus. As a result, transaction latency may have higher impact on program performance.
62H	See Table 18-62.	BUS_DRDY_ CLOCKS.(Bus Agents)	Bus cycles when data is sent on the bus.	This event counts the number of bus cycles during which the DRDY (Data Ready) signal is asserted on the bus. The DRDY signal is asserted when data is sent on the bus. With the 'THIS_AGENT' mask this event counts the number of bus cycles during which this agent (the processor) writes data on the bus back to memory or to other bus agents. This includes all explicit and implicit data writebacks, as well as partial writes. With the 'ALL_AGENTS' mask, this event counts the number of bus cycles during which any bus agent sends data on the bus. This includes all data reads and writes on the bus.

Table 19-25. Non-Architectural Performance Events in Processors Based on Intel® Core™ Microarchitecture (Contd.)

Event Num	Umask Value	Event Name	Definition	Description and Comment
63H	See Table 18-61 and Table 18-62.	BUS_LOCK_ CLOCKS.(Core and Bus Agents)	Bus cycles when a LOCK signal asserted.	This event counts the number of bus cycles, during which the LOCK signal is asserted on the bus. A LOCK signal is asserted when there is a locked memory access, due to: ▪ Uncacheable memory. ▪ Locked operation that spans two cache lines. ▪ Page-walk from an uncacheable page table. Bus locks have a very high performance penalty and it is highly recommended to avoid such accesses.
64H	See Table 18-61.	BUS_DATA_ RCV.(Core)	Bus cycles while processor receives data.	This event counts the number of bus cycles during which the processor is busy receiving data.
65H	See Table 18-61 and Table 18-62.	BUS_TRANS_BRD.(Core and Bus Agents)	Burst read bus transactions.	This event counts the number of burst read transactions including: ▪ L1 data cache read misses (and L1 data cache hardware prefetches). ▪ L2 hardware prefetches by the DPL and L2 streamer. ▪ IFU read misses of cacheable lines. It does not include RFO transactions.
66H	See Table 18-61 and Table 18-62.	BUS_TRANS_RFO.(Core and Bus Agents)	RFO bus transactions.	This event counts the number of Read For Ownership (RFO) bus transactions, due to store operations that miss the L1 data cache and the L2 cache. It also counts RFO bus transactions due to locked operations.
67H	See Table 18-61 and Table 18-62.	BUS_TRANS_WB. (Core and Bus Agents)	Explicit writeback bus transactions.	This event counts all explicit writeback bus transactions due to dirty line evictions. It does not count implicit writebacks due to invalidation by a snoop request.
68H	See Table 18-61 and Table 18-62.	BUS_TRANS_ IFETCH.(Core and Bus Agents)	Instruction-fetch bus transactions.	This event counts all instruction fetch full cache line bus transactions.
69H	See Table 18-61 and Table 18-62.	BUS_TRANS_ INVAL.(Core and Bus Agents)	Invalidate bus transactions.	This event counts all invalidate transactions. Invalidate transactions are generated when: ▪ A store operation hits a shared line in the L2 cache. ▪ A full cache line write misses the L2 cache or hits a shared line in the L2 cache.
6AH	See Table 18-61 and Table 18-62.	BUS_TRANS_ PWR.(Core and Bus Agents)	Partial write bus transaction.	This event counts partial write bus transactions.

Table 19-25. Non-Architectural Performance Events in Processors Based on Intel® Core™ Microarchitecture (Contd.)

Event Num	Umask Value	Event Name	Definition	Description and Comment
6BH	See Table 18-61 and Table 18-62.	BUS_TRANS _P.(Core and Bus Agents)	Partial bus transactions.	This event counts all (read and write) partial bus transactions.
6CH	See Table 18-61 and Table 18-62.	BUS_TRANS_IO.(Core and Bus Agents)	IO bus transactions.	This event counts the number of completed I/O bus transactions as a result of IN and OUT instructions. The count does not include memory mapped IO.
6DH	See Table 18-61 and Table 18-62.	BUS_TRANS_ DEF.(Core and Bus Agents)	Deferred bus transactions.	This event counts the number of deferred transactions.
6EH	See Table 18-61 and Table 18-62.	BUS_TRANS_ BURST.(Core and Bus Agents)	Burst (full cache-line) bus transactions.	This event counts burst (full cache line) transactions including: ▪ Burst reads. ▪ RFOs. ▪ Explicit writebacks. ▪ Write combine lines.
6FH	See Table 18-61 and Table 18-62.	BUS_TRANS_ MEM.(Core and Bus Agents)	Memory bus transactions.	This event counts all memory bus transactions including: ▪ Burst transactions. ▪ Partial reads and writes - invalidate transactions. The BUS_TRANS_MEM count is the sum of BUS_TRANS_BURST, BUS_TRANS_P and BUS_TRANS_IVAL.
70H	See Table 18-61 and Table 18-62.	BUS_TRANS_ ANY.(Core and Bus Agents)	All bus transactions.	This event counts all bus transactions. This includes: ▪ Memory transactions. ▪ IO transactions (non memory-mapped). ▪ Deferred transaction completion. ▪ Other less frequent transactions, such as interrupts.
77H	See Table 18-61 and Table 18-65.	EXT_SNOOP. (Bus Agents, Snoop Response)	External snoops.	This event counts the snoop responses to bus transactions. Responses can be counted separately by type and by bus agent. With the 'THIS_AGENT' mask, the event counts snoop responses from this processor to bus transactions sent by this processor. With the 'ALL_AGENTS' mask the event counts all snoop responses seen on the bus.
78H	See Table 18-61 and Table 18-66.	CMP_SNOOP.(Core, Snoop Type)	L1 data cache snooped by other core.	This event counts the number of times the L1 data cache is snooped for a cache line that is needed by the other core in the same processor. The cache line is either missing in the L1 instruction or data caches of the other core, or is available for reading only and the other core wishes to write the cache line.

Table 19-25. Non-Architectural Performance Events in Processors Based on Intel® Core™ Microarchitecture (Contd.)

Event Num	Umask Value	Event Name	Definition	Description and Comment
				The snoop operation may change the cache line state. If the other core issued a read request that hit this core in E state, typically the state changes to S state in this core. If the other core issued a read for ownership request (due a write miss or hit to S state) that hits this core's cache line in E or S state, this typically results in invalidation of the cache line in this core. If the snoop hits a line in M state, the state is changed at a later opportunity. These snoops are performed through the L1 data cache store port. Therefore, frequent snoops may conflict with extensive stores to the L1 data cache, which may increase store latency and impact performance.
7AH	See Table 18-62.	BUS_HIT_DRV. (Bus Agents)	HIT signal asserted.	This event counts the number of bus cycles during which the processor drives the HIT# pin to signal HIT snoop response.
7BH	See Table 18-62.	BUS_HITM_DRV. (Bus Agents)	HITM signal asserted.	This event counts the number of bus cycles during which the processor drives the HITM# pin to signal HITM snoop response.
7DH	See Table 18-61.	BUSQ_EMPTY. (Core)	Bus queue empty.	This event counts the number of cycles during which the core did not have any pending transactions in the bus queue. It also counts when the core is halted and the other core is not halted. This event can count occurrences for this core or both cores.
7EH	See Table 18-61 and Table 18-62.	SNOOP_STALL_ DRV.(Core and Bus Agents)	Bus stalled for snoops.	This event counts the number of times that the bus snoop stall signal is asserted. To obtain the number of bus cycles during which snoops on the bus are prohibited, multiply the event count by two. During the snoop stall cycles, no new bus transactions requiring a snoop response can be initiated on the bus. A bus agent asserts a snoop stall signal if it cannot response to a snoop request within three bus cycles.
7FH	See Table 18-61.	BUS_IO_WAIT. (Core)	IO requests waiting in the bus queue.	This event counts the number of core cycles during which IO requests wait in the bus queue. With the SELF modifier this event counts IO requests per core. With the BOTH_CORE modifier, this event increments by one for any cycle for which there is a request from either core.
80H	00H	L1I_READS	Instruction fetches.	This event counts all instruction fetches, including uncacheable fetches that bypass the Instruction Fetch Unit (IFU).
81H	00H	L1I_MISSES	Instruction Fetch Unit misses.	This event counts all instruction fetches that miss the Instruction Fetch Unit (IFU) or produce memory requests. This includes uncacheable fetches. An instruction fetch miss is counted only once and not once for every cycle it is outstanding.
82H	02H	ITLB.SMALL_MISS	ITLB small page misses.	This event counts the number of instruction fetches from small pages that miss the ITLB.
82H	10H	ITLB.LARGE_MISS	ITLB large page misses.	This event counts the number of instruction fetches from large pages that miss the ITLB.

Table 19-25. Non-Architectural Performance Events in Processors Based on Intel® Core™ Microarchitecture (Contd.)

Event Num	Umask Value	Event Name	Definition	Description and Comment
82H	40H	ITLB.FLUSH	ITLB flushes.	This event counts the number of ITLB flushes. This usually happens upon CR3 or CR0 writes, which are executed by the operating system during process switches.
82H	12H	ITLB.MISSES	ITLB misses.	This event counts the number of instruction fetches from either small or large pages that miss the ITLB.
83H	02H	INST_QUEUE.FULL	Cycles during which the instruction queue is full.	This event counts the number of cycles during which the instruction queue is full. In this situation, the core front end stops fetching more instructions. This is an indication of very long stalls in the back-end pipeline stages.
86H	00H	CYCLES_L1I_ MEM_STALLED	Cycles during which instruction fetches stalled.	This event counts the number of cycles for which an instruction fetch stalls, including stalls due to any of the following reasons: ▪ Instruction Fetch Unit cache misses. ▪ Instruction TLB misses. ▪ Instruction TLB faults.
87H	00H	ILD_STALL	Instruction Length Decoder stall cycles due to a length changing prefix.	This event counts the number of cycles during which the instruction length decoder uses the slow length decoder. Usually, instruction length decoding is done in one cycle. When the slow decoder is used, instruction decoding requires 6 cycles. The slow decoder is used in the following cases: ▪ Operand override prefix (66H) preceding an instruction with immediate data. ▪ Address override prefix (67H) preceding an instruction with a modr/m in real, big real, 16-bit protected or 32-bit protected modes. To avoid instruction length decoding stalls, generate code using imm8 or imm32 values instead of imm16 values. If you must use an imm16 value, store the value in a register using "mov reg, imm32" and use the register format of the instruction.
88H	00H	BR_INST_EXEC	Branch instructions executed.	This event counts all executed branches (not necessarily retired). This includes only instructions and not micro-op branches. Frequent branching is not necessarily a major performance issue. However frequent branch mispredictions may be a problem.
89H	00H	BR_MISSP_EXEC	Mispredicted branch instructions executed.	This event counts the number of mispredicted branch instructions that were executed.
8AH	00H	BR_BAC_ MISSP_EXEC	Branch instructions mispredicted at decoding.	This event counts the number of branch instructions that were mispredicted at decoding.
8BH	00H	BR_CND_EXEC	Conditional branch instructions executed.	This event counts the number of conditional branch instructions executed, but not necessarily retired.
8CH	00H	BR_CND_ MISSP_EXEC	Mispredicted conditional branch instructions executed.	This event counts the number of mispredicted conditional branch instructions that were executed.
8DH	00H	BR_IND_EXEC	Indirect branch instructions executed.	This event counts the number of indirect branch instructions that were executed.

Table 19-25. Non-Architectural Performance Events in Processors Based on Intel® Core™ Microarchitecture (Contd.)

Event Num	Umask Value	Event Name	Definition	Description and Comment
8EH	00H	BR_IND_MISSP _EXEC	Mispredicted indirect branch instructions executed.	This event counts the number of mispredicted indirect branch instructions that were executed.
8FH	00H	BR_RET_EXEC	RET instructions executed.	This event counts the number of RET instructions that were executed.
90H	00H	BR_RET_ MISSP_EXEC	Mispredicted RET instructions executed.	This event counts the number of mispredicted RET instructions that were executed.
91H	00H	BR_RET_BAC_ MISSP_EXEC	RET instructions executed mispredicted at decoding.	This event counts the number of RET instructions that were executed and were mispredicted at decoding.
92H	00H	BR_CALL_EXEC	CALL instructions executed.	This event counts the number of CALL instructions executed.
93H	00H	BR_CALL_ MISSP_EXEC	Mispredicted CALL instructions executed.	This event counts the number of mispredicted CALL instructions that were executed.
94H	00H	BR_IND_CALL_ EXEC	Indirect CALL instructions executed.	This event counts the number of indirect CALL instructions that were executed.
97H	00H	BR_TKN_ BUBBLE_1	Branch predicted taken with bubble 1.	The events BR_TKN_BUBBLE_1 and BR_TKN_BUBBLE_2 together count the number of times a taken branch prediction incurred a one-cycle penalty. The penalty incurs when: ▪ Too many taken branches are placed together. To avoid this, unroll loops and add a non-taken branch in the middle of the taken sequence. ▪ The branch target is unaligned. To avoid this, align the branch target.
98H	00H	BR_TKN_ BUBBLE_2	Branch predicted taken with bubble 2.	The events BR_TKN_BUBBLE_1 and BR_TKN_BUBBLE_2 together count the number of times a taken branch prediction incurred a one-cycle penalty. The penalty incurs when: ▪ Too many taken branches are placed together. To avoid this, unroll loops and add a non-taken branch in the middle of the taken sequence. ▪ The branch target is unaligned. To avoid this, align the branch target.
A0H	00H	RS_UOPS_ DISPATCHED	Micro-ops dispatched for execution.	This event counts the number of micro-ops dispatched for execution. Up to six micro-ops can be dispatched in each cycle.
A1H	01H	RS_UOPS_ DISPATCHED.PORT0	Cycles micro-ops dispatched for execution on port 0.	This event counts the number of cycles for which micro-ops dispatched for execution. Each cycle, at most one micro-op can be dispatched on the port. Issue Ports are described in *Intel® 64 and IA-32 Architectures Optimization Reference Manual*. Use IA32_PMC0 only.
A1H	02H	RS_UOPS_ DISPATCHED.PORT1	Cycles micro-ops dispatched for execution on port 1.	This event counts the number of cycles for which micro-ops dispatched for execution. Each cycle, at most one micro-op can be dispatched on the port. Use IA32_PMC0 only.
A1H	04H	RS_UOPS_ DISPATCHED.PORT2	Cycles micro-ops dispatched for execution on port 2.	This event counts the number of cycles for which micro-ops dispatched for execution. Each cycle, at most one micro-op can be dispatched on the port. Use IA32_PMC0 only.

Table 19-25. Non-Architectural Performance Events in Processors Based on Intel® Core™ Microarchitecture (Contd.)

Event Num	Umask Value	Event Name	Definition	Description and Comment
A1H	08H	RS_UOPS_ DISPATCHED.PORT3	Cycles micro-ops dispatched for execution on port 3.	This event counts the number of cycles for which micro-ops dispatched for execution. Each cycle, at most one micro-op can be dispatched on the port. Use IA32_PMC0 only.
A1H	10H	RS_UOPS_ DISPATCHED.PORT4	Cycles micro-ops dispatched for execution on port 4.	This event counts the number of cycles for which micro-ops dispatched for execution. Each cycle, at most one micro-op can be dispatched on the port. Use IA32_PMC0 only.
A1H	20H	RS_UOPS_ DISPATCHED.PORT5	Cycles micro-ops dispatched for execution on port 5.	This event counts the number of cycles for which micro-ops dispatched for execution. Each cycle, at most one micro-op can be dispatched on the port. Use IA32_PMC0 only.
AAH	01H	MACRO_INSTS. DECODED	Instructions decoded.	This event counts the number of instructions decoded (but not necessarily executed or retired).
AAH	08H	MACRO_INSTS. CISC_DECODED	CISC Instructions decoded.	This event counts the number of complex instructions decoded. Complex instructions usually have more than four micro-ops. Only one complex instruction can be decoded at a time.
ABH	01H	ESP.SYNCH	ESP register content synchron-ization.	This event counts the number of times that the ESP register is explicitly used in the address expression of a load or store operation, after it is implicitly used, for example by a push or a pop instruction. ESP synch micro-op uses resources from the rename pipe-stage and up to retirement. The expected ratio of this event divided by the number of ESP implicit changes is 0,2. If the ratio is higher, consider rearranging your code to avoid ESP synchronization events.
ABH	02H	ESP.ADDITIONS	ESP register automatic additions.	This event counts the number of ESP additions performed automatically by the decoder. A high count of this event is good, since each automatic addition performed by the decoder saves a micro-op from the execution units. To maximize the number of ESP additions performed automatically by the decoder, choose instructions that implicitly use the ESP, such as PUSH, POP, CALL, and RET instructions whenever possible.
B0H	00H	SIMD_UOPS_EXEC	SIMD micro-ops executed (excluding stores).	This event counts all the SIMD micro-ops executed. It does not count MOVQ and MOVD stores from register to memory.
B1H	00H	SIMD_SAT_UOP_ EXEC	SIMD saturated arithmetic micro-ops executed.	This event counts the number of SIMD saturated arithmetic micro-ops executed.
B3H	01H	SIMD_UOP_ TYPE_EXEC.MUL	SIMD packed multiply micro-ops executed.	This event counts the number of SIMD packed multiply micro-ops executed.
B3H	02H	SIMD_UOP_TYPE_EXEC.SHI FT	SIMD packed shift micro-ops executed.	This event counts the number of SIMD packed shift micro-ops executed.
B3H	04H	SIMD_UOP_TYPE_EXEC.PA CK	SIMD pack micro-ops executed.	This event counts the number of SIMD pack micro-ops executed.
B3H	08H	SIMD_UOP_TYPE_EXEC.UN PACK	SIMD unpack micro-ops executed.	This event counts the number of SIMD unpack micro-ops executed.
B3H	10H	SIMD_UOP_TYPE_EXEC.LO GICAL	SIMD packed logical micro-ops executed.	This event counts the number of SIMD packed logical micro-ops executed.

Table 19-25. Non-Architectural Performance Events in Processors Based on Intel® Core™ Microarchitecture (Contd.)

Event Num	Umask Value	Event Name	Definition	Description and Comment
B3H	20H	SIMD_UOP_TYPE_EXEC.ARITHMETIC	SIMD packed arithmetic micro-ops executed.	This event counts the number of SIMD packed arithmetic micro-ops executed.
C0H	00H	INST_RETIRED.ANY_P	Instructions retired.	This event counts the number of instructions that retire execution. For instructions that consist of multiple micro-ops, this event counts the retirement of the last micro-op of the instruction. The counter continues counting during hardware interrupts, traps, and inside interrupt handlers. INST_RETIRED.ANY_P is an architectural performance event.
C0H	01H	INST_RETIRED.LOADS	Instructions retired, which contain a load.	This event counts the number of instructions retired that contain a load operation.
C0H	02H	INST_RETIRED.STORES	Instructions retired, which contain a store.	This event counts the number of instructions retired that contain a store operation.
C0H	04H	INST_RETIRED.OTHER	Instructions retired, with no load or store operation.	This event counts the number of instructions retired that do not contain a load or a store operation.
C1H	01H	X87_OPS_RETIRED.FXCH	FXCH instructions retired.	This event counts the number of FXCH instructions retired. Modern compilers generate more efficient code and are less likely to use this instruction. If you obtain a high count for this event consider recompiling the code.
C1H	FEH	X87_OPS_RETIRED.ANY	Retired floating-point computational operations (precise event).	This event counts the number of floating-point computational operations retired. It counts: ▪ Floating point computational operations executed by the assist handler. ▪ Sub-operations of complex floating-point instructions like transcendental instructions. This event does not count: ▪ Floating-point computational operations that cause traps or assists. ▪ Floating-point loads and stores. When this event is captured with the precise event mechanism, the collected samples contain the address of the instruction that was executed immediately after the instruction that caused the event.
C2H	01H	UOPS_RETIRED.LD_IND_BR	Fused load+op or load+indirect branch retired.	This event counts the number of retired micro-ops that fused a load with another operation. This includes: ▪ Fusion of a load and an arithmetic operation, such as with the following instruction: ADD EAX, [EBX] where the content of the memory location specified by EBX register is loaded, added to EXA register, and the result is stored in EAX. ▪ Fusion of a load and a branch in an indirect branch operation, such as with the following instructions: ▪ JMP [RDI+200] ▪ RET ▪ Fusion decreases the number of micro-ops in the processor pipeline. A high value for this event count indicates that the code is using the processor resources effectively.

Table 19-25. Non-Architectural Performance Events in Processors Based on Intel® Core™ Microarchitecture (Contd.)

Event Num	Umask Value	Event Name	Definition	Description and Comment
C2H	02H	UOPS_RETIRED. STD_STA	Fused store address + data retired.	This event counts the number of store address calculations that are fused with store data emission into one micro-op. Traditionally, each store operation required two micro-ops. This event counts fusion of retired micro-ops only. Fusion decreases the number of micro-ops in the processor pipeline. A high value for this event count indicates that the code is using the processor resources effectively.
C2H	04H	UOPS_RETIRED. MACRO_FUSION	Retired instruction pairs fused into one micro-op.	This event counts the number of times CMP or TEST instructions were fused with a conditional branch instruction into one micro-op. It counts fusion by retired micro-ops only. Fusion decreases the number of micro-ops in the processor pipeline. A high value for this event count indicates that the code uses the processor resources more effectively.
C2H	07H	UOPS_RETIRED. FUSED	Fused micro-ops retired.	This event counts the total number of retired fused micro-ops. The counts include the following fusion types: ▪ Fusion of load operation with an arithmetic operation or with an indirect branch (counted by event UOPS_RETIRED.LD_IND_BR) ▪ Fusion of store address and data (counted by event UOPS_RETIRED.STD_STA) ▪ Fusion of CMP or TEST instruction with a conditional branch instruction (counted by event UOPS_RETIRED.MACRO_FUSION) Fusion decreases the number of micro-ops in the processor pipeline. A high value for this event count indicates that the code is using the processor resources effectively.
C2H	08H	UOPS_RETIRED. NON_FUSED	Non-fused micro-ops retired.	This event counts the number of micro-ops retired that were not fused.
C2H	0FH	UOPS_RETIRED. ANY	Micro-ops retired.	This event counts the number of micro-ops retired. The processor decodes complex macro instructions into a sequence of simpler micro-ops. Most instructions are composed of one or two micro-ops. Some instructions are decoded into longer sequences such as repeat instructions, floating point transcendental instructions, and assists. In some cases micro-op sequences are fused or whole instructions are fused into one micro-op. See other UOPS_RETIRED events for differentiating retired fused and non-fused micro-ops.
C3H	01H	MACHINE_ NUKES.SMC	Self-Modifying Code detected.	This event counts the number of times that a program writes to a code section. Self-modifying code causes a severe penalty in all Intel 64 and IA-32 processors.

Table 19-25. Non-Architectural Performance Events in Processors Based on Intel® Core™ Microarchitecture (Contd.)

Event Num	Umask Value	Event Name	Definition	Description and Comment
C3H	04H	MACHINE_NUKES.MEM_ORDER	Execution pipeline restart due to memory ordering conflict or memory disambiguation misprediction.	This event counts the number of times the pipeline is restarted due to either multi-threaded memory ordering conflicts or memory disambiguation misprediction. A multi-threaded memory ordering conflict occurs when a store, which is executed in another core, hits a load that is executed out of order in this core but not yet retired. As a result, the load needs to be restarted to satisfy the memory ordering model. See Chapter 8, "Multiple-Processor Management" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*. To count memory disambiguation mispredictions, use the event MEMORY_DISAMBIGUATION.RESET.
C4H	00H	BR_INST_RETIRED.ANY	Retired branch instructions.	This event counts the number of branch instructions retired. This is an architectural performance event.
C4H	01H	BR_INST_RETIRED.PRED_NOT_TAKEN	Retired branch instructions that were predicted not-taken.	This event counts the number of branch instructions retired that were correctly predicted to be not-taken.
C4H	02H	BR_INST_RETIRED.MISPRED_NOT_TAKEN	Retired branch instructions that were mispredicted not-taken.	This event counts the number of branch instructions retired that were mispredicted and not-taken.
C4H	04H	BR_INST_RETIRED.PRED_TAKEN	Retired branch instructions that were predicted taken.	This event counts the number of branch instructions retired that were correctly predicted to be taken.
C4H	08H	BR_INST_RETIRED.MISPRED_TAKEN	Retired branch instructions that were mispredicted taken.	This event counts the number of branch instructions retired that were mispredicted and taken.
C4H	0CH	BR_INST_RETIRED.TAKEN	Retired taken branch instructions.	This event counts the number of branches retired that were taken.
C5H	00H	BR_INST_RETIRED.MISPRED	Retired mispredicted branch instructions. (precise event)	This event counts the number of retired branch instructions that were mispredicted by the processor. A branch misprediction occurs when the processor predicts that the branch would be taken, but it is not, or vice-versa. This is an architectural performance event.
C6H	01H	CYCLES_INT_MASKED	Cycles during which interrupts are disabled.	This event counts the number of cycles during which interrupts are disabled.
C6H	02H	CYCLES_INT_PENDING_AND_MASKED	Cycles during which interrupts are pending and disabled.	This event counts the number of cycles during which there are pending interrupts but interrupts are disabled.
C7H	01H	SIMD_INST_RETIRED.PACKED_SINGLE	Retired SSE packed-single instructions.	This event counts the number of SSE packed-single instructions retired.
C7H	02H	SIMD_INST_RETIRED.SCALAR_SINGLE	Retired SSE scalar-single instructions.	This event counts the number of SSE scalar-single instructions retired.
C7H	04H	SIMD_INST_RETIRED.PACKED_DOUBLE	Retired SSE2 packed-double instructions.	This event counts the number of SSE2 packed-double instructions retired.
C7H	08H	SIMD_INST_RETIRED.SCALAR_DOUBLE	Retired SSE2 scalar-double instructions.	This event counts the number of SSE2 scalar-double instructions retired.

Table 19-25. Non-Architectural Performance Events in Processors Based on Intel® Core™ Microarchitecture (Contd.)

Event Num	Umask Value	Event Name	Definition	Description and Comment
C7H	10H	SIMD_INST_ RETIRED.VECTOR	Retired SSE2 vector integer instructions.	This event counts the number of SSE2 vector integer instructions retired.
C7H	1FH	SIMD_INST_ RETIRED.ANY	Retired Streaming SIMD instructions (precise event).	This event counts the overall number of retired SIMD instructions that use XMM registers. To count each type of SIMD instruction separately, use the following events: ▪ SIMD_INST_RETIRED.PACKED_SINGLE ▪ SIMD_INST_RETIRED.SCALAR_SINGLE ▪ SIMD_INST_RETIRED.PACKED_DOUBLE ▪ SIMD_INST_RETIRED.SCALAR_DOUBLE ▪ and SIMD_INST_RETIRED.VECTOR When this event is captured with the precise event mechanism, the collected samples contain the address of the instruction that was executed immediately after the instruction that caused the event.
C8H	00H	HW_INT_RCV	Hardware interrupts received.	This event counts the number of hardware interrupts received by the processor.
C9H	00H	ITLB_MISS_ RETIRED	Retired instructions that missed the ITLB.	This event counts the number of retired instructions that missed the ITLB when they were fetched.
CAH	01H	SIMD_COMP_ INST_RETIRED. PACKED_SINGLE	Retired computational SSE packed-single instructions.	This event counts the number of computational SSE packed-single instructions retired. Computational instructions perform arithmetic computations (for example: add, multiply and divide). Instructions that perform load and store operations or logical operations, like XOR, OR, and AND are not counted by this event.
CAH	02H	SIMD_COMP_ INST_RETIRED. SCALAR_SINGLE	Retired computational SSE scalar-single instructions.	This event counts the number of computational SSE scalar-single instructions retired. Computational instructions perform arithmetic computations (for example: add, multiply and divide). Instructions that perform load and store operations or logical operations, like XOR, OR, and AND are not counted by this event.
CAH	04H	SIMD_COMP_ INST_RETIRED. PACKED_DOUBLE	Retired computational SSE2 packed-double instructions.	This event counts the number of computational SSE2 packed-double instructions retired. Computational instructions perform arithmetic computations (for example: add, multiply and divide). Instructions that perform load and store operations or logical operations, like XOR, OR, and AND are not counted by this event.
CAH	08H	SIMD_COMP_INST_RETIRE D.SCALAR_DOUBLE	Retired computational SSE2 scalar-double instructions.	This event counts the number of computational SSE2 scalar-double instructions retired. Computational instructions perform arithmetic computations (for example: add, multiply and divide). Instructions that perform load and store operations or logical operations, like XOR, OR, and AND are not counted by this event.

Table 19-25. Non-Architectural Performance Events in Processors Based on Intel® Core™ Microarchitecture (Contd.)

Event Num	Umask Value	Event Name	Definition	Description and Comment
CBH	01H	MEM_LOAD_ RETIRED.L1D _MISS	Retired loads that miss the L1 data cache (precise event).	This event counts the number of retired load operations that missed the L1 data cache. This includes loads from cache lines that are currently being fetched, due to a previous L1 data cache miss to the same cache line. This event counts loads from cacheable memory only. The event does not count loads by software prefetches. When this event is captured with the precise event mechanism, the collected samples contain the address of the instruction that was executed immediately after the instruction that caused the event. Use IA32_PMC0 only.
CBH	02H	MEM_LOAD_ RETIRED.L1D_ LINE_MISS	L1 data cache line missed by retired loads (precise event).	This event counts the number of load operations that miss the L1 data cache and send a request to the L2 cache to fetch the missing cache line. That is the missing cache line fetching has not yet started. The event count is equal to the number of cache lines fetched from the L2 cache by retired loads. This event counts loads from cacheable memory only. The event does not count loads by software prefetches. The event might not be counted if the load is blocked (see LOAD_BLOCK events). When this event is captured with the precise event mechanism, the collected samples contain the address of the instruction that was executed immediately after the instruction that caused the event. Use IA32_PMC0 only.
CBH	04H	MEM_LOAD_ RETIRED.L2_MISS	Retired loads that miss the L2 cache (precise event).	This event counts the number of retired load operations that missed the L2 cache. This event counts loads from cacheable memory only. It does not count loads by software prefetches. When this event is captured with the precise event mechanism, the collected samples contain the address of the instruction that was executed immediately after the instruction that caused the event. Use IA32_PMC0 only.

Table 19-25. Non-Architectural Performance Events in Processors Based on Intel® Core™ Microarchitecture (Contd.)

Event Num	Umask Value	Event Name	Definition	Description and Comment
CBH	08H	MEM_LOAD_ RETIRED.L2_LINE_MISS	L2 cache line missed by retired loads (precise event).	This event counts the number of load operations that miss the L2 cache and result in a bus request to fetch the missing cache line. That is the missing cache line fetching has not yet started. This event count is equal to the number of cache lines fetched from memory by retired loads. This event counts loads from cacheable memory only. The event does not count loads by software prefetches. The event might not be counted if the load is blocked (see LOAD_BLOCK events). When this event is captured with the precise event mechanism, the collected samples contain the address of the instruction that was executed immediately after the instruction that caused the event. Use IA32_PMC0 only.
CBH	10H	MEM_LOAD_ RETIRED.DTLB_ MISS	Retired loads that miss the DTLB (precise event).	This event counts the number of retired loads that missed the DTLB. The DTLB miss is not counted if the load operation causes a fault. This event counts loads from cacheable memory only. The event does not count loads by software prefetches. When this event is captured with the precise event mechanism, the collected samples contain the address of the instruction that was executed immediately after the instruction that caused the event. Use IA32_PMC0 only.
CCH	01H	FP_MMX_TRANS_TO_MMX	Transitions from Floating Point to MMX Instructions.	This event counts the first MMX instructions following a floating-point instruction. Use this event to estimate the penalties for the transitions between floating-point and MMX states.
CCH	02H	FP_MMX_TRANS_TO_FP	Transitions from MMX Instructions to Floating Point Instructions.	This event counts the first floating-point instructions following any MMX instruction. Use this event to estimate the penalties for the transitions between floating-point and MMX states.
CDH	00H	SIMD_ASSIST	SIMD assists invoked.	This event counts the number of SIMD assists invoked. SIMD assists are invoked when an EMMS instruction is executed, changing the MMX state in the floating point stack.
CEH	00H	SIMD_INSTR_ RETIRED	SIMD Instructions retired.	This event counts the number of retired SIMD instructions that use MMX registers.
CFH	00H	SIMD_SAT_INSTR_RETIRED	Saturated arithmetic instructions retired.	This event counts the number of saturated arithmetic SIMD instructions that retired.
D2H	01H	RAT_STALLS. ROB_READ_PORT	ROB read port stalls cycles.	This event counts the number of cycles when ROB read port stalls occurred, which did not allow new micro-ops to enter the out-of-order pipeline. Note that, at this stage in the pipeline, additional stalls may occur at the same cycle and prevent the stalled micro-ops from entering the pipe. In such a case, micro-ops retry entering the execution pipe in the next cycle and the ROB-read-port stall is counted again.

Table 19-25. Non-Architectural Performance Events in Processors Based on Intel® Core™ Microarchitecture (Contd.)

Event Num	Umask Value	Event Name	Definition	Description and Comment
D2H	02H	RAT_STALLS. PARTIAL_CYCLES	Partial register stall cycles.	This event counts the number of cycles instruction execution latency became longer than the defined latency because the instruction uses a register that was partially written by previous instructions.
D2H	04H	RAT_STALLS. FLAGS	Flag stall cycles.	This event counts the number of cycles during which execution stalled due to several reasons, one of which is a partial flag register stall. A partial register stall may occur when two conditions are met: ▪ An instruction modifies some, but not all, of the flags in the flag register. ▪ The next instruction, which depends on flags, depends on flags that were not modified by this instruction.
D2H	08H	RAT_STALLS. FPSW	FPU status word stall.	This event indicates that the FPU status word (FPSW) is written. To obtain the number of times the FPSW is written divide the event count by 2. The FPSW is written by instructions with long latency; a small count may indicate a high penalty.
D2H	0FH	RAT_STALLS. ANY	All RAT stall cycles.	This event counts the number of stall cycles due to conditions described by: ▪ RAT_STALLS.ROB_READ_PORT ▪ RAT_STALLS.PARTIAL ▪ RAT_STALLS.FLAGS ▪ RAT_STALLS.FPSW.
D4H	01H	SEG_RENAME_ STALLS.ES	Segment rename stalls - ES.	This event counts the number of stalls due to the lack of renaming resources for the ES segment register. If a segment is renamed, but not retired and a second update to the same segment occurs, a stall occurs in the front end of the pipeline until the renamed segment retires.
D4H	02H	SEG_RENAME_ STALLS.DS	Segment rename stalls - DS.	This event counts the number of stalls due to the lack of renaming resources for the DS segment register. If a segment is renamed, but not retired and a second update to the same segment occurs, a stall occurs in the front end of the pipeline until the renamed segment retires.
D4H	04H	SEG_RENAME_ STALLS.FS	Segment rename stalls - FS.	This event counts the number of stalls due to the lack of renaming resources for the FS segment register. If a segment is renamed, but not retired and a second update to the same segment occurs, a stall occurs in the front end of the pipeline until the renamed segment retires.
D4H	08H	SEG_RENAME_ STALLS.GS	Segment rename stalls - GS.	This event counts the number of stalls due to the lack of renaming resources for the GS segment register. If a segment is renamed, but not retired and a second update to the same segment occurs, a stall occurs in the front end of the pipeline until the renamed segment retires.
D4H	0FH	SEG_RENAME_ STALLS.ANY	Any (ES/DS/FS/GS) segment rename stall.	This event counts the number of stalls due to the lack of renaming resources for the ES, DS, FS, and GS segment registers. If a segment is renamed but not retired and a second update to the same segment occurs, a stall occurs in the front end of the pipeline until the renamed segment retires.

Table 19-25. Non-Architectural Performance Events in Processors Based on Intel® Core™ Microarchitecture (Contd.)

Event Num	Umask Value	Event Name	Definition	Description and Comment
D5H	01H	SEG_REG_ RENAMES.ES	Segment renames - ES.	This event counts the number of times the ES segment register is renamed.
D5H	02H	SEG_REG_ RENAMES.DS	Segment renames - DS.	This event counts the number of times the DS segment register is renamed.
D5H	04H	SEG_REG_ RENAMES.FS	Segment renames - FS.	This event counts the number of times the FS segment register is renamed.
D5H	08H	SEG_REG_ RENAMES.GS	Segment renames - GS.	This event counts the number of times the GS segment register is renamed.
D5H	0FH	SEG_REG_ RENAMES.ANY	Any (ES/DS/FS/GS) segment rename.	This event counts the number of times any of the four segment registers (ES/DS/FS/GS) is renamed.
DCH	01H	RESOURCE_ STALLS.ROB_FULL	Cycles during which the ROB full.	This event counts the number of cycles when the number of instructions in the pipeline waiting for retirement reaches the limit the processor can handle. A high count for this event indicates that there are long latency operations in the pipe (possibly load and store operations that miss the L2 cache, and other instructions that depend on these cannot execute until the former instructions complete execution). In this situation new instructions cannot enter the pipe and start execution.
DCH	02H	RESOURCE_ STALLS.RS_FULL	Cycles during which the RS full.	This event counts the number of cycles when the number of instructions in the pipeline waiting for execution reaches the limit the processor can handle. A high count of this event indicates that there are long latency operations in the pipe (possibly load and store operations that miss the L2 cache, and other instructions that depend on these cannot execute until the former instructions complete execution). In this situation new instructions cannot enter the pipe and start execution.
DCH	04	RESOURCE_ STALLS.LD_ST	Cycles during which the pipeline has exceeded load or store limit or waiting to commit all stores.	This event counts the number of cycles while resource-related stalls occur due to: ▪ The number of load instructions in the pipeline reached the limit the processor can handle. The stall ends when a loading instruction retires. ▪ The number of store instructions in the pipeline reached the limit the processor can handle. The stall ends when a storing instruction commits its data to the cache or memory. ▪ There is an instruction in the pipe that can be executed only when all previous stores complete and their data is committed in the caches or memory. For example, the SFENCE and MFENCE instructions require this behavior.
DCH	08H	RESOURCE_ STALLS.FPCW	Cycles stalled due to FPU control word write.	This event counts the number of cycles while execution was stalled due to writing the floating-point unit (FPU) control word.
DCH	10H	RESOURCE_ STALLS.BR_MISS_CLEAR	Cycles stalled due to branch misprediction.	This event counts the number of cycles after a branch misprediction is detected at execution until the branch and all older micro-ops retire. During this time new micro-ops cannot enter the out-of-order pipeline.

Table 19-25. Non-Architectural Performance Events in Processors Based on Intel® Core™ Microarchitecture (Contd.)

Event Num	Umask Value	Event Name	Definition	Description and Comment
DCH	1FH	RESOURCE_ STALLS.ANY	Resource related stalls.	This event counts the number of cycles while resource-related stalls occurs for any conditions described by the following events: ▪ RESOURCE_STALLS.ROB_FULL ▪ RESOURCE_STALLS.RS_FULL ▪ RESOURCE_STALLS.LD_ST ▪ RESOURCE_STALLS.FPCW ▪ RESOURCE_STALLS.BR_MISS_CLEAR
E0H	00H	BR_INST_ DECODED	Branch instructions decoded.	This event counts the number of branch instructions decoded.
E4H	00H	BOGUS_BR	Bogus branches.	This event counts the number of byte sequences that were mistakenly detected as taken branch instructions. This results in a BACLEAR event. This occurs mainly after task switches.
E6H	00H	BACLEARS	BACLEARS asserted.	This event counts the number of times the front end is resteered, mainly when the BPU cannot provide a correct prediction and this is corrected by other branch handling mechanisms at the front and. This can occur if the code has many branches such that they cannot be consumed by the BPU. Each BACLEAR asserted costs approximately 7 cycles of instruction fetch. The effect on total execution time depends on the surrounding code.
F0H	00H	PREF_RQSTS_UP	Upward prefetches issued from DPL.	This event counts the number of upward prefetches issued from the Data Prefetch Logic (DPL) to the L2 cache. A prefetch request issued to the L2 cache cannot be cancelled and the requested cache line is fetched to the L2 cache.
F8H	00H	PREF_RQSTS_DN	Downward prefetches issued from DPL.	This event counts the number of downward prefetches issued from the Data Prefetch Logic (DPL) to the L2 cache. A prefetch request issued to the L2 cache cannot be cancelled and the requested cache line is fetched to the L2 cache.

19.13 PERFORMANCE MONITORING EVENTS FOR PROCESSORS BASED ON THE GOLDMONT MICROARCHITECTURE

Next Generation Intel Atom processors based on the Goldmont microarchitecture support the architectural performance-monitoring events listed in Table 19-1 and fixed-function performance events using a fixed counter. In addition, they also support the following non-architectural performance-monitoring events listed in Table 19-26. These events also apply to processors with CPUID signatures of 06_5CH and 06_5FH.

Performance monitoring event descriptions may refer to terminology described in Section B.2, "Intel® Xeon® processor 5500 Series," in Appendix B of the *Intel® 64 and IA-32 Architectures Optimization Reference Manual*.

In Goldmont microarchitecture, performance monitoring events that support Processor Event Based Sampling (PEBS) and PEBS records that contain processor state information that are associated with at-retirement tagging are marked by "Precise Event".

Table 19-26. Non-Architectural Performance Events for the Goldmont Microarchitecture

Event Num.	Umask Value	Event Name	Description	Comment
03H	10H	LD_BLOCKS.ALL_BLOCK	Counts anytime a load that retires is blocked for any reason.	Precise Event
03H	08H	LD_BLOCKS.UTLB_MISS	Counts loads blocked because they are unable to find their physical address in the micro TLB (UTLB).	Precise Event
03H	02H	LD_BLOCKS.STORE_FORWARD	Counts a load blocked from using a store forward because of an address/size mismatch; only one of the loads blocked from each store will be counted.	Precise Event
03H	01H	LD_BLOCKS.DATA_UNKNOWN	Counts a load blocked from using a store forward, but did not occur because the store data was not available at the right time. The forward might occur subsequently when the data is available.	Precise Event
03H	04H	LD_BLOCKS.4K_ALIAS	Counts loads that block because their address modulo 4K matches a pending store.	Precise Event
05H	01H	PAGE_WALKS.D_SIDE_CYCLES	Counts every core cycle when a Data-side (walks due to data operation) page walk is in progress.	
05H	02H	PAGE_WALKS.I_SIDE_CYCLES	Counts every core cycle when an Instruction-side (walks due to an instruction fetch) page walk is in progress.	
05H	03H	PAGE_WALKS.CYCLES	Counts every core cycle a page-walk is in progress due to either a data memory operation, or an instruction fetch.	
0EH	00H	UOPS_ISSUED.ANY	Counts uops issued by the front end and allocated into the back end of the machine. This event counts uops that retire as well as uops that were speculatively executed but didn't retire. The sort of speculative uops that might be counted includes, but is not limited to those uops issued in the shadow of a mispredicted branch, those uops that are inserted during an assist (such as for a denormal floating-point result), and (previously allocated) uops that might be canceled during a machine clear.	
13H	02H	MISALIGN_MEM_REF.LOAD_PAGE_SPLIT	Counts when a memory load of a uop that spans a page boundary (a split) is retired.	Precise Event
13H	04H	MISALIGN_MEM_REF.STORE_PAGE_SPLIT	Counts when a memory store of a uop that spans a page boundary (a split) is retired.	Precise Event
2EH	4FH	LONGEST_LAT_CACHE.REFERENCE	Counts memory requests originating from the core that reference a cache line in the L2 cache.	
2EH	41H	LONGEST_LAT_CACHE.MISS	Counts memory requests originating from the core that miss in the L2 cache.	

Table 19-26. Non-Architectural Performance Events for the Goldmont Microarchitecture (Contd.)

Event Num.	Umask Value	Event Name	Description	Comment
30H	00H	L2_REJECT_XQ.ALL	Counts the number of demand and prefetch transactions that the L2 XQ rejects due to a full or near full condition which likely indicates back pressure from the intra-die interconnect (IDI) fabric. The XQ may reject transactions from the L2Q (non-cacheable requests), L2 misses and L2 write-back victims.	
31H	00H	CORE_REJECT_L2Q.ALL	Counts the number of demand and L1 prefetcher requests rejected by the L2Q due to a full or nearly full condition which likely indicates back pressure from L2Q. It also counts requests that would have gone directly to the XQ, but are rejected due to a full or nearly full condition, indicating back pressure from the IDI link. The L2Q may also reject transactions from a core to ensure fairness between cores, or to delay a core's dirty eviction when the address conflicts with incoming external snoops.	
3CH	00H	CPU_CLK_UNHALTED.CORE_P	Core cycles when core is not halted. This event uses a programmable general purpose performance counter.	
3CH	01H	CPU_CLK_UNHALTED.REF	Reference cycles when core is not halted. This event uses a programmable general purpose performance counter.	
51H	01H	DL1.DIRTY_EVICTION	Counts when a modified (dirty) cache line is evicted from the data L1 cache and needs to be written back to memory. No count will occur if the evicted line is clean, and hence does not require a writeback.	
80H	01H	ICACHE.HIT	Counts requests to the Instruction Cache (ICache) for one or more bytes in an ICache Line and that cache line is in the Icache (hit). The event strives to count on a cache line basis, so that multiple accesses which hit in a single cache line count as one ICACHE.HIT. Specifically, the event counts when straight line code crosses the cache line boundary, or when a branch target is to a new line, and that cache line is in the ICache. This event counts differently than Intel processors based on the Silvermont microarchitecture.	
80H	02H	ICACHE.MISSES	Counts requests to the Instruction Cache (ICache) for one or more bytes in an ICache Line and that cache line is not in the Icache (miss). The event strives to count on a cache line basis, so that multiple accesses which miss in a single cache line count as one ICACHE.MISS. Specifically, the event counts when straight line code crosses the cache line boundary, or when a branch target is to a new line, and that cache line is not in the ICache. This event counts differently than Intel processors based on the Silvermont microarchitecture.	
80H	03H	ICACHE.ACCESSES	Counts requests to the Instruction Cache (ICache) for one or more bytes in an ICache Line. The event strives to count on a cache line basis, so that multiple fetches to a single cache line count as one ICACHE.ACCESS. Specifically, the event counts when accesses from straight line code crosses the cache line boundary, or when a branch target is to a new line. This event counts differently than Intel processors based on the Silvermont microarchitecture.	
81H	04H	ITLB.MISS	Counts the number of times the machine was unable to find a translation in the Instruction Translation Lookaside Buffer (ITLB) for a linear address of an instruction fetch. It counts when new translations are filled into the ITLB. The event is speculative in nature, but will not count translations (page walks) that are begun and not finished, or translations that are finished but not filled into the ITLB.	

Table 19-26. Non-Architectural Performance Events for the Goldmont Microarchitecture (Contd.)

Event Num.	Umask Value	Event Name	Description	Comment
86H	02H	FETCH_STALL.ICACHE_FILL_PENDING_CYCLES	Counts cycles that an ICache miss is outstanding, and instruction fetch is stalled. That is, the decoder queue is able to accept bytes, but the fetch unit is unable to provide bytes, while an Icache miss is outstanding. Note this event is not the same as cycles to retrieve an instruction due to an Icache miss. Rather, it is the part of the Instruction Cache (ICache) miss time where no bytes are available for the decoder.	
9CH	00H	UOPS_NOT_DELIVERED.ANY	This event is used to measure front-end inefficiencies, i.e., when the front end of the machine is not delivering uops to the back end and the back end has not stalled. This event can be used to identify if the machine is truly front-end bound. When this event occurs, it is an indication that the front end of the machine is operating at less than its theoretical peak performance. Background: We can think of the processor pipeline as being divided into 2 broader parts: the front end and the back end. The front end is responsible for fetching the instruction, decoding into uops in machine understandable format and putting them into a uop queue to be consumed by the back end. The back end then takes these uops and allocates the required resources. When all resources are ready, uops are executed. If the back end is not ready to accept uops from the front end, then we do not want to count these as front-end bottlenecks. However, whenever we have bottlenecks in the back end, we will have allocation unit stalls and eventually force the front end to wait until the back end is ready to receive more uops. This event counts only when the back end is requesting more micro-uops and the front end is not able to provide them. When 3 uops are requested and no uops are delivered, the event counts 3. When 3 are requested, and only 1 is delivered, the event counts 2. When only 2 are delivered, the event counts 1. Alternatively stated, the event will not count if 3 uops are delivered, or if the back end is stalled and not requesting any uops at all. Counts indicate missed opportunities for the front end to deliver a uop to the back end. Some examples of conditions that cause front-end efficiencies are: Icache misses, ITLB misses, and decoder restrictions that limit the front-end bandwidth. Known Issues: Some uops require multiple allocation slots. These uops will not be charged as a front end 'not delivered' opportunity, and will be regarded as a back-end problem. For example, the INC instruction has one uop that requires 2 issue slots. A stream of INC instructions will not count as UOPS_NOT_DELIVERED, even though only one instruction can be issued per clock. The low uop issue rate for a stream of INC instructions is considered to be a back-end issue.	
B7H	01H, 02H	OFFCORE_RESPONSE	Requires MSR_OFFCORE_RESP[0,1] to specify request type and response. (Duplicated for both MSRs.)	
C0H	00H	INST_RETIRED.ANY_P	Counts the number of instructions that retire execution. For instructions that consist of multiple uops, this event counts the retirement of the last uop of the instruction. The event continues counting during hardware interrupts, traps, and inside interrupt handlers. This is an architectural performance event. This event uses a programmable general purpose performance counter. *This event is a Precise Event: the EventingRIP field in the PEBS record is precise to the address of the instruction which caused the event. Note: Because PEBS records can be collected only on IA32_PMC0, only one event can use the PEBS facility at a time.	Precise Event

Table 19-26. Non-Architectural Performance Events for the Goldmont Microarchitecture (Contd.)

Event Num.	Umask Value	Event Name	Description	Comment
C2H	00H	UOPS_RETIRED.ANY	Counts uops which have retired.	Precise Event
C2H	01H	UOPS_RETIRED.MS	Counts uops retired that are from the complex flows issued by the micro-sequencer (MS). Counts both the uops from a micro-coded instruction, and the uops that might be generated from a micro-coded assist.	Precise Event
C3H	01H	MACHINE_CLEARS.SMC	Counts the number of times that the processor detects that a program is writing to a code section and has to perform a machine clear because of that modification. Self-modifying code (SMC) causes a severe penalty in all Intel architecture processors.	
C3H	02H	MACHINE_CLEARS.MEMORY_ORDERING	Counts machine clears due to memory ordering issues. This occurs when a snoop request happens and the machine is uncertain if memory ordering will be preserved as another core is in the process of modifying the data.	
C3H	04H	MACHINE_CLEARS.FP_ASSIST	Counts machine clears due to floating-point (FP) operations needing assists. For instance, if the result was a floating-point denormal, the hardware clears the pipeline and reissues uops to produce the correct IEEE compliant denormal result.	
C3H	08H	MACHINE_CLEARS.DISAMBIGUATION	Counts machine clears due to memory disambiguation. Memory disambiguation happens when a load which has been issued conflicts with a previous un-retired store in the pipeline whose address was not known at issue time, but is later resolved to be the same as the load address.	
C3H	00H	MACHINE_CLEARS.ALL	Counts machine clears for any reason.	
C4H	00H	BR_INST_RETIRED.ALL_BRANCHES	Counts branch instructions retired for all branch types. This is an architectural performance event.	Precise Event
C4H	7EH	BR_INST_RETIRED.JCC	Counts retired Jcc (Jump on Conditional Code/Jump if Condition is Met) branch instructions retired, including both when the branch was taken and when it was not taken.	Precise Event
C4H	FEH	BR_INST_RETIRED.TAKEN_JCC	Counts Jcc (Jump on Conditional Code/Jump if Condition is Met) branch instructions retired that were taken and does not count when the Jcc branch instruction were not taken.	Precise Event
C4H	F9H	BR_INST_RETIRED.CALL	Counts near CALL branch instructions retired.	Precise Event
C4H	FDH	BR_INST_RETIRED.REL_CALL	Counts near relative CALL branch instructions retired.	Precise Event
C4H	FBH	BR_INST_RETIRED.IND_CALL	Counts near indirect CALL branch instructions retired.	Precise Event
C4H	F7H	BR_INST_RETIRED.RETURN	Counts near return branch instructions retired.	Precise Event
C4H	EBH	BR_INST_RETIRED.NON_RETURN_IND	Counts near indirect call or near indirect jmp branch instructions retired.	Precise Event
C4H	BFH	BR_INST_RETIRED.FAR_BRANCH	Counts far branch instructions retired. This includes far jump, far call and return, and Interrupt call and return.	Precise Event
C5H	00H	BR_MISP_RETIRED.ALL_BRANCHES	Counts mispredicted branch instructions retired including all branch types.	Precise Event
C5H	7EH	BR_MISP_RETIRED.JCC	Counts mispredicted retired Jcc (Jump on Conditional Code/Jump if Condition is Met) branch instructions retired, including both when the branch was supposed to be taken and when it was not supposed to be taken (but the processor predicted the opposite condition).	Precise Event

Table 19-26. Non-Architectural Performance Events for the Goldmont Microarchitecture (Contd.)

Event Num.	Umask Value	Event Name	Description	Comment
C5H	FEH	BR_MISP_RETIRED.TAKEN_JCC	Counts mispredicted retired Jcc (Jump on Conditional Code/Jump if Condition is Met) branch instructions retired that were supposed to be taken but the processor predicted that it would not be taken.	Precise Event
C5H	FBH	BR_MISP_RETIRED.IND_CALL	Counts mispredicted near indirect CALL branch instructions retired, where the target address taken was not what the processor predicted.	Precise Event
C5H	F7H	BR_MISP_RETIRED.RETURN	Counts mispredicted near RET branch instructions retired, where the return address taken was not what the processor predicted.	Precise Event
C5H	EBH	BR_MISP_RETIRED.NON_RETURN_IND	Counts mispredicted branch instructions retired that were near indirect call or near indirect jmp, where the target address taken was not what the processor predicted.	Precise Event
CAH	01H	ISSUE_SLOTS_NOT_CONSUMED.RESOURCE_FULL	Counts the number of issue slots per core cycle that were not consumed because of a full resource in the back end. Including but not limited to resources include the Re-order Buffer (ROB), reservation stations (RS), load/store buffers, physical registers, or any other needed machine resource that is currently unavailable. Note that uops must be available for consumption in order for this event to fire. If a uop is not available (Instruction Queue is empty), this event will not count.	
CAH	02H	ISSUE_SLOTS_NOT_CONSUMED.RECOVERY	Counts the number of issue slots per core cycle that were not consumed by the back end because allocation is stalled waiting for a mispredicted jump to retire or other branch-like conditions (e.g. the event is relevant during certain microcode flows). Counts all issue slots blocked while within this window, including slots where uops were not available in the Instruction Queue.	
CAH	00H	ISSUE_SLOTS_NOT_CONSUMED.ANY	Counts the number of issue slots per core cycle that were not consumed by the back end due to either a full resource in the back end (RESOURCE_FULL), or due to the processor recovering from some event (RECOVERY).	
CBH	01H	HW_INTERRUPTS.RECEIVED	Counts hardware interrupts received by the processor.	
CBH	04H	HW_INTERRUPTS.PENDING_AND_MASKED	Counts core cycles during which there are pending interrupts, but interrupts are masked (EFLAGS.IF = 0).	
CDH	00H	CYCLES_DIV_BUSY.ALL	Counts core cycles if either divide unit is busy.	
CDH	01H	CYCLES_DIV_BUSY.IDIV	Counts core cycles if the integer divide unit is busy.	
CDH	02H	CYCLES_DIV_BUSY.FPDIV	Counts core cycles if the floating point divide unit is busy.	
D0H	81H	MEM_UOPS_RETIRED.ALL_LOADS	Counts the number of load uops retired.	Precise Event
D0H	82H	MEM_UOPS_RETIRED.ALL_STORES	Counts the number of store uops retired.	Precise Event
D0H	83H	MEM_UOPS_RETIRED.ALL	Counts the number of memory uops retired that are either a load or a store or both.	Precise Event
D0H	11H	MEM_UOPS_RETIRED.DTLB_MISS_LOADS	Counts load uops retired that caused a DTLB miss.	Precise Event
D0H	12H	MEM_UOPS_RETIRED.DTLB_MISS_STORES	Counts store uops retired that caused a DTLB miss.	Precise Event

Table 19-26. Non-Architectural Performance Events for the Goldmont Microarchitecture (Contd.)

Event Num.	Umask Value	Event Name	Description	Comment
D0H	13H	MEM_UOPS_RETIRED.DTLB_MISS	Counts uops retired that had a DTLB miss on load, store or either. Note that when two distinct memory operations to the same page miss the DTLB, only one of them will be recorded as a DTLB miss.	Precise Event
D0H	21H	MEM_UOPS_RETIRED.LOCK_LOADS	Counts locked memory uops retired. This includes 'regular' locks and bus locks. To specifically count bus locks only, see the offcore response event. A locked access is one with a lock prefix, or an exchange to memory.	Precise Event
D0H	41H	MEM_UOPS_RETIRED.SPLIT_LOADS	Counts load uops retired where the data requested spans a 64 byte cache line boundary.	Precise Event
D0H	42H	MEM_UOPS_RETIRED.SPLIT_STORES	Counts store uops retired where the data requested spans a 64 byte cache line boundary.	Precise Event
D0H	43H	MEM_UOPS_RETIRED.SPLIT	Counts memory uops retired where the data requested spans a 64 byte cache line boundary.	Precise Event
D1H	01H	MEM_LOAD_UOPS_RETIRED.L1_HIT	Counts load uops retired that hit the L1 data cache.	Precise Event
D1H	08H	MEM_LOAD_UOPS_RETIRED.L1_MISS	Counts load uops retired that miss the L1 data cache.	Precise Event
D1H	02H	MEM_LOAD_UOPS_RETIRED.L2_HIT	Counts load uops retired that hit in the L2 cache.	Precise Event
0xD1H	10H	MEM_LOAD_UOPS_RETIRED.L2_MISS	Counts load uops retired that miss in the L2 cache.	Precise Event
D1H	20H	MEM_LOAD_UOPS_RETIRED.HITM	Counts load uops retired where the cache line containing the data was in the modified state of another core or modules cache (HITM). More specifically, this means that when the load address was checked by other caching agents (typically another processor) in the system, one of those caching agents indicated that they had a dirty copy of the data. Loads that obtain a HITM response incur greater latency than most that is typical for a load. In addition, since HITM indicates that some other processor had this data in its cache, it implies that the data was shared between processors, or potentially was a lock or semaphore value. This event is useful for locating sharing, false sharing, and contended locks.	Precise Event
D1H	40H	MEM_LOAD_UOPS_RETIRED.WCB_HIT	Counts memory load uops retired where the data is retrieved from the WCB (or fill buffer), indicating that the load found its data while that data was in the process of being brought into the L1 cache. Typically a load will receive this indication when some other load or prefetch missed the L1 cache and was in the process of retrieving the cache line containing the data, but that process had not yet finished (and written the data back to the cache). For example, consider load X and Y, both referencing the same cache line that is not in the L1 cache. If load X misses cache first, it obtains and WCB (or fill buffer) begins the process of requesting the data. When load Y requests the data, it will either hit the WCB, or the L1 cache, depending on exactly what time the request to Y occurs.	Precise Event
D1H	80H	MEM_LOAD_UOPS_RETIRED.DRAM_HIT	Counts memory load uops retired where the data is retrieved from DRAM. Event is counted at retirement, so the speculative loads are ignored. A memory load can hit (or miss) the L1 cache, hit (or miss) the L2 cache, hit DRAM, hit in the WCB or receive a HITM response.	Precise Event

Table 19-26. Non-Architectural Performance Events for the Goldmont Microarchitecture (Contd.)

Event Num.	Umask Value	Event Name	Description	Comment
E6H	01H	BACLEARS.ALL	Counts the number of times a BACLEAR is signaled for any reason, including, but not limited to indirect branch/call, Jcc (Jump on Conditional Code/Jump if Condition is Met) branch, unconditional branch/call, and returns.	
E6H	08H	BACLEARS.RETURN	Counts BACLEARS on return instructions.	
E6H	10H	BACLEARS.COND	Counts BACLEARS on Jcc (Jump on Conditional Code/Jump if Condition is Met) branches.	
E7H	01H	MS_DECODED.MS_ENTRY	Counts the number of times the Microcode Sequencer (MS) starts a flow of uops from the MSROM. It does not count every time a uop is read from the MSROM. The most common case that this counts is when a micro-coded instruction is encountered by the front end of the machine. Other cases include when an instruction encounters a fault, trap, or microcode assist of any sort that initiates a flow of uops. The event will count MS startups for uops that are speculative, and subsequently cleared by branch mispredict or a machine clear.	
E9H	01H	DECODE_RESTRICTION.PREDECODE_WRONG	Counts the number of times the prediction (from the pre-decode cache) for instruction length is incorrect.	

19.14 PERFORMANCE MONITORING EVENTS FOR PROCESSORS BASED ON THE SILVERMONT MICROARCHITECTURE

Processors based on the Silvermont microarchitecture support the architectural performance-monitoring events listed in Table 19-1 and fixed-function performance events using fixed counter. In addition, they also support the following non-architectural performance-monitoring events listed in Table 19-27. These processors have the CPUID signatures of 06_37H, 06_4AH, 06_4DH, 06_5AH, and 06_5DH.

Performance monitoring event descriptions may refer to terminology described in Section B.2, "Intel® Xeon® processor 5500 Series," in Appendix B of the *Intel® 64 and IA-32 Architectures Optimization Reference Manual*.

Table 19-27. Performance Events for Silvermont Microarchitecture

Event Num.	Umask Value	Event Name	Definition	Description and Comment
03H	01H	REHABQ.LD_BLOCK_ST_FORWARD	Loads blocked due to store forward restriction.	This event counts the number of retired loads that were prohibited from receiving forwarded data from the store because of address mismatch.
03H	02H	REHABQ.LD_BLOCK_STD_NOTREADY	Loads blocked due to store data not ready.	This event counts the cases where a forward was technically possible, but did not occur because the store data was not available at the right time.
03H	04H	REHABQ.ST_SPLITS	Store uops that split cache line boundary.	This event counts the number of retire stores that experienced cache line boundary splits.
03H	08H	REHABQ.LD_SPLITS	Load uops that split cache line boundary.	This event counts the number of retire loads that experienced cache line boundary splits.
03H	10H	REHABQ.LOCK	Uops with lock semantics.	This event counts the number of retired memory operations with lock semantics. These are either implicit locked instructions such as the XCHG instruction or instructions with an explicit LOCK prefix (F0H).
03H	20H	REHABQ.STA_FULL	Store address buffer full.	This event counts the number of retired stores that are delayed because there is not a store address buffer available.

Table 19-27. Performance Events for Silvermont Microarchitecture

Event Num.	Umask Value	Event Name	Definition	Description and Comment
03H	40H	REHABQ.ANY_LD	Any reissued load uops.	This event counts the number of load uops reissued from Rehabq.
03H	80H	REHABQ.ANY_ST	Any reissued store uops.	This event counts the number of store uops reissued from Rehabq.
04H	01H	MEM_UOPS_RETIRED.L1_MISS_LOADS	Loads retired that missed L1 data cache.	This event counts the number of load ops retired that miss in L1 Data cache. Note that prefetch misses will not be counted.
04H	02H	MEM_UOPS_RETIRED.L2_HIT_LOADS	Loads retired that hit L2.	This event counts the number of load micro-ops retired that hit L2.
04H	04H	MEM_UOPS_RETIRED.L2_MISS_LOADS	Loads retired that missed L2.	This event counts the number of load micro-ops retired that missed L2.
04H	08H	MEM_UOPS_RETIRED.DTLB_MISS_LOADS	Loads missed DTLB.	This event counts the number of load ops retired that had DTLB miss.
04H	10H	MEM_UOPS_RETIRED.UTLB_MISS	Loads missed UTLB.	This event counts the number of load ops retired that had UTLB miss.
04H	20H	MEM_UOPS_RETIRED.HITM	Cross core or cross module hitm.	This event counts the number of load ops retired that got data from the other core or from the other module.
04H	40H	MEM_UOPS_RETIRED.ALL_LOADS	All Loads.	This event counts the number of load ops retired.
04H	80H	MEM_UOP_RETIRED.ALL_STORES	All Stores.	This event counts the number of store ops retired.
05H	01H	PAGE_WALKS.D_SIDE_CYCLES	Duration of D-side page-walks in core cycles.	This event counts every cycle when a D-side (walks due to a load) page walk is in progress. Page walk duration divided by number of page walks is the average duration of page-walks. Edge trigger bit must be cleared. Set Edge to count the number of page walks.
05H	02H	PAGE_WALKS.I_SIDE_CYCLES	Duration of I-side page-walks in core cycles.	This event counts every cycle when an I-side (walks due to an instruction fetch) page walk is in progress. Page walk duration divided by number of page walks is the average duration of page-walks. Edge trigger bit must be cleared. Set Edge to count the number of page walks.
05H	03H	PAGE_WALKS.WALKS	Total number of page-walks that are completed (I-side and D-side).	This event counts when a data (D) page walk or an instruction (I) page walk is completed or started. Since a page walk implies a TLB miss, the number of TLB misses can be counted by counting the number of pagewalks. Edge trigger bit must be set. Clear Edge to count the number of cycles.
2EH	41H	LONGEST_LAT_CACHE.MISS	L2 cache request misses.	This event counts the total number of L2 cache references and the number of L2 cache misses respectively. L3 is not supported in Silvermont microarchitecture.
2EH	4FH	LONGEST_LAT_CACHE.REFERENCE	L2 cache requests from this core.	This event counts requests originating from the core that references a cache line in the L2 cache. L3 is not supported in Silvermont microarchitecture.
30H	00H	L2_REJECT_XQ.ALL	Counts the number of request from the L2 that were not accepted into the XQ.	This event counts the number of demand and prefetch transactions that the L2 XQ rejects due to a full or near full condition which likely indicates back pressure from the IDI link. The XQ may reject transactions from the L2Q (non-cacheable requests), BBS (L2 misses) and WOB (L2 write-back victims).

Table 19-27. Performance Events for Silvermont Microarchitecture

Event Num.	Umask Value	Event Name	Definition	Description and Comment
31H	00H	CORE_REJECT_L2Q.ALL	Counts the number of request that were not accepted into the L2Q because the L2Q is FULL.	This event counts the number of demand and L1 prefetcher requests rejected by the L2Q due to a full or nearly full condition which likely indicates back pressure from L2Q. It also counts requests that would have gone directly to the XQ, but are rejected due to a full or nearly full condition, indicating back pressure from the IDI link. The L2Q may also reject transactions from a core to insure fairness between cores, or to delay a core's dirty eviction when the address conflicts incoming external snoops. (Note that L2 prefetcher requests that are dropped are not counted by this event.).
3CH	00H	CPU_CLK_UNHALTED.CORE_P	Core cycles when core is not halted.	This event counts the number of core cycles while the core is not in a halt state. The core enters the halt state when it is running the HLT instruction. In mobile systems the core frequency may change from time to time. For this reason this event may have a changing ratio with regards to time.
N/A	N/A	CPU_CLK_UNHALTED.CORE	Core cycles when core is not halted.	This uses the fixed counter 1 to count the same condition as CPU_CLK_UNHALTED.CORE_P does.
3CH	01H	CPU_CLK_UNHALTED.REF_P	Bus cycles when core is not halted.	This event counts the number of bus cycles that the core is not in a halt state. The core enters the halt state when it is running the HLT instruction. In mobile systems the core frequency may change from time. This event is not affected by core frequency changes.
N/A	N/A	CPU_CLK_UNHALTED.REF_TSC	Reference cycles when core is not halted.	This event counts the number of reference cycles at a TSC rate that the core is not in a halt state. The core enters the halt state when it is running the HLT instruction. In mobile systems the core frequency may change from time. This event is not affected by core frequency changes.
80H	01H	ICACHE.HIT	Instruction fetches from Icache.	This event counts all instruction fetches from the instruction cache.
80H	02H	ICACHE.MISSES	Icache miss.	This event counts all instruction fetches that miss the Instruction cache or produce memory requests. This includes uncacheable fetches. An instruction fetch miss is counted only once and not once for every cycle it is outstanding.
80H	03H	ICACHE.ACCESSES	Instruction fetches.	This event counts all instruction fetches, including uncacheable fetches.
B7H	01H	OFFCORE_RESPONSE_0	See Section 18.5.2.2.	Requires MSR_OFFCORE_RESP0 to specify request type and response.
B7H	02H	OFFCORE_RESPONSE_1	See Section 18.5.2.2.	Requires MSR_OFFCORE_RESP1 to specify request type and response.
C0H	00H	INST_RETIRED.ANY_P	Instructions retired (PEBS supported with IA32_PMC0).	This event counts the number of instructions that retire execution. For instructions that consist of multiple micro-ops, this event counts the retirement of the last micro-op of the instruction. The counter continues counting during hardware interrupts, traps, and inside interrupt handlers.
N/A	N/A	INST_RETIRED.ANY	Instructions retired.	This uses the fixed counter 0 to count the same condition as INST_RETIRED.ANY_P does.
C2H	01H	UOPS_RETIRED.MS	MSROM micro-ops retired.	This event counts the number of micro-ops retired that were supplied from MSROM.
C2H	10H	UOPS_RETIRED.ALL	Micro-ops retired.	This event counts the number of micro-ops retired.

Table 19-27. Performance Events for Silvermont Microarchitecture

Event Num.	Umask Value	Event Name	Definition	Description and Comment
C3H	01H	MACHINE_CLEARS.SMC	Self-Modifying Code detected.	This event counts the number of times that a program writes to a code section. Self-modifying code causes a severe penalty in all Intel® architecture processors.
C3H	02H	MACHINE_CLEARS.MEMORY_ORDERING	Stalls due to Memory ordering.	This event counts the number of times that pipeline was cleared due to memory ordering issues.
C3H	04H	MACHINE_CLEARS.FP_ASSIST	Stalls due to FP assists.	This event counts the number of times that pipeline stalled due to FP operations needing assists.
C3H	08H	MACHINE_CLEARS.ALL	Stalls due to any causes.	This event counts the number of times that pipeline stalled due to due to any causes (including SMC, MO, FP assist, etc.).
C4H	00H	BR_INST_RETIRED.ALL_BRANCHES	Retired branch instructions.	This event counts the number of branch instructions retired.
C4H	7EH	BR_INST_RETIRED.JCC	Retired branch instructions that were conditional jumps.	This event counts the number of branch instructions retired that were conditional jumps.
C4H	BFH	BR_INST_RETIRED.FAR_BRANCH	Retired far branch instructions.	This event counts the number of far branch instructions retired.
C4H	EBH	BR_INST_RETIRED.NON_RETURN_IND	Retired instructions of near indirect Jmp or call.	This event counts the number of branch instructions retired that were near indirect call or near indirect jmp.
C4H	F7H	BR_INST_RETIRED.RETURN	Retired near return instructions.	This event counts the number of near RET branch instructions retired.
C4H	F9H	BR_INST_RETIRED.CALL	Retired near call instructions.	This event counts the number of near CALL branch instructions retired.
C4H	FBH	BR_INST_RETIRED.IND_CALL	Retired near indirect call instructions.	This event counts the number of near indirect CALL branch instructions retired.
C4H	FDH	BR_INST_RETIRED.REL_CALL	Retired near relative call instructions.	This event counts the number of near relative CALL branch instructions retired.
C4H	FEH	BR_INST_RETIRED.TAKEN_JCC	Retired conditional jumps that were taken.	This event counts the number of branch instructions retired that were conditional jumps and taken.
C5H	00H	BR_MISP_RETIRED.ALL_BRANCHES	Retired mispredicted branch instructions.	This event counts the number of mispredicted branch instructions retired.
C5H	7EH	BR_MISP_RETIRED.JCC	Retired mispredicted conditional jumps.	This event counts the number of mispredicted branch instructions retired that were conditional jumps.
C5H	BFH	BR_MISP_RETIRED.FAR	Retired mispredicted far branch instructions.	This event counts the number of mispredicted far branch instructions retired.
C5H	EBH	BR_MISP_RETIRED.NON_RETURN_IND	Retired mispredicted instructions of near indirect Jmp or call.	This event counts the number of mispredicted branch instructions retired that were near indirect call or near indirect jmp.
C5H	F7H	BR_MISP_RETIRED.RETURN	Retired mispredicted near return instructions.	This event counts the number of mispredicted near RET branch instructions retired.
C5H	F9H	BR_MISP_RETIRED.CALL	Retired mispredicted near call instructions.	This event counts the number of mispredicted near CALL branch instructions retired.
C5H	FBH	BR_MISP_RETIRED.IND_CALL	Retired mispredicted near indirect call instructions.	This event counts the number of mispredicted near indirect CALL branch instructions retired.

Table 19-27. Performance Events for Silvermont Microarchitecture

Event Num.	Umask Value	Event Name	Definition	Description and Comment
C5H	FDH	BR_MISP_RETIRED.REL_CALL	Retired mispredicted near relative call instructions	This event counts the number of mispredicted near relative CALL branch instructions retired.
C5H	FEH	BR_MISP_RETIRED.TAKEN_JCC	Retired mispredicted conditional jumps that were taken.	This event counts the number of mispredicted branch instructions retired that were conditional jumps and taken.
CAH	01H	NO_ALLOC_CYCLES.ROB_FULL	Counts the number of cycles when no uops are allocated and the ROB is full (less than 2 entries available).	Counts the number of cycles when no uops are allocated and the ROB is full (less than 2 entries available).
CAH	20H	NO_ALLOC_CYCLES.RAT_STALL	Counts the number of cycles when no uops are allocated and a RATstall is asserted.	Counts the number of cycles when no uops are allocated and a RATstall is asserted.
CAH	3FH	NO_ALLOC_CYCLES.ALL	Front end not delivering.	This event counts the number of cycles when the front end does not provide any instructions to be allocated for any reason.
CAH	50H	NO_ALLOC_CYCLES.NOT_DELIVERED	Front end not delivering back end not stalled.	This event counts the number of cycles when the front end does not provide any instructions to be allocated but the back end is not stalled.
CBH	01H	RS_FULL_STALL.MEC	MEC RS full.	This event counts the number of cycles the allocation pipe line stalled due to the RS for the MEC cluster is full.
CBH	1FH	RS_FULL_STALL.ALL	Any RS full.	This event counts the number of cycles that the allocation pipe line stalled due to any one of the RS is full.
CDH	01H	CYCLES_DIV_BUSY.ANY	Divider Busy.	This event counts the number of cycles the divider is busy.
E6H	01H	BACLEARS.ALL	BACLEARS asserted for any branch.	This event counts the number of baclears for any type of branch.
E6H	08H	BACLEARS.RETURN	BACLEARS asserted for return branch.	This event counts the number of baclears for return branches.
E6H	10H	BACLEARS.COND	BACLEARS asserted for conditional branch.	This event counts the number of baclears for conditional branches.
E7H	01H	MS_DECODED.MS_ENTRY	MS Decode starts.	This event counts the number of times the MSROM starts a flow of UOPS.

19.14.1 Performance Monitoring Events for Processors Based on the Airmont Microarchitecture

Intel processors based on the Airmont microarchitecture support the same architectural and the non-architectural performance monitoring events as processors based on the Silvermont microarchitecture. All of the events listed in Table 19-27 apply. These processors have the CPUID signatures that include 06_4CH.

19.15 PERFORMANCE MONITORING EVENTS FOR 45 NM AND 32 NM INTEL® ATOM™ PROCESSORS

45 nm and 32 nm processors based on the Intel® Atom™ microarchitecture support the architectural performance-monitoring events listed in Table 19-1 and fixed-function performance events using fixed counter listed in Table 19-24. In addition, they also support the following non-architectural performance-monitoring events listed in Table 19-28.

Table 19-28. Non-Architectural Performance Events for 45 nm, 32 nm Intel® Atom™ Processors

Event Num.	Umask Value	Event Name	Definition	Description and Comment
02H	81H	STORe_FORWARDS.GOOD	Good store forwards.	This event counts the number of times store data was forwarded directly to a load.
06H	00H	SEGMENT_REG_LOADS.ANY	Number of segment register loads.	This event counts the number of segment register load operations. Instructions that load new values into segment registers cause a penalty. This event indicates performance issues in 16-bit code. If this event occurs frequently, it may be useful to calculate the number of instructions retired per segment register load. If the resulting calculation is low (on average a small number of instructions are executed between segment register loads), then the code's segment register usage should be optimized. As a result of branch misprediction, this event is speculative and may include segment register loads that do not actually occur. However, most segment register loads are internally serialized and such speculative effects are minimized.
07H	01H	PREFETCH.PREFETCHT0	Streaming SIMD Extensions (SSE) PrefetchT0 instructions executed.	This event counts the number of times the SSE instruction prefetchT0 is executed. This instruction prefetches the data to the L1 data cache and L2 cache.
07H	06H	PREFETCH.SW_L2	Streaming SIMD Extensions (SSE) PrefetchT1 and PrefetchT2 instructions executed.	This event counts the number of times the SSE instructions prefetchT1 and prefetchT2 are executed. These instructions prefetch the data to the L2 cache.
07H	08H	PREFETCH.PREFETCHNTA	Streaming SIMD Extensions (SSE) Prefetch NTA instructions executed.	This event counts the number of times the SSE instruction prefetchNTA is executed. This instruction prefetches the data to the L1 data cache.
08H	07H	DATA_TLB_MISSES.DTLB_MISS	Memory accesses that missed the DTLB.	This event counts the number of Data Table Lookaside Buffer (DTLB) misses. The count includes misses detected as a result of speculative accesses. Typically a high count for this event indicates that the code accesses a large number of data pages.
08H	05H	DATA_TLB_MISSES.DTLB_MISS_LD	DTLB misses due to load operations.	This event counts the number of Data Table Lookaside Buffer (DTLB) misses due to load operations. This count includes misses detected as a result of speculative accesses.
08H	09H	DATA_TLB_MISSES.L0_DTLB_MISS_LD	L0_DTLB misses due to load operations.	This event counts the number of L0_DTLB misses due to load operations. This count includes misses detected as a result of speculative accesses.
08H	06H	DATA_TLB_MISSES.DTLB_MISS_ST	DTLB misses due to store operations.	This event counts the number of Data Table Lookaside Buffer (DTLB) misses due to store operations. This count includes misses detected as a result of speculative accesses.

Table 19-28. Non-Architectural Performance Events for 45 nm, 32 nm Intel® Atom™ Processors (Contd.)

Event Num.	Umask Value	Event Name	Definition	Description and Comment
0CH	03H	PAGE_WALKS.WALKS	Number of page-walks executed.	This event counts the number of page-walks executed due to either a DTLB or ITLB miss. The page walk duration, PAGE_WALKS.CYCLES, divided by number of page walks is the average duration of a page walk. This can hint to whether most of the page-walks are satisfied by the caches or cause an L2 cache miss. Edge trigger bit must be set.
0CH	03H	PAGE_WALKS.CYCLES	Duration of page-walks in core cycles.	This event counts the duration of page-walks in core cycles. The paging mode in use typically affects the duration of page walks. Page walk duration divided by number of page walks is the average duration of page-walks. This can hint at whether most of the page-walks are satisfied by the caches or cause an L2 cache miss. Edge trigger bit must be cleared.
10H	01H	X87_COMP_OPS_EXE.ANY.S	Floating point computational micro-ops executed.	This event counts the number of x87 floating point computational micro-ops executed.
10H	81H	X87_COMP_OPS_EXE.ANY.AR	Floating point computational micro-ops retired.	This event counts the number of x87 floating point computational micro-ops retired.
11H	01H	FP_ASSIST	Floating point assists.	This event counts the number of floating point operations executed that required micro-code assist intervention. These assists are required in the following cases. X87 instructions: 1. NaN or denormal are loaded to a register or used as input from memory. 2. Division by 0. 3. Underflow output.
11H	81H	FP_ASSIST.AR	Floating point assists.	This event counts the number of floating point operations executed that required micro-code assist intervention. These assists are required in the following cases. X87 instructions: 1. NaN or denormal are loaded to a register or used as input from memory. 2. Division by 0. 3. Underflow output.
12H	01H	MUL.S	Multiply operations executed.	This event counts the number of multiply operations executed. This includes integer as well as floating point multiply operations.
12H	81H	MUL.AR	Multiply operations retired.	This event counts the number of multiply operations retired. This includes integer as well as floating point multiply operations.
13H	01H	DIV.S	Divide operations executed.	This event counts the number of divide operations executed. This includes integer divides, floating point divides and square-root operations executed.
13H	81H	DIV.AR	Divide operations retired.	This event counts the number of divide operations retired. This includes integer divides, floating point divides and square-root operations executed.

Table 19-28. Non-Architectural Performance Events for 45 nm, 32 nm Intel® Atom™ Processors (Contd.)

Event Num.	Umask Value	Event Name	Definition	Description and Comment
14H	01H	CYCLES_DIV_BUSY	Cycles the driver is busy.	This event counts the number of cycles the divider is busy executing divide or square root operations. The divide can be integer, X87 or Streaming SIMD Extensions (SSE). The square root operation can be either X87 or SSE.
21H	See Table 18-61	L2_ADS	Cycles L2 address bus is in use.	This event counts the number of cycles the L2 address bus is being used for accesses to the L2 cache or bus queue. This event can count occurrences for this core or both cores.
22H	See Table 18-61	L2_DBUS_BUSY	Cycles the L2 cache data bus is busy.	This event counts core cycles during which the L2 cache data bus is busy transferring data from the L2 cache to the core. It counts for all L1 cache misses (data and instruction) that hit the L2 cache. The count will increment by two for a full cache-line request.
24H	See Table 18-61 and Table 18-63	L2_LINES_IN	L2 cache misses.	This event counts the number of cache lines allocated in the L2 cache. Cache lines are allocated in the L2 cache as a result of requests from the L1 data and instruction caches and the L2 hardware prefetchers to cache lines that are missing in the L2 cache. This event can count occurrences for this core or both cores. This event can also count demand requests and L2 hardware prefetch requests together or separately.
25H	See Table 18-61	L2_M_LINES_IN	L2 cache line modifications.	This event counts whenever a modified cache line is written back from the L1 data cache to the L2 cache. This event can count occurrences for this core or both cores.
26H	See Table 18-61 and Table 18-63	L2_LINES_OUT	L2 cache lines evicted.	This event counts the number of L2 cache lines evicted. This event can count occurrences for this core or both cores. This event can also count evictions due to demand requests and L2 hardware prefetch requests together or separately.
27H	See Table 18-61 and Table 18-63	L2_M_LINES_OUT	Modified lines evicted from the L2 cache.	This event counts the number of L2 modified cache lines evicted. These lines are written back to memory unless they also exist in a shared-state in one of the L1 data caches. This event can count occurrences for this core or both cores. This event can also count evictions due to demand requests and L2 hardware prefetch requests together or separately.
28H	See Table 18-61 and Table 18-64	L2_IFETCH	L2 cacheable instruction fetch requests.	This event counts the number of instruction cache line requests from the ICache. It does not include fetch requests from uncacheable memory. It does not include ITLB miss accesses. This event can count occurrences for this core or both cores. This event can also count accesses to cache lines at different MESI states.

Table 19-28. Non-Architectural Performance Events for 45 nm, 32 nm Intel® Atom™ Processors (Contd.)

Event Num.	Umask Value	Event Name	Definition	Description and Comment
29H	See Table 18-61, Table 18-63 and Table 18-64	L2_LD	L2 cache reads.	This event counts L2 cache read requests coming from the L1 data cache and L2 prefetchers. This event can count occurrences for this core or both cores. This event can count occurrences - for this core or both cores. - due to demand requests and L2 hardware prefetch requests together or separately. - of accesses to cache lines at different MESI states.
2AH	See Table 18-61 and Table 18-64	L2_ST	L2 store requests.	This event counts all store operations that miss the L1 data cache and request the data from the L2 cache. This event can count occurrences for this core or both cores. This event can also count accesses to cache lines at different MESI states.
2BH	See Table 18-61 and Table 18-64	L2_LOCK	L2 locked accesses.	This event counts all locked accesses to cache lines that miss the L1 data cache. This event can count occurrences for this core or both cores. This event can also count accesses to cache lines at different MESI states.
2EH	See Table 18-61, Table 18-63 and Table 18-64	L2_RQSTS	L2 cache requests.	This event counts all completed L2 cache requests. This includes L1 data cache reads, writes, and locked accesses, L1 data prefetch requests, instruction fetches, and all L2 hardware prefetch requests. This event can count occurrences - for this core or both cores. - due to demand requests and L2 hardware prefetch requests together, or separately. - of accesses to cache lines at different MESI states.
2EH	41H	L2_RQSTS.SELF.DEMAND.I_STATE	L2 cache demand requests from this core that missed the L2.	This event counts all completed L2 cache demand requests from this core that miss the L2 cache. This includes L1 data cache reads, writes, and locked accesses, L1 data prefetch requests, and instruction fetches. This is an architectural performance event.
2EH	4FH	L2_RQSTS.SELF.DEMAND.MESI	L2 cache demand requests from this core.	This event counts all completed L2 cache demand requests from this core. This includes L1 data cache reads, writes, and locked accesses, L1 data prefetch requests, and instruction fetches. This is an architectural performance event.

Table 19-28. Non-Architectural Performance Events for 45 nm, 32 nm Intel® Atom™ Processors (Contd.)

Event Num.	Umask Value	Event Name	Definition	Description and Comment
30H	See Table 18-61, Table 18-63 and Table 18-64	L2_REJECT_BUSQ	Rejected L2 cache requests.	This event indicates that a pending L2 cache request that requires a bus transaction is delayed from moving to the bus queue. Some of the reasons for this event are: - The bus queue is full. - The bus queue already holds an entry for a cache line in the same set. The number of events is greater or equal to the number of requests that were rejected. - For this core or both cores. - Due to demand requests and L2 hardware prefetch requests together, or separately. - Of accesses to cache lines at different MESI states.
32H	See Table 18-61	L2_NO_REQ	Cycles no L2 cache requests are pending.	This event counts the number of cycles that no L2 cache requests are pending.
3AH	00H	EIST_TRANS	Number of Enhanced Intel SpeedStep(R) Technology (EIST) transitions.	This event counts the number of Enhanced Intel SpeedStep(R) Technology (EIST) transitions that include a frequency change, either with or without VID change. This event is incremented only while the counting core is in C0 state. In situations where an EIST transition was caused by hardware as a result of CxE state transitions, those EIST transitions will also be registered in this event. Enhanced Intel Speedstep Technology transitions are commonly initiated by OS, but can be initiated by HW internally. For example: CxE states are C-states (C1,C2,C3...) which not only place the CPU into a sleep state by turning off the clock and other components, but also lower the voltage (which reduces the leakage power consumption). The same is true for thermal throttling transition which uses Enhanced Intel Speedstep Technology internally.
3BH	C0H	THERMAL_TRIP	Number of thermal trips.	This event counts the number of thermal trips. A thermal trip occurs whenever the processor temperature exceeds the thermal trip threshold temperature. Following a thermal trip, the processor automatically reduces frequency and voltage. The processor checks the temperature every millisecond, and returns to normal when the temperature falls below the thermal trip threshold temperature.

Table 19-28. Non-Architectural Performance Events for 45 nm, 32 nm Intel® Atom™ Processors (Contd.)

Event Num.	Umask Value	Event Name	Definition	Description and Comment
3CH	00H	CPU_CLK_UNHALTED.CORE_P	Core cycles when core is not halted.	This event counts the number of core cycles while the core is not in a halt state. The core enters the halt state when it is running the HLT instruction. This event is a component in many key event ratios. In mobile systems the core frequency may change from time to time. For this reason this event may have a changing ratio with regards to time. In systems with a constant core frequency, this event can give you a measurement of the elapsed time while the core was not in halt state by dividing the event count by the core frequency. -This is an architectural performance event. - The event CPU_CLK_UNHALTED.CORE_P is counted by a programmable counter. - The event CPU_CLK_UNHALTED.CORE is counted by a designated fixed counter, leaving the two programmable counters available for other events.
3CH	01H	CPU_CLK_UNHALTED.BUS	Bus cycles when core is not halted.	This event counts the number of bus cycles while the core is not in the halt state. This event can give you a measurement of the elapsed time while the core was not in the halt state, by dividing the event count by the bus frequency. The core enters the halt state when it is running the HLT instruction. The event also has a constant ratio with CPU_CLK_UNHALTED.REF event, which is the maximum bus to processor frequency ratio. Non-halted bus cycles are a component in many key event ratios.
3CH	02H	CPU_CLK_UNHALTED.NO_OTHER	Bus cycles when core is active and the other is halted.	This event counts the number of bus cycles during which the core remains non-halted, and the other core on the processor is halted. This event can be used to determine the amount of parallelism exploited by an application or a system. Divide this event count by the bus frequency to determine the amount of time that only one core was in use.
40H	21H	L1D_CACHE.LD	L1 Cacheable Data Reads.	This event counts the number of data reads from cacheable memory.
40H	22H	L1D_CACHE.ST	L1 Cacheable Data Writes.	This event counts the number of data writes to cacheable memory.
60H	See Table 18-61 and Table 18-62.	BUS_REQUEST_OUTSTANDING	Outstanding cacheable data read bus requests duration.	This event counts the number of pending full cache line read transactions on the bus occurring in each cycle. A read transaction is pending from the cycle it is sent on the bus until the full cache line is received by the processor. NOTE: This event is thread-independent and will not provide a count per logical processor when AnyThr is disabled.

Table 19-28. Non-Architectural Performance Events for 45 nm, 32 nm Intel® Atom™ Processors (Contd.)

Event Num.	Umask Value	Event Name	Definition	Description and Comment
61H	See Table 18-62.	BUS_BNR_DRV	Number of Bus Not Ready signals asserted.	This event counts the number of Bus Not Ready (BNR) signals that the processor asserts on the bus to suspend additional bus requests by other bus agents. A bus agent asserts the BNR signal when the number of data and snoop transactions is close to the maximum that the bus can handle. While this signal is asserted, new transactions cannot be submitted on the bus. As a result, transaction latency may have higher impact on program performance. NOTE: This event is thread-independent and will not provide a count per logical processor when AnyThr is disabled.
62H	See Table 18-62.	BUS_DRDY_CLOCKS	Bus cycles when data is sent on the bus.	This event counts the number of bus cycles during which the DRDY (Data Ready) signal is asserted on the bus. The DRDY signal is asserted when data is sent on the bus. This event counts the number of bus cycles during which this agent (the processor) writes data on the bus back to memory or to other bus agents. This includes all explicit and implicit data writebacks, as well as partial writes. Note: This event is thread-independent and will not provide a count per logical processor when AnyThr is disabled.
63H	See Table 18-61 and Table 18-62.	BUS_LOCK_CLOCKS	Bus cycles when a LOCK signal is asserted.	This event counts the number of bus cycles, during which the LOCK signal is asserted on the bus. A LOCK signal is asserted when there is a locked memory access, due to: - Uncacheable memory. - Locked operation that spans two cache lines. - Page-walk from an uncacheable page table. Bus locks have a very high performance penalty and it is highly recommended to avoid such accesses. NOTE: This event is thread-independent and will not provide a count per logical processor when AnyThr is disabled.
64H	See Table 18-61.	BUS_DATA_RCV	Bus cycles while processor receives data.	This event counts the number of cycles during which the processor is busy receiving data. NOTE: This event is thread-independent and will not provide a count per logical processor when AnyThr is disabled.
65H	See Table 18-61 and Table 18-62.	BUS_TRANS_BRD	Burst read bus transactions.	This event counts the number of burst read transactions including: - L1 data cache read misses (and L1 data cache hardware prefetches). - L2 hardware prefetches by the DPL and L2 streamer. - IFU read misses of cacheable lines. It does not include RFO transactions.
66H	See Table 18-61 and Table 18-62.	BUS_TRANS_RFO	RFO bus transactions.	This event counts the number of Read For Ownership (RFO) bus transactions, due to store operations that miss the L1 data cache and the L2 cache. This event also counts RFO bus transactions due to locked operations.

Table 19-28. Non-Architectural Performance Events for 45 nm, 32 nm Intel® Atom™ Processors (Contd.)

Event Num.	Umask Value	Event Name	Definition	Description and Comment
67H	See Table 18-61 and Table 18-62.	BUS_TRANS_WB	Explicit writeback bus transactions.	This event counts all explicit writeback bus transactions due to dirty line evictions. It does not count implicit writebacks due to invalidation by a snoop request.
68H	See Table 18-61 and Table 18-62.	BUS_TRANS_IFETCH	Instruction-fetch bus transactions.	This event counts all instruction fetch full cache line bus transactions.
69H	See Table 18-61 and Table 18-62.	BUS_TRANS_INVAL	Invalidate bus transactions.	This event counts all invalidate transactions. Invalidate transactions are generated when: - A store operation hits a shared line in the L2 cache. - A full cache line write misses the L2 cache or hits a shared line in the L2 cache.
6AH	See Table 18-61 and Table 18-62.	BUS_TRANS_PWR	Partial write bus transaction.	This event counts partial write bus transactions.
6BH	See Table 18-61 and Table 18-62.	BUS_TRANS_P	Partial bus transactions.	This event counts all (read and write) partial bus transactions.
6CH	See Table 18-61 and Table 18-62.	BUS_TRANS_IO	IO bus transactions.	This event counts the number of completed I/O bus transactions as a result of IN and OUT instructions. The count does not include memory mapped IO.
6DH	See Table 18-61 and Table 18-62.	BUS_TRANS_DEF	Deferred bus transactions.	This event counts the number of deferred transactions.
6EH	See Table 18-61 and Table 18-62.	BUS_TRANS_BURST	Burst (full cache-line) bus transactions.	This event counts burst (full cache line) transactions including: - Burst reads. - RFOs. - Explicit writebacks. - Write combine lines.

Table 19-28. Non-Architectural Performance Events for 45 nm, 32 nm Intel® Atom™ Processors (Contd.)

Event Num.	Umask Value	Event Name	Definition	Description and Comment
6FH	See Table 18-61 and Table 18-62.	BUS_TRANS_MEM	Memory bus transactions.	This event counts all memory bus transactions including: - Burst transactions. - Partial reads and writes. - Invalidate transactions. The BUS_TRANS_MEM count is the sum of BUS_TRANS_BURST, BUS_TRANS_P and BUS_TRANS_INVAL.
70H	See Table 18-61 and Table 18-62.	BUS_TRANS_ANY	All bus transactions.	This event counts all bus transactions. This includes: - Memory transactions. - IO transactions (non memory-mapped). - Deferred transaction completion. - Other less frequent transactions, such as interrupts.
77H	See Table 18-61 and Table 18-64.	EXT_SNOOP	External snoops.	This event counts the snoop responses to bus transactions. Responses can be counted separately by type and by bus agent. Note: This event is thread-independent and will not provide a count per logical processor when AnyThr is disabled.
7AH	See Table 18-62.	BUS_HIT_DRV	HIT signal asserted.	This event counts the number of bus cycles during which the processor drives the HIT# pin to signal HIT snoop response. Note: This event is thread-independent and will not provide a count per logical processor when AnyThr is disabled.
7BH	See Table 18-62.	BUS_HITM_DRV	HITM signal asserted.	This event counts the number of bus cycles during which the processor drives the HITM# pin to signal HITM snoop response. NOTE: This event is thread-independent and will not provide a count per logical processor when AnyThr is disabled.
7DH	See Table 18-61.	BUSQ_EMPTY	Bus queue is empty.	This event counts the number of cycles during which the core did not have any pending transactions in the bus queue. Note: This event is thread-independent and will not provide a count per logical processor when AnyThr is disabled.
7EH	See Table 18-61 and Table 18-62.	SNOOP_STALL_DRV	Bus stalled for snoops.	This event counts the number of times that the bus snoop stall signal is asserted. During the snoop stall cycles no new bus transactions requiring a snoop response can be initiated on the bus. Note: This event is thread-independent and will not provide a count per logical processor when AnyThr is disabled.
7FH	See Table 18-61.	BUS_IO_WAIT	IO requests waiting in the bus queue.	This event counts the number of core cycles during which IO requests wait in the bus queue. This event counts IO requests from the core.
80H	03H	ICACHE.ACCESSES	Instruction fetches.	This event counts all instruction fetches, including uncacheable fetches.
80H	02H	ICACHE.MISSES	Icache miss.	This event counts all instruction fetches that miss the Instruction cache or produce memory requests. This includes uncacheable fetches. An instruction fetch miss is counted only once and not once for every cycle it is outstanding.
82H	04H	ITLB.FLUSH	ITLB flushes.	This event counts the number of ITLB flushes.
82H	02H	ITLB.MISSES	ITLB misses.	This event counts the number of instruction fetches that miss the ITLB.

Table 19-28. Non-Architectural Performance Events for 45 nm, 32 nm Intel® Atom™ Processors (Contd.)

Event Num.	Umask Value	Event Name	Definition	Description and Comment
AAH	02H	MACRO_INSTS.CISC_DECODED	CISC macro instructions decoded.	This event counts the number of complex instructions decoded, but not necessarily executed or retired. Only one complex instruction can be decoded at a time.
AAH	03H	MACRO_INSTS.ALL_DECODED	All Instructions decoded.	This event counts the number of instructions decoded.
B0H	00H	SIMD_UOPS_EXEC.S	SIMD micro-ops executed (excluding stores).	This event counts all the SIMD micro-ops executed. This event does not count MOVQ and MOVD stores from register to memory.
B0H	80H	SIMD_UOPS_EXEC.AR	SIMD micro-ops retired (excluding stores).	This event counts the number of SIMD saturated arithmetic micro-ops executed.
B1H	00H	SIMD_SAT_UOP_EXEC.S	SIMD saturated arithmetic micro-ops executed.	This event counts the number of SIMD saturated arithmetic micro-ops executed.
B1H	80H	SIMD_SAT_UOP_EXEC.AR	SIMD saturated arithmetic micro-ops retired.	This event counts the number of SIMD saturated arithmetic micro-ops retired.
B3H	01H	SIMD_UOP_TYPE_EXEC.MUL.S	SIMD packed multiply micro-ops executed.	This event counts the number of SIMD packed multiply micro-ops executed.
B3H	81H	SIMD_UOP_TYPE_EXEC.MUL.AR	SIMD packed multiply micro-ops retired.	This event counts the number of SIMD packed multiply micro-ops retired.
B3H	02H	SIMD_UOP_TYPE_EXEC.SHIFT.S	SIMD packed shift micro-ops executed.	This event counts the number of SIMD packed shift micro-ops executed.
B3H	82H	SIMD_UOP_TYPE_EXEC.SHIFT.AR	SIMD packed shift micro-ops retired.	This event counts the number of SIMD packed shift micro-ops retired.
B3H	04H	SIMD_UOP_TYPE_EXEC.PACK.S	SIMD pack micro-ops executed.	This event counts the number of SIMD pack micro-ops executed.
B3H	84H	SIMD_UOP_TYPE_EXEC.PACK.AR	SIMD pack micro-ops retired.	This event counts the number of SIMD pack micro-ops retired.
B3H	08H	SIMD_UOP_TYPE_EXEC.UNPACK.S	SIMD unpack micro-ops executed.	This event counts the number of SIMD unpack micro-ops executed.
B3H	88H	SIMD_UOP_TYPE_EXEC.UNPACK.AR	SIMD unpack micro-ops retired.	This event counts the number of SIMD unpack micro-ops retired.
B3H	10H	SIMD_UOP_TYPE_EXEC.LOGICAL.S	SIMD packed logical micro-ops executed.	This event counts the number of SIMD packed logical micro-ops executed.
B3H	90H	SIMD_UOP_TYPE_EXEC.LOGICAL.AR	SIMD packed logical micro-ops retired.	This event counts the number of SIMD packed logical micro-ops retired.
B3H	20H	SIMD_UOP_TYPE_EXEC.ARITHMETIC.S	SIMD packed arithmetic micro-ops executed.	This event counts the number of SIMD packed arithmetic micro-ops executed.
B3H	A0H	SIMD_UOP_TYPE_EXEC.ARITHMETIC.AR	SIMD packed arithmetic micro-ops retired.	This event counts the number of SIMD packed arithmetic micro-ops retired.
C0H	00H	INST_RETIRED.ANY_P	Instructions retired (precise event).	This event counts the number of instructions that retire execution. For instructions that consist of multiple micro-ops, this event counts the retirement of the last micro-op of the instruction. The counter continues counting during hardware interrupts, traps, and inside interrupt handlers.

Table 19-28. Non-Architectural Performance Events for 45 nm, 32 nm Intel® Atom™ Processors (Contd.)

Event Num.	Umask Value	Event Name	Definition	Description and Comment
N/A	00H	INST_RETIRED.ANY	Instructions retired.	This event counts the number of instructions that retire execution. For instructions that consist of multiple micro-ops, this event counts the retirement of the last micro-op of the instruction. The counter continues counting during hardware interrupts, traps, and inside interrupt handlers.
C2H	10H	UOPS_RETIRED.ANY	Micro-ops retired.	This event counts the number of micro-ops retired. The processor decodes complex macro instructions into a sequence of simpler micro-ops. Most instructions are composed of one or two micro-ops. Some instructions are decoded into longer sequences such as repeat instructions, floating point transcendental instructions, and assists. In some cases micro-op sequences are fused or whole instructions are fused into one micro-op. See other UOPS_RETIRED events for differentiating retired fused and non-fused micro-ops.
C3H	01H	MACHINE_CLEARS.SMC	Self-Modifying Code detected.	This event counts the number of times that a program writes to a code section. Self-modifying code causes a severe penalty in all Intel® architecture processors.
C4H	00H	BR_INST_RETIRED.ANY	Retired branch instructions.	This event counts the number of branch instructions retired. **This is an architectural performance event.**
C4H	01H	BR_INST_RETIRED.PRED_NOT_TAKEN	Retired branch instructions that were predicted not-taken.	This event counts the number of branch instructions retired that were correctly predicted to be not-taken.
C4H	02H	BR_INST_RETIRED.MISPRED_NOT_TAKEN	Retired branch instructions that were mispredicted not-taken.	This event counts the number of branch instructions retired that were mispredicted and not-taken.
C4H	04H	BR_INST_RETIRED.PRED_TAKEN	Retired branch instructions that were predicted taken.	This event counts the number of branch instructions retired that were correctly predicted to be taken.
C4H	08H	BR_INST_RETIRED.MISPRED_TAKEN	Retired branch instructions that were mispredicted taken.	This event counts the number of branch instructions retired that were mispredicted and taken.
C4H	0AH	BR_INST_RETIRED.MISPRED	Retired mispredicted branch instructions (precise event).	This event counts the number of retired branch instructions that were mispredicted by the processor. A branch misprediction occurs when the processor predicts that the branch would be taken, but it is not, or vice-versa. Mispredicted branches degrade the performance because the processor starts executing instructions along a wrong path it predicts. When the misprediction is discovered, all the instructions executed in the wrong path must be discarded, and the processor must start again on the correct path. Using the Profile-Guided Optimization (PGO) features of the Intel® C++ compiler may help reduce branch mispredictions. See the compiler documentation for more information on this feature.

Table 19-28. Non-Architectural Performance Events for 45 nm, 32 nm Intel® Atom™ Processors (Contd.)

Event Num.	Umask Value	Event Name	Definition	Description and Comment
				To determine the branch misprediction ratio, divide the BR_INST_RETIRED.MISPRED event count by the number of BR_INST_RETIRED.ANY event count. To determine the number of mispredicted branches per instruction, divide the number of mispredicted branches by the INST_RETIRED.ANY event count. To measure the impact of the branch mispredictions use the event RESOURCE_STALLS.BR_MISS_CLEAR. Tips: - See the optimization guide for tips on reducing branch mispredictions. - PGO's purpose is to have straight line code for the most frequent execution paths, reducing branches taken and increasing the "basic block" size, possibly also reducing the code footprint or working-set.
C4H	0CH	BR_INST_RETIRED.TAKEN	Retired taken branch instructions.	This event counts the number of branches retired that were taken.
C4H	0FH	BR_INST_RETIRED.ANY1	Retired branch instructions.	This event counts the number of branch instructions retired that were mispredicted. This event is a duplicate of BR_INST_RETIRED.MISPRED.
C5H	00H	BR_INST_RETIRED.MISPRED	Retired mispredicted branch instructions (precise event).	This event counts the number of retired branch instructions that were mispredicted by the processor. A branch misprediction occurs when the processor predicts that the branch would be taken, but it is not, or vice-versa. Mispredicted branches degrade the performance because the processor starts executing instructions along a wrong path it predicts. When the misprediction is discovered, all the instructions executed in the wrong path must be discarded, and the processor must start again on the correct path. Using the Profile-Guided Optimization (PGO) features of the Intel® C++ compiler may help reduce branch mispredictions. See the compiler documentation for more information on this feature. To determine the branch misprediction ratio, divide the BR_INST_RETIRED.MISPRED event count by the number of BR_INST_RETIRED.ANY event count. To determine the number of mispredicted branches per instruction, divide the number of mispredicted branches by the INST_RETIRED.ANY event count. To measure the impact of the branch mispredictions use the event RESOURCE_STALLS.BR_MISS_CLEAR. Tips: - See the optimization guide for tips on reducing branch mispredictions. - PGO's purpose is to have straight line code for the most frequent execution paths, reducing branches taken and increasing the "basic block" size, possibly also reducing the code footprint or working-set.
C6H	01H	CYCLES_INT_MASKED.CYCLES_INT_MASKED	Cycles during which interrupts are disabled.	This event counts the number of cycles during which interrupts are disabled.
C6H	02H	CYCLES_INT_MASKED.CYCLES_INT_PENDING_AND_MASKED	Cycles during which interrupts are pending and disabled.	This event counts the number of cycles during which there are pending interrupts but interrupts are disabled.

Table 19-28. Non-Architectural Performance Events for 45 nm, 32 nm Intel® Atom™ Processors (Contd.)

Event Num.	Umask Value	Event Name	Definition	Description and Comment
C7H	01H	SIMD_INST_RETIRED.PACKED_SINGLE	Retired Streaming SIMD Extensions (SSE) packed-single instructions.	This event counts the number of SSE packed-single instructions retired.
C7H	02H	SIMD_INST_RETIRED.SCALAR_SINGLE	Retired Streaming SIMD Extensions (SSE) scalar-single instructions.	This event counts the number of SSE scalar-single instructions retired.
C7H	04H	SIMD_INST_RETIRED.PACKED_DOUBLE	Retired Streaming SIMD Extensions 2 (SSE2) packed-double instructions.	This event counts the number of SSE2 packed-double instructions retired.
C7H	08H	SIMD_INST_RETIRED.SCALAR_DOUBLE	Retired Streaming SIMD Extensions 2 (SSE2) scalar-double instructions.	This event counts the number of SSE2 scalar-double instructions retired.
C7H	10H	SIMD_INST_RETIRED.VECTOR	Retired Streaming SIMD Extensions 2 (SSE2) vector instructions.	This event counts the number of SSE2 vector instructions retired.
C7H	1FH	SIMD_INST_RETIRED.ANY	Retired Streaming SIMD instructions.	This event counts the overall number of SIMD instructions retired. To count each type of SIMD instruction separately, use the following events: SIMD_INST_RETIRED.PACKED_SINGLE SIMD_INST_RETIRED.SCALAR_SINGLE SIMD_INST_RETIRED.PACKED_DOUBLE SIMD_INST_RETIRED.SCALAR_DOUBLE SIMD_INST_RETIRED.VECTOR.
C8H	00H	HW_INT_RCV	Hardware interrupts received.	This event counts the number of hardware interrupts received by the processor. This event will count twice for dual-pipe micro-ops.
CAH	01H	SIMD_COMP_INST_RETIRED.PACKED_SINGLE	Retired computational Streaming SIMD Extensions (SSE) packed-single instructions.	This event counts the number of computational SSE packed-single instructions retired. Computational instructions perform arithmetic computations, like add, multiply and divide. Instructions that perform load and store operations or logical operations, like XOR, OR, and AND are not counted by this event.
CAH	02H	SIMD_COMP_INST_RETIRED.SCALAR_SINGLE	Retired computational Streaming SIMD Extensions (SSE) scalar-single instructions.	This event counts the number of computational SSE scalar-single instructions retired. Computational instructions perform arithmetic computations, like add, multiply and divide. Instructions that perform load and store operations or logical operations, like XOR, OR, and AND are not counted by this event.
CAH	04H	SIMD_COMP_INST_RETIRED.PACKED_DOUBLE	Retired computational Streaming SIMD Extensions 2 (SSE2) packed-double instructions.	This event counts the number of computational SSE2 packed-double instructions retired. Computational instructions perform arithmetic computations, like add, multiply and divide. Instructions that perform load and store operations or logical operations, like XOR, OR, and AND are not counted by this event.

Table 19-28. Non-Architectural Performance Events for 45 nm, 32 nm Intel® Atom™ Processors (Contd.)

Event Num.	Umask Value	Event Name	Definition	Description and Comment
CAH	08H	SIMD_COMP_INST_RETIRED.SCALAR_DOUBLE	Retired computational Streaming SIMD Extensions 2 (SSE2) scalar-double instructions.	This event counts the number of computational SSE2 scalar-double instructions retired. Computational instructions perform arithmetic computations, like add, multiply and divide. Instructions that perform load and store operations or logical operations, like XOR, OR, and AND are not counted by this event.
CBH	01H	MEM_LOAD_RETIRED.L2_HIT	Retired loads that hit the L2 cache (precise event).	This event counts the number of retired load operations that missed the L1 data cache and hit the L2 cache.
CBH	02H	MEM_LOAD_RETIRED.L2_MISS	Retired loads that miss the L2 cache (precise event).	This event counts the number of retired load operations that missed the L2 cache.
CBH	04H	MEM_LOAD_RETIRED.DTLB_MISS	Retired loads that miss the DTLB (precise event).	This event counts the number of retired loads that missed the DTLB. The DTLB miss is not counted if the load operation causes a fault.
CDH	00H	SIMD_ASSIST	SIMD assists invoked.	This event counts the number of SIMD assists invoked. SIMD assists are invoked when an EMMS instruction is executed after MMX™ technology code has changed the MMX state in the floating point stack. For example, these assists are required in the following cases. Streaming SIMD Extensions (SSE) instructions: 1. Denormal input when the DAZ (Denormals Are Zeros) flag is off. 2. Underflow result when the FTZ (Flush To Zero) flag is off.
CEH	00H	SIMD_INSTR_RETIRED	SIMD Instructions retired.	This event counts the number of SIMD instructions that retired.
CFH	00H	SIMD_SAT_INSTR_RETIRED	Saturated arithmetic instructions retired.	This event counts the number of saturated arithmetic SIMD instructions that retired.
E0H	01H	BR_INST_DECODED	Branch instructions decoded.	This event counts the number of branch instructions decoded.
E4H	01H	BOGUS_BR	Bogus branches.	This event counts the number of byte sequences that were mistakenly detected as taken branch instructions. This results in a BACLEAR event and the BTB is flushed. This occurs mainly after task switches.
E6H	01H	BACLEARS.ANY	BACLEARS asserted.	This event counts the number of times the front end is redirected for a branch prediction, mainly when an early branch prediction is corrected by other branch handling mechanisms in the front end. This can occur if the code has many branches such that they cannot be consumed by the branch predictor. Each Baclear asserted costs approximately 7 cycles. The effect on total execution time depends on the surrounding code.

19.16 PERFORMANCE MONITORING EVENTS FOR INTEL® CORE™ SOLO AND INTEL® CORE™ DUO PROCESSORS

Table 19-29 lists non-architectural performance events for Intel® Core™ Duo processors. If a non-architectural event requires qualification in core specificity, it is indicated in the comment column. Table 19-29 also applies to Intel® Core™ Solo processors; bits in the unit mask corresponding to core-specificity are reserved and should be 00B.

Table 19-29. Non-Architectural Performance Events in Intel® Core™ Solo and Intel® Core™ Duo Processors

Event Num.	Event Mask Mnemonic	Umask Value	Description	Comment
03H	LD_Blocks	00H	Load operations delayed due to store buffer blocks. The preceding store may be blocked due to unknown address, unknown data, or conflict due to partial overlap between the load and store.	
04H	SD_Drains	00H	Cycles while draining store buffers.	
05H	Misalign_Mem_Ref	00H	Misaligned data memory references (MOB splits of loads and stores).	
06H	Seg_Reg_Loads	00H	Segment register loads.	
07H	SSE_PrefNta_Ret	00H	SSE software prefetch instruction PREFETCHNTA retired.	
07H	SSE_PrefT1_Ret	01H	SSE software prefetch instruction PREFETCHT1 retired.	
07H	SSE_PrefT2_Ret	02H	SSE software prefetch instruction PREFETCHT2 retired.	
07H	SSE_NTStores_Ret	03H	SSE streaming store instruction retired.	
10H	FP_Comps_Op_Exe	00H	FP computational Instruction executed. FADD, FSUB, FCOM, FMULs, MUL, IMUL, FDIVs, DIV, IDIV, FPREMs, FSQRT are included; but exclude FADD or FMUL used in the middle of a transcendental instruction.	
11H	FP_Assist	00H	FP exceptions experienced microcode assists.	IA32_PMC1 only.
12H	Mul	00H	Multiply operations (a speculative count, including FP and integer multiplies).	IA32_PMC1 only.
13H	Div	00H	Divide operations (a speculative count, including FP and integer divisions).	IA32_PMC1 only.
14H	Cycles_Div_Busy	00H	Cycles the divider is busy.	IA32_PMC0 only.
21H	L2_ADS	00H	L2 Address strobes.	Requires core-specificity.
22H	Dbus_Busy	00H	Core cycle during which data bus was busy (increments by 4).	Requires core-specificity.
23H	Dbus_Busy_Rd	00H	Cycles data bus is busy transferring data to a core (increments by 4).	Requires core-specificity.
24H	L2_Lines_In	00H	L2 cache lines allocated.	Requires core-specificity and HW prefetch qualification.
25H	L2_M_Lines_In	00H	L2 Modified-state cache lines allocated.	Requires core-specificity.

Table 19-29. Non-Architectural Performance Events in Intel® Core™ Solo and Intel® Core™ Duo Processors (Contd.)

Event Num.	Event Mask Mnemonic	Umask Value	Description	Comment
26H	L2_Lines_Out	00H	L2 cache lines evicted.	Requires core-specificity and HW prefetch qualification.
27H	L2_M_Lines_Out	00H	L2 Modified-state cache lines evicted.	
28H	L2_IFetch	Requires MESI qualification	L2 instruction fetches from instruction fetch unit (includes speculative fetches).	Requires core-specificity.
29H	L2_LD	Requires MESI qualification	L2 cache reads.	Requires core-specificity.
2AH	L2_ST	Requires MESI qualification	L2 cache writes (includes speculation).	Requires core-specificity.
2EH	L2_Rqsts	Requires MESI qualification	L2 cache reference requests.	Requires core-specificity, HW prefetch qualification.
30H	L2_Reject_Cycles	Requires MESI qualification	Cycles L2 is busy and rejecting new requests.	
32H	L2_No_Request_ Cycles	Requires MESI qualification	Cycles there is no request to access L2.	
3AH	EST_Trans_All	00H	Any Intel Enhanced SpeedStep(R) Technology transitions.	
3AH	EST_Trans_All	10H	Intel Enhanced SpeedStep Technology frequency transitions.	
3BH	Thermal_Trip	C0H	Duration in a thermal trip based on the current core clock.	Use edge trigger to count occurrence.
3CH	NonHlt_Ref_Cycles	01H	Non-halted bus cycles.	
3CH	Serial_Execution_ Cycles	02H	Non-halted bus cycles of this core executing code while the other core is halted.	
40H	DCache_Cache_LD	Requires MESI qualification	L1 cacheable data read operations.	
41H	DCache_Cache_ST	Requires MESI qualification	L1 cacheable data write operations.	
42H	DCache_Cache_ Lock	Requires MESI qualification	L1 cacheable lock read operations to invalid state.	
43H	Data_Mem_Ref	01H	L1 data read and writes of cacheable and non-cacheable types.	
44H	Data_Mem_Cache_ Ref	02H	L1 data cacheable read and write operations.	
45H	DCache_Repl	0FH	L1 data cache line replacements.	
46H	DCache_M_Repl	00H	L1 data M-state cache line allocated.	
47H	DCache_M_Evict	00H	L1 data M-state cache line evicted.	
48H	DCache_Pend_Miss	00H	Weighted cycles of L1 miss outstanding.	Use Cmask =1 to count duration.
49H	Dtlb_Miss	00H	Data references that missed TLB.	
4BH	SSE_PrefNta_Miss	00H	PREFETCHNTA missed all caches.	
4BH	SSE_PrefT1_Miss	01H	PREFETCHT1 missed all caches.	
4BH	SSE_PrefT2_Miss	02H	PREFETCHT2 missed all caches.	
4BH	SSE_NTStores_ Miss	03H	SSE streaming store instruction missed all caches.	

Table 19-29. Non-Architectural Performance Events in Intel® Core™ Solo and Intel® Core™ Duo Processors (Contd.)

Event Num.	Event Mask Mnemonic	Umask Value	Description	Comment
4FH	L1_Pref_Req	00H	L1 prefetch requests due to DCU cache misses.	May overcount if request re-submitted.
60H	Bus_Req_ Outstanding	00; Requires core-specificity, and agent specificity	Weighted cycles of cacheable bus data read requests. This event counts full-line read request from DCU or HW prefetcher, but not RFO, write, instruction fetches, or others.	Use Cmask =1 to count duration. Use Umask bit 12 to include HWP or exclude HWP separately.
61H	Bus_BNR_Clocks	00H	External bus cycles while BNR asserted.	
62H	Bus_DRDY_Clocks	00H	External bus cycles while DRDY asserted.	Requires agent specificity.
63H	Bus_Locks_Clocks	00H	External bus cycles while bus lock signal asserted.	Requires core specificity.
64H	Bus_Data_Rcv	40H	Number of data chunks received by this processor.	
65H	Bus_Trans_Brd	See comment.	Burst read bus transactions (data or code).	Requires core specificity.
66H	Bus_Trans_RFO	See comment.	Completed read for ownership (RFO) transactions.	Requires agent specificity. Requires core specificity. Each transaction counts its address strobe. Retried transaction may be counted more than once.
68H	Bus_Trans_Ifetch	See comment.	Completed instruction fetch transactions.	
69H	Bus_Trans_Inval	See comment.	Completed invalidate transactions.	
6AH	Bus_Trans_Pwr	See comment.	Completed partial write transactions.	
6BH	Bus_Trans_P	See comment.	Completed partial transactions (include partial read + partial write + line write).	
6CH	Bus_Trans_IO	See comment.	Completed I/O transactions (read and write).	
6DH	Bus_Trans_Def	20H	Completed defer transactions.	Requires core specificity. Retried transaction may be counted more than once.
67H	Bus_Trans_WB	C0H	Completed writeback transactions from DCU (does not include L2 writebacks).	Requires agent specificity. Each transaction counts its address strobe. Retried transaction may be counted more than once.
6EH	Bus_Trans_Burst	C0H	Completed burst transactions (full line transactions include reads, write, RFO, and writebacks).	
6FH	Bus_Trans_Mem	C0H	Completed memory transactions. This includes Bus_Trans_Burst + Bus_Trans_P+Bus_Trans_Inval.	
70H	Bus_Trans_Any	C0H	Any completed bus transactions.	
77H	Bus_Snoops	00H	Counts any snoop on the bus.	Requires MESI qualification. Requires agent specificity.
78H	DCU_Snoop_To_ Share	01H	DCU snoops to share-state L1 cache line due to L1 misses.	Requires core specificity.
7DH	Bus_Not_In_Use	00H	Number of cycles there is no transaction from the core.	Requires core specificity.
7EH	Bus_Snoop_Stall	00H	Number of bus cycles while bus snoop is stalled.	

Table 19-29. Non-Architectural Performance Events in Intel® Core™ Solo and Intel® Core™ Duo Processors (Contd.)

Event Num.	Event Mask Mnemonic	Umask Value	Description	Comment
80H	ICache_Reads	00H	Number of instruction fetches from ICache, streaming buffers (both cacheable and uncacheable fetches).	
81H	ICache_Misses	00H	Number of instruction fetch misses from ICache, streaming buffers.	
85H	ITLB_Misses	00H	Number of iITLB misses.	
86H	IFU_Mem_Stall	00H	Cycles IFU is stalled while waiting for data from memory.	
87H	ILD_Stall	00H	Number of instruction length decoder stalls (Counts number of LCP stalls).	
88H	Br_Inst_Exec	00H	Branch instruction executed (includes speculation).	
89H	Br_Missp_Exec	00H	Branch instructions executed and mispredicted at execution (includes branches that do not have prediction or mispredicted).	
8AH	Br_BAC_Missp_ Exec	00H	Branch instructions executed that were mispredicted at front end.	
8BH	Br_Cnd_Exec	00H	Conditional branch instructions executed.	
8CH	Br_Cnd_Missp_ Exec	00H	Conditional branch instructions executed that were mispredicted.	
8DH	Br_Ind_Exec	00H	Indirect branch instructions executed.	
8EH	Br_Ind_Missp_Exec	00H	Indirect branch instructions executed that were mispredicted.	
8FH	Br_Ret_Exec	00H	Return branch instructions executed.	
90H	Br_Ret_Missp_Exec	00H	Return branch instructions executed that were mispredicted.	
91H	Br_Ret_BAC_Missp_ Exec	00H	Return branch instructions executed that were mispredicted at the front end.	
92H	Br_Call_Exec	00H	Return call instructions executed.	
93H	Br_Call_Missp_Exec	00H	Return call instructions executed that were mispredicted.	
94H	Br_Ind_Call_Exec	00H	Indirect call branch instructions executed.	
A2H	Resource_Stall	00H	Cycles while there is a resource related stall (renaming, buffer entries) as seen by allocator.	
B0H	MMX_Instr_Exec	00H	Number of MMX instructions executed (does not include MOVQ and MOVD stores).	
B1H	SIMD_Int_Sat_Exec	00H	Number of SIMD Integer saturating instructions executed.	
B3H	SIMD_Int_Pmul_ Exec	01H	Number of SIMD Integer packed multiply instructions executed.	
B3H	SIMD_Int_Psft_Exec	02H	Number of SIMD Integer packed shift instructions executed.	
B3H	SIMD_Int_Pck_Exec	04H	Number of SIMD Integer pack operations instruction executed.	
B3H	SIMD_Int_Upck_ Exec	08H	Number of SIMD Integer unpack instructions executed.	

Table 19-29. Non-Architectural Performance Events in Intel® Core™ Solo and Intel® Core™ Duo Processors (Contd.)

Event Num.	Event Mask Mnemonic	Umask Value	Description	Comment
B3H	SIMD_Int_Plog_ Exec	10H	Number of SIMD Integer packed logical instructions executed.	
B3H	SIMD_Int_Pari_Exec	20H	Number of SIMD Integer packed arithmetic instructions executed.	
C0H	Instr_Ret	00H	Number of instruction retired (Macro fused instruction count as 2).	
C1H	FP_Comp_Instr_Ret	00H	Number of FP compute instructions retired (X87 instruction or instruction that contains X87 operations).	Use IA32_PMC0 only.
C2H	Uops_Ret	00H	Number of micro-ops retired (include fused uops).	
C3H	SMC_Detected	00H	Number of times self-modifying code condition detected.	
C4H	Br_Instr_Ret	00H	Number of branch instructions retired.	
C5H	Br_MisPred_Ret	00H	Number of mispredicted branch instructions retired.	
C6H	Cycles_Int_Masked	00H	Cycles while interrupt is disabled.	
C7H	Cycles_Int_Pedning_ Masked	00H	Cycles while interrupt is disabled and interrupts are pending.	
C8H	HW_Int_Rx	00H	Number of hardware interrupts received.	
C9H	Br_Taken_Ret	00H	Number of taken branch instruction retired.	
CAH	Br_MisPred_Taken_ Ret	00H	Number of taken and mispredicted branch instructions retired.	
CCH	MMX_FP_Trans	00H	Number of transitions from MMX to X87.	
CCH	FP_MMX_Trans	01H	Number of transitions from X87 to MMX.	
CDH	MMX_Assist	00H	Number of EMMS executed.	
CEH	MMX_Instr_Ret	00H	Number of MMX instruction retired.	
D0H	Instr_Decoded	00H	Number of instruction decoded.	
D7H	ESP_Uops	00H	Number of ESP folding instruction decoded.	
D8H	SIMD_FP_SP_Ret	00H	Number of SSE/SSE2 single precision instructions retired (packed and scalar).	
D8H	SIMD_FP_SP_S_ Ret	01H	Number of SSE/SSE2 scalar single precision instructions retired.	
D8H	SIMD_FP_DP_P_ Ret	02H	Number of SSE/SSE2 packed double precision instructions retired.	
D8H	SIMD_FP_DP_S_ Ret	03H	Number of SSE/SSE2 scalar double precision instructions retired.	
D8H	SIMD_Int_128_Ret	04H	Number of SSE2 128 bit integer instructions retired.	
D9H	SIMD_FP_SP_P_ Comp_Ret	00H	Number of SSE/SSE2 packed single precision compute instructions retired (does not include AND, OR, XOR).	
D9H	SIMD_FP_SP_S_ Comp_Ret	01H	Number of SSE/SSE2 scalar single precision compute instructions retired (does not include AND, OR, XOR).	

Table 19-29. Non-Architectural Performance Events in Intel® Core™ Solo and Intel® Core™ Duo Processors (Contd.)

Event Num.	Event Mask Mnemonic	Umask Value	Description	Comment
D9H	SIMD_FP_DP_P_ Comp_Ret	02H	Number of SSE/SSE2 packed double precision compute instructions retired (does not include AND, OR, XOR).	
D9H	SIMD_FP_DP_S_ Comp_Ret	03H	Number of SSE/SSE2 scalar double precision compute instructions retired (does not include AND, OR, XOR).	
DAH	Fused_Uops_Ret	00H	All fused uops retired.	
DAH	Fused_Ld_Uops_ Ret	01H	Fused load uops retired.	
DAH	Fused_St_Uops_Ret	02H	Fused store uops retired.	
DBH	Unfusion	00H	Number of unfusion events in the ROB (due to exception).	
E0H	Br_Instr_Decoded	00H	Branch instructions decoded.	
E2H	BTB_Misses	00H	Number of branches the BTB did not produce a prediction.	
E4H	Br_Bogus	00H	Number of bogus branches.	
E6H	BAClears	00H	Number of BAClears asserted.	
F0H	Pref_Rqsts_Up	00H	Number of hardware prefetch requests issued in forward streams.	
F8H	Pref_Rqsts_Dn	00H	Number of hardware prefetch requests issued in backward streams.	

19.17 PENTIUM® 4 AND INTEL® XEON® PROCESSOR PERFORMANCE-MONITORING EVENTS

Tables 19-30, 19-31 and 19-32 list performance-monitoring events that can be counted or sampled on processors based on Intel NetBurst® microarchitecture. Table 19-30 lists the non-retirement events, and Table 19-31 lists the at-retirement events. Tables 19-33, 19-34, and 19-35 describes three sets of parameters that are available for three of the at-retirement counting events defined in Table 19-31. Table 19-36 shows which of the non-retirement and at retirement events are logical processor specific (TS) (see Section 18.6.4.4, "Performance Monitoring Events") and which are non-logical processor specific (TI).

Some of the Pentium 4 and Intel Xeon processor performance-monitoring events may be available only to specific models. The performance-monitoring events listed in Tables 19-30 and 19-31 apply to processors with CPUID signature that matches family encoding 15, model encoding 0, 1, 2 3, 4, or 6. Table applies to processors with a CPUID signature that matches family encoding 15, model encoding 3, 4 or 6.

The functionality of performance-monitoring events in Pentium 4 and Intel Xeon processors is also available when IA-32e mode is enabled.

Table 19-30. Performance Monitoring Events Supported by Intel NetBurst® Microarchitecture for Non-Retirement Counting

Event Name	Event Parameters	Parameter Value	Description
TC_deliver_mode			This event counts the duration (in clock cycles) of the operating modes of the trace cache and decode engine in the processor package. The mode is specified by one or more of the event mask bits.

Table 19-30. Performance Monitoring Events Supported by Intel NetBurst® Microarchitecture for Non-Retirement Counting (Contd.)

Event Name	Event Parameters	Parameter Value	Description
	ESCR restrictions	MSR_TC_ESCR0 MSR_TC_ESCR1	
	Counter numbers per ESCR	ESCR0: 4, 5 ESCR1: 6, 7	
	ESCR Event Select	01H	ESCR[31:25]
	ESCR Event Mask	Bit 0: DD 1: DB 2: DI 3: BD 4: BB 5: BI 6: ID 7: IB	ESCR[24:9] Both logical processors are in deliver mode. Logical processor 0 is in deliver mode and logical processor 1 is in build mode. Logical processor 0 is in deliver mode and logical processor 1 is either halted, under a machine clear condition or transitioning to a long microcode flow. Logical processor 0 is in build mode and logical processor 1 is in deliver mode. Both logical processors are in build mode. Logical processor 0 is in build mode and logical processor 1 is either halted, under a machine clear condition or transitioning to a long microcode flow. Logical processor 0 is either halted, under a machine clear condition or transitioning to a long microcode flow. Logical processor 1 is in deliver mode. Logical processor 0 is either halted, under a machine clear condition or transitioning to a long microcode flow. Logical processor 1 is in build mode.
	CCCR Select	01H	CCCR[15:13]
	Event Specific Notes		If only one logical processor is available from a physical processor package, the event mask should be interpreted as logical processor 1 is halted. Event mask bit 2 was previously known as "DELIVER", bit 5 was previously known as "BUILD".
BPU_fetch_ request			This event counts instruction fetch requests of specified request type by the Branch Prediction unit. Specify one or more mask bits to qualify the request type(s).
	ESCR restrictions	MSR_BPU_ESCR0 MSR_BPU_ESCR1	
	Counter numbers per ESCR	ESCR0: 0, 1 ESCR1: 2, 3	
	ESCR Event Select	03H	ESCR[31:25]

Table 19-30. Performance Monitoring Events Supported by Intel NetBurst® Microarchitecture for Non-Retirement Counting (Contd.)

Event Name	Event Parameters	Parameter Value	Description
	ESCR Event Mask	Bit 0: TCMISS	ESCR[24:9] Trace cache lookup miss
	CCCR Select	00H	CCCR[15:13]
ITLB_reference			This event counts translations using the Instruction Translation Look-aside Buffer (ITLB).
	ESCR restrictions	MSR_ITLB_ESCR0 MSR_ITLB_ESCR1	
	Counter numbers per ESCR	ESCR0: 0, 1 ESCR1: 2, 3	
	ESCR Event Select	18H	ESCR[31:25]
	ESCR Event Mask	Bit 0: HIT 1: MISS 2: HIT_UC	ESCR[24:9] ITLB hit ITLB miss Uncacheable ITLB hit
	CCCR Select	03H	CCCR[15:13]
	Event Specific Notes		All page references regardless of the page size are looked up as actual 4-KByte pages. Use the page_walk_type event with the ITMISS mask for a more conservative count.
memory_cancel			This event counts the canceling of various type of request in the Data cache Address Control unit (DAC). Specify one or more mask bits to select the type of requests that are canceled.
	ESCR restrictions	MSR_DAC_ESCR0 MSR_DAC_ESCR1	
	Counter numbers per ESCR	ESCR0: 8, 9 ESCR1: 10, 11	
	ESCR Event Select	02H	ESCR[31:25]
	ESCR Event Mask	Bit 2: ST_RB_FULL 3: 64K_CONF	ESCR[24:9] Replayed because no store request buffer is available. Conflicts due to 64-KByte aliasing.
	CCCR Select	05H	CCCR[15:13]
	Event Specific Notes		All_CACHE_MISS includes uncacheable memory in count.
memory_ complete			This event counts the completion of a load split, store split, uncacheable (UC) split, or UC load. Specify one or more mask bits to select the operations to be counted.
	ESCR restrictions	MSR_SAAT_ESCR0 MSR_SAAT_ESCR1	

Table 19-30. Performance Monitoring Events Supported by Intel NetBurst® Microarchitecture for Non-Retirement Counting (Contd.)

Event Name	Event Parameters	Parameter Value	Description
	Counter numbers per ESCR	ESCR0: 8, 9 ESCR1: 10, 11	
	ESCR Event Select	08H	ESCR[31:25]
	ESCR Event Mask	Bit 0: LSC 1: SSC	ESCR[24:9] Load split completed, excluding UC/WC loads. Any split stores completed.
	CCCR Select	02H	CCCR[15:13]
load_port_replay			This event counts replayed events at the load port. Specify one or more mask bits to select the cause of the replay.
	ESCR restrictions	MSR_SAAT_ESCR0 MSR_SAAT_ESCR1	
	Counter numbers per ESCR	ESCR0: 8, 9 ESCR1: 10, 11	
	ESCR Event Select	04H	ESCR[31:25]
	ESCR Event Mask	Bit 1: SPLIT_LD	ESCR[24:9] Split load.
	CCCR Select	02H	CCCR[15:13]
	Event Specific Notes		Must use ESCR1 for at-retirement counting.
store_port_replay			This event counts replayed events at the store port. Specify one or more mask bits to select the cause of the replay.
	ESCR restrictions	MSR_SAAT_ESCR0 MSR_SAAT_ESCR1	
	Counter numbers per ESCR	ESCR0: 8, 9 ESCR1: 10, 11	
	ESCR Event Select	05H	ESCR[31:25]
	ESCR Event Mask	Bit 1: SPLIT_ST	ESCR[24:9] Split store
	CCCR Select	02H	CCCR[15:13]
	Event Specific Notes		Must use ESCR1 for at-retirement counting.
MOB_load_replay			This event triggers if the memory order buffer (MOB) caused a load operation to be replayed. Specify one or more mask bits to select the cause of the replay.
	ESCR restrictions	MSR_MOB_ESCR0 MSR_MOB_ESCR1	
	Counter numbers per ESCR	ESCR0: 0, 1 ESCR1: 2, 3	
	ESCR Event Select	03H	ESCR[31:25]

Table 19-30. Performance Monitoring Events Supported by Intel NetBurst® Microarchitecture for Non-Retirement Counting (Contd.)

Event Name	Event Parameters	Parameter Value	Description
	ESCR Event Mask	Bit 1: NO_STA 3: NO_STD	ESCR[24:9] Replayed because of unknown store address. Replayed because of unknown store data.
		4: PARTIAL_DATA 5: UNALGN_ADDR	Replayed because of partially overlapped data access between the load and store operations. Replayed because the lower 4 bits of the linear address do not match between the load and store operations.
	CCCR Select	02H	CCCR[15:13]
page_walk_type			This event counts various types of page walks that the page miss handler (PMH) performs.
	ESCR restrictions	MSR_PMH_ESCR0 MSR_PMH_ESCR1	
	Counter numbers per ESCR	ESCR0: 0, 1 ESCR1: 2, 3	
	ESCR Event Select	01H	ESCR[31:25]
	ESCR Event Mask	Bit 0: DTMISS 1: ITMISS	ESCR[24:9] Page walk for a data TLB miss (either load or store). Page walk for an instruction TLB miss.
	CCCR Select	04H	CCCR[15:13]
BSQ_cache_reference			This event counts cache references (2nd level cache or 3rd level cache) as seen by the bus unit. Specify one or more mask bit to select an access according to the access type (read type includes both load and RFO, write type includes writebacks and evictions) and the access result (hit, misses).
	ESCR restrictions	MSR_BSU_ESCR0 MSR_BSU_ESCR1	
	Counter numbers per ESCR	ESCR0: 0, 1 ESCR1: 2, 3	
	ESCR Event Select	0CH	ESCR[31:25]

Table 19-30. Performance Monitoring Events Supported by Intel NetBurst® Microarchitecture for Non-Retirement Counting (Contd.)

Event Name	Event Parameters	Parameter Value	Description
		Bit	ESCR[24:9]
		0: RD_2ndL_HITS	Read 2nd level cache hit Shared (includes load and RFO).
		1: RD_2ndL_HITE	Read 2nd level cache hit Exclusive (includes load and RFO).
		2: RD_2ndL_HITM	Read 2nd level cache hit Modified (includes load and RFO).
		3: RD_3rdL_HITS	Read 3rd level cache hit Shared (includes load and RFO).
		4: RD_3rdL_HITE	Read 3rd level cache hit Exclusive (includes load and RFO).
		5: RD_3rdL_HITM	Read 3rd level cache hit Modified (includes load and RFO).
	ESCR Event Mask	8: RD_2ndL_MISS	Read 2nd level cache miss (includes load and RFO).
		9: RD_3rdL_MISS	Read 3rd level cache miss (includes load and RFO).
		10: WR_2ndL_MISS	A Writeback lookup from DAC misses the 2nd level cache (unlikely to happen).
	CCCR Select	07H	CCCR[15:13]
	Event Specific Notes		1: The implementation of this event in current Pentium 4 and Xeon processors treats either a load operation or a request for ownership (RFO) request as a "read" type operation. 2: Currently this event causes both over and undercounting by as much as a factor of two due to an erratum. 3: It is possible for a transaction that is started as a prefetch to change the transaction's internal status, making it no longer a prefetch. or change the access result status (hit, miss) as seen by this event.
IOQ_allocation			This event counts the various types of transactions on the bus. A count is generated each time a transaction is allocated into the IOQ that matches the specified mask bits. An allocated entry can be a sector (64 bytes) or a chunks of 8 bytes. Requests are counted once per retry. The event mask bits constitute 4 bit fields. A transaction type is specified by interpreting the values of each bit field. Specify one or more event mask bits in a bit field to select the value of the bit field. Each field (bits 0-4 are one field) are independent of and can be ORed with the others. The request type field is further combined with bit 5 and 6 to form a binary expression. Bits 7 and 8 form a bit field to specify the memory type of the target address. Bits 13 and 14 form a bit field to specify the source agent of the request. Bit 15 affects read operation only. The event is triggered by evaluating the logical expression: (((Request type) OR Bit 5 OR Bit 6) OR (Memory type)) AND (Source agent).

Table 19-30. Performance Monitoring Events Supported by Intel NetBurst® Microarchitecture for Non-Retirement Counting (Contd.)

Event Name	Event Parameters	Parameter Value	Description
	ESCR restrictions	MSR_FSB_ESCR0, MSR_FSB_ESCR1	
	Counter numbers per ESCR	ESCR0: 0, 1; ESCR1: 2, 3	
	ESCR Event Select	03H	ESCR[31:25]
	ESCR Event Mask		ESCR[24:9]
		Bits	
		0-4 (single field)	Bus request type (use 00001 for invalid or default).
		5: ALL_READ	Count read entries.
		6: ALL_WRITE	Count write entries.
		7: MEM_UC	Count UC memory access entries.
		8: MEM_WC	Count WC memory access entries.
		9: MEM_WT	Count write-through (WT) memory access entries.
		10: MEM_WP	Count write-protected (WP) memory access entries.
		11: MEM_WB	Count WB memory access entries.
		13: OWN	Count all store requests driven by processor, as opposed to other processor or DMA.
		14: OTHER	Count all requests driven by other processors or DMA.
		15: PREFETCH	Include HW and SW prefetch requests in the count.
	CCCR Select	06H	CCCR[15:13]
	Event Specific Notes		1: If PREFETCH bit is cleared, sectors fetched using prefetch are excluded in the counts. If PREFETCH bit is set, all sectors or chunks read are counted. 2: Specify the edge trigger in CCCR to avoid double counting.
			3: The mapping of interpreted bit field values to transaction types may differ with different processor model implementations of the Pentium 4 processor family. Applications that program performance monitoring events should use CPUID to determine processor models when using this event. The logic equations that trigger the event are model-specific (see 4a and 4b below). 4a:For Pentium 4 and Xeon Processors starting with CPUID Model field encoding equal to 2 or greater, this event is triggered by evaluating the logical expression ((Request type) and (Bit 5 or Bit 6) and (Memory type) and (Source agent)). 4b:For Pentium 4 and Xeon Processors with CPUID Model field encoding less than 2, this event is triggered by evaluating the logical expression [((Request type) or Bit 5 or Bit 6) or (Memory type)] and (Source agent). Note that event mask bits for memory type are ignored if either ALL_READ or ALL_WRITE is specified. 5: This event is known to ignore CPL in early implementations of Pentium 4 and Xeon Processors. Both user requests and OS requests are included in the count. This behavior is fixed starting with Pentium 4 and Xeon Processors with CPUID signature F27H (Family 15, Model 2, Stepping 7).

Table 19-30. Performance Monitoring Events Supported by Intel NetBurst® Microarchitecture for Non-Retirement Counting (Contd.)

Event Name	Event Parameters	Parameter Value	Description
			6: For write-through (WT) and write-protected (WP) memory types, this event counts reads as the number of 64-byte sectors. Writes are counted by individual chunks. 7: For uncacheable (UC) memory types, this event counts the number of 8-byte chunks allocated. 8: For Pentium 4 and Xeon Processors with CPUID Signature less than F27H, only MSR_FSB_ESCR0 is available.
IOQ_active_ entries			This event counts the number of entries (clipped at 15) in the IOQ that are active. An allocated entry can be a sector (64 bytes) or a chunks of 8 bytes. The event must be programmed in conjunction with IOQ_allocation. Specify one or more event mask bits to select the transactions that is counted.
	ESCR restrictions	MSR_FSB_ESCR1	
	Counter numbers per ESCR	ESCR1: 2, 3	
	ESCR Event Select	01AH	ESCR[30:25]
	ESCR Event Mask	Bits 0-4 (single field) 5: ALL_READ 6: ALL_WRITE 7: MEM_UC 8: MEM_WC 9: MEM_WT 10: MEM_WP 11: MEM_WB 13: OWN 14: OTHER 15: PREFETCH	ESCR[24:9] Bus request type (use 00001 for invalid or default). Count read entries. Count write entries. Count UC memory access entries. Count WC memory access entries. Count write-through (WT) memory access entries. Count write-protected (WP) memory access entries. Count WB memory access entries. Count all store requests driven by processor, as opposed to other processor or DMA. Count all requests driven by other processors or DMA. Include HW and SW prefetch requests in the count.
	CCCR Select	06H	CCCR[15:13]
	Event Specific Notes		1: Specified desired mask bits in ESCR0 and ESCR1. 2: See the ioq_allocation event for descriptions of the mask bits. 3: Edge triggering should not be used when counting cycles. 4: The mapping of interpreted bit field values to transaction types may differ across different processor model implementations of the Pentium 4 processor family. Applications that programs performance monitoring events should use the CPUID instruction to detect processor models when using this event. The logical expression that triggers this event as describe below: 5a:For Pentium 4 and Xeon Processors starting with CPUID MODEL field encoding equal to 2 or greater, this event is triggered by evaluating the logical expression ((Request type) and (Bit 5 or Bit 6) and (Memory type) and (Source agent)).

Table 19-30. Performance Monitoring Events Supported by Intel NetBurst® Microarchitecture for Non-Retirement Counting (Contd.)

Event Name	Event Parameters	Parameter Value	Description
			5b:For Pentium 4 and Xeon Processors starting with CPUID MODEL field encoding less than 2, this event is triggered by evaluating the logical expression [((Request type) or Bit 5 or Bit 6) or (Memory type)] and (Source agent). Event mask bits for memory type are ignored if either ALL_READ or ALL_WRITE is specified. 5c: This event is known to ignore CPL in the current implementations of Pentium 4 and Xeon Processors Both user requests and OS requests are included in the count.
			6: An allocated entry can be a full line (64 bytes) or in individual chunks of 8 bytes.
FSB_data_ activity			This event increments once for each DRDY or DBSY event that occurs on the front side bus. The event allows selection of a specific DRDY or DBSY event.
	ESCR restrictions	MSR_FSB_ESCR0 MSR_FSB_ESCR1	
	Counter numbers per ESCR	ESCR0: 0, 1 ESCR1: 2, 3	
	ESCR Event Select	17H	ESCR[31:25]
	ESCR Event Mask		ESCR[24:9]
		Bit 0: DRDY_DRV	Count when this processor drives data onto the bus - includes writes and implicit writebacks. Asserted two processor clock cycles for partial writes and 4 processor clocks (usually in consecutive bus clocks) for full line writes.
		1: DRDY_OWN	Count when this processor reads data from the bus - includes loads and some PIC transactions. Asserted two processor clock cycles for partial reads and 4 processor clocks (usually in consecutive bus clocks) for full line reads. Count DRDY events that we drive. Count DRDY events sampled that we own.
		2: DRDY_OTHER	Count when data is on the bus but not being sampled by the processor. It may or may not be being driven by this processor. Asserted two processor clock cycles for partial transactions and 4 processor clocks (usually in consecutive bus clocks) for full line transactions.
		3: DBSY_DRV	Count when this processor reserves the bus for use in the next bus cycle in order to drive data. Asserted for two processor clock cycles for full line writes and not at all for partial line writes. May be asserted multiple times (in consecutive bus clocks) if we stall the bus waiting for a cache lock to complete.
		4: DBSY_OWN	Count when some agent reserves the bus for use in the next bus cycle to drive data that this processor will sample. Asserted for two processor clock cycles for full line writes and not at all for partial line writes. May be asserted multiple times (all one bus clock apart) if we stall the bus for some reason.

Table 19-30. Performance Monitoring Events Supported by Intel NetBurst® Microarchitecture for Non-Retirement Counting (Contd.)

Event Name	Event Parameters	Parameter Value	Description
		5:DBSY_OTHER	Count when some agent reserves the bus for use in the next bus cycle to drive data that this processor will NOT sample. It may or may not be being driven by this processor. Asserted two processor clock cycles for partial transactions and 4 processor clocks (usually in consecutive bus clocks) for full line transactions.
	CCCR Select	06H	CCCR[15:13]
	Event Specific Notes		Specify edge trigger in the CCCR MSR to avoid double counting. DRDY_OWN and DRDY_OTHER are mutually exclusive; similarly for DBSY_OWN and DBSY_OTHER.
BSQ_allocation			This event counts allocations in the Bus Sequence Unit (BSQ) according to the specified mask bit encoding. The event mask bits consist of four sub-groups: ▪ Request type. ▪ Request length. ▪ Memory type. ▪ Sub-group consisting mostly of independent bits (bits 5, 6, 7, 8, 9, and 10). Specify an encoding for each sub-group.
	ESCR restrictions	MSR_BSU_ESCR0	
	Counter numbers per ESCR	ESCR0: 0, 1	
	ESCR Event Select	05H	ESCR[31:25]
	ESCR Event Mask	Bit	ESCR[24:9]
		0: REQ_TYPE0 1: REQ_TYPE1	Request type encoding (bit 0 and 1) are: 0 – Read (excludes read invalidate). 1 – Read invalidate. 2 – Write (other than writebacks). 3 – Writeback (evicted from cache). (public)
		2: REQ_LEN0 3: REQ_LEN1	Request length encoding (bit 2, 3) are: 0 – 0 chunks 1 – 1 chunks 3 – 8 chunks
		5: REQ_IO_TYPE	Request type is input or output.
		6: REQ_LOCK_ TYPE	Request type is bus lock.
		7: REQ_CACHE_ TYPE	Request type is cacheable.
		8: REQ_SPLIT_ TYPE	Request type is a bus 8-byte chunk split across 8-byte boundary.
		9: REQ_DEM_TYPE	Request type is a demand if set. Request type is HW.SW prefetch if 0.
		10: REQ_ORD_ TYPE	Request is an ordered type.

Table 19-30. Performance Monitoring Events Supported by Intel NetBurst® Microarchitecture for Non-Retirement Counting (Contd.)

Event Name	Event Parameters	Parameter Value	Description
		11: MEM_TYPE0 12: MEM_TYPE1 13: MEM_TYPE2	Memory type encodings (bit 11-13) are: 0 – UC 1 – WC 4 – WT 5 – WP 6 – WB
	CCCR Select	07H	CCCR[15:13]
	Event Specific Notes		1: Specify edge trigger in CCCR to avoid double counting. 2: A writebacks to 3rd level cache from 2nd level cache counts as a separate entry, this is in additional to the entry allocated for a request to the bus. 3: A read request to WB memory type results in a request to the 64-byte sector, containing the target address, followed by a prefetch request to an adjacent sector. 4: For Pentium 4 and Xeon processors with CPUID model encoding value equals to 0 and 1, an allocated BSQ entry includes both the demand sector and prefetched 2nd sector. 5: An allocated BSQ entry for a data chunk is any request less than 64 bytes. 6a: This event may undercount for requests of split type transactions if the data address straddled across modulo-64 byte boundary. 6b: This event may undercount for requests of read request of 16-byte operands from WC or UC address. 6c: This event may undercount WC partial requests originated from store operands that are dwords.
bsq_active_ entries			This event represents the number of BSQ entries (clipped at 15) currently active (valid) which meet the subevent mask criteria during allocation in the BSQ. Active request entries are allocated on the BSQ until de-allocated. De-allocation of an entry does not necessarily imply the request is filled. This event must be programmed in conjunction with BSQ_allocation. Specify one or more event mask bits to select the transactions that is counted.
	ESCR restrictions	ESCR1	
	Counter numbers per ESCR	ESCR1: 2, 3	
	ESCR Event Select	06H	ESCR[30:25]
	ESCR Event Mask		ESCR[24:9]
	CCCR Select	07H	CCCR[15:13]
	Event Specific Notes		1: Specified desired mask bits in ESCR0 and ESCR1. 2: See the BSQ_allocation event for descriptions of the mask bits. 3: Edge triggering should not be used when counting cycles.
			4: This event can be used to estimate the latency of a transaction from allocation to de-allocation in the BSQ. The latency observed by BSQ_allocation includes the latency of FSB, plus additional overhead.

Table 19-30. Performance Monitoring Events Supported by Intel NetBurst® Microarchitecture for Non-Retirement Counting (Contd.)

Event Name	Event Parameters	Parameter Value	Description
			5: Additional overhead may include the time it takes to issue two requests (the sector by demand and the adjacent sector via prefetch). Since adjacent sector prefetches have lower priority that demand fetches, on a heavily used system there is a high probability that the adjacent sector prefetch will have to wait until the next bus arbitration. 6: For Pentium 4 and Xeon processors with CPUID model encoding value less than 3, this event is updated every clock. 7: For Pentium 4 and Xeon processors with CPUID model encoding value equals to 3 or 4, this event is updated every other clock.
SSE_input_assist			This event counts the number of times an assist is requested to handle problems with input operands for SSE/SSE2/SSE3 operations; most notably denormal source operands when the DAZ bit is not set. Set bit 15 of the event mask to use this event.
	ESCR restrictions	MSR_FIRM_ESCR0 MSR_FIRM_ESCR1	
	Counter numbers per ESCR	ESCR0: 8, 9 ESCR1: 10, 11	
	ESCR Event Select	34H	ESCR[31:25]
	ESCR Event Mask	15: ALL	ESCR[24:9] Count assists for SSE/SSE2/SSE3 μops.
	CCCR Select	01H	CCCR[15:13]
	Event Specific Notes		1: Not all requests for assists are actually taken. This event is known to overcount in that it counts requests for assists from instructions on the non-retired path that do not incur a performance penalty. An assist is actually taken only for non-bogus μops. Any appreciable counts for this event are an indication that the DAZ or FTZ bit should be set and/or the source code should be changed to eliminate the condition. 2: Two common situations for an SSE/SSE2/SSE3 operation needing an assist are: (1) when a denormal constant is used as an input and the Denormals-Are-Zero (DAZ) mode is not set, (2) when the input operand uses the underflowed result of a previous SSE/SSE2/SSE3 operation and neither the DAZ nor Flush-To-Zero (FTZ) modes are set. 3: Enabling the DAZ mode prevents SSE/SSE2/SSE3 operations from needing assists in the first situation. Enabling the FTZ mode prevents SSE/SSE2/SSE3 operations from needing assists in the second situation.
packed_SP_uop			This event increments for each packed single-precision μop, specified through the event mask for detection.
	ESCR restrictions	MSR_FIRM_ESCR0 MSR_FIRM_ESCR1	
	Counter numbers per ESCR	ESCR0: 8, 9 ESCR1: 10, 11	
	ESCR Event Select	08H	ESCR[31:25]

Table 19-30. Performance Monitoring Events Supported by Intel NetBurst® Microarchitecture for Non-Retirement Counting (Contd.)

Event Name	Event Parameters	Parameter Value	Description
	ESCR Event Mask	Bit 15: ALL	ESCR[24:9] Count all μops operating on packed single-precision operands.
	CCCR Select	01H	CCCR[15:13]
	Event Specific Notes		1: If an instruction contains more than one packed SP μops, each packed SP μop that is specified by the event mask will be counted. 2: This metric counts instances of packed memory μops in a repeat move string.
packed_DP_uop			This event increments for each packed double-precision μop, specified through the event mask for detection.
	ESCR restrictions	MSR_FIRM_ESCR0 MSR_FIRM_ESCR1	
	Counter numbers per ESCR	ESCR0: 8, 9 ESCR1: 10, 11	
	ESCR Event Select	0CH	ESCR[31:25]
	ESCR Event Mask	Bit 15: ALL	ESCR[24:9] Count all μops operating on packed double-precision operands.
	CCCR Select	01H	CCCR[15:13]
	Event Specific Notes		If an instruction contains more than one packed DP μops, each packed DP μop that is specified by the event mask will be counted.
scalar_SP_uop			This event increments for each scalar single-precision μop, specified through the event mask for detection.
	ESCR restrictions	MSR_FIRM_ESCR0 MSR_FIRM_ESCR1	
	Counter numbers per ESCR	ESCR0: 8, 9 ESCR1: 10, 11	
	ESCR Event Select	0AH	ESCR[31:25]
	ESCR Event Mask	Bit 15: ALL	ESCR[24:9] Count all μops operating on scalar single-precision operands.
	CCCR Select	01H	CCCR[15:13]
	Event Specific Notes		If an instruction contains more than one scalar SP μops, each scalar SP μop that is specified by the event mask will be counted.
scalar_DP_uop			This event increments for each scalar double-precision μop, specified through the event mask for detection.
	ESCR restrictions	MSR_FIRM_ESCR0 MSR_FIRM_ESCR1	
	Counter numbers per ESCR	ESCR0: 8, 9 ESCR1: 10, 11	
	ESCR Event Select	0EH	ESCR[31:25]
	ESCR Event Mask	Bit 15: ALL	ESCR[24:9] Count all μops operating on scalar double-precision operands.
	CCCR Select	01H	CCCR[15:13]

Table 19-30. Performance Monitoring Events Supported by Intel NetBurst® Microarchitecture for Non-Retirement Counting (Contd.)

Event Name	Event Parameters	Parameter Value	Description
	Event Specific Notes		If an instruction contains more than one scalar DP μops, each scalar DP μop that is specified by the event mask is counted.
64bit_MMX_uop			This event increments for each MMX instruction, which operate on 64-bit SIMD operands.
	ESCR restrictions	MSR_FIRM_ESCR0 MSR_FIRM_ESCR1	
	Counter numbers per ESCR	ESCR0: 8, 9 ESCR1: 10, 11	
	ESCR Event Select	02H	ESCR[31:25]
	ESCR Event Mask	Bit 15: ALL	ESCR[24:9] Count all μops operating on 64- bit SIMD integer operands in memory or MMX registers.
	CCCR Select	01H	CCCR[15:13]
	Event Specific Notes		If an instruction contains more than one 64-bit MMX μops, each 64-bit MMX μop that is specified by the event mask will be counted.
128bit_MMX_uop			This event increments for each integer SIMD SSE2 instruction, which operate on 128-bit SIMD operands.
	ESCR restrictions	MSR_FIRM_ESCR0 MSR_FIRM_ESCR1	
	Counter numbers per ESCR	ESCR0: 8, 9 ESCR1: 10, 11	
	ESCR Event Select	1AH	ESCR[31:25]
	ESCR Event Mask	Bit 15: ALL	ESCR[24:9] Count all μops operating on 128-bit SIMD integer operands in memory or XMM registers.
	CCCR Select	01H	CCCR[15:13]
	Event Specific Notes		If an instruction contains more than one 128-bit MMX μops, each 128-bit MMX μop that is specified by the event mask will be counted.
x87_FP_uop			This event increments for each x87 floating-point μop, specified through the event mask for detection.
	ESCR restrictions	MSR_FIRM_ESCR0 MSR_FIRM_ESCR1	
	Counter numbers per ESCR	ESCR0: 8, 9 ESCR1: 10, 11	
	ESCR Event Select	04H	ESCR[31:25]
	ESCR Event Mask	Bit 15: ALL	ESCR[24:9] Count all x87 FP μops.
	CCCR Select	01H	CCCR[15:13]

Table 19-30. Performance Monitoring Events Supported by Intel NetBurst® Microarchitecture for Non-Retirement Counting (Contd.)

Event Name	Event Parameters	Parameter Value	Description
	Event Specific Notes		1: If an instruction contains more than one x87 FP μops, each x87 FP μop that is specified by the event mask will be counted. 2: This event does not count x87 FP μop for load, store, move between registers.
TC_misc			This event counts miscellaneous events detected by the TC. The counter will count twice for each occurrence.
	ESCR restrictions	MSR_TC_ESCR0 MSR_TC_ESCR1	
	Counter numbers per ESCR	ESCR0: 4, 5 ESCR1: 6, 7	
	ESCR Event Select	06H	ESCR[31:25]
	CCCR Select	01H	CCCR[15:13]
	ESCR Event Mask	Bit 4: FLUSH	ESCR[24:9] Number of flushes
global_power _events			This event accumulates the time during which a processor is not stopped.
	ESCR restrictions	MSR_FSB_ESCR0 MSR_FSB_ESCR1	
	Counter numbers per ESCR	ESCR0: 0, 1 ESCR1: 2, 3	
	ESCR Event Select	013H	ESCR[31:25]
	ESCR Event Mask	Bit 0: Running	ESCR[24:9] The processor is active (includes the handling of HLT STPCLK and throttling.
	CCCR Select	06H	CCCR[15:13]
tc_ms_xfer			This event counts the number of times that uop delivery changed from TC to MS ROM.
	ESCR restrictions	MSR_MS_ESCR0 MSR_MS_ESCR1	
	Counter numbers per ESCR	ESCR0: 4, 5 ESCR1: 6, 7	
	ESCR Event Select	05H	ESCR[31:25]
	ESCR Event Mask	Bit 0: CISC	ESCR[24:9] A TC to MS transfer occurred.
	CCCR Select	0H	CCCR[15:13]
uop_queue_ writes			This event counts the number of valid uops written to the uop queue. Specify one or more mask bits to select the source type of writes.
	ESCR restrictions	MSR_MS_ESCR0 MSR_MS_ESCR1	
	Counter numbers per ESCR	ESCR0: 4, 5 ESCR1: 6, 7	

Table 19-30. Performance Monitoring Events Supported by Intel NetBurst® Microarchitecture for Non-Retirement Counting (Contd.)

Event Name	Event Parameters	Parameter Value	Description
	ESCR Event Select	09H	ESCR[31:25]
	ESCR Event Mask	Bit 0: FROM_TC_ BUILD 1: FROM_TC_ DELIVER 2: FROM_ROM	ESCR[24:9] The uops being written are from TC build mode. The uops being written are from TC deliver mode. The uops being written are from microcode ROM.
	CCCR Select	0H	CCCR[15:13]
retired_mispred _branch_type			This event counts retiring mispredicted branches by type.
	ESCR restrictions	MSR_TBPU_ESCR0 MSR_TBPU_ESCR1	
	Counter numbers per ESCR	ESCR0: 4, 5 ESCR1: 6, 7	
	ESCR Event Select	05H	ESCR[30:25]
	ESCR Event Mask	Bit 1: CONDITIONAL 2: CALL	ESCR[24:9] Conditional jumps. Indirect call branches.
		3: RETURN 4: INDIRECT	Return branches. Returns, indirect calls, or indirect jumps.
	CCCR Select	02H	CCCR[15:13]
	Event Specific Notes		This event may overcount conditional branches if: ▪ Mispredictions cause the trace cache and delivery engine to build new traces. ▪ When the processor's pipeline is being cleared.
retired_branch _type			This event counts retiring branches by type. Specify one or more mask bits to qualify the branch by its type.
	ESCR restrictions	MSR_TBPU_ESCR0 MSR_TBPU_ESCR1	
	Counter numbers per ESCR	ESCR0: 4, 5 ESCR1: 6, 7	
	ESCR Event Select	04H	ESCR[30:25]
	ESCR Event Mask	Bit 1: CONDITIONAL 2: CALL 3: RETURN 4: INDIRECT	ESCR[24:9] Conditional jumps. Direct or indirect calls. Return branches. Returns, indirect calls, or indirect jumps.
	CCCR Select	02H	CCCR[15:13]

Table 19-30. Performance Monitoring Events Supported by Intel NetBurst® Microarchitecture for Non-Retirement Counting (Contd.)

Event Name	Event Parameters	Parameter Value	Description
	Event Specific Notes		This event may overcount conditional branches if : ▪ Mispredictions cause the trace cache and delivery engine to build new traces. ▪ When the processor's pipeline is being cleared.
resource_stall			This event monitors the occurrence or latency of stalls in the Allocator.
	ESCR restrictions	MSR_ALF_ESCR0 MSR_ALF_ESCR1	
	Counter numbers per ESCR	ESCR0: 12, 13, 16 ESCR1: 14, 15, 17	
	ESCR Event Select	01H	ESCR[30:25]
	Event Masks	Bit 5: SBFULL	ESCR[24:9] A Stall due to lack of store buffers.
	CCCR Select	01H	CCCR[15:13]
	Event Specific Notes		This event may not be supported in all models of the processor family.
WC_Buffer			This event counts Write Combining Buffer operations that are selected by the event mask.
	ESCR restrictions	MSR_DAC_ESCR0 MSR_DAC_ESCR1	
	Counter numbers per ESCR	ESCR0: 8, 9 ESCR1: 10, 11	
	ESCR Event Select	05H	ESCR[30:25]
	Event Masks	Bit 0: WCB_EVICTS	ESCR[24:9] WC Buffer evictions of all causes.
		1: WCB_FULL_ EVICT	WC Buffer eviction: no WC buffer is available.
	CCCR Select	05H	CCCR[15:13]
	Event Specific Notes		This event is useful for detecting the subset of 64K aliasing cases that are more costly (i.e. 64K aliasing cases involving stores) as long as there are no significant contributions due to write combining buffer full or hit-modified conditions.
b2b_cycles			This event can be configured to count the number back-to-back bus cycles using sub-event mask bits 1 through 6.
	ESCR restrictions	MSR_FSB_ESCR0 MSR_FSB_ESCR1	
	Counter numbers per ESCR	ESCR0: 0, 1 ESCR1: 2, 3	
	ESCR Event Select	016H	ESCR[30:25]
	Event Masks	Bit	ESCR[24:9]

Table 19-30. Performance Monitoring Events Supported by Intel NetBurst® Microarchitecture for Non-Retirement Counting (Contd.)

Event Name	Event Parameters	Parameter Value	Description
	CCCR Select	03H	CCCR[15:13]
	Event Specific Notes		This event may not be supported in all models of the processor family.
bnr			This event can be configured to count bus not ready conditions using sub-event mask bits 0 through 2.
	ESCR restrictions	MSR_FSB_ESCR0 MSR_FSB_ESCR1	
	Counter numbers per ESCR	ESCR0: 0, 1 ESCR1: 2, 3	
	ESCR Event Select	08H	ESCR[30:25]
	Event Masks	Bit	ESCR[24:9]
	CCCR Select	03H	CCCR[15:13]
	Event Specific Notes		This event may not be supported in all models of the processor family.
snoop			This event can be configured to count snoop hit modified bus traffic using sub-event mask bits 2, 6 and 7.
	ESCR restrictions	MSR_FSB_ESCR0 MSR_FSB_ESCR1	
	Counter numbers per ESCR	ESCR0: 0, 1 ESCR1: 2, 3	
	ESCR Event Select	06H	ESCR[30:25]
	Event Masks	Bit	ESCR[24:9]
	CCCR Select	03H	CCCR[15:13]
	Event Specific Notes		This event may not be supported in all models of the processor family.
Response			This event can be configured to count different types of responses using sub-event mask bits 1,2, 8, and 9.
	ESCR restrictions	MSR_FSB_ESCR0 MSR_FSB_ESCR1	
	Counter numbers per ESCR	ESCR0: 0, 1 ESCR1: 2, 3	
	ESCR Event Select	04H	ESCR[30:25]
	Event Masks	Bit	ESCR[24:9]
	CCCR Select	03H	CCCR[15:13]
	Event Specific Notes		This event may not be supported in all models of the processor family.

Table 19-31. Performance Monitoring Events For Intel NetBurst® Microarchitecture for At-Retirement Counting

Event Name	Event Parameters	Parameter Value	Description
front_end_event			This event counts the retirement of tagged μops, which are specified through the front-end tagging mechanism. The event mask specifies bogus or non-bogus μops.
	ESCR restrictions	MSR_CRU_ESCR2 MSR_CRU_ESCR3	
	Counter numbers per ESCR	ESCR2: 12, 13, 16 ESCR3: 14, 15, 17	
	ESCR Event Select	08H	ESCR[31:25]
	ESCR Event Mask	 Bit 0: NBOGUS 1: BOGUS	ESCR[24:9] The marked μops are not bogus. The marked μops are bogus.
	CCCR Select	05H	CCCR[15:13]
	Can Support PEBS	Yes	
	Require Additional MSRs for tagging	Selected ESCRs and/or MSR_TC_ PRECISE_EVENT	See list of metrics supported by Front_end tagging in Table A-3
execution_event			This event counts the retirement of tagged μops, which are specified through the execution tagging mechanism. The event mask allows from one to four types of μops to be specified as either bogus or non-bogus μops to be tagged.
	ESCR restrictions	MSR_CRU_ESCR2 MSR_CRU_ESCR3	
	Counter numbers per ESCR	ESCR2: 12, 13, 16 ESCR3: 14, 15, 17	
	ESCR Event Select	0CH	ESCR[31:25]
	ESCR Event Mask	 Bit 0: NBOGUS0 1: NBOGUS1 2: NBOGUS2 3: NBOGUS3 4: BOGUS0 5: BOGUS1 6: BOGUS2 7: BOGUS3	ESCR[24:9] The marked μops are not bogus. The marked μops are not bogus. The marked μops are not bogus. The marked μops are not bogus. The marked μops are bogus. The marked μops are bogus. The marked μops are bogus. The marked μops are bogus.
	CCCR Select	05H	CCCR[15:13]
	Event Specific Notes		Each of the 4 slots to specify the bogus/non-bogus μops must be coordinated with the 4 TagValue bits in the ESCR (for example, NBOGUS0 must accompany a '1' in the lowest bit of the TagValue field in ESCR, NBOGUS1 must accompany a '1' in the next but lowest bit of the TagValue field).
	Can Support PEBS	Yes	

Table 19-31. Performance Monitoring Events For Intel NetBurst® Microarchitecture for At-Retirement Counting (Contd.)

Event Name	Event Parameters	Parameter Value	Description
	Require Additional MSRs for tagging	An ESCR for an upstream event	See list of metrics supported by execution tagging in Table A-4.
replay_event			This event counts the retirement of tagged μops, which are specified through the replay tagging mechanism. The event mask specifies bogus or non-bogus μops.
	ESCR restrictions	MSR_CRU_ESCR2 MSR_CRU_ESCR3	
	Counter numbers per ESCR	ESCR2: 12, 13, 16 ESCR3: 14, 15, 17	
	ESCR Event Select	09H	ESCR[31:25]
	ESCR Event Mask	 Bit 0: NBOGUS 1: BOGUS	ESCR[24:9] The marked μops are not bogus. The marked μops are bogus.
	CCCR Select	05H	CCCR[15:13]
	Event Specific Notes		Supports counting tagged μops with additional MSRs.
	Can Support PEBS	Yes	
	Require Additional MSRs for tagging	IA32_PEBS_ ENABLE MSR_PEBS_ MATRIX_VERT Selected ESCR	See list of metrics supported by replay tagging in Table A-5.
instr_retired			This event counts instructions that are retired during a clock cycle. Mask bits specify bogus or non-bogus (and whether they are tagged using the front-end tagging mechanism).
	ESCR restrictions	MSR_CRU_ESCR0 MSR_CRU_ESCR1	
	Counter numbers per ESCR	ESCR0: 12, 13, 16 ESCR1: 14, 15, 17	
	ESCR Event Select	02H	ESCR[31:25]
	ESCR Event Mask	 Bit 0: NBOGUSNTAG 1: NBOGUSTAG 2: BOGUSNTAG 3: BOGUSTAG	ESCR[24:9] Non-bogus instructions that are not tagged. Non-bogus instructions that are tagged. Bogus instructions that are not tagged. Bogus instructions that are tagged.
	CCCR Select	04H	CCCR[15:13]
	Event Specific Notes		1: The event count may vary depending on the microarchitectural states of the processor when the event detection is enabled. 2: The event may count more than once for some instructions with complex uop flows and were interrupted before retirement.

Table 19-31. Performance Monitoring Events For Intel NetBurst® Microarchitecture for At-Retirement Counting (Contd.)

Event Name	Event Parameters	Parameter Value	Description
	Can Support PEBS	No	
uops_retired			This event counts μops that are retired during a clock cycle. Mask bits specify bogus or non-bogus.
	ESCR restrictions	MSR_CRU_ESCR0 MSR_CRU_ESCR1	
	Counter numbers per ESCR	ESCR0: 12, 13, 16 ESCR1: 14, 15, 17	
	ESCR Event Select	01H	ESCR[31:25]
	ESCR Event Mask	 Bit 0: NBOGUS 1: BOGUS	ESCR[24:9] The marked μops are not bogus. The marked μops are bogus.
	CCCR Select	04H	CCCR[15:13]
	Event Specific Notes		P6: EMON_UOPS_RETIRED
	Can Support PEBS	No	
uop_type			This event is used in conjunction with the front-end at-retirement mechanism to tag load and store μops.
	ESCR restrictions	MSR_RAT_ESCR0 MSR_RAT_ESCR1	
	Counter numbers per ESCR	ESCR0: 12, 13, 16 ESCR1: 14, 15, 17	
	ESCR Event Select	02H	ESCR[31:25]
	ESCR Event Mask	 Bit 1: TAGLOADS 2: TAGSTORES	ESCR[24:9] The μop is a load operation. The μop is a store operation.
	CCCR Select	02H	CCCR[15:13]
	Event Specific Notes		Setting the TAGLOADS and TAGSTORES mask bits does not cause a counter to increment. They are only used to tag uops.
	Can Support PEBS	No	
branch_retired			This event counts the retirement of a branch. Specify one or more mask bits to select any combination of taken, not-taken, predicted and mispredicted.
	ESCR restrictions	MSR_CRU_ESCR2 MSR_CRU_ESCR3	See Table 18-70 for the addresses of the ESCR MSRs
	Counter numbers per ESCR	ESCR2: 12, 13, 16 ESCR3: 14, 15, 17	The counter numbers associated with each ESCR are provided. The performance counters and corresponding CCCRs can be obtained from Table 18-70.
	ESCR Event Select	06H	ESCR[31:25]

Table 19-31. Performance Monitoring Events For Intel NetBurst® Microarchitecture for At-Retirement Counting (Contd.)

Event Name	Event Parameters	Parameter Value	Description
	ESCR Event Mask	Bit 0: MMNP 1: MMNM 2: MMTP 3: MMTM	ESCR[24:9] Branch not-taken predicted Branch not-taken mispredicted Branch taken predicted Branch taken mispredicted
	CCCR Select	05H	CCCR[15:13]
	Event Specific Notes		P6: EMON_BR_INST_RETIRED
	Can Support PEBS	No	
mispred_branch_ retired			This event represents the retirement of mispredicted branch instructions.
	ESCR restrictions	MSR_CRU_ESCR0 MSR_CRU_ESCR1	
	Counter numbers per ESCR	ESCR0: 12, 13, 16 ESCR1: 14, 15, 17	
	ESCR Event Select	03H	ESCR[31:25]
	ESCR Event Mask	Bit 0: NBOGUS	ESCR[24:9] The retired instruction is not bogus.
	CCCR Select	04H	CCCR[15:13]
	Can Support PEBS	No	
x87_assist			This event counts the retirement of x87 instructions that required special handling. Specifies one or more event mask bits to select the type of assistance.
	ESCR restrictions	MSR_CRU_ESCR2 MSR_CRU_ESCR3	
	Counter numbers per ESCR	ESCR2: 12, 13, 16 ESCR3: 14, 15, 17	
	ESCR Event Select	03H	ESCR[31:25]
	ESCR Event Mask	Bit 0: FPSU 1: FPSO 2: POAO 3: POAU 4: PREA	ESCR[24:9] Handle FP stack underflow. Handle FP stack overflow. Handle x87 output overflow. Handle x87 output underflow. Handle x87 input assist.
	CCCR Select	05H	CCCR[15:13]
	Can Support PEBS	No	

Table 19-31. Performance Monitoring Events For Intel NetBurst® Microarchitecture for At-Retirement Counting (Contd.)

Event Name	Event Parameters	Parameter Value	Description
machine_clear			This event increments according to the mask bit specified while the entire pipeline of the machine is cleared. Specify one of the mask bit to select the cause.
	ESCR restrictions	MSR_CRU_ESCR2 MSR_CRU_ESCR3	
	Counter numbers per ESCR	ESCR2: 12, 13, 16 ESCR3: 14, 15, 17	
	ESCR Event Select	02H	ESCR[31:25]
	ESCR Event Mask	Bit 0: CLEAR 2: MOCLEAR 6: SMCLEAR	ESCR[24:9] Counts for a portion of the many cycles while the machine is cleared for any cause. Use Edge triggering for this bit only to get a count of occurrence versus a duration. Increments each time the machine is cleared due to memory ordering issues. Increments each time the machine is cleared due to self-modifying code issues.
	CCCR Select	05H	CCCR[15:13]
	Can Support PEBS	No	

Table 19-32. Intel NetBurst® Microarchitecture Model-Specific Performance Monitoring Events (For Model Encoding 3, 4 or 6)

Event Name	Event Parameters	Parameter Value	Description
instr_completed			This event counts instructions that have completed and retired during a clock cycle. Mask bits specify whether the instruction is bogus or non-bogus and whether they are:
	ESCR restrictions	MSR_CRU_ESCR0 MSR_CRU_ESCR1	
	Counter numbers per ESCR	ESCR0: 12, 13, 16 ESCR1: 14, 15, 17	
	ESCR Event Select	07H	ESCR[31:25]
	ESCR Event Mask	Bit 0: NBOGUS 1: BOGUS	ESCR[24:9] Non-bogus instructions Bogus instructions
	CCCR Select	04H	CCCR[15:13]
	Event Specific Notes		This metric differs from instr_retired, since it counts instructions completed, rather than the number of times that instructions started.
	Can Support PEBS	No	

Table 19-33. List of Metrics Available for Front_end Tagging (For Front_end Event Only)

Front-end metric[1]	MSR_ TC_PRECISE_EVENT MSR Bit field	Additional MSR	Event mask value for Front_end_event
memory_loads	None	Set TAGLOADS bit in ESCR corresponding to event Uop_Type.	NBOGUS
memory_stores	None	Set TAGSTORES bit in the ESCR corresponding to event Uop_Type.	NBOGUS

NOTES:

1. There may be some undercounting of front end events when there is an overflow or underflow of the floating point stack.

Table 19-34. List of Metrics Available for Execution Tagging (For Execution Event Only)

Execution metric	Upstream ESCR	TagValue in Upstream ESCR	Event mask value for execution_event
packed_SP_retired	Set ALL bit in event mask, TagUop bit in ESCR of packed_SP_uop.	1	NBOGUS0
packed_DP_retired	Set ALL bit in event mask, TagUop bit in ESCR of packed_DP_uop.	1	NBOGUS0
scalar_SP_retired	Set ALL bit in event mask, TagUop bit in ESCR of scalar_SP_uop.	1	NBOGUS0
scalar_DP_retired	Set ALL bit in event mask, TagUop bit in ESCR of scalar_DP_uop.	1	NBOGUS0
128_bit_MMX_retired	Set ALL bit in event mask, TagUop bit in ESCR of 128_bit_MMX_uop.	1	NBOGUS0
64_bit_MMX_retired	Set ALL bit in event mask, TagUop bit in ESCR of 64_bit_MMX_uop.	1	NBOGUS0
X87_FP_retired	Set ALL bit in event mask, TagUop bit in ESCR of x87_FP_uop.	1	NBOGUS0
X87_SIMD_memory_moves_retired	Set ALLP0, ALLP2 bits in event mask, TagUop bit in ESCR of X87_SIMD_ moves_uop.	1	NBOGUS0

Table 19-35. List of Metrics Available for Replay Tagging (For Replay Event Only)

Replay metric[1]	IA32_PEBS_ ENABLE Field to Set	MSR_PEBS_ MATRIX_VERT Bit Field to Set	Additional MSR/ Event	Event Mask Value for Replay_event
1stL_cache_load _miss_retired	Bit 0, Bit 24, Bit 25	Bit 0	None	NBOGUS
2ndL_cache_load _miss_retired[2]	Bit 1, Bit 24, Bit 25	Bit 0	None	NBOGUS
DTLB_load_miss _retired	Bit 2, Bit 24, Bit 25	Bit 0	None	NBOGUS
DTLB_store_miss _retired	Bit 2, Bit 24, Bit 25	Bit 1	None	NBOGUS
DTLB_all_miss _retired	Bit 2, Bit 24, Bit 25	Bit 0, Bit 1	None	NBOGUS
Tagged_mispred_ branch	Bit 15, Bit 16, Bit 24, Bit 25	Bit 4	None	NBOGUS

Table 19-35. List of Metrics Available for Replay Tagging (For Replay Event Only) (Contd.)

Replay metric[1]	IA32_PEBS_ ENABLE Field to Set	MSR_PEBS_ MATRIX_VERT Bit Field to Set	Additional MSR/ Event	Event Mask Value for Replay_event
MOB_load _replay_retired[3]	Bit 9, Bit 24, Bit 25	Bit 0	Select MOB_load_replay event and set PARTIAL_DATA and UNALGN_ADDR bit.	NBOGUS
split_load_retired	Bit 10, Bit 24, Bit 25	Bit 0	Select load_port_replay event with the MSR_SAAT_ESCR1 MSR and set the SPLIT_LD mask bit.	NBOGUS
split_store_retired	Bit 10, Bit 24, Bit 25	Bit 1	Select store_port_replay event with the MSR_SAAT_ESCR0 MSR and set the SPLIT_ST mask bit.	NBOGUS

NOTES:

1. Certain kinds of μops cannot be tagged. These include I/O operations, UC and locked accesses, returns, and far transfers.
2. 2nd-level misses retired does not count all 2nd-level misses. It only includes those references that are found to be misses by the fast detection logic and not those that are later found to be misses.
3. While there are several causes for a MOB replay, the event counted with this event mask setting is the case where the data from a load that would otherwise be forwarded is not an aligned subset of the data from a preceding store.

Table 19-36. Event Mask Qualification for Logical Processors

Event Type	Event Name	Event Masks, ESCR[24:9]	TS or TI
Non-Retirement	BPU_fetch_request	Bit 0: TCMISS	TS
Non-Retirement	BSQ_allocation	Bit	
		0: REQ_TYPE0	TS
		1: REQ_TYPE1	TS
		2: REQ_LEN0	TS
		3: REQ_LEN1	TS
		5: REQ_IO_TYPE	TS
		6: REQ_LOCK_TYPE	TS
		7: REQ_CACHE_TYPE	TS
		8: REQ_SPLIT_TYPE	TS
		9: REQ_DEM_TYPE	TS
		10: REQ_ORD_TYPE	TS
		11: MEM_TYPE0	TS
		12: MEM_TYPE1	TS
		13: MEM_TYPE2	TS
Non-Retirement	BSQ_cache_reference	Bit	
		0: RD_2ndL_HITS	TS
		1: RD_2ndL_HITE	TS
		2: RD_2ndL_HITM	TS
		3: RD_3rdL_HITS	TS
		4: RD_3rdL_HITE	TS
		5: RD_3rdL_HITM	TS
		6: WR_2ndL_HIT	TS
		7: WR_3rdL_HIT	TS
		8: RD_2ndL_MISS	TS
		9: RD_3rdL_MISS	TS
		10: WR_2ndL_MISS	TS
		11: WR_3rdL_MISS	TS
Non-Retirement	memory_cancel	Bit	
		2: ST_RB_FULL	TS
		3: 64K_CONF	TS
Non-Retirement	SSE_input_assist	Bit 15: ALL	TI
Non-Retirement	64bit_MMX_uop	Bit 15: ALL	TI
Non-Retirement	packed_DP_uop	Bit 15: ALL	TI
Non-Retirement	packed_SP_uop	Bit 15: ALL	TI
Non-Retirement	scalar_DP_uop	Bit 15: ALL	TI
Non-Retirement	scalar_SP_uop	Bit 15: ALL	TI
Non-Retirement	128bit_MMX_uop	Bit 15: ALL	TI
Non-Retirement	x87_FP_uop	Bit 15: ALL	TI

Table 19-36. Event Mask Qualification for Logical Processors (Contd.)

Event Type	Event Name	Event Masks, ESCR[24:9]	TS or TI
Non-Retirement	x87_SIMD_moves_uop	Bit	
		3: ALLP0	TI
		4: ALLP2	TI
Non-Retirement	FSB_data_activity	Bit	
		0: DRDY_DRV	TI
		1: DRDY_OWN	TI
		2: DRDY_OTHER	TI
		3: DBSY_DRV	TI
		4: DBSY_OWN	TI
		5: DBSY_OTHER	TI
Non-Retirement	IOQ_allocation	Bit	
		0: ReqA0	TS
		1: ReqA1	TS
		2: ReqA2	TS
		3: ReqA3	TS
		4: ReqA4	TS
		5: ALL_READ	TS
		6: ALL_WRITE	TS
		7: MEM_UC	TS
		8: MEM_WC	TS
		9: MEM_WT	TS
		10: MEM_WP	TS
		11: MEM_WB	TS
		13: OWN	TS
		14: OTHER	TS
		15: PREFETCH	TS
Non-Retirement	IOQ_active_entries	Bit	TS
		0: ReqA0	
		1:ReqA1	TS
		2: ReqA2	TS
		3: ReqA3	TS
		4: ReqA4	TS
		5: ALL_READ	TS
		6: ALL_WRITE	TS
		7: MEM_UC	TS
		8: MEM_WC	TS
		9: MEM_WT	TS
		10: MEM_WP	TS
		11: MEM_WB	TS

Table 19-36. Event Mask Qualification for Logical Processors (Contd.)

Event Type	Event Name	Event Masks, ESCR[24:9]	TS or TI
		13: OWN	TS
		14: OTHER	TS
		15: PREFETCH	TS
Non-Retirement	global_power_events	Bit 0: RUNNING	TS
Non-Retirement	ITLB_reference	Bit	
		0: HIT	TS
		1: MISS	TS
		2: HIT_UC	TS
Non-Retirement	MOB_load_replay	Bit	
		1: NO_STA	TS
		3: NO_STD	TS
		4: PARTIAL_DATA	TS
		5: UNALGN_ADDR	TS
Non-Retirement	page_walk_type	Bit	
		0: DTMISS	TI
		1: ITMISS	TI
Non-Retirement	uop_type	Bit	
		1: TAGLOADS	TS
		2: TAGSTORES	TS
Non-Retirement	load_port_replay	Bit 1: SPLIT_LD	TS
Non-Retirement	store_port_replay	Bit 1: SPLIT_ST	TS
Non-Retirement	memory_complete	Bit	
		0: LSC	TS
		1: SSC	TS
		2: USC	TS
		3: ULC	TS
Non-Retirement	retired_mispred_branch_ type	Bit	
		0: UNCONDITIONAL	TS
		1: CONDITIONAL	TS
		2: CALL	TS
		3: RETURN	TS
		4: INDIRECT	TS
Non-Retirement	retired_branch_type	Bit	
		0: UNCONDITIONAL	TS
		1: CONDITIONAL	TS
		2: CALL	TS
		3: RETURN	TS
		4: INDIRECT	TS

Table 19-36. Event Mask Qualification for Logical Processors (Contd.)

Event Type	Event Name	Event Masks, ESCR[24:9]	TS or TI
Non-Retirement	tc_ms_xfer	Bit 0: CISC	TS
Non-Retirement	tc_misc	Bit 4: FLUSH	TS
Non-Retirement	TC_deliver_mode	Bit 0: DD 1: DB 2: DI 3: BD 4: BB 5: BI 6: ID 7: IB	TI TI TI TI TI TI TI TI
Non-Retirement	uop_queue_writes	Bit 0: FROM_TC_BUILD 1: FROM_TC_DELIVER 2: FROM_ROM	TS TS TS
Non-Retirement	resource_stall	Bit 5: SBFULL	TS
Non-Retirement	WC_Buffer	Bit 0: WCB_EVICTS 1: WCB_FULL_EVICT 2: WCB_HITM_EVICT	TI TI TI TI
At Retirement	instr_retired	Bit 0: NBOGUSNTAG 1: NBOGUSTAG 2: BOGUSNTAG 3: BOGUSTAG	TS TS TS TS
At Retirement	machine_clear	Bit 0: CLEAR 2: MOCLEAR 6: SMCCLEAR	TS TS TS
At Retirement	front_end_event	Bit 0: NBOGUS 1: BOGUS	TS TS
At Retirement	replay_event	Bit 0: NBOGUS 1: BOGUS	TS TS
At Retirement	execution_event	Bit 0: NONBOGUS0 1: NONBOGUS1	TS TS

Table 19-36. Event Mask Qualification for Logical Processors (Contd.)

Event Type	Event Name	Event Masks, ESCR[24:9]	TS or TI
		2: NONBOGUS2	TS
		3: NONBOGUS3	TS
		4: BOGUS0	TS
		5: BOGUS1	TS
		6: BOGUS2	TS
		7: BOGUS3	TS
At Retirement	x87_assist	Bit 0: FPSU	TS
		1: FPSO	TS
		2: POAO	TS
		3: POAU	TS
		4: PREA	TS
At Retirement	branch_retired	Bit 0: MMNP	TS
		1: MMNM	TS
		2: MMTP	TS
		3: MMTM	TS
At Retirement	mispred_branch_retired	Bit 0: NBOGUS	TS
At Retirement	uops_retired	Bit 0: NBOGUS	TS
		1: BOGUS	TS
At Retirement	instr_completed	Bit 0: NBOGUS	TS
		1: BOGUS	TS

19.18 PERFORMANCE MONITORING EVENTS FOR INTEL® PENTIUM® M PROCESSORS

The Pentium M processor's performance-monitoring events are based on monitoring events for the P6 family of processors. All of these performance events are model specific for the Pentium M processor and are not available in this form in other processors. Table 19-37 lists the Performance-Monitoring events that were added in the Pentium M processor.

Table 19-37. Performance Monitoring Events on Intel® Pentium® M Processors

Name	Hex Values	Descriptions
Power Management		
EMON_EST_TRANS	58H	Number of Enhanced Intel SpeedStep technology transitions: Mask = 00H - All transitions Mask = 02H - Only Frequency transitions
EMON_THERMAL_TRIP	59H	Duration/Occurrences in thermal trip; to count number of thermal trips: bit 22 in PerfEvtSel0/1 needs to be set to enable edge detect.
BPU		
BR_INST_EXEC	88H	Branch instructions that were executed (not necessarily retired).
BR_MISSP_EXEC	89H	Branch instructions executed that were mispredicted at execution.
BR_BAC_MISSP_EXEC	8AH	Branch instructions executed that were mispredicted at front end (BAC).
BR_CND_EXEC	8BH	Conditional branch instructions that were executed.
BR_CND_MISSP_EXEC	8CH	Conditional branch instructions executed that were mispredicted.
BR_IND_EXEC	8DH	Indirect branch instructions executed.
BR_IND_MISSP_EXEC	8EH	Indirect branch instructions executed that were mispredicted.
BR_RET_EXEC	8FH	Return branch instructions executed.
BR_RET_MISSP_EXEC	90H	Return branch instructions executed that were mispredicted at execution.
BR_RET_BAC_MISSP_EXEC	91H	Return branch instructions executed that were mispredicted at front end (BAC).
BR_CALL_EXEC	92H	CALL instruction executed.
BR_CALL_MISSP_EXEC	93H	CALL instruction executed and miss predicted.
BR_IND_CALL_EXEC	94H	Indirect CALL instructions executed.
Decoder		
EMON_SIMD_INSTR_RETIRED	CEH	Number of retired MMX instructions.
EMON_SYNCH_UOPS	D3H	Sync micro-ops
EMON_ESP_UOPS	D7H	Total number of micro-ops
EMON_FUSED_UOPS_RET	DAH	Number of retired fused micro-ops: Mask = 0 - Fused micro-ops Mask = 1 - Only load+Op micro-ops Mask = 2 - Only std+sta micro-ops
EMON_UNFUSION	DBH	Number of unfusion events in the ROB, happened on a FP exception to a fused μop.
Prefetcher		
EMON_PREF_RQSTS_UP	F0H	Number of upward prefetches issued.
EMON_PREF_RQSTS_DN	F8H	Number of downward prefetches issued.

A number of P6 family processor performance monitoring events are modified for the Pentium M processor. Table 19-38 lists the performance monitoring events that were changed in the Pentium M processor, and differ from performance monitoring events for the P6 family of processors.

Table 19-38. Performance Monitoring Events Modified on Intel® Pentium® M Processors

Name	Hex Values	Descriptions	
CPU_CLK_UNHALTED	79H	Number of cycles during which the processor is not halted, and not in a thermal trip.	
EMON_SSE_SSE2_INST_ RETIRED	D8H	Streaming SIMD Extensions Instructions Retired: Mask = 0 – SSE packed single and scalar single Mask = 1 – SSE scalar-single Mask = 2 – SSE2 packed-double Mask = 3 – SSE2 scalar-double	
EMON_SSE_SSE2_COMP_INST_ RETIRED	D9H	Computational SSE Instructions Retired: Mask = 0 – SSE packed single Mask = 1 – SSE Scalar-single Mask = 2 – SSE2 packed-double Mask = 3 – SSE2 scalar-double	
L2_LD	29H	L2 data loads	Mask[0] = 1 – count I state lines Mask[1] = 1 – count S state lines Mask[2] = 1 – count E state lines Mask[3] = 1 – count M state lines Mask[5:4]: 00H – Excluding hardware-prefetched lines 01H - Hardware-prefetched lines only 02H/03H – All (HW-prefetched lines and non HW -- Prefetched lines)
L2_LINES_IN	24H	L2 lines allocated	
L2_LINES_OUT	26H	L2 lines evicted	
L2_M_LINES_OUT	27H	Lw M-state lines evicted	

19.19 P6 FAMILY PROCESSOR PERFORMANCE-MONITORING EVENTS

Table 19-39 lists the events that can be counted with the performance-monitoring counters and read with the RDPMC instruction for the P6 family processors. The unit column gives the microarchitecture or bus unit that produces the event; the event number column gives the hexadecimal number identifying the event; the mnemonic event name column gives the name of the event; the unit mask column gives the unit mask required (if any); the description column describes the event; and the comments column gives additional information about the event.

All of these performance events are model specific for the P6 family processors and are not available in this form in the Pentium 4 processors or the Pentium processors. Some events (such as those added in later generations of the P6 family processors) are only available in specific processors in the P6 family. All performance event encodings not listed in Table 19-39 are reserved and their use will result in undefined counter results.

See the end of the table for notes related to certain entries in the table.

Table 19-39. Events That Can Be Counted with the P6 Family Performance-Monitoring Counters

Unit	Event Num.	Mnemonic Event Name	Unit Mask	Description	Comments
Data Cache Unit (DCU)	43H	DATA_MEM_REFS	00H	All loads from any memory type. All stores to any memory type. Each part of a split is counted separately. The internal logic counts not only memory loads and stores, but also internal retries. 80-bit floating-point accesses are double counted, since they are decomposed into a 16-bit exponent load and a 64-bit mantissa load. Memory accesses are only counted when they are actually performed (such as a load that gets squashed because a previous cache miss is outstanding to the same address, and which finally gets performed, is only counted once). Does not include I/O accesses, or other nonmemory accesses.	
	45H	DCU_LINES_IN	00H	Total lines allocated in DCU.	
	46H	DCU_M_LINES_IN	00H	Number of M state lines allocated in DCU.	
	47H	DCU_M_LINES_ OUT	00H	Number of M state lines evicted from DCU. This includes evictions via snoop HITM, intervention or replacement.	
	48H	DCU_MISS_ OUTSTANDING	00H	Weighted number of cycles while a DCU miss is outstanding, incremented by the number of outstanding cache misses at any particular time. Cacheable read requests only are considered. Uncacheable requests are excluded. Read-for-ownerships are counted, as well as line fills, invalidates, and stores.	An access that also misses the L2 is short-changed by 2 cycles (i.e., if counts N cycles, should be N+2 cycles). Subsequent loads to the same cache line will not result in any additional counts. Count value not precise, but still useful.
Instruction Fetch Unit (IFU)	80H	IFU_IFETCH	00H	Number of instruction fetches, both cacheable and noncacheable, including UC fetches.	
	81H	IFU_IFETCH_ MISS	00H	Number of instruction fetch misses All instruction fetches that do not hit the IFU (i.e., that produce memory requests). This includes UC accesses.	
	85H	ITLB_MISS	00H	Number of ITLB misses.	
	86H	IFU_MEM_STALL	00H	Number of cycles instruction fetch is stalled, for any reason. Includes IFU cache misses, ITLB misses, ITLB faults, and other minor stalls.	
	87H	ILD_STALL	00H	Number of cycles that the instruction length decoder is stalled.	
L2 Cache[1]	28H	L2_IFETCH	MESI 0FH	Number of L2 instruction fetches. This event indicates that a normal instruction fetch was received by the L2.	

Table 19-39. Events That Can Be Counted with the P6 Family Performance-Monitoring Counters (Contd.)

Unit	Event Num.	Mnemonic Event Name	Unit Mask	Description	Comments
				The count includes only L2 cacheable instruction fetches; it does not include UC instruction fetches. It does not include ITLB miss accesses.	
	29H	L2_LD	MESI 0FH	Number of L2 data loads. This event indicates that a normal, unlocked, load memory access was received by the L2. It includes only L2 cacheable memory accesses; it does not include I/O accesses, other nonmemory accesses, or memory accesses such as UC/WT memory accesses. It does include L2 cacheable TLB miss memory accesses.	
	2AH	L2_ST	MESI 0FH	Number of L2 data stores. This event indicates that a normal, unlocked, store memory access was received by the L2. it indicates that the DCU sent a read-for-ownership request to the L2. It also includes Invalid to Modified requests sent by the DCU to the L2. It includes only L2 cacheable memory accesses; it does not include I/O accesses, other nonmemory accesses, or memory accesses such as UC/WT memory accesses. It includes TLB miss memory accesses.	
	24H	L2_LINES_IN	00H	Number of lines allocated in the L2.	
	26H	L2_LINES_OUT	00H	Number of lines removed from the L2 for any reason.	
	25H	L2_M_LINES_INM	00H	Number of modified lines allocated in the L2.	
	27H	L2_M_LINES_ OUTM	00H	Number of modified lines removed from the L2 for any reason.	
	2EH	L2_RQSTS	MESI 0FH	Total number of L2 requests.	
	21H	L2_ADS	00H	Number of L2 address strobes.	
	22H	L2_DBUS_BUSY	00H	Number of cycles during which the L2 cache data bus was busy.	
	23H	L2_DBUS_BUSY_ RD	00H	Number of cycles during which the data bus was busy transferring read data from L2 to the processor.	
External Bus Logic (EBL)[2]	62H	BUS_DRDY_ CLOCKS	00H (Self) 20H (Any)	Number of clocks during which DRDY# is asserted. Utilization of the external system data bus during data transfers.	Unit Mask = 00H counts bus clocks when the processor is driving DRDY#. Unit Mask = 20H counts in processor clocks when any agent is driving DRDY#.

Table 19-39. Events That Can Be Counted with the P6 Family Performance-Monitoring Counters (Contd.)

Unit	Event Num.	Mnemonic Event Name	Unit Mask	Description	Comments
	63H	BUS_LOCK_ CLOCKS	00H (Self) 20H (Any)	Number of clocks during which LOCK# is asserted on the external system bus.[3]	Always counts in processor clocks.
	60H	BUS_REQ_ OUTSTANDING	00H (Self)	Number of bus requests outstanding. This counter is incremented by the number of cacheable read bus requests outstanding in any given cycle.	Counts only DCU full-line cacheable reads, not RFOs, writes, instruction fetches, or anything else. Counts "waiting for bus to complete" (last data chunk received).
	65H	BUS_TRAN_BRD	00H (Self) 20H (Any)	Number of burst read transactions.	
	66H	BUS_TRAN_RFO	00H (Self) 20H (Any)	Number of completed read for ownership transactions.	
	67H	BUS_TRANS_WB	00H (Self) 20H (Any)	Number of completed write back transactions.	
	68H	BUS_TRAN_ IFETCH	00H (Self) 20H (Any)	Number of completed instruction fetch transactions.	
	69H	BUS_TRAN_INVA L	00H (Self) 20H (Any)	Number of completed invalidate transactions.	
	6AH	BUS_TRAN_PWR	00H (Self) 20H (Any)	Number of completed partial write transactions.	
	6BH	BUS_TRANS_P	00H (Self) 20H (Any)	Number of completed partial transactions.	
	6CH	BUS_TRANS_IO	00H (Self) 20H (Any)	Number of completed I/O transactions.	
	6DH	BUS_TRAN_DEF	00H (Self) 20H (Any)	Number of completed deferred transactions.	

Table 19-39. Events That Can Be Counted with the P6 Family Performance-Monitoring Counters (Contd.)

Unit	Event Num.	Mnemonic Event Name	Unit Mask	Description	Comments
	6EH	BUS_TRAN_ BURST	00H (Self) 20H (Any)	Number of completed burst transactions.	
	70H	BUS_TRAN_ANY	00H (Self) 20H (Any)	Number of all completed bus transactions. Address bus utilization can be calculated knowing the minimum address bus occupancy. Includes special cycles, etc.	
	6FH	BUS_TRAN_MEM	00H (Self) 20H (Any)	Number of completed memory transactions.	
	64H	BUS_DATA_RCV	00H (Self)	Number of bus clock cycles during which this processor is receiving data.	
	61H	BUS_BNR_DRV	00H (Self)	Number of bus clock cycles during which this processor is driving the BNR# pin.	
	7AH	BUS_HIT_DRV	00H (Self)	Number of bus clock cycles during which this processor is driving the HIT# pin.	Includes cycles due to snoop stalls. The event counts correctly, but BPM*i* (breakpoint monitor) pins function as follows based on the setting of the PC bits (bit 19 in the PerfEvtSel0 and PerfEvtSel1 registers): ▪ If the core-clock-to- bus-clock ratio is 2:1 or 3:1, and a PC bit is set, the BPM*i* pins will be asserted for a single clock when the counters overflow. ▪ If the PC bit is clear, the processor toggles the BPM*i* pins when the counter overflows. ▪ If the clock ratio is not 2:1 or 3:1, the BPM*i* pins will not function for these performance-monitoring counter events.
	7BH	BUS_HITM_DRV	00H (Self)	Number of bus clock cycles during which this processor is driving the HITM# pin.	Includes cycles due to snoop stalls. The event counts correctly, but BPM*i* (breakpoint monitor) pins function as follows based on the setting of the PC bits (bit 19 in the PerfEvtSel0 and PerfEvtSel1 registers): ▪ If the core-clock-to- bus-clock ratio is 2:1 or 3:1, and a PC bit is set, the BPM*i* pins will be asserted for a single clock when the counters overflow.

Table 19-39. Events That Can Be Counted with the P6 Family Performance-Monitoring Counters (Contd.)

Unit	Event Num.	Mnemonic Event Name	Unit Mask	Description	Comments
					▪ If the PC bit is clear, the processor toggles the BPM*i*pins when the counter overflows. ▪ If the clock ratio is not 2:1 or 3:1, the BPM*i* pins will not function for these performance-monitoring counter events.
	7EH	BUS_SNOOP_ STALL	00H (Self)	Number of clock cycles during which the bus is snoop stalled.	
Floating-Point Unit	C1H	FLOPS	00H	Number of computational floating-point operations retired. Excludes floating-point computational operations that cause traps or assists. Includes floating-point computational operations executed by the assist handler. Includes internal sub-operations for complex floating-point instructions like transcendentals. Excludes floating-point loads and stores.	Counter 0 only.
	10H	FP_COMP_OPS_ EXE	00H	Number of computational floating-point operations executed. The number of FADD, FSUB, FCOM, FMULs, integer MULs and IMULs, FDIVs, FPREMs, FSQRTS, integer DIVs, and IDIVs. This number does not include the number of cycles, but the number of operations. This event does not distinguish an FADD used in the middle of a transcendental flow from a separate FADD instruction.	Counter 0 only.
	11H	FP_ASSIST	00H	Number of floating-point exception cases handled by microcode.	Counter 1 only. This event includes counts due to speculative execution.
	12H	MUL	00H	Number of multiplies. This count includes integer as well as FP multiplies and is speculative.	Counter 1 only.
	13H	DIV	00H	Number of divides. This count includes integer as well as FP divides and is speculative.	Counter 1 only.
	14H	CYCLES_DIV_ BUSY	00H	Number of cycles during which the divider is busy, and cannot accept new divides. This includes integer and FP divides, FPREM, FPSQRT, etc. and is speculative.	Counter 0 only.

Table 19-39. Events That Can Be Counted with the P6 Family Performance-Monitoring Counters (Contd.)

Unit	Event Num.	Mnemonic Event Name	Unit Mask	Description	Comments
Memory Ordering	03H	LD_BLOCKS	00H	Number of load operations delayed due to store buffer blocks. Includes counts caused by preceding stores whose addresses are unknown, preceding stores whose addresses are known but whose data is unknown, and preceding stores that conflicts with the load but which incompletely overlap the load.	
	04H	SB_DRAINS	00H	Number of store buffer drain cycles. Incremented every cycle the store buffer is draining. Draining is caused by serializing operations like CPUID, synchronizing operations like XCHG, interrupt acknowledgment, as well as other conditions (such as cache flushing).	
	05H	MISALIGN_ MEM_REF	00H	Number of misaligned data memory references. Incremented by 1 every cycle, during which either the processor's load or store pipeline dispatches a misaligned μop. Counting is performed if it is the first or second half, or if it is blocked, squashed, or missed. In this context, misaligned means crossing a 64-bit boundary.	MISALIGN_MEM_ REF is only an approximation to the true number of misaligned memory references. The value returned is roughly proportional to the number of misaligned memory accesses (the size of the problem).
	07H	EMON_KNI_PREF _DISPATCHED	 00H 01H 02H 03H	Number of Streaming SIMD extensions prefetch/weakly-ordered instructions dispatched (speculative prefetches are included in counting): 0: prefetch NTA 1: prefetch T1 2: prefetch T2 3: weakly ordered stores	Counters 0 and 1. Pentium III processor only.
	4BH	EMON_KNI_PREF _MISS	 00H 01H 02H 03H	Number of prefetch/weakly-ordered instructions that miss all caches: 0: prefetch NTA 1: prefetch T1 2: prefetch T2 3: weakly ordered stores	Counters 0 and 1. Pentium III processor only.
Instruction Decoding and Retirement	C0H	INST_RETIRED	00H	Number of instructions retired.	A hardware interrupt received during/after the last iteration of the REP STOS flow causes the counter to undercount by 1 instruction.
					An SMI received while executing a HLT instruction will cause the performance counter to not count the RSM instruction and undercount by 1.

Table 19-39. Events That Can Be Counted with the P6 Family Performance-Monitoring Counters (Contd.)

Unit	Event Num.	Mnemonic Event Name	Unit Mask	Description	Comments
	C2H	UOPS_RETIRED	00H	Number of μops retired.	
	D0H	INST_DECODED	00H	Number of instructions decoded.	
	D8H	EMON_KNI_INST_RETIRED	00H 01H	Number of Streaming SIMD extensions retired: 0: packed & scalar 1: scalar	Counters 0 and 1. Pentium III processor only.
	D9H	EMON_KNI_COMP_INST_RET	00H 01H	Number of Streaming SIMD extensions computation instructions retired: 0: packed and scalar 1: scalar	Counters 0 and 1. Pentium III processor only.
Interrupts	C8H	HW_INT_RX	00H	Number of hardware interrupts received.	
	C6H	CYCLES_INT_MASKED	00H	Number of processor cycles for which interrupts are disabled.	
	C7H	CYCLES_INT_PENDING_AND_MASKED	00H	Number of processor cycles for which interrupts are disabled and interrupts are pending.	
Branches	C4H	BR_INST_RETIRED	00H	Number of branch instructions retired.	
	C5H	BR_MISS_PRED_RETIRED	00H	Number of mispredicted branches retired.	
	C9H	BR_TAKEN_RETIRED	00H	Number of taken branches retired.	
	CAH	BR_MISS_PRED_TAKEN_RET	00H	Number of taken mispredictions branches retired.	
	E0H	BR_INST_DECODED	00H	Number of branch instructions decoded.	
	E2H	BTB_MISSES	00H	Number of branches for which the BTB did not produce a prediction.	
	E4H	BR_BOGUS	00H	Number of bogus branches.	
	E6H	BACLEARS	00H	Number of times BACLEAR is asserted. This is the number of times that a static branch prediction was made, in which the branch decoder decided to make a branch prediction because the BTB did not.	
Stalls	A2H	RESOURCE_STALLS	00H	Incremented by 1 during every cycle for which there is a resource related stall. Includes register renaming buffer entries, memory buffer entries.	

Table 19-39. Events That Can Be Counted with the P6 Family Performance-Monitoring Counters (Contd.)

Unit	Event Num.	Mnemonic Event Name	Unit Mask	Description	Comments
				Does not include stalls due to bus queue full, too many cache misses, etc. In addition to resource related stalls, this event counts some other events. Includes stalls arising during branch misprediction recovery, such as if retirement of the mispredicted branch is delayed and stalls arising while store buffer is draining from synchronizing operations.	
	D2H	PARTIAL_RAT_ STALLS	00H	Number of cycles or events for partial stalls. This includes flag partial stalls.	
Segment Register Loads	06H	SEGMENT_REG_ LOADS	00H	Number of segment register loads.	
Clocks	79H	CPU_CLK_ UNHALTED	00H	Number of cycles during which the processor is not halted.	
MMX Unit	B0H	MMX_INSTR_ EXEC	00H	Number of MMX Instructions Executed.	Available in Intel Celeron, Pentium II and Pentium II Xeon processors only. Does not account for MOVQ and MOVD stores from register to memory.
	B1H	MMX_SAT_ INSTR_EXEC	00H	Number of MMX Saturating Instructions Executed.	Available in Pentium II and Pentium III processors only.
	B2H	MMX_UOPS_ EXEC	0FH	Number of MMX μops Executed.	Available in Pentium II and Pentium III processors only.
	B3H	MMX_INSTR_ TYPE_EXEC	01H 02H 04H 08H 10H 20H	MMX packed multiply instructions executed. MMX packed shift instructions executed. MMX pack operation instructions executed. MMX unpack operation instructions executed. MMX packed logical instructions executed. MMX packed arithmetic instructions executed.	Available in Pentium II and Pentium III processors only.
	CCH	FP_MMX_TRANS	00H 01H	Transitions from MMX instruction to floating-point instructions. Transitions from floating-point instructions to MMX instructions.	Available in Pentium II and Pentium III processors only.
	CDH	MMX_ASSIST	00H	Number of MMX Assists (that is, the number of EMMS instructions executed).	Available in Pentium II and Pentium III processors only.
	CEH	MMX_INSTR_RET	00H	Number of MMX Instructions Retired.	Available in Pentium II processors only.
Segment Register Renaming	D4H	SEG_RENAME_ STALLS		Number of Segment Register Renaming Stalls:	Available in Pentium II and Pentium III processors only.

Table 19-39. Events That Can Be Counted with the P6 Family Performance-Monitoring Counters (Contd.)

Unit	Event Num.	Mnemonic Event Name	Unit Mask	Description	Comments
			02H	Segment register ES	
			04H	Segment register DS	
			08H	Segment register FS	
			0FH	Segment register FS	
				Segment registers ES + DS + FS + GS	
	D5H	SEG_REG_ RENAMES		Number of Segment Register Renames:	Available in Pentium II and Pentium III processors only.
			01H	Segment register ES	
			02H	Segment register DS	
			04H	Segment register FS	
			08H	Segment register FS	
			0FH	Segment registers ES + DS + FS + GS	
	D6H	RET_SEG_ RENAMES	00H	Number of segment register rename events retired.	Available in Pentium II and Pentium III processors only.

NOTES:

1. Several L2 cache events, where noted, can be further qualified using the Unit Mask (UMSK) field in the PerfEvtSel0 and PerfEvtSel1 registers. The lower 4 bits of the Unit Mask field are used in conjunction with L2 events to indicate the cache state or cache states involved.

   The P6 family processors identify cache states using the "MESI" protocol and consequently each bit in the Unit Mask field represents one of the four states: UMSK[3] = M (8H) state, UMSK[2] = E (4H) state, UMSK[1] = S (2H) state, and UMSK[0] = I (1H) state. UMSK[3:0] = MESI" (FH) should be used to collect data for all states; UMSK = 0H, for the applicable events, will result in nothing being counted.

2. All of the external bus logic (EBL) events, except where noted, can be further qualified using the Unit Mask (UMSK) field in the PerfEvtSel0 and PerfEvtSel1 registers.

   Bit 5 of the UMSK field is used in conjunction with the EBL events to indicate whether the processor should count transactions that are self- generated (UMSK[5] = 0) or transactions that result from any processor on the bus (UMSK[5] = 1).

3. L2 cache locks, so it is possible to have a zero count.

19.20 PENTIUM PROCESSOR PERFORMANCE-MONITORING EVENTS

Table 19-40 lists the events that can be counted with the performance-monitoring counters for the Pentium processor. The Event Number column gives the hexadecimal code that identifies the event and that is entered in the ES0 or ES1 (event select) fields of the CESR MSR. The Mnemonic Event Name column gives the name of the event, and the Description and Comments columns give detailed descriptions of the events. Most events can be counted with either counter 0 or counter 1; however, some events can only be counted with only counter 0 or only counter 1 (as noted).

NOTE

The events in the table that are shaded are implemented only in the Pentium processor with MMX technology.

Table 19-40. Events That Can Be Counted with Pentium Processor Performance-Monitoring Counters

Event Num.	Mnemonic Event Name	Description	Comments
00H	DATA_READ	Number of memory data reads (internal data cache hit and miss combined).	Split cycle reads are counted individually. Data Memory Reads that are part of TLB miss processing are not included. These events may occur at a maximum of two per clock. I/O is not included.
01H	DATA_WRITE	Number of memory data writes (internal data cache hit and miss combined); I/O not included.	Split cycle writes are counted individually. These events may occur at a maximum of two per clock. I/O is not included.
0H2	DATA_TLB_MISS	Number of misses to the data cache translation look-aside buffer.	
03H	DATA_READ_MISS	Number of memory read accesses that miss the internal data cache whether or not the access is cacheable or noncacheable.	Additional reads to the same cache line after the first BRDY# of the burst line fill is returned but before the final (fourth) BRDY# has been returned, will not cause the counter to be incremented additional times. Data accesses that are part of TLB miss processing are not included. Accesses directed to I/O space are not included.
04H	DATA WRITE MISS	Number of memory write accesses that miss the internal data cache whether or not the access is cacheable or noncacheable.	Data accesses that are part of TLB miss processing are not included. Accesses directed to I/O space are not included.
05H	WRITE_HIT_TO_ M-_OR_E-STATE_LINES	Number of write hits to exclusive or modified lines in the data cache.	These are the writes that may be held up if EWBE# is inactive. These events may occur a maximum of two per clock.
06H	DATA_CACHE_ LINES_ WRITTEN_BACK	Number of dirty lines (all) that are written back, regardless of the cause.	Replacements and internal and external snoops can all cause writeback and are counted.
07H	EXTERNAL_ SNOOPS	Number of accepted external snoops whether they hit in the code cache or data cache or neither.	Assertions of EADS# outside of the sampling interval are not counted, and no internal snoops are counted.
08H	EXTERNAL_DATA_ CACHE_SNOOP_ HITS	Number of external snoops to the data cache.	Snoop hits to a valid line in either the data cache, the data line fill buffer, or one of the write back buffers are all counted as hits.
09H	MEMORY ACCESSES IN BOTH PIPES	Number of data memory reads or writes that are paired in both pipes of the pipeline.	These accesses are not necessarily run in parallel due to cache misses, bank conflicts, etc.
0AH	BANK CONFLICTS	Number of actual bank conflicts.	
0BH	MISALIGNED DATA MEMORY OR I/O REFERENCES	Number of memory or I/O reads or writes that are misaligned.	A 2- or 4-byte access is misaligned when it crosses a 4-byte boundary; an 8-byte access is misaligned when it crosses an 8-byte boundary. Ten byte accesses are treated as two separate accesses of 8 and 2 bytes each.
0CH	CODE READ	Number of instruction reads; whether the read is cacheable or noncacheable.	Individual 8-byte noncacheable instruction reads are counted.
0DH	CODE TLB MISS	Number of instruction reads that miss the code TLB whether the read is cacheable or noncacheable.	Individual 8-byte noncacheable instruction reads are counted.
0EH	CODE CACHE MISS	Number of instruction reads that miss the internal code cache; whether the read is cacheable or noncacheable.	Individual 8-byte noncacheable instruction reads are counted.

Table 19-40. Events That Can Be Counted with Pentium Processor Performance-Monitoring Counters (Contd.)

Event Num.	Mnemonic Event Name	Description	Comments
0FH	ANY SEGMENT REGISTER LOADED	Number of writes into any segment register in real or protected mode including the LDTR, GDTR, IDTR, and TR.	Segment loads are caused by explicit segment register load instructions, far control transfers, and task switches. Far control transfers and task switches causing a privilege level change will signal this event twice. Interrupts and exceptions may initiate a far control transfer.
10H	Reserved		
11H	Reserved		
12H	Branches	Number of taken and not taken branches, including: conditional branches, jumps, calls, returns, software interrupts, and interrupt returns.	Also counted as taken branches are serializing instructions, VERR and VERW instructions, some segment descriptor loads, hardware interrupts (including FLUSH#), and programmatic exceptions that invoke a trap or fault handler. The pipe is not necessarily flushed. The number of branches actually executed is measured, not the number of predicted branches.
13H	BTB_HITS	Number of BTB hits that occur.	Hits are counted only for those instructions that are actually executed.
14H	TAKEN_BRANCH_ OR_BTB_HIT	Number of taken branches or BTB hits that occur.	This event type is a logical OR of taken branches and BTB hits. It represents an event that may cause a hit in the BTB. Specifically, it is either a candidate for a space in the BTB or it is already in the BTB.
15H	PIPELINE FLUSHES	Number of pipeline flushes that occur Pipeline flushes are caused by BTB misses on taken branches, mispredictions, exceptions, interrupts, and some segment descriptor loads.	The counter will not be incremented for serializing instructions (serializing instructions cause the prefetch queue to be flushed but will not trigger the Pipeline Flushed event counter) and software interrupts (software interrupts do not flush the pipeline).
16H	INSTRUCTIONS_ EXECUTED	Number of instructions executed (up to two per clock).	Invocations of a fault handler are considered instructions. All hardware and software interrupts and exceptions will also cause the count to be incremented. Repeat prefixed string instructions will only increment this counter once despite the fact that the repeat loop executes the same instruction multiple times until the loop criteria is satisfied. This applies to all the Repeat string instruction prefixes (i.e., REP, REPE, REPZ, REPNE, and REPNZ). This counter will also only increment once per each HLT instruction executed regardless of how many cycles the processor remains in the HALT state.
17H	INSTRUCTIONS_ EXECUTED_ V PIPE	Number of instructions executed in the V_pipe. The event indicates the number of instructions that were paired.	This event is the same as the 16H event except it only counts the number of instructions actually executed in the V-pipe.
18H	BUS_CYCLE_ DURATION	Number of clocks while a bus cycle is in progress. This event measures bus use.	The count includes HLDA, AHOLD, and BOFF# clocks.
19H	WRITE_BUFFER_ FULL_STALL_ DURATION	Number of clocks while the pipeline is stalled due to full write buffers.	Full write buffers stall data memory read misses, data memory write misses, and data memory write hits to S-state lines. Stalls on I/O accesses are not included.

Table 19-40. Events That Can Be Counted with Pentium Processor Performance-Monitoring Counters (Contd.)

Event Num.	Mnemonic Event Name	Description	Comments
1AH	WAITING_FOR_ DATA_MEMORY_ READ_STALL_ DURATION	Number of clocks while the pipeline is stalled while waiting for data memory reads.	Data TLB Miss processing is also included in the count. The pipeline stalls while a data memory read is in progress including attempts to read that are not bypassed while a line is being filled.
1BH	STALL ON WRITE TO AN E- OR M-STATE LINE	Number of stalls on writes to E- or M-state lines.	
1CH	LOCKED BUS CYCLE	Number of locked bus cycles that occur as the result of the LOCK prefix or LOCK instruction, page-table updates, and descriptor table updates.	Only the read portion of the locked read-modify-write is counted. Split locked cycles (SCYC active) count as two separate accesses. Cycles restarted due to BOFF# are not re-counted.
1DH	I/O READ OR WRITE CYCLE	Number of bus cycles directed to I/O space.	Misaligned I/O accesses will generate two bus cycles. Bus cycles restarted due to BOFF# are not re-counted.
1EH	NONCACHEABLE_ MEMORY_READS	Number of noncacheable instruction or data memory read bus cycles. The count includes read cycles caused by TLB misses, but does not include read cycles to I/O space.	Cycles restarted due to BOFF# are not re-counted.
1FH	PIPELINE_AGI_ STALLS	Number of address generation interlock (AGI) stalls. An AGI occurring in both the U- and V-pipelines in the same clock signals this event twice.	An AGI occurs when the instruction in the execute stage of either of U- or V-pipelines is writing to either the index or base address register of an instruction in the D2 (address generation) stage of either the U- or V- pipelines.
20H	Reserved		
21H	Reserved		
22H	FLOPS	Number of floating-point operations that occur.	Number of floating-point adds, subtracts, multiplies, divides, remainders, and square roots are counted. The transcendental instructions consist of multiple adds and multiplies and will signal this event multiple times. Instructions generating the divide-by-zero, negative square root, special operand, or stack exceptions will not be counted. Instructions generating all other floating-point exceptions will be counted. The integer multiply instructions and other instructions which use the x87 FPU will be counted.
23H	BREAKPOINT MATCH ON DR0 REGISTER	Number of matches on register DR0 breakpoint.	The counters is incremented regardless if the breakpoints are enabled or not. However, if breakpoints are not enabled, code breakpoint matches will not be checked for instructions executed in the V-pipe and will not cause this counter to be incremented. (They are checked on instruction executed in the U-pipe only when breakpoints are not enabled.) These events correspond to the signals driven on the BP[3:0] pins. Refer to Chapter 17, "Debug, Branch Profile, TSC, and Resource Monitoring Features" for more information.
24H	BREAKPOINT MATCH ON DR1 REGISTER	Number of matches on register DR1 breakpoint.	See comment for 23H event.

Table 19-40. Events That Can Be Counted with Pentium Processor Performance-Monitoring Counters (Contd.)

Event Num.	Mnemonic Event Name	Description	Comments
25H	BREAKPOINT MATCH ON DR2 REGISTER	Number of matches on register DR2 breakpoint.	See comment for 23H event.
26H	BREAKPOINT MATCH ON DR3 REGISTER	Number of matches on register DR3 breakpoint.	See comment for 23H event.
27H	HARDWARE INTERRUPTS	Number of taken INTR and NMI interrupts.	
28H	DATA_READ_OR_ WRITE	Number of memory data reads and/or writes (internal data cache hit and miss combined).	Split cycle reads and writes are counted individually. Data Memory Reads that are part of TLB miss processing are not included. These events may occur at a maximum of two per clock. I/O is not included.
29H	DATA_READ_MISS OR_WRITE MISS	Number of memory read and/or write accesses that miss the internal data cache, whether or not the access is cacheable or noncacheable.	Additional reads to the same cache line after the first BRDY# of the burst line fill is returned but before the final (fourth) BRDY# has been returned, will not cause the counter to be incremented additional times. Data accesses that are part of TLB miss processing are not included. Accesses directed to I/O space are not included.
2AH	BUS_OWNERSHIP_ LATENCY (Counter 0)	The time from LRM bus ownership request to bus ownership granted (that is, the time from the earlier of a PBREQ (0), PHITM# or HITM# assertion to a PBGNT assertion)	The ratio of the 2AH events counted on counter 0 and counter 1 is the average stall time due to bus ownership conflict.
2AH	BUS OWNERSHIP TRANSFERS (Counter 1)	The number of buss ownership transfers (that is, the number of PBREQ (0) assertions	The ratio of the 2AH events counted on counter 0 and counter 1 is the average stall time due to bus ownership conflict.
2BH	MMX_ INSTRUCTIONS_ EXECUTED_ U-PIPE (Counter 0)	Number of MMX instructions executed in the U-pipe	
2BH	MMX_ INSTRUCTIONS_ EXECUTED_ V-PIPE (Counter 1)	Number of MMX instructions executed in the V-pipe	
2CH	CACHE_M-STATE_LINE_ SHARING (Counter 0)	Number of times a processor identified a hit to a modified line due to a memory access in the other processor (PHITM (0))	If the average memory latencies of the system are known, this event enables the user to count the Write Backs on PHITM(0) penalty and the Latency on Hit Modified(I) penalty.
2CH	CACHE_LINE_ SHARING (Counter 1)	Number of shared data lines in the L1 cache (PHIT (0))	
2DH	EMMS_ INSTRUCTIONS_ EXECUTED (Counter 0)	Number of EMMS instructions executed	

Table 19-40. Events That Can Be Counted with Pentium Processor Performance-Monitoring Counters (Contd.)

Event Num.	Mnemonic Event Name	Description	Comments
2DH	TRANSITIONS_ BETWEEN_MMX_ AND_FP_ INSTRUCTIONS (Counter 1)	Number of transitions between MMX and floating-point instructions or vice versa An even count indicates the processor is in MMX state. an odd count indicates it is in FP state.	This event counts the first floating-point instruction following an MMX instruction or first MMX instruction following a floating-point instruction. The count may be used to estimate the penalty in transitions between floating-point state and MMX state.
2EH	BUS_UTILIZATION_ DUE_TO_ PROCESSOR_ ACTIVITY (Counter 0)	Number of clocks the bus is busy due to the processor's own activity (the bus activity that is caused by the processor)	
2EH	WRITES_TO_ NONCACHEABLE_ MEMORY (Counter 1)	Number of write accesses to noncacheable memory	The count includes write cycles caused by TLB misses and I/O write cycles. Cycles restarted due to BOFF# are not re-counted.
2FH	SATURATING_ MMX_ INSTRUCTIONS_ EXECUTED (Counter 0)	Number of saturating MMX instructions executed, independently of whether they actually saturated.	
2FH	SATURATIONS_ PERFORMED (Counter 1)	Number of MMX instructions that used saturating arithmetic when at least one of its results actually saturated	If an MMX instruction operating on 4 doublewords saturated in three out of the four results, the counter will be incremented by one only.
30H	NUMBER_OF_ CYCLES_NOT_IN_ HALT_STATE (Counter 0)	Number of cycles the processor is not idle due to HLT instruction	This event will enable the user to calculate "net CPI". Note that during the time that the processor is executing the HLT instruction, the Time-Stamp Counter is not disabled. Since this event is controlled by the Counter Controls CC0, CC1 it can be used to calculate the CPI at CPL=3, which the TSC cannot provide.
30H	DATA_CACHE_ TLB_MISS_ STALL_DURATION (Counter 1)	Number of clocks the pipeline is stalled due to a data cache translation look-aside buffer (TLB) miss	
31H	MMX_ INSTRUCTION_ DATA_READS (Counter 0)	Number of MMX instruction data reads	
31H	MMX_ INSTRUCTION_ DATA_READ_ MISSES (Counter 1)	Number of MMX instruction data read misses	
32H	FLOATING_POINT_S TALLS_DURATION (Counter 0)	Number of clocks while pipe is stalled due to a floating-point freeze	
32H	TAKEN_BRANCHES (Counter 1)	Number of taken branches	

Table 19-40. Events That Can Be Counted with Pentium Processor Performance-Monitoring Counters (Contd.)

Event Num.	Mnemonic Event Name	Description	Comments
33H	D1_STARVATION_ AND_FIFO_IS_ EMPTY (Counter 0)	Number of times D1 stage cannot issue ANY instructions since the FIFO buffer is empty	The D1 stage can issue 0, 1, or 2 instructions per clock if those are available in an instructions FIFO buffer.
33H	D1_STARVATION_ AND_ONLY_ONE_ INSTRUCTION_IN_ FIFO (Counter 1)	Number of times the D1 stage issues a single instruction (since the FIFO buffer had just one instruction ready)	The D1 stage can issue 0, 1, or 2 instructions per clock if those are available in an instructions FIFO buffer. When combined with the previously defined events, Instruction Executed (16H) and Instruction Executed in the V-pipe (17H), this event enables the user to calculate the numbers of time pairing rules prevented issuing of two instructions.
34H	MMX_ INSTRUCTION_ DATA_WRITES (Counter 0)	Number of data writes caused by MMX instructions	
34H	MMX_ INSTRUCTION_ DATA_WRITE_ MISSES (Counter 1)	Number of data write misses caused by MMX instructions	
35H	PIPELINE_ FLUSHES_DUE_ TO_WRONG_ BRANCH_ PREDICTIONS (Counter 0)	Number of pipeline flushes due to wrong branch predictions resolved in either the E-stage or the WB-stage	The count includes any pipeline flush due to a branch that the pipeline did not follow correctly. It includes cases where a branch was not in the BTB, cases where a branch was in the BTB but was mispredicted, and cases where a branch was correctly predicted but to the wrong address. Branches are resolved in either the Execute stage (E-stage) or the Writeback stage (WB-stage). In the later case, the misprediction penalty is larger by one clock. The difference between the 35H event count in counter 0 and counter 1 is the number of E-stage resolved branches.
35H	PIPELINE_ FLUSHES_DUE_ TO_WRONG_ BRANCH_ PREDICTIONS_ RESOLVED_IN_ WB-STAGE (Counter 1)	Number of pipeline flushes due to wrong branch predictions resolved in the WB-stage	See note for event 35H (Counter 0).
36H	MISALIGNED_ DATA_MEMORY_ REFERENCE_ON_ MMX_ INSTRUCTIONS (Counter 0)	Number of misaligned data memory references when executing MMX instructions	
36H	PIPELINE_ ISTALL_FOR_MMX_ INSTRUCTION_ DATA_MEMORY_ READS (Counter 1)	Number clocks during pipeline stalls caused by waits form MMX instruction data memory reads	T3:

Table 19-40. Events That Can Be Counted with Pentium Processor Performance-Monitoring Counters (Contd.)

Event Num.	Mnemonic Event Name	Description	Comments
37H	MISPREDICTED_ OR_ UNPREDICTED_ RETURNS (Counter 1)	Number of returns predicted incorrectly or not predicted at all	The count is the difference between the total number of executed returns and the number of returns that were correctly predicted. Only RET instructions are counted (for example, IRET instructions are not counted).
37H	PREDICTED_ RETURNS (Counter 1)	Number of predicted returns (whether they are predicted correctly and incorrectly	Only RET instructions are counted (for example, IRET instructions are not counted).
38H	MMX_MULTIPLY_ UNIT_INTERLOCK (Counter 0)	Number of clocks the pipe is stalled since the destination of previous MMX multiply instruction is not ready yet	The counter will not be incremented if there is another cause for a stall. For each occurrence of a multiply interlock, this event will be counted twice (if the stalled instruction comes on the next clock after the multiply) or by once (if the stalled instruction comes two clocks after the multiply).
38H	MOVD/MOVQ_ STORE_STALL_ DUE_TO_ PREVIOUS_MMX_ OPERATION (Counter 1)	Number of clocks a MOVD/MOVQ instruction store is stalled in D2 stage due to a previous MMX operation with a destination to be used in the store instruction.	
39H	RETURNS (Counter 0)	Number or returns executed.	Only RET instructions are counted; IRET instructions are not counted. Any exception taken on a RET instruction and any interrupt recognized by the processor on the instruction boundary prior to the execution of the RET instruction will also cause this counter to be incremented.
39H	Reserved		
3AH	BTB_FALSE_ ENTRIES (Counter 0)	Number of false entries in the Branch Target Buffer	False entries are causes for misprediction other than a wrong prediction.
3AH	BTB_MISS_ PREDICTION_ON_ NOT-TAKEN_ BRANCH (Counter 1)	Number of times the BTB predicted a not-taken branch as taken	
3BH	FULL_WRITE_ BUFFER_STALL_ DURATION_ WHILE_ EXECUTING_MMX_I NSTRUCTIONS (Counter 0)	Number of clocks while the pipeline is stalled due to full write buffers while executing MMX instructions	
3BH	STALL_ON_MMX_ INSTRUCTION_ WRITE_TO E-_OR_ M-STATE_LINE (Counter 1)	Number of clocks during stalls on MMX instructions writing to E- or M-state lines	

16.Updates to Chapter 20, Volume 3B

Change bars show changes to Chapter 20 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3B:* System Programming Guide, Part 2.

Changes to this chapter: Correction to Section 20.1 “Real-Address Mode”. Updates to Sections 20.3 “Interrupt and Exception Handling in Virtual-8086 Mode”, and 20.4 “Protected-Mode Virtual Interrupts”.

CHAPTER 20
8086 EMULATION

IA-32 processors (beginning with the Intel386 processor) provide two ways to execute new or legacy programs that are assembled and/or compiled to run on an Intel 8086 processor:

- Real-address mode.
- Virtual-8086 mode.

Figure 2-3 shows the relationship of these operating modes to protected mode and system management mode (SMM).

When the processor is powered up or reset, it is placed in the real-address mode. This operating mode almost exactly duplicates the execution environment of the Intel 8086 processor, with some extensions. Virtually any program assembled and/or compiled to run on an Intel 8086 processor will run on an IA-32 processor in this mode.

When running in protected mode, the processor can be switched to virtual-8086 mode to run 8086 programs. This mode also duplicates the execution environment of the Intel 8086 processor, with extensions. In virtual-8086 mode, an 8086 program runs as a separate protected-mode task. Legacy 8086 programs are thus able to run under an operating system (such as Microsoft Windows*) that takes advantage of protected mode and to use protected-mode facilities, such as the protected-mode interrupt- and exception-handling facilities. Protected-mode multitasking permits multiple virtual-8086 mode tasks (with each task running a separate 8086 program) to be run on the processor along with other non-virtual-8086 mode tasks.

This section describes both the basic real-address mode execution environment and the virtual-8086-mode execution environment, available on the IA-32 processors beginning with the Intel386 processor.

20.1 REAL-ADDRESS MODE

The IA-32 architecture's real-address mode runs programs written for the Intel 8086, Intel 8088, Intel 80186, and Intel 80188 processors, or for the real-address mode of the Intel 286, Intel386, Intel486, Pentium, P6 family, Pentium 4, and Intel Xeon processors.

The execution environment of the processor in real-address mode is designed to duplicate the execution environment of the Intel 8086 processor. To an 8086 program, a processor operating in real-address mode behaves like a high-speed 8086 processor. The principal features of this architecture are defined in Chapter 3, "Basic Execution Environment", of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*.

The following is a summary of the core features of the real-address mode execution environment as would be seen by a program written for the 8086:

- The processor supports a nominal 1-MByte physical address space (see Section 20.1.1, "Address Translation in Real-Address Mode", for specific details). This address space is divided into segments, each of which can be up to 64 KBytes in length. The base of a segment is specified with a 16-bit segment selector, which is shifted left by 4 bits to form a 20-bit offset from address 0 in the address space. An operand within a segment is addressed with a 16-bit offset from the base of the segment. A physical address is thus formed by adding the offset to the 20-bit segment base (see Section 20.1.1, "Address Translation in Real-Address Mode").
- All operands in "native 8086 code" are 8-bit or 16-bit values. (Operand size override prefixes can be used to access 32-bit operands.)
- Eight 16-bit general-purpose registers are provided: AX, BX, CX, DX, SP, BP, SI, and DI. The extended 32 bit registers (EAX, EBX, ECX, EDX, ESP, EBP, ESI, and EDI) are accessible to programs that explicitly perform a size override operation.
- Four segment registers are provided: CS, DS, SS, and ES. (The FS and GS registers are accessible to programs that explicitly access them.) The CS register contains the segment selector for the code segment; the DS and ES registers contain segment selectors for data segments; and the SS register contains the segment selector for the stack segment.
- The 8086 16-bit instruction pointer (IP) is mapped to the lower 16-bits of the EIP register. Note this register is a 32-bit register and unintentional address wrapping may occur.

- The 16-bit FLAGS register contains status and control flags. (This register is mapped to the 16 least significant bits of the 32-bit EFLAGS register.)
- All of the Intel 8086 instructions are supported (see Section 20.1.3, "Instructions Supported in Real-Address Mode").
- A single, 16-bit-wide stack is provided for handling procedure calls and invocations of interrupt and exception handlers. This stack is contained in the stack segment identified with the SS register. The SP (stack pointer) register contains an offset into the stack segment. The stack grows down (toward lower segment offsets) from the stack pointer. The BP (base pointer) register also contains an offset into the stack segment that can be used as a pointer to a parameter list. When a CALL instruction is executed, the processor pushes the current instruction pointer (the 16 least-significant bits of the EIP register and, on far calls, the current value of the CS register) onto the stack. On a return, initiated with a RET instruction, the processor pops the saved instruction pointer from the stack into the EIP register (and CS register on far returns). When an implicit call to an interrupt or exception handler is executed, the processor pushes the EIP, CS, and EFLAGS (low-order 16-bits only) registers onto the stack. On a return from an interrupt or exception handler, initiated with an IRET instruction, the processor pops the saved instruction pointer and EFLAGS image from the stack into the EIP, CS, and EFLAGS registers.
- A single interrupt table, called the "interrupt vector table" or "interrupt table," is provided for handling interrupts and exceptions (see Figure 20-2). The interrupt table (which has 4-byte entries) takes the place of the interrupt descriptor table (IDT, with 8-byte entries) used when handling protected-mode interrupts and exceptions. Interrupt and exception vector numbers provide an index to entries in the interrupt table. Each entry provides a pointer (called a "vector") to an interrupt- or exception-handling procedure. See Section 20.1.4, "Interrupt and Exception Handling", for more details. It is possible for software to relocate the IDT by means of the LIDT instruction on IA-32 processors beginning with the Intel386 processor.
- The x87 FPU is active and available to execute x87 FPU instructions in real-address mode. Programs written to run on the Intel 8087 and Intel 287 math coprocessors can be run in real-address mode without modification.

The following extensions to the Intel 8086 execution environment are available in the IA-32 architecture's real-address mode. If backwards compatibility to Intel 286 and Intel 8086 processors is required, these features should not be used in new programs written to run in real-address mode.

- Two additional segment registers (FS and GS) are available.
- Many of the integer and system instructions that have been added to later IA-32 processors can be executed in real-address mode (see Section 20.1.3, "Instructions Supported in Real-Address Mode").
- The 32-bit operand prefix can be used in real-address mode programs to execute the 32-bit forms of instructions. This prefix also allows real-address mode programs to use the processor's 32-bit general-purpose registers.
- The 32-bit address prefix can be used in real-address mode programs, allowing 32-bit offsets.

The following sections describe address formation, registers, available instructions, and interrupt and exception handling in real-address mode. For information on I/O in real-address mode, see Chapter 18, "Input/Output", of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*.

20.1.1 Address Translation in Real-Address Mode

In real-address mode, the processor does not interpret segment selectors as indexes into a descriptor table; instead, it uses them directly to form linear addresses as the 8086 processor does. It shifts the segment selector left by 4 bits to form a 20-bit base address (see Figure 20-1). The offset into a segment is added to the base address to create a linear address that maps directly to the physical address space.

When using 8086-style address translation, it is possible to specify addresses larger than 1 MByte. For example, with a segment selector value of FFFFH and an offset of FFFFH, the linear (and physical) address would be 10FFEFH (1 megabyte plus 64 KBytes). The 8086 processor, which can form addresses only up to 20 bits long, truncates the high-order bit, thereby "wrapping" this address to FFEFH. When operating in real-address mode, however, the processor does not truncate such an address and uses it as a physical address. (Note, however, that for IA-32 processors beginning with the Intel486 processor, the A20M# signal can be used in real-address mode to mask address line A20, thereby mimicking the 20-bit wrap-around behavior of the 8086 processor.) Care should be take to ensure that A20M# based address wrapping is handled correctly in multiprocessor based system.

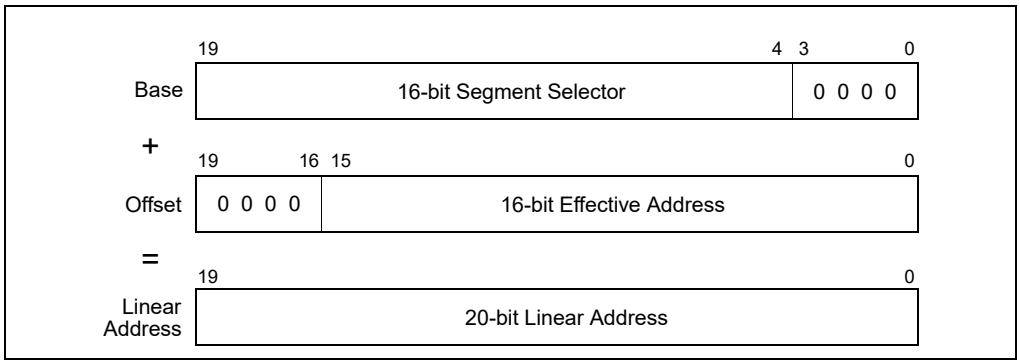


Figure 20-1. Real-Address Mode Address Translation

The IA-32 processors beginning with the Intel386 processor can generate 32-bit offsets using an address override prefix; however, in real-address mode, the value of a 32-bit offset may not exceed FFFFH without causing an exception.

For full compatibility with Intel 286 real-address mode, pseudo-protection faults (interrupt 12 or 13) occur if a 32-bit offset is generated outside the range 0 through FFFFH.

20.1.2 Registers Supported in Real-Address Mode

The register set available in real-address mode includes all the registers defined for the 8086 processor plus the new registers introduced in later IA-32 processors, such as the FS and GS segment registers, the debug registers, the control registers, and the floating-point unit registers. The 32-bit operand prefix allows a real-address mode program to use the 32-bit general-purpose registers (EAX, EBX, ECX, EDX, ESP, EBP, ESI, and EDI).

20.1.3 Instructions Supported in Real-Address Mode

The following instructions make up the core instruction set for the 8086 processor. If backwards compatibility to the Intel 286 and Intel 8086 processors is required, only these instructions should be used in a new program written to run in real-address mode.

- Move (MOV) instructions that move operands between general-purpose registers, segment registers, and between memory and general-purpose registers.
- The exchange (XCHG) instruction.
- Load segment register instructions LDS and LES.
- Arithmetic instructions ADD, ADC, SUB, SBB, MUL, IMUL, DIV, IDIV, INC, DEC, CMP, and NEG.
- Logical instructions AND, OR, XOR, and NOT.
- Decimal instructions DAA, DAS, AAA, AAS, AAM, and AAD.
- Stack instructions PUSH and POP (to general-purpose registers and segment registers).
- Type conversion instructions CWD, CDQ, CBW, and CWDE.
- Shift and rotate instructions SAL, SHL, SHR, SAR, ROL, ROR, RCL, and RCR.
- TEST instruction.
- Control instructions JMP, J*cc*, CALL, RET, LOOP, LOOPE, and LOOPNE.
- Interrupt instructions INT *n*, INTO, and IRET.
- EFLAGS control instructions STC, CLC, CMC, CLD, STD, LAHF, SAHF, PUSHF, and POPF.
- I/O instructions IN, INS, OUT, and OUTS.
- Load effective address (LEA) instruction, and translate (XLATB) instruction.

- LOCK prefix.
- Repeat prefixes REP, REPE, REPZ, REPNE, and REPNZ.
- Processor halt (HLT) instruction.
- No operation (NOP) instruction.

The following instructions, added to later IA-32 processors (some in the Intel 286 processor and the remainder in the Intel386 processor), can be executed in real-address mode, if backwards compatibility to the Intel 8086 processor is not required.

- Move (MOV) instructions that operate on the control and debug registers.
- Load segment register instructions LSS, LFS, and LGS.
- Generalized multiply instructions and multiply immediate data.
- Shift and rotate by immediate counts.
- Stack instructions PUSHA, PUSHAD, POPA and POPAD, and PUSH immediate data.
- Move with sign extension instructions MOVSX and MOVZX.
- Long-displacement J*cc* instructions.
- Exchange instructions CMPXCHG, CMPXCHG8B, and XADD.
- String instructions MOVS, CMPS, SCAS, LODS, and STOS.
- Bit test and bit scan instructions BT, BTS, BTR, BTC, BSF, and BSR; the byte-set-on condition instruction SET*cc*; and the byte swap (BSWAP) instruction.
- Double shift instructions SHLD and SHRD.
- EFLAGS control instructions PUSHF and POPF.
- ENTER and LEAVE control instructions.
- BOUND instruction.
- CPU identification (CPUID) instruction.
- System instructions CLTS, INVD, WINVD, INVLPG, LGDT, SGDT, LIDT, SIDT, LMSW, SMSW, RDMSR, WRMSR, RDTSC, and RDPMC.

Execution of any of the other IA-32 architecture instructions (not given in the previous two lists) in real-address mode result in an invalid-opcode exception (#UD) being generated.

20.1.4 Interrupt and Exception Handling

When operating in real-address mode, software must provide interrupt and exception-handling facilities that are separate from those provided in protected mode. Even during the early stages of processor initialization when the processor is still in real-address mode, elementary real-address mode interrupt and exception-handling facilities must be provided to insure reliable operation of the processor, or the initialization code must insure that no interrupts or exceptions will occur.

The IA-32 processors handle interrupts and exceptions in real-address mode similar to the way they handle them in protected mode. When a processor receives an interrupt or generates an exception, it uses the vector number of the interrupt or exception as an index into the interrupt table. (In protected mode, the interrupt table is called the **interrupt descriptor table (IDT)**, but in real-address mode, the table is usually called the **interrupt vector table**, or simply the **interrupt table**.) The entry in the interrupt vector table provides a pointer to an interrupt- or exception-handler procedure. (The pointer consists of a segment selector for a code segment and a 16-bit offset into the segment.) The processor performs the following actions to make an implicit call to the selected handler:

1. Pushes the current values of the CS and EIP registers onto the stack. (Only the 16 least-significant bits of the EIP register are pushed.)
2. Pushes the low-order 16 bits of the EFLAGS register onto the stack.
3. Clears the IF flag in the EFLAGS register to disable interrupts.
4. Clears the TF, RF, and AC flags, in the EFLAGS register.

5. Transfers program control to the location specified in the interrupt vector table.

An IRET instruction at the end of the handler procedure reverses these steps to return program control to the interrupted program. Exceptions do not return error codes in real-address mode.

The interrupt vector table is an array of 4-byte entries (see Figure 20-2). Each entry consists of a far pointer to a handler procedure, made up of a segment selector and an offset. The processor scales the interrupt or exception vector by 4 to obtain an offset into the interrupt table. Following reset, the base of the interrupt vector table is located at physical address 0 and its limit is set to 3FFH. In the Intel 8086 processor, the base address and limit of the interrupt vector table cannot be changed. In the later IA-32 processors, the base address and limit of the interrupt vector table are contained in the IDTR register and can be changed using the LIDT instruction.

(For backward compatibility to Intel 8086 processors, the default base address and limit of the interrupt vector table should not be changed.)

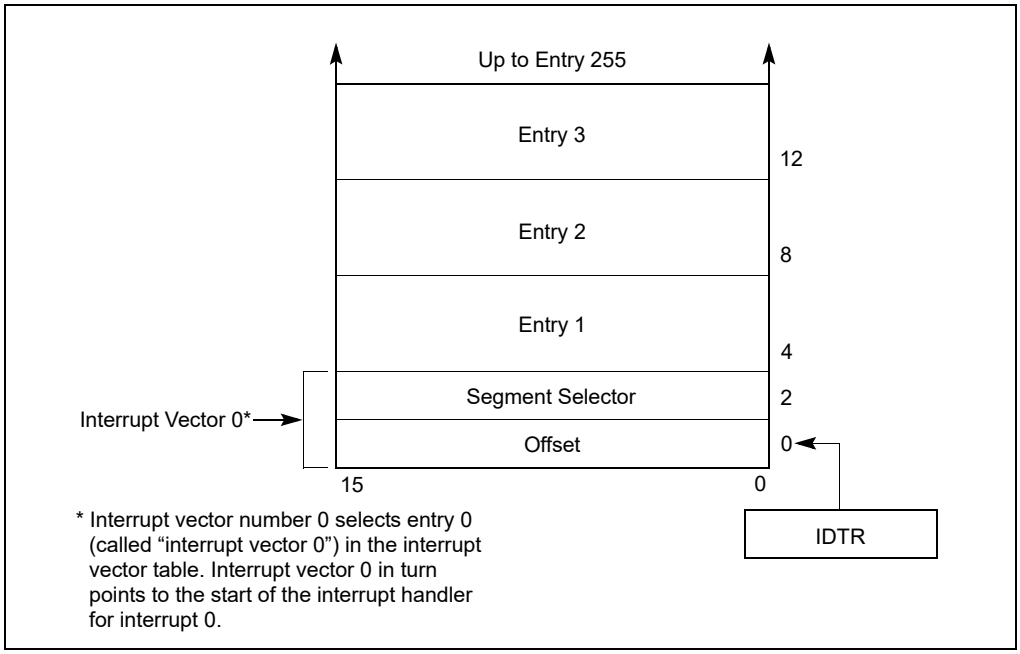


Figure 20-2. Interrupt Vector Table in Real-Address Mode

Table 20-1 shows the interrupt and exception vectors that can be generated in real-address mode and virtual-8086 mode, and in the Intel 8086 processor. See Chapter 6, “Interrupt and Exception Handling”, for a description of the exception conditions.

20.2 VIRTUAL-8086 MODE

Virtual-8086 mode is actually a special type of a task that runs in protected mode. When the operating-system or executive switches to a virtual-8086-mode task, the processor emulates an Intel 8086 processor. The execution environment of the processor while in the 8086-emulation state is the same as is described in Section 20.1, “Real-Address Mode” for real-address mode, including the extensions. The major difference between the two modes is that in virtual-8086 mode the 8086 emulator uses some protected-mode services (such as the protected-mode interrupt and exception-handling and paging facilities).

As in real-address mode, any new or legacy program that has been assembled and/or compiled to run on an Intel 8086 processor will run in a virtual-8086-mode task. And several 8086 programs can be run as virtual-8086-mode tasks concurrently with normal protected-mode tasks, using the processor's multitasking facilities.

Table 20-1. Real-Address Mode Exceptions and Interrupts

Vector No.	Description	Real-Address Mode	Virtual-8086 Mode	Intel 8086 Processor
0	Divide Error (#DE)	Yes	Yes	Yes
1	Debug Exception (#DB)	Yes	Yes	No
2	NMI Interrupt	Yes	Yes	Yes
3	Breakpoint (#BP)	Yes	Yes	Yes
4	Overflow (#OF)	Yes	Yes	Yes
5	BOUND Range Exceeded (#BR)	Yes	Yes	Reserved
6	Invalid Opcode (#UD)	Yes	Yes	Reserved
7	Device Not Available (#NM)	Yes	Yes	Reserved
8	Double Fault (#DF)	Yes	Yes	Reserved
9	(Intel reserved. Do not use.)	Reserved	Reserved	Reserved
10	Invalid TSS (#TS)	Reserved	Yes	Reserved
11	Segment Not Present (#NP)	Reserved	Yes	Reserved
12	Stack Fault (#SS)	Yes	Yes	Reserved
13	General Protection (#GP)*	Yes	Yes	Reserved
14	Page Fault (#PF)	Reserved	Yes	Reserved
15	(Intel reserved. Do not use.)	Reserved	Reserved	Reserved
16	Floating-Point Error (#MF)	Yes	Yes	Reserved
17	Alignment Check (#AC)	Reserved	Yes	Reserved
18	Machine Check (#MC)	Yes	Yes	Reserved
19-31	(Intel reserved. Do not use.)	Reserved	Reserved	Reserved
32-255	User Defined Interrupts	Yes	Yes	Yes

NOTE:

* In the real-address mode, vector 13 is the segment overrun exception. In protected and virtual-8086 modes, this exception covers all general-protection error conditions, including traps to the virtual-8086 monitor from virtual-8086 mode.

20.2.1 Enabling Virtual-8086 Mode

The processor runs in virtual-8086 mode when the VM (virtual machine) flag in the EFLAGS register is set. This flag can only be set when the processor switches to a new protected-mode task or resumes virtual-8086 mode via an IRET instruction.

System software cannot change the state of the VM flag directly in the EFLAGS register (for example, by using the POPFD instruction). Instead it changes the flag in the image of the EFLAGS register stored in the TSS or on the stack following a call to an interrupt- or exception-handler procedure. For example, software sets the VM flag in the EFLAGS image in the TSS when first creating a virtual-8086 task.

The processor tests the VM flag under three general conditions:

- When loading segment registers, to determine whether to use 8086-style address translation.
- When decoding instructions, to determine which instructions are not supported in virtual-8086 mode and which instructions are sensitive to IOPL.

- When checking privileged instructions, on page accesses, or when performing other permission checks. (Virtual-8086 mode always executes at CPL 3.)

20.2.2 Structure of a Virtual-8086 Task

A virtual-8086-mode task consists of the following items:

- A 32-bit TSS for the task.
- The 8086 program.
- A virtual-8086 monitor.
- 8086 operating-system services.

The TSS of the new task must be a 32-bit TSS, not a 16-bit TSS, because the 16-bit TSS does not load the most-significant word of the EFLAGS register, which contains the VM flag. All TSS's, stacks, data, and code used to handle exceptions when in virtual-8086 mode must also be 32-bit segments.

The processor enters virtual-8086 mode to run the 8086 program and returns to protected mode to run the virtual-8086 monitor.

The virtual-8086 monitor is a 32-bit protected-mode code module that runs at a CPL of 0. The monitor consists of initialization, interrupt- and exception-handling, and I/O emulation procedures that emulate a personal computer or other 8086-based platform. Typically, the monitor is either part of or closely associated with the protected-mode general-protection (#GP) exception handler, which also runs at a CPL of 0. As with any protected-mode code module, code-segment descriptors for the virtual-8086 monitor must exist in the GDT or in the task's LDT. The virtual-8086 monitor also may need data-segment descriptors so it can examine the IDT or other parts of the 8086 program in the first 1 MByte of the address space. The linear addresses above 10FFEFH are available for the monitor, the operating system, and other system software.

The 8086 operating-system services consists of a kernel and/or operating-system procedures that the 8086 program makes calls to. These services can be implemented in either of the following two ways:

- They can be included in the 8086 program. This approach is desirable for either of the following reasons:
  - The 8086 program code modifies the 8086 operating-system services.
  - There is not sufficient development time to merge the 8086 operating-system services into main operating system or executive.
- They can be implemented or emulated in the virtual-8086 monitor. This approach is desirable for any of the following reasons:
  - The 8086 operating-system procedures can be more easily coordinated among several virtual-8086 tasks.
  - Memory can be saved by not duplicating 8086 operating-system procedure code for several virtual-8086 tasks.
  - The 8086 operating-system procedures can be easily emulated by calls to the main operating system or executive.

The approach chosen for implementing the 8086 operating-system services may result in different virtual-8086-mode tasks using different 8086 operating-system services.

20.2.3 Paging of Virtual-8086 Tasks

Even though a program running in virtual-8086 mode can use only 20-bit linear addresses, the processor converts these addresses into 32-bit linear addresses before mapping them to the physical address space. If paging is being used, the 8086 address space for a program running in virtual-8086 mode can be paged and located in a set of pages in physical address space. If paging is used, it is transparent to the program running in virtual-8086 mode just as it is for any task running on the processor.

Paging is not necessary for a single virtual-8086-mode task, but paging is useful or necessary in the following situations:

- When running multiple virtual-8086-mode tasks. Here, paging allows the lower 1 MByte of the linear address space for each virtual-8086-mode task to be mapped to a different physical address location.
- When emulating the 8086 address-wraparound that occurs at 1 MByte. When using 8086-style address translation, it is possible to specify addresses larger than 1 MByte. These addresses automatically wraparound in the Intel 8086 processor (see Section 20.1.1, "Address Translation in Real-Address Mode"). If any 8086 programs depend on address wraparound, the same effect can be achieved in a virtual-8086-mode task by mapping the linear addresses between 100000H and 110000H and linear addresses between 0 and 10000H to the same physical addresses.
- When sharing the 8086 operating-system services or ROM code that is common to several 8086 programs running as different 8086-mode tasks.
- When redirecting or trapping references to memory-mapped I/O devices.

20.2.4 Protection within a Virtual-8086 Task

Protection is not enforced between the segments of an 8086 program. Either of the following techniques can be used to protect the system software running in a virtual-8086-mode task from the 8086 program:

- Reserve the first 1 MByte plus 64 KBytes of each task's linear address space for the 8086 program. An 8086 processor task cannot generate addresses outside this range.
- Use the U/S flag of page-table entries to protect the virtual-8086 monitor and other system software in the virtual-8086 mode task space. When the processor is in virtual-8086 mode, the CPL is 3. Therefore, an 8086 processor program has only user privileges. If the pages of the virtual-8086 monitor have supervisor privilege, they cannot be accessed by the 8086 program.

20.2.5 Entering Virtual-8086 Mode

Figure 20-3 summarizes the methods of entering and leaving virtual-8086 mode. The processor switches to virtual-8086 mode in either of the following situations:

- Task switch when the VM flag is set to 1 in the EFLAGS register image stored in the TSS for the task. Here the task switch can be initiated in either of two ways:
  - A CALL or JMP instruction.
  - An IRET instruction, where the NT flag in the EFLAGS image is set to 1.
- Return from a protected-mode interrupt or exception handler when the VM flag is set to 1 in the EFLAGS register image on the stack.

When a task switch is used to enter virtual-8086 mode, the TSS for the virtual-8086-mode task must be a 32-bit TSS. (If the new TSS is a 16-bit TSS, the upper word of the EFLAGS register is not in the TSS, causing the processor to clear the VM flag when it loads the EFLAGS register.) The processor updates the VM flag prior to loading the segment registers from their images in the new TSS. The new setting of the VM flag determines whether the processor interprets the contents of the segment registers as 8086-style segment selectors or protected-mode segment selectors. When the VM flag is set, the segment registers are loaded from the TSS, using 8086-style address translation to form base addresses.

See Section 20.3, "Interrupt and Exception Handling in Virtual-8086 Mode", for information on entering virtual-8086 mode on a return from an interrupt or exception handler.

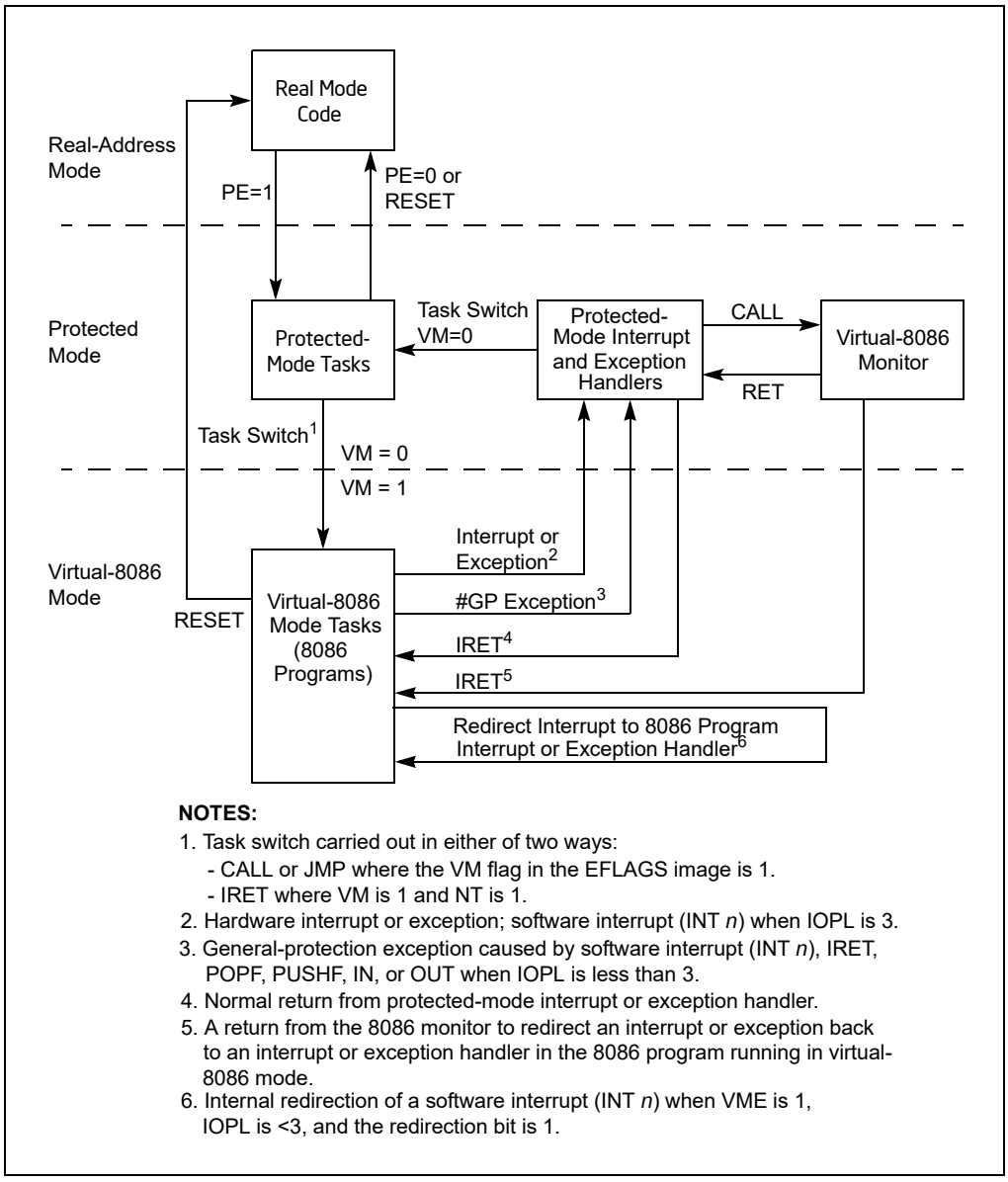


Figure 20-3. Entering and Leaving Virtual-8086 Mode

20.2.6 Leaving Virtual-8086 Mode

The processor can leave the virtual-8086 mode only through an interrupt or exception. The following are situations where an interrupt or exception will lead to the processor leaving virtual-8086 mode (see Figure 20-3):

- The processor services a hardware interrupt generated to signal the suspension of execution of the virtual-8086 application. This hardware interrupt may be generated by a timer or other external mechanism. Upon receiving the hardware interrupt, the processor enters protected mode and switches to a protected-mode (or another virtual-8086 mode) task either through a task gate in the protected-mode IDT or through a trap or interrupt gate that points to a handler that initiates a task switch. A task switch from a virtual-8086 task to another task loads the EFLAGS register from the TSS of the new task. The value of the VM flag in the new EFLAGS determines if the new task executes in virtual-8086 mode or not.
- The processor services an exception caused by code executing the virtual-8086 task or services a hardware interrupt that "belongs to" the virtual-8086 task. Here, the processor enters protected mode and services the

exception or hardware interrupt through the protected-mode IDT (normally through an interrupt or trap gate) and the protected-mode exception- and interrupt-handlers. The processor may handle the exception or interrupt within the context of the virtual 8086 task and return to virtual-8086 mode on a return from the handler procedure. The processor may also execute a task switch and handle the exception or interrupt in the context of another task.

- The processor services a software interrupt generated by code executing in the virtual-8086 task (such as a software interrupt to call a MS-DOS* operating system routine). The processor provides several methods of handling these software interrupts, which are discussed in detail in Section 20.3.3, "Class 3—Software Interrupt Handling in Virtual-8086 Mode". Most of them involve the processor entering protected mode, often by means of a general-protection (#GP) exception. In protected mode, the processor can send the interrupt to the virtual-8086 monitor for handling and/or redirect the interrupt back to the application program running in virtual-8086 mode task for handling.

  IA-32 processors that incorporate the virtual mode extension (enabled with the VME flag in control register CR4) are capable of redirecting software-generated interrupts back to the program's interrupt handlers without leaving virtual-8086 mode. See Section 20.3.3.4, "Method 5: Software Interrupt Handling", for more information on this mechanism.
- A hardware reset initiated by asserting the RESET or INIT pin is a special kind of interrupt. When a RESET or INIT is signaled while the processor is in virtual-8086 mode, the processor leaves virtual-8086 mode and enters real-address mode.
- Execution of the HLT instruction in virtual-8086 mode will cause a general-protection (GP#) fault, which the protected-mode handler generally sends to the virtual-8086 monitor. The virtual-8086 monitor then determines the correct execution sequence after verifying that it was entered as a result of a HLT execution.

See Section 20.3, "Interrupt and Exception Handling in Virtual-8086 Mode", for information on leaving virtual-8086 mode to handle an interrupt or exception generated in virtual-8086 mode.

20.2.7 Sensitive Instructions

When an IA-32 processor is running in virtual-8086 mode, the CLI, STI, PUSHF, POPF, INT *n*, and IRET instructions are sensitive to IOPL. The IN, INS, OUT, and OUTS instructions, which are sensitive to IOPL in protected mode, are not sensitive in virtual-8086 mode.

The CPL is always 3 while running in virtual-8086 mode; if the IOPL is less than 3, an attempt to use the IOPL-sensitive instructions listed above triggers a general-protection exception (#GP). These instructions are sensitive to IOPL to give the virtual-8086 monitor a chance to emulate the facilities they affect.

20.2.8 Virtual-8086 Mode I/O

Many 8086 programs written for non-multitasking systems directly access I/O ports. This practice may cause problems in a multitasking environment. If more than one program accesses the same port, they may interfere with each other. Most multitasking systems require application programs to access I/O ports through the operating system. This results in simplified, centralized control.

The processor provides I/O protection for creating I/O that is compatible with the environment and transparent to 8086 programs. Designers may take any of several possible approaches to protecting I/O ports:

- Protect the I/O address space and generate exceptions for all attempts to perform I/O directly.
- Let the 8086 program perform I/O directly.
- Generate exceptions on attempts to access specific I/O ports.
- Generate exceptions on attempts to access specific memory-mapped I/O ports.

The method of controlling access to I/O ports depends upon whether they are I/O-port mapped or memory mapped.

20.2.8.1 I/O-Port-Mapped I/O

The I/O permission bit map in the TSS can be used to generate exceptions on attempts to access specific I/O port addresses. The I/O permission bit map of each virtual-8086-mode task determines which I/O addresses generate exceptions for that task. Because each task may have a different I/O permission bit map, the addresses that generate exceptions for one task may be different from the addresses for another task. This differs from protected mode in which, if the CPL is less than or equal to the IOPL, I/O access is allowed without checking the I/O permission bit map. See Chapter 18, "Input/Output", in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*, for more information about the I/O permission bit map.

20.2.8.2 Memory-Mapped I/O

In systems which use memory-mapped I/O, the paging facilities of the processor can be used to generate exceptions for attempts to access I/O ports. The virtual-8086 monitor may use paging to control memory-mapped I/O in these ways:

- Map part of the linear address space of each task that needs to perform I/O to the physical address space where I/O ports are placed. By putting the I/O ports at different addresses (in different pages), the paging mechanism can enforce isolation between tasks.
- Map part of the linear address space to pages that are not-present. This generates an exception whenever a task attempts to perform I/O to those pages. System software then can interpret the I/O operation being attempted.

Software emulation of the I/O space may require too much operating system intervention under some conditions. In these cases, it may be possible to generate an exception for only the first attempt to access I/O. The system software then may determine whether a program can be given exclusive control of I/O temporarily, the protection of the I/O space may be lifted, and the program allowed to run at full speed.

20.2.8.3 Special I/O Buffers

Buffers of intelligent controllers (for example, a bit-mapped frame buffer) also can be emulated using page mapping. The linear space for the buffer can be mapped to a different physical space for each virtual-8086-mode task. The virtual-8086 monitor then can control which virtual buffer to copy onto the real buffer in the physical address space.

20.3 INTERRUPT AND EXCEPTION HANDLING IN VIRTUAL-8086 MODE

When the processor receives an interrupt or detects an exception condition while in virtual-8086 mode, it invokes an interrupt or exception handler, just as it does in protected or real-address mode. The interrupt or exception handler that is invoked and the mechanism used to invoke it depends on the class of interrupt or exception that has been detected or generated and the state of various system flags and fields.

In virtual-8086 mode, the interrupts and exceptions are divided into three classes for the purposes of handling:

- **Class 1** — All processor-generated exceptions and all hardware interrupts, including the NMI interrupt and the hardware interrupts sent to the processor's external interrupt delivery pins. All class 1 exceptions and interrupts are handled by the protected-mode exception and interrupt handlers.
- **Class 2** — Special case for maskable hardware interrupts (Section 6.3.2, "Maskable Hardware Interrupts") when the virtual mode extensions are enabled.
- **Class 3** — All software-generated interrupts, that is interrupts generated with the INT *n* instruction[1].

The method the processor uses to handle class 2 and 3 interrupts depends on the setting of the following flags and fields:

- **IOPL field (bits 12 and 13 in the EFLAGS register)** — Controls how class 3 software interrupts are handled when the processor is in virtual-8086 mode (see Section 2.3, "System Flags and Fields in the EFLAGS

1. The INT 3 instruction is a special case (see the description of the INT *n* instruction in Chapter 3, "Instruction Set Reference, A-L", of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A*).

Register"). This field also controls the enabling of the VIF and VIP flags in the EFLAGS register when the VME flag is set. The VIF and VIP flags are provided to assist in the handling of class 2 maskable hardware interrupts.

- **VME flag (bit 0 in control register CR4)** — Enables the virtual mode extension for the processor when set (see Section 2.5, "Control Registers").
- **Software interrupt redirection bit map (32 bytes in the TSS, see Figure 20-5)** — Contains 256 flags that indicates how class 3 software interrupts should be handled when they occur in virtual-8086 mode. A software interrupt can be directed either to the interrupt and exception handlers in the currently running 8086 program or to the protected-mode interrupt and exception handlers.
- **The virtual interrupt flag (VIF) and virtual interrupt pending flag (VIP) in the EFLAGS register** — Provides **virtual interrupt support** for the handling of class 2 maskable hardware interrupts (see Section 20.3.2, "Class 2—Maskable Hardware Interrupt Handling in Virtual-8086 Mode Using the Virtual Interrupt Mechanism").

NOTE

The VME flag, software interrupt redirection bit map, and VIF and VIP flags are only available in IA-32 processors that support the virtual mode extensions. These extensions were introduced in the IA-32 architecture with the Pentium processor.

The following sections describe the actions that processor takes and the possible actions of interrupt and exception handlers for the two classes of interrupts described in the previous paragraphs. These sections describe three possible types of interrupt and exception handlers:

- **Protected-mode interrupt and exceptions handlers** — These are the standard handlers that the processor calls through the protected-mode IDT.
- **Virtual-8086 monitor interrupt and exception handlers** — These handlers are resident in the virtual-8086 monitor, and they are commonly accessed through a general-protection exception (#GP, interrupt 13) that is directed to the protected-mode general-protection exception handler.
- **8086 program interrupt and exception handlers** — These handlers are part of the 8086 program that is running in virtual-8086 mode.

The following sections describe how these handlers are used, depending on the selected class and method of interrupt and exception handling.

20.3.1 Class 1—Hardware Interrupt and Exception Handling in Virtual-8086 Mode

In virtual-8086 mode, the Pentium, P6 family, Pentium 4, and Intel Xeon processors handle hardware interrupts and exceptions in the same manner as they are handled by the Intel486 and Intel386 processors. They invoke the protected-mode interrupt or exception handler that the interrupt or exception vector points to in the IDT. Here, the IDT entry must contain either a 32-bit trap or interrupt gate or a task gate. The following sections describe various ways that a virtual-8086 mode interrupt or exception can be handled after the protected-mode handler has been invoked.

See Section 20.3.2, "Class 2—Maskable Hardware Interrupt Handling in Virtual-8086 Mode Using the Virtual Interrupt Mechanism", for a description of the virtual interrupt mechanism that is available for handling maskable hardware interrupts while in virtual-8086 mode. When this mechanism is either not available or not enabled, maskable hardware interrupts are handled in the same manner as exceptions, as described in the following sections.

20.3.1.1 Handling an Interrupt or Exception Through a Protected-Mode Trap or Interrupt Gate

When an interrupt or exception vector points to a 32-bit trap or interrupt gate in the IDT, the gate must in turn point to a nonconforming, privilege-level 0, code segment. When accessing this code segment, processor performs the following steps.

1. Switches to 32-bit protected mode and privilege level 0.
2. Saves the state of the processor on the privilege-level 0 stack. The states of the EIP, CS, EFLAGS, ESP, SS, ES, DS, FS, and GS registers are saved (see Figure 20-4).

3. Clears the segment registers. Saving the DS, ES, FS, and GS registers on the stack and then clearing the registers lets the interrupt or exception handler safely save and restore these registers regardless of the type segment selectors they contain (protected-mode or 8086-style). The interrupt and exception handlers, which may be called in the context of either a protected-mode task or a virtual-8086-mode task, can use the same code sequences for saving and restoring the registers for any task. Clearing these registers before execution of the IRET instruction does not cause a trap in the interrupt handler. Interrupt procedures that expect values in the segment registers or that return values in the segment registers must use the register images saved on the stack for privilege level 0.
4. Clears VM, NT, RF and TF flags (in the EFLAGS register). If the gate is an interrupt gate, clears the IF flag.
5. Begins executing the selected interrupt or exception handler.

If the trap or interrupt gate references a procedure in a conforming segment or in a segment at a privilege level other than 0, the processor generates a general-protection exception (#GP). Here, the error code is the segment selector of the code segment to which a call was attempted.

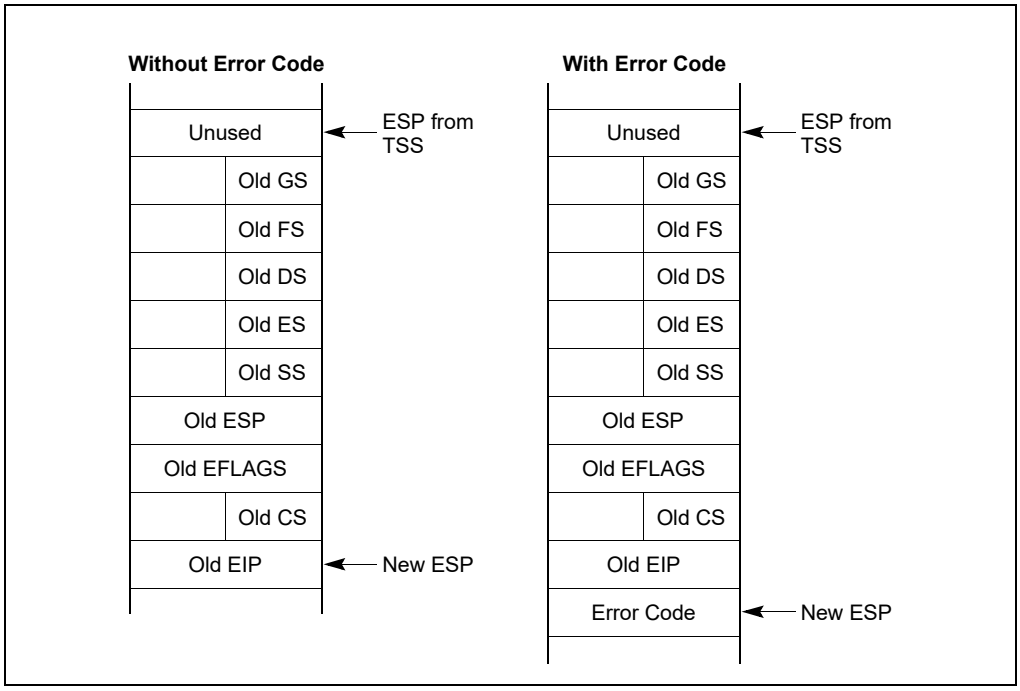


Figure 20-4. Privilege Level 0 Stack After Interrupt or Exception in Virtual-8086 Mode

Interrupt and exception handlers can examine the VM flag on the stack to determine if the interrupted procedure was running in virtual-8086 mode. If so, the interrupt or exception can be handled in one of three ways:

- The protected-mode interrupt or exception handler that was called can handle the interrupt or exception.
- The protected-mode interrupt or exception handler can call the virtual-8086 monitor to handle the interrupt or exception.
- The virtual-8086 monitor (if called) can in turn pass control back to the 8086 program's interrupt and exception handler.

If the interrupt or exception is handled with a protected-mode handler, the handler can return to the interrupted program in virtual-8086 mode by executing an IRET instruction. This instruction loads the EFLAGS and segment registers from the images saved in the privilege level 0 stack (see Figure 20-4). A set VM flag in the EFLAGS image causes the processor to switch back to virtual-8086 mode. The CPL at the time the IRET instruction is executed must be 0, otherwise the processor does not change the state of the VM flag.

The virtual-8086 monitor runs at privilege level 0, like the protected-mode interrupt and exception handlers. It is commonly closely tied to the protected-mode general-protection exception (#GP, vector 13) handler. If the protected-mode interrupt or exception handler calls the virtual-8086 monitor to handle the interrupt or exception, the return from the virtual-8086 monitor to the interrupted virtual-8086 mode program requires two return instructions: a RET instruction to return to the protected-mode handler and an IRET instruction to return to the interrupted program.

The virtual-8086 monitor has the option of directing the interrupt and exception back to an interrupt or exception handler that is part of the interrupted 8086 program, as described in Section 20.3.1.2, "Handling an Interrupt or Exception With an 8086 Program Interrupt or Exception Handler".

20.3.1.2 Handling an Interrupt or Exception With an 8086 Program Interrupt or Exception Handler

Because it was designed to run on an 8086 processor, an 8086 program running in a virtual-8086-mode task contains an 8086-style interrupt vector table, which starts at linear address 0. If the virtual-8086 monitor correctly directs an interrupt or exception vector back to the virtual-8086-mode task it came from, the handlers in the 8086 program can handle the interrupt or exception. The virtual-8086 monitor must carry out the following steps to send an interrupt or exception back to the 8086 program:

1. Use the 8086 interrupt vector to locate the appropriate handler procedure in the 8086 program interrupt table.
2. Store the EFLAGS (low-order 16 bits only), CS and EIP values of the 8086 program on the privilege-level 3 stack. This is the stack that the virtual-8086-mode task is using. (The 8086 handler may use or modify this information.)
3. Change the return link on the privilege-level 0 stack to point to the privilege-level 3 handler procedure.
4. Execute an IRET instruction to pass control to the 8086 program handler.
5. When the IRET instruction from the privilege-level 3 handler triggers a general-protection exception (#GP) and thus effectively again calls the virtual-8086 monitor, restore the return link on the privilege-level 0 stack to point to the original, interrupted, privilege-level 3 procedure.
6. Copy the low order 16 bits of the EFLAGS image from the privilege-level 3 stack to the privilege-level 0 stack (because some 8086 handlers modify these flags to return information to the code that caused the interrupt).
7. Execute an IRET instruction to pass control back to the interrupted 8086 program.

Note that if an operating system intends to support all 8086 MS-DOS-based programs, it is necessary to use the actual 8086 interrupt and exception handlers supplied with the program. The reason for this is that some programs modify their own interrupt vector table to substitute (or hook in series) their own specialized interrupt and exception handlers.

20.3.1.3 Handling an Interrupt or Exception Through a Task Gate

When an interrupt or exception vector points to a task gate in the IDT, the processor performs a task switch to the selected interrupt- or exception-handling task. The following actions are carried out as part of this task switch:

1. The EFLAGS register with the VM flag set is saved in the current TSS.
2. The link field in the TSS of the called task is loaded with the segment selector of the TSS for the interrupted virtual-8086-mode task.
3. The EFLAGS register is loaded from the image in the new TSS, which clears the VM flag and causes the processor to switch to protected mode.
4. The NT flag in the EFLAGS register is set.
5. The processor begins executing the selected interrupt- or exception-handler task.

When an IRET instruction is executed in the handler task and the NT flag in the EFLAGS register is set, the processors switches from a protected-mode interrupt- or exception-handler task back to a virtual-8086-mode task. Here, the EFLAGS and segment registers are loaded from images saved in the TSS for the virtual-8086-mode task. If the VM flag is set in the EFLAGS image, the processor switches back to virtual-8086 mode on the task switch. The CPL at the time the IRET instruction is executed must be 0, otherwise the processor does not change the state of the VM flag.

20.3.2 Class 2—Maskable Hardware Interrupt Handling in Virtual-8086 Mode Using the Virtual Interrupt Mechanism

Maskable hardware interrupts are those interrupts that are delivered through the INTR# pin or through an interrupt request to the local APIC (see Section 6.3.2, "Maskable Hardware Interrupts"). These interrupts can be inhibited (masked) from interrupting an executing program or task by clearing the IF flag in the EFLAGS register.

When the VME flag in control register CR4 is set and the IOPL field in the EFLAGS register is less than 3, two additional flags are activated in the EFLAGS register:

- VIF (virtual interrupt) flag, bit 19 of the EFLAGS register.
- VIP (virtual interrupt pending) flag, bit 20 of the EFLAGS register.

These flags provide the virtual-8086 monitor with more efficient control over handling maskable hardware interrupts that occur during virtual-8086 mode tasks. They also reduce interrupt-handling overhead, by eliminating the need for all IF related operations (such as PUSHF, POPF, CLI, and STI instructions) to trap to the virtual-8086 monitor. The purpose and use of these flags are as follows.

NOTE

The VIF and VIP flags are only available in IA-32 processors that support the virtual mode extensions. These extensions were introduced in the IA-32 architecture with the Pentium processor. When this mechanism is either not available or not enabled, maskable hardware interrupts are handled as class 1 interrupts. Here, if VIF and VIP flags are needed, the virtual-8086 monitor can implement them in software.

Existing 8086 programs commonly set and clear the IF flag in the EFLAGS register to enable and disable maskable hardware interrupts, respectively; for example, to disable interrupts while handling another interrupt or an exception. This practice works well in single task environments, but can cause problems in multitasking and multiple-processor environments, where it is often desirable to prevent an application program from having direct control over the handling of hardware interrupts. When using earlier IA-32 processors, this problem was often solved by creating a virtual IF flag in software. The IA-32 processors (beginning with the Pentium processor) provide hardware support for this virtual IF flag through the VIF and VIP flags.

The VIF flag is a virtualized version of the IF flag, which an application program running from within a virtual-8086 task can used to control the handling of maskable hardware interrupts. When the VIF flag is enabled, the CLI and STI instructions operate on the VIF flag instead of the IF flag. When an 8086 program executes the CLI instruction, the processor clears the VIF flag to request that the virtual-8086 monitor inhibit maskable hardware interrupts from interrupting program execution; when it executes the STI instruction, the processor sets the VIF flag requesting that the virtual-8086 monitor enable maskable hardware interrupts for the 8086 program. But actually the IF flag, managed by the operating system, always controls whether maskable hardware interrupts are enabled. Also, if under these circumstances an 8086 program tries to read or change the IF flag using the PUSHF or POPF instructions, the processor will change the VIF flag instead, leaving IF unchanged.

The VIP flag provides software a means of recording the existence of a deferred (or pending) maskable hardware interrupt. This flag is read by the processor but never explicitly written by the processor; it can only be written by software.

If the IF flag is set and the VIF and VIP flags are enabled, and the processor receives a maskable hardware interrupt (interrupt vector 0 through 255), the processor performs and the interrupt handler software should perform the following operations:

1. The processor invokes the protected-mode interrupt handler for the interrupt received, as described in the following steps. These steps are almost identical to those described for method 1 interrupt and exception handling in Section 20.3.1.1, "Handling an Interrupt or Exception Through a Protected-Mode Trap or Interrupt Gate":
   a. Switches to 32-bit protected mode and privilege level 0.
   b. Saves the state of the processor on the privilege-level 0 stack. The states of the EIP, CS, EFLAGS, ESP, SS, ES, DS, FS, and GS registers are saved (see Figure 20-4).
   c. Clears the segment registers.

d. Clears the VM flag in the EFLAGS register.

e. Begins executing the selected protected-mode interrupt handler.

2. The recommended action of the protected-mode interrupt handler is to read the VM flag from the EFLAGS image on the stack. If this flag is set, the handler makes a call to the virtual-8086 monitor.
3. The virtual-8086 monitor should read the VIF flag in the EFLAGS register.
   — If the VIF flag is clear, the virtual-8086 monitor sets the VIP flag in the EFLAGS image on the stack to indicate that there is a deferred interrupt pending and returns to the protected-mode handler.
   — If the VIF flag is set, the virtual-8086 monitor can handle the interrupt if it "belongs" to the 8086 program running in the interrupted virtual-8086 task; otherwise, it can call the protected-mode interrupt handler to handle the interrupt.
4. The protected-mode handler executes a return to the program executing in virtual-8086 mode.
5. Upon returning to virtual-8086 mode, the processor continues execution of the 8086 program.

When the 8086 program is ready to receive maskable hardware interrupts, it executes the STI instruction to set the VIF flag (enabling maskable hardware interrupts). Prior to setting the VIF flag, the processor automatically checks the VIP flag and does one of the following, depending on the state of the flag:

- If the VIP flag is clear (indicating no pending interrupts), the processor sets the VIF flag.
- If the VIP flag is set (indicating a pending interrupt), the processor generates a general-protection exception (#GP).

The recommended action of the protected-mode general-protection exception handler is to then call the virtual-8086 monitor and let it handle the pending interrupt. After handling the pending interrupt, the typical action of the virtual-8086 monitor is to clear the VIP flag and set the VIF flag in the EFLAGS image on the stack, and then execute a return to the virtual-8086 mode. The next time the processor receives a maskable hardware interrupt, it will then handle it as described in steps 1 through 5 earlier in this section.

If the processor finds that both the VIF and VIP flags are set at the beginning of an instruction, it generates a general-protection exception. This action allows the virtual-8086 monitor to handle the pending interrupt for the virtual-8086 mode task for which the VIF flag is enabled. Note that this situation can only occur immediately following execution of a POPF or IRET instruction or upon entering a virtual-8086 mode task through a task switch.

Note that the states of the VIF and VIP flags are not modified in real-address mode or during transitions between real-address and protected modes.

NOTE

The virtual interrupt mechanism described in this section is also available for use in protected mode, see Section 20.4, "Protected-Mode Virtual Interrupts".

20.3.3 Class 3—Software Interrupt Handling in Virtual-8086 Mode

When the processor receives a software interrupt (an interrupt generated with the INT *n* instruction) while in virtual-8086 mode, it can use any of six different methods to handle the interrupt. The method selected depends on the settings of the VME flag in control register CR4, the IOPL field in the EFLAGS register, and the software interrupt redirection bit map in the TSS. Table 20-2 lists the six methods of handling software interrupts in virtual-8086 mode and the respective settings of the VME flag, IOPL field, and the bits in the interrupt redirection bit map for each method. The table also summarizes the various actions the processor takes for each method.

The VME flag enables the virtual mode extensions for the Pentium and later IA-32 processors. When this flag is clear, the processor responds to interrupts and exceptions in virtual-8086 mode in the same manner as an Intel386 or Intel486 processor does. When this flag is set, the virtual mode extension provides the following enhancements to virtual-8086 mode:

- Speeds up the handling of software-generated interrupts in virtual-8086 mode by allowing the processor to bypass the virtual-8086 monitor and redirect software interrupts back to the interrupt handlers that are part of the currently running 8086 program.
- Supports virtual interrupts for software written to run on the 8086 processor.

The IOPL value interacts with the VME flag and the bits in the interrupt redirection bit map to determine how specific software interrupts should be handled.

The software interrupt redirection bit map (see Figure 20-5) is a 32-byte field in the TSS. This map is located directly below the I/O permission bit map in the TSS. Each bit in the interrupt redirection bit map is mapped to an interrupt vector. Bit 0 in the interrupt redirection bit map (which maps to vector zero in the interrupt table) is located at the I/O base map address in the TSS minus 32 bytes. When a bit in this bit map is set, it indicates that the associated software interrupt (interrupt generated with an INT *n* instruction) should be handled through the protected-mode IDT and interrupt and exception handlers. When a bit in this bit map is clear, the processor redirects the associated software interrupt back to the interrupt table in the 8086 program (located at linear address 0 in the program's address space).

NOTE

The software interrupt redirection bit map does not affect hardware generated interrupts and exceptions. Hardware generated interrupts and exceptions are always handled by the protected-mode interrupt and exception handlers.

Table 20-2. Software Interrupt Handling Methods While in Virtual-8086 Mode

Method	VME	IOPL	Bit in Redir. Bitmap*	Processor Action
1	0	3	X	Interrupt directed to a protected-mode interrupt handler: ▪ Switches to privilege-level 0 stack ▪ Pushes GS, FS, DS and ES onto privilege-level 0 stack ▪ Pushes SS, ESP, EFLAGS, CS and EIP of interrupted task onto privilege-level 0 stack ▪ Clears VM, RF, NT, and TF flags ▪ If serviced through interrupt gate, clears IF flag ▪ Clears GS, FS, DS and ES to 0 ▪ Sets CS and EIP from interrupt gate
2	0	< 3	X	Interrupt directed to protected-mode general-protection exception (#GP) handler.
3	1	< 3	1	Interrupt directed to a protected-mode general-protection exception (#GP) handler; VIF and VIP flag support for handling class 2 maskable hardware interrupts.
4	1	3	1	Interrupt directed to protected-mode interrupt handler: (see method 1 processor action).
5	1	3	0	Interrupt redirected to 8086 program interrupt handler: ▪ Pushes EFLAGS ▪ Pushes CS and EIP (lower 16 bits only) ▪ Clears IF flag ▪ Clears TF flag ▪ Loads CS and EIP (lower 16 bits only) from selected entry in the interrupt vector table of the current virtual-8086 task
6	1	< 3	0	Interrupt redirected to 8086 program interrupt handler; VIF and VIP flag support for handling class 2 maskable hardware interrupts: ▪ Pushes EFLAGS with IOPL set to 3 and VIF copied to IF ▪ Pushes CS and EIP (lower 16 bits only) ▪ Clears the VIF flag ▪ Clears TF flag ▪ Loads CS and EIP (lower 16 bits only) from selected entry in the interrupt vector table of the current virtual-8086 task

NOTE:

* When set to 0, software interrupt is redirected back to the 8086 program interrupt handler; when set to 1, interrupt is directed to protected-mode handler.

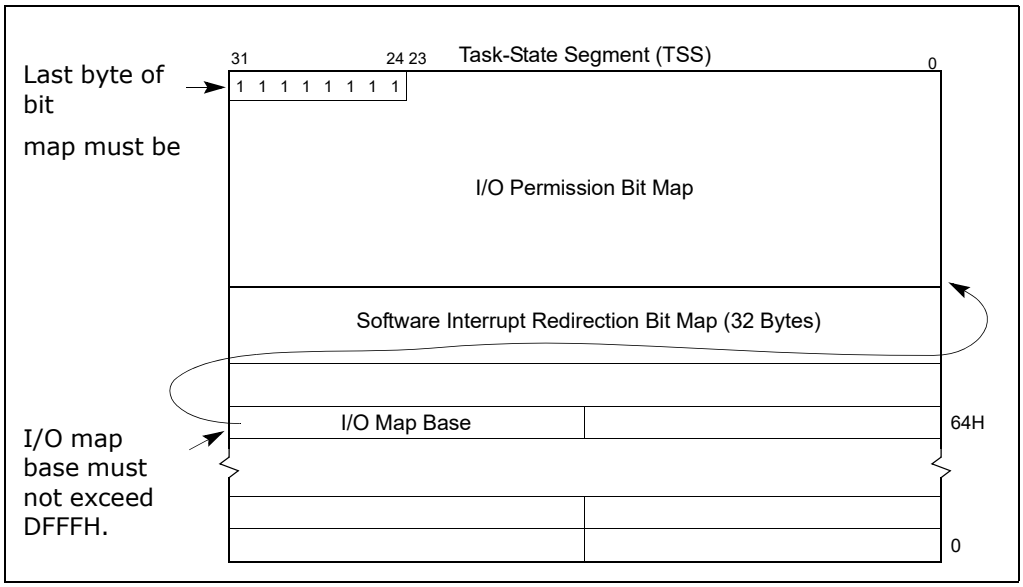


Figure 20-5. Software Interrupt Redirection Bit Map in TSS

Redirecting software interrupts back to the 8086 program potentially speeds up interrupt handling because a switch back and forth between virtual-8086 mode and protected mode is not required. This latter interrupt-handling technique is particularly useful for 8086 operating systems (such as MS-DOS) that use the INT *n* instruction to call operating system procedures.

The CPUID instruction can be used to verify that the virtual mode extension is implemented on the processor. Bit 1 of the feature flags register (EDX) indicates the availability of the virtual mode extension (see "CPUID—CPU Identification" in Chapter 3, "Instruction Set Reference, A-L", of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A*).

The following sections describe the six methods (or mechanisms) for handling software interrupts in virtual-8086 mode. See Section 20.3.2, "Class 2—Maskable Hardware Interrupt Handling in Virtual-8086 Mode Using the Virtual Interrupt Mechanism", for a description of the use of the VIF and VIP flags in the EFLAGS register for handling maskable hardware interrupts.

20.3.3.1 Method 1: Software Interrupt Handling

When the VME flag in control register CR4 is clear and the IOPL field is 3, a Pentium or later IA-32 processor handles software interrupts in the same manner as they are handled by an Intel386 or Intel486 processor. It executes an implicit call to the interrupt handler in the protected-mode IDT pointed to by the interrupt vector. See Section 20.3.1, "Class 1—Hardware Interrupt and Exception Handling in Virtual-8086 Mode", for a complete description of this mechanism and its possible uses.

20.3.3.2 Methods 2 and 3: Software Interrupt Handling

When a software interrupt occurs in virtual-8086 mode and the method 2 or 3 conditions are present, the processor generates a general-protection exception (#GP). Method 2 is enabled when the VME flag is set to 0 and the IOPL value is less than 3. Here the IOPL value is used to bypass the protected-mode interrupt handlers and cause any software interrupt that occurs in virtual-8086 mode to be treated as a protected-mode general-protection exception (#GP). The general-protection exception handler calls the virtual-8086 monitor, which can then emulate an 8086-program interrupt handler or pass control back to the 8086 program's handler, as described in Section 20.3.1.2, "Handling an Interrupt or Exception With an 8086 Program Interrupt or Exception Handler".

Method 3 is enabled when the VME flag is set to 1, the IOPL value is less than 3, and the corresponding bit for the software interrupt in the software interrupt redirection bit map is set to 1. Here, the processor performs the same

operation as it does for method 2 software interrupt handling. If the corresponding bit for the software interrupt in the software interrupt redirection bit map is set to 0, the interrupt is handled using method 6 (see Section 20.3.3.5, "Method 6: Software Interrupt Handling").

20.3.3.3 Method 4: Software Interrupt Handling

Method 4 handling is enabled when the VME flag is set to 1, the IOPL value is 3, and the bit for the interrupt vector in the redirection bit map is set to 1. Method 4 software interrupt handling allows method 1 style handling when the virtual mode extension is enabled; that is, the interrupt is directed to a protected-mode handler (see Section 20.3.3.1, "Method 1: Software Interrupt Handling").

20.3.3.4 Method 5: Software Interrupt Handling

Method 5 software interrupt handling provides a streamlined method of redirecting software interrupts (invoked with the INT *n* instruction) that occur in virtual 8086 mode back to the 8086 program's interrupt vector table and its interrupt handlers. Method 5 handling is enabled when the VME flag is set to 1, the IOPL value is 3, and the bit for the interrupt vector in the redirection bit map is set to 0. The processor performs the following actions to make an implicit call to the selected 8086 program interrupt handler:

1. Pushes the low-order 16 bits of the EFLAGS register onto the stack.
2. Pushes the current values of the CS and EIP registers onto the current stack. (Only the 16 least-significant bits of the EIP register are pushed and no stack switch occurs.)
3. Clears the IF flag in the EFLAGS register to disable interrupts.
4. Clears the TF flag, in the EFLAGS register.
5. Locates the 8086 program interrupt vector table at linear address 0 for the 8086-mode task.
6. Loads the CS and EIP registers with values from the interrupt vector table entry pointed to by the interrupt vector number. Only the 16 low-order bits of the EIP are loaded and the 16 high-order bits are set to 0. The interrupt vector table is assumed to be at linear address 0 of the current virtual-8086 task.
7. Begins executing the selected interrupt handler.

An IRET instruction at the end of the handler procedure reverses these steps to return program control to the interrupted 8086 program.

Note that with method 5 handling, a mode switch from virtual-8086 mode to protected mode does not occur. The processor remains in virtual-8086 mode throughout the interrupt-handling operation.

The method 5 handling actions are virtually identical to the actions the processor takes when handling software interrupts in real-address mode. The benefit of using method 5 handling to access the 8086 program handlers is that it avoids the overhead of methods 2 and 3 handling, which requires first going to the virtual-8086 monitor, then to the 8086 program handler, then back again to the virtual-8086 monitor, before returning to the interrupted 8086 program (see Section 20.3.1.2, "Handling an Interrupt or Exception With an 8086 Program Interrupt or Exception Handler").

NOTE

Methods 1 and 4 handling can handle a software interrupt in a virtual-8086 task with a regular protected-mode handler, but this approach requires all virtual-8086 tasks to use the same software interrupt handlers, which generally does not give sufficient latitude to the programs running in the virtual-8086 tasks, particularly MS-DOS programs.

20.3.3.5 Method 6: Software Interrupt Handling

Method 6 handling is enabled when the VME flag is set to 1, the IOPL value is less than 3, and the bit for the interrupt or exception vector in the redirection bit map is set to 0. With method 6 interrupt handling, software interrupts are handled in the same manner as was described for method 5 handling (see Section 20.3.3.4, "Method 5: Software Interrupt Handling").

Method 6 differs from method 5 in that with the IOPL value set to less than 3, the VIF and VIP flags in the EFLAGS register are enabled, providing virtual interrupt support for handling class 2 maskable hardware interrupts (see Section 20.3.2, "Class 2—Maskable Hardware Interrupt Handling in Virtual-8086 Mode Using the Virtual Interrupt Mechanism"). These flags provide the virtual-8086 monitor with an efficient means of handling maskable hardware interrupts that occur during a virtual-8086 mode task. Also, because the IOPL value is less than 3 and the VIF flag is enabled, the information pushed on the stack by the processor when invoking the interrupt handler is slightly different between methods 5 and 6 (see Table 20-2).

20.4 PROTECTED-MODE VIRTUAL INTERRUPTS

The IA-32 processors (beginning with the Pentium processor) also support the VIF and VIP flags in the EFLAGS register in protected mode by setting the PVI (protected-mode virtual interrupt) flag in the CR4 register. Setting the PVI flag allows applications running at privilege level 3 to execute the CLI and STI instructions without causing a general-protection exception (#GP) or affecting hardware interrupts.

When the PVI flag is set to 1, the CPL is 3, and the IOPL is less than 3, the STI and CLI instructions set and clear the VIF flag in the EFLAGS register, leaving IF unaffected. In this mode of operation, an application running in protected mode and at a CPL of 3 can inhibit interrupts in the same manner as is described in Section 20.3.2, "Class 2—Maskable Hardware Interrupt Handling in Virtual-8086 Mode Using the Virtual Interrupt Mechanism", for a virtual-8086 mode task. When the application executes the CLI instruction, the processor clears the VIF flag. If the processor receives a maskable hardware interrupt, the processor invokes the protected-mode interrupt handler. This handler checks the state of the VIF flag in the EFLAGS register. If the VIF flag is clear (indicating that the active task does not want to have interrupts handled now), the handler sets the VIP flag in the EFLAGS image on the stack and returns to the privilege-level 3 application, which continues program execution. When the application executes a STI instruction to set the VIF flag, the processor automatically invokes the general-protection exception handler, which can then handle the pending interrupt. After handing the pending interrupt, the handler typically sets the VIF flag and clears the VIP flag in the EFLAGS image on the stack and executes a return to the application program. The next time the processor receives a maskable hardware interrupt, the processor will handle it in the normal manner for interrupts received while the processor is operating at a CPL of 3.

If the protected-mode virtual interrupt extension is enabled, CPL = 3, and the processor finds that both the VIF and VIP flags are set at the beginning of an instruction, a general-protection exception is generated.

Because the protected-mode virtual interrupt extension changes only the treatment of EFLAGS.IF (by having CLI and STI update EFLAGS.VIF instead), it affects only the masking of maskable hardware interrupts (interrupt vectors 32 through 255). NMI interrupts and exceptions are handled in the normal manner.

(When protected-mode virtual interrupts are disabled (that is, when the PVI flag in control register CR4 is set to 0, the CPL is less than 3, or the IOPL value is 3), then the CLI and STI instructions execute in a manner compatible with the Intel486 processor. That is, if the CPL is greater (less privileged) than the I/O privilege level (IOPL), a general-protection exception occurs. If the IOPL value is 3, CLI and STI clear or set the IF flag, respectively.)

PUSHF, POPF, IRET and INT are executed like in the Intel486 processor, regardless of whether protected-mode virtual interrupts are enabled.

It is only possible to enter virtual-8086 mode through a task switch or the execution of an IRET instruction, and it is only possible to leave virtual-8086 mode by faulting to a protected-mode interrupt handler (typically the general-protection exception handler, which in turn calls the virtual 8086-mode monitor). In both cases, the EFLAGS register is saved and restored. This is not true, however, in protected mode when the PVI flag is set and the processor is not in virtual-8086 mode. Here, it is possible to call a procedure at a different privilege level, in which case the EFLAGS register is not saved or modified. However, the states of VIF and VIP flags are never examined by the processor when the CPL is not 3.

17. Updates to Chapter 23, Volume 3B

Change bars show changes to Chapter 23 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3B:* System Programming Guide, Part 2.

Change to chapter: Correction to Section 23.8 "Restrictions on VMX Operation".

CHAPTER 23
INTRODUCTION TO VIRTUAL MACHINE EXTENSIONS

23.1 OVERVIEW

This chapter describes the basics of virtual machine architecture and an overview of the virtual-machine extensions (VMX) that support virtualization of processor hardware for multiple software environments.

Information about VMX instructions is provided in *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2B*. Other aspects of VMX and system programming considerations are described in chapters of *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3B*.

23.2 VIRTUAL MACHINE ARCHITECTURE

Virtual-machine extensions define processor-level support for virtual machines on IA-32 processors. Two principal classes of software are supported:

- **Virtual-machine monitors (VMM) —** A VMM acts as a host and has full control of the processor(s) and other platform hardware. A VMM presents guest software (see next paragraph) with an abstraction of a virtual processor and allows it to execute directly on a logical processor. A VMM is able to retain selective control of processor resources, physical memory, interrupt management, and I/O.
- **Guest software —** Each virtual machine (VM) is a guest software environment that supports a stack consisting of operating system (OS) and application software. Each operates independently of other virtual machines and uses on the same interface to processor(s), memory, storage, graphics, and I/O provided by a physical platform. The software stack acts as if it were running on a platform with no VMM. Software executing in a virtual machine must operate with reduced privilege so that the VMM can retain control of platform resources.

23.3 INTRODUCTION TO VMX OPERATION

Processor support for virtualization is provided by a form of processor operation called VMX operation. There are two kinds of VMX operation: VMX root operation and VMX non-root operation. In general, a VMM will run in VMX root operation and guest software will run in VMX non-root operation. Transitions between VMX root operation and VMX non-root operation are called VMX transitions. There are two kinds of VMX transitions. Transitions into VMX non-root operation are called VM entries. Transitions from VMX non-root operation to VMX root operation are called VM exits.

Processor behavior in VMX root operation is very much as it is outside VMX operation. The principal differences are that a set of new instructions (the VMX instructions) is available and that the values that can be loaded into certain control registers are limited (see Section 23.8).

Processor behavior in VMX non-root operation is restricted and modified to facilitate virtualization. Instead of their ordinary operation, certain instructions (including the new VMCALL instruction) and events cause VM exits to the VMM. Because these VM exits replace ordinary behavior, the functionality of software in VMX non-root operation is limited. It is this limitation that allows the VMM to retain control of processor resources.

There is no software-visible bit whose setting indicates whether a logical processor is in VMX non-root operation. This fact may allow a VMM to prevent guest software from determining that it is running in a virtual machine.

Because VMX operation places restrictions even on software running with current privilege level (CPL) 0, guest software can run at the privilege level for which it was originally designed. This capability may simplify the development of a VMM.

23.4 LIFE CYCLE OF VMM SOFTWARE

Figure 23-1 illustrates the life cycle of a VMM and its guest software as well as the interactions between them. The following items summarize that life cycle:

- Software enters VMX operation by executing a VMXON instruction.
- Using VM entries, a VMM can then enter guests into virtual machines (one at a time). The VMM effects a VM entry using instructions VMLAUNCH and VMRESUME; it regains control using VM exits.
- VM exits transfer control to an entry point specified by the VMM. The VMM can take action appropriate to the cause of the VM exit and can then return to the virtual machine using a VM entry.
- Eventually, the VMM may decide to shut itself down and leave VMX operation. It does so by executing the VMXOFF instruction.

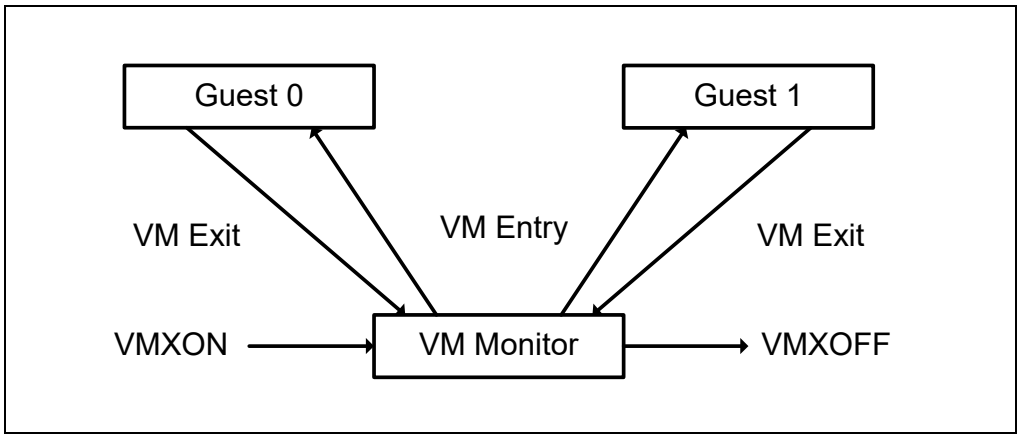


Figure 23-1. Interaction of a Virtual-Machine Monitor and Guests

23.5 VIRTUAL-MACHINE CONTROL STRUCTURE

VMX non-root operation and VMX transitions are controlled by a data structure called a virtual-machine control structure (VMCS).

Access to the VMCS is managed through a component of processor state called the VMCS pointer (one per logical processor). The value of the VMCS pointer is the 64-bit address of the VMCS. The VMCS pointer is read and written using the instructions VMPTRST and VMPTRLD. The VMM configures a VMCS using the VMREAD, VMWRITE, and VMCLEAR instructions.

A VMM could use a different VMCS for each virtual machine that it supports. For a virtual machine with multiple logical processors (virtual processors), the VMM could use a different VMCS for each virtual processor.

23.6 DISCOVERING SUPPORT FOR VMX

Before system software enters into VMX operation, it must discover the presence of VMX support in the processor. System software can determine whether a processor supports VMX operation using CPUID. If CPUID.1:ECX.VMX[bit 5] = 1, then VMX operation is supported. See Chapter 3, "Instruction Set Reference, A-L" of *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A*.

The VMX architecture is designed to be extensible so that future processors in VMX operation can support additional features not present in first-generation implementations of the VMX architecture. The availability of extensible VMX features is reported to software using a set of VMX capability MSRs (see Appendix A, "VMX Capability Reporting Facility").

23.7 ENABLING AND ENTERING VMX OPERATION

Before system software can enter VMX operation, it enables VMX by setting CR4.VMXE[bit 13] = 1. VMX operation is then entered by executing the VMXON instruction. VMXON causes an invalid-opcode exception (#UD) if executed with CR4.VMXE = 0. Once in VMX operation, it is not possible to clear CR4.VMXE (see Section 23.8). System software leaves VMX operation by executing the VMXOFF instruction. CR4.VMXE can be cleared outside of VMX operation after executing of VMXOFF.

VMXON is also controlled by the IA32_FEATURE_CONTROL MSR (MSR address 3AH). This MSR is cleared to zero when a logical processor is reset. The relevant bits of the MSR are:

- **Bit 0 is the lock bit.** If this bit is clear, VMXON causes a general-protection exception. If the lock bit is set, WRMSR to this MSR causes a general-protection exception; the MSR cannot be modified until a power-up reset condition. System BIOS can use this bit to provide a setup option for BIOS to disable support for VMX. To enable VMX support in a platform, BIOS must set bit 1, bit 2, or both (see below), as well as the lock bit.
- **Bit 1 enables VMXON in SMX operation.** If this bit is clear, execution of VMXON in SMX operation causes a general-protection exception. Attempts to set this bit on logical processors that do not support both VMX operation (see Section 23.6) and SMX operation (see Chapter 6, "Safer Mode Extensions Reference," in *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2D*) cause general-protection exceptions.
- **Bit 2 enables VMXON outside SMX operation.** If this bit is clear, execution of VMXON outside SMX operation causes a general-protection exception. Attempts to set this bit on logical processors that do not support VMX operation (see Section 23.6) cause general-protection exceptions.

NOTE

A logical processor is in SMX operation if GETSEC[SEXIT] has not been executed since the last execution of GETSEC[SENTER]. A logical processor is outside SMX operation if GETSEC[SENTER] has not been executed or if GETSEC[SEXIT] was executed after the last execution of GETSEC[SENTER]. See Chapter 6, "Safer Mode Extensions Reference," in *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2D*.

Before executing VMXON, software should allocate a naturally aligned 4-KByte region of memory that a logical processor may use to support VMX operation.[1] This region is called the **VMXON region**. The address of the VMXON region (the VMXON pointer) is provided in an operand to VMXON. Section 24.11.5, "VMXON Region," details how software should initialize and access the VMXON region.

23.8 RESTRICTIONS ON VMX OPERATION

VMX operation places restrictions on processor operation. These are detailed below:

- In VMX operation, processors may fix certain bits in CR0 and CR4 to specific values and not support other values. VMXON fails if any of these bits contains an unsupported value (see "VMXON—Enter VMX Operation" in Chapter 30). Any attempt to set one of these bits to an unsupported value while in VMX operation (including VMX root operation) using any of the CLTS, LMSW, or MOV CR instructions causes a general-protection exception. VM entry or VM exit cannot set any of these bits to an unsupported value. Software should consult the VMX capability MSRs IA32_VMX_CR0_FIXED0 and IA32_VMX_CR0_FIXED1 to determine how bits in CR0 are fixed (see Appendix A.7). For CR4, software should consult the VMX capability MSRs IA32_VMX_CR4_FIXED0 and IA32_VMX_CR4_FIXED1 (see Appendix A.8).

NOTES

The first processors to support VMX operation require that the following bits be 1 in VMX operation: CR0.PE, CR0.NE, CR0.PG, and CR4.VMXE. The restrictions on CR0.PE and CR0.PG imply that VMX operation is supported only in paged protected mode (including IA-32e mode). Therefore, guest software cannot be run in unpaged protected mode or in real-address mode. See Section 31.2,

1. Future processors may require that a different amount of memory be reserved. If so, this fact is reported to software using the VMX capability-reporting mechanism.

"Supporting Processor Operating Modes in Guest Environments," for a discussion of how a VMM might support guest software that expects to run in unpaged protected mode or in real-address mode.

Later processors support a VM-execution control called "unrestricted guest" (see Section 24.6.2). If this control is 1, CR0.PE and CR0.PG may be 0 in VMX non-root operation (even if the capability MSR IA32_VMX_CR0_FIXED0 reports otherwise).[1] Such processors allow guest software to run in unpaged protected mode or in real-address mode.

- VMXON fails if a logical processor is in A20M mode (see "VMXON—Enter VMX Operation" in Chapter 30). Once the processor is in VMX operation, A20M interrupts are blocked. Thus, it is impossible to be in A20M mode in VMX operation.
- The INIT signal is blocked whenever a logical processor is in VMX root operation. It is not blocked in VMX non-root operation. Instead, INITs cause VM exits (see Section 25.2, "Other Causes of VM Exits").
- Intel® Processor Trace (Intel PT) can be used in VMX operation only if IA32_VMX_MISC[14] is read as 1 (see Appendix A.6). On processors that support Intel PT but which do not allow it to be used in VMX operation, execution of VMXON clears IA32_RTIT_CTL.TraceEn (see "VMXON—Enter VMX Operation" in Chapter 30); any attempt to write IA32_RTIT_CTL while in VMX operation (including VMX root operation) causes a general-protection exception.

1. "Unrestricted guest" is a secondary processor-based VM-execution control. If bit 31 of the primary processor-based VM-execution controls is 0, VMX non-root operation functions as if the "unrestricted guest" VM-execution control were 0. See Section 24.6.2.

18. Updates to Chapter 25, Volume 3C

Change bars show changes to Chapter 25 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3C:* System Programming Guide, Part 3.

Changes to this chapter: Typo corrections in LMSW listing of Section 25.1.3 "Instructions That Cause VM Exits Conditionally". Typo corrections in SMSW and WRMSR listings of Section 25.3 "Changes to Instruction Behavior in VMX Non-Root Operation".

CHAPTER 25
VMX NON-ROOT OPERATION

In a virtualized environment using VMX, the guest software stack typically runs on a logical processor in VMX non-root operation. This mode of operation is similar to that of ordinary processor operation outside of the virtualized environment. This chapter describes the differences between VMX non-root operation and ordinary processor operation with special attention to causes of VM exits (which bring a logical processor from VMX non-root operation to root operation). The differences between VMX non-root operation and ordinary processor operation are described in the following sections:

- Section 25.1, "Instructions That Cause VM Exits"
- Section 25.2, "Other Causes of VM Exits"
- Section 25.3, "Changes to Instruction Behavior in VMX Non-Root Operation"
- Section 25.4, "Other Changes in VMX Non-Root Operation"
- Section 25.5, "Features Specific to VMX Non-Root Operation"
- Section 25.6, "Unrestricted Guests"

Chapter 26, "VM Entries," describes the data control structures that govern VMX non-root operation. Chapter 26, "VM Entries," describes the operation of VM entries by which the processor transitions from VMX root operation to VMX non-root operation. Chapter 25, "VMX Non-Root Operation," describes the operation of VM exits by which the processor transitions from VMX non-root operation to VMX root operation.

Chapter 28, "VMX Support for Address Translation," describes two features that support address translation in VMX non-root operation. Chapter 29, "APIC Virtualization and Virtual Interrupts," describes features that support virtualization of interrupts and the Advanced Programmable Interrupt Controller (APIC) in VMX non-root operation.

25.1 INSTRUCTIONS THAT CAUSE VM EXITS

Certain instructions may cause VM exits if executed in VMX non-root operation. Unless otherwise specified, such VM exits are "fault-like," meaning that the instruction causing the VM exit does not execute and no processor state is updated by the instruction. Section 27.1 details architectural state in the context of a VM exit.

Section 25.1.1 defines the prioritization between faults and VM exits for instructions subject to both. Section 25.1.2 identifies instructions that cause VM exits whenever they are executed in VMX non-root operation (and thus can never be executed in VMX non-root operation). Section 25.1.3 identifies instructions that cause VM exits depending on the settings of certain VM-execution control fields (see Section 24.6).

25.1.1 Relative Priority of Faults and VM Exits

The following principles describe the ordering between existing faults and VM exits:

- Certain exceptions have priority over VM exits. These include invalid-opcode exceptions, faults based on privilege level,[1] and general-protection exceptions that are based on checking I/O permission bits in the task-state segment (TSS). For example, execution of RDMSR with CPL = 3 generates a general-protection exception and not a VM exit.[2]
- Faults incurred while fetching instruction operands have priority over VM exits that are conditioned based on the contents of those operands (see LMSW in Section 25.1.3).
- VM exits caused by execution of the INS and OUTS instructions (resulting either because the "unconditional I/O exiting" VM-execution control is 1 or because the "use I/O bitmaps control is 1) have priority over the following faults:

1. These include faults generated by attempts to execute, in virtual-8086 mode, privileged instructions that are not recognized in that mode.
2. MOV DR is an exception to this rule; see Section 25.1.3.

  - A general-protection fault due to the relevant segment (ES for INS; DS for OUTS unless overridden by an instruction prefix) being unusable
  - A general-protection fault due to an offset beyond the limit of the relevant segment
  - An alignment-check exception
- Fault-like VM exits have priority over exceptions other than those mentioned above. For example, RDMSR of a non-existent MSR with CPL = 0 generates a VM exit and not a general-protection exception.

When Section 25.1.2 or Section 25.1.3 (below) identify an instruction execution that may lead to a VM exit, it is assumed that the instruction does not incur a fault that takes priority over a VM exit.

25.1.2 Instructions That Cause VM Exits Unconditionally

The following instructions cause VM exits when they are executed in VMX non-root operation: CPUID, GETSEC,[1] INVD, and XSETBV. This is also true of instructions introduced with VMX, which include: INVEPT, INVVPID, VMCALL,[2] VMCLEAR, VMLAUNCH, VMPTRLD, VMPTRST, VMRESUME, VMXOFF, and VMXON.

25.1.3 Instructions That Cause VM Exits Conditionally

Certain instructions cause VM exits in VMX non-root operation depending on the setting of the VM-execution controls. The following instructions can cause "fault-like" VM exits based on the conditions described:[3]

- **CLTS.** The CLTS instruction causes a VM exit if the bits in position 3 (corresponding to CR0.TS) are set in both the CR0 guest/host mask and the CR0 read shadow.
- **ENCLS.** The ENCLS instruction causes a VM exit if the "enable ENCLS exiting" VM-execution control is 1 and one of the following is true:
  - The value of EAX is less than 63 and the corresponding bit in the ENCLS-exiting bitmap is 1 (see Section 24.6.16).
  - The value of EAX is greater than or equal to 63 and bit 63 in the ENCLS-exiting bitmap is 1.
- **HLT.** The HLT instruction causes a VM exit if the "HLT exiting" VM-execution control is 1.
- **IN, INS/INSB/INSW/INSD, OUT, OUTS/OUTSB/OUTSW/OUTSD.** The behavior of each of these instructions is determined by the settings of the "unconditional I/O exiting" and "use I/O bitmaps" VM-execution controls:
  - If both controls are 0, the instruction executes normally.
  - If the "unconditional I/O exiting" VM-execution control is 1 and the "use I/O bitmaps" VM-execution control is 0, the instruction causes a VM exit.
  - If the "use I/O bitmaps" VM-execution control is 1, the instruction causes a VM exit if it attempts to access an I/O port corresponding to a bit set to 1 in the appropriate I/O bitmap (see Section 24.6.4). If an I/O operation "wraps around" the 16-bit I/O-port space (accesses ports FFFFH and 0000H), the I/O instruction causes a VM exit (the "unconditional I/O exiting" VM-execution control is ignored if the "use I/O bitmaps" VM-execution control is 1).

  See Section 25.1.1 for information regarding the priority of VM exits relative to faults that may be caused by the INS and OUTS instructions.
- **INVLPG.** The INVLPG instruction causes a VM exit if the "INVLPG exiting" VM-execution control is 1.
- **INVPCID.** The INVPCID instruction causes a VM exit if the "INVLPG exiting" and "enable INVPCID" VM-execution controls are both 1.

1. An execution of GETSEC in VMX non-root operation causes a VM exit if CR4.SMXE[Bit 14] = 1 regardless of the value of CPL or RAX. An execution of GETSEC causes an invalid-opcode exception (#UD) if CR4.SMXE[Bit 14] = 0.
2. Under the dual-monitor treatment of SMIs and SMM, executions of VMCALL cause SMM VM exits in VMX root operation outside SMM. See Section 34.15.2.
3. Many of the items in this section refer to secondary processor-based VM-execution controls. If bit 31 of the primary processor-based VM-execution controls is 0, VMX non-root operation functions as if these controls were all 0. See Section 24.6.2.

- **LGDT, LIDT, LLDT, LTR, SGDT, SIDT, SLDT, STR.** These instructions cause VM exits if the "descriptor-table exiting" VM-execution control is 1.
- **LMSW.** In general, the LMSW instruction causes a VM exit if it would write, for any bit set in the low 4 bits of the CR0 guest/host mask, a value different than the corresponding bit in the CR0 read shadow. LMSW never clears bit 0 of CR0 (CR0.PE); thus, LMSW causes a VM exit if either of the following are true:
  - The bits in position 0 (corresponding to CR0.PE) are set in both the CR0 guest/host mask and the source operand, and the bit in position 0 is clear in the CR0 read shadow.
  - For any bit position in the range 3:1, the bit in that position is set in the CR0 guest/host mask and the values of the corresponding bits in the source operand and the CR0 read shadow differ.
- **MONITOR.** The MONITOR instruction causes a VM exit if the "MONITOR exiting" VM-execution control is 1.
- **MOV from CR3.** The MOV from CR3 instruction causes a VM exit if the "CR3-store exiting" VM-execution control is 1. The first processors to support the virtual-machine extensions supported only the 1-setting of this control.
- **MOV from CR8.** The MOV from CR8 instruction causes a VM exit if the "CR8-store exiting" VM-execution control is 1.
- **MOV to CR0.** The MOV to CR0 instruction causes a VM exit unless the value of its source operand matches, for the position of each bit set in the CR0 guest/host mask, the corresponding bit in the CR0 read shadow. (If every bit is clear in the CR0 guest/host mask, MOV to CR0 cannot cause a VM exit.)
- **MOV to CR3.** The MOV to CR3 instruction causes a VM exit unless the "CR3-load exiting" VM-execution control is 0 or the value of its source operand is equal to one of the CR3-target values specified in the VMCS. Only the first *n* CR3-target values are considered, where *n* is the CR3-target count. If the "CR3-load exiting" VM-execution control is 1 and the CR3-target count is 0, MOV to CR3 always causes a VM exit.

  The first processors to support the virtual-machine extensions supported only the 1-setting of the "CR3-load exiting" VM-execution control. These processors always consult the CR3-target controls to determine whether an execution of MOV to CR3 causes a VM exit.
- **MOV to CR4.** The MOV to CR4 instruction causes a VM exit unless the value of its source operand matches, for the position of each bit set in the CR4 guest/host mask, the corresponding bit in the CR4 read shadow.
- **MOV to CR8.** The MOV to CR8 instruction causes a VM exit if the "CR8-load exiting" VM-execution control is 1.
- **MOV DR.** The MOV DR instruction causes a VM exit if the "MOV-DR exiting" VM-execution control is 1. Such VM exits represent an exception to the principles identified in Section 25.1.1 in that they take priority over the following: general-protection exceptions based on privilege level; and invalid-opcode exceptions that occur because CR4.DE=1 and the instruction specified access to DR4 or DR5.
- **MWAIT.** The MWAIT instruction causes a VM exit if the "MWAIT exiting" VM-execution control is 1. If this control is 0, the behavior of the MWAIT instruction may be modified (see Section 25.3).
- **PAUSE.** The behavior of each of this instruction depends on CPL and the settings of the "PAUSE exiting" and "PAUSE-loop exiting" VM-execution controls:
  - CPL = 0.
    - If the "PAUSE exiting" and "PAUSE-loop exiting" VM-execution controls are both 0, the PAUSE instruction executes normally.
    - If the "PAUSE exiting" VM-execution control is 1, the PAUSE instruction causes a VM exit (the "PAUSE-loop exiting" VM-execution control is ignored if CPL = 0 and the "PAUSE exiting" VM-execution control is 1).
    - If the "PAUSE exiting" VM-execution control is 0 and the "PAUSE-loop exiting" VM-execution control is 1, the following treatment applies.

      The processor determines the amount of time between this execution of PAUSE and the previous execution of PAUSE at CPL 0. If this amount of time exceeds the value of the VM-execution control field PLE_Gap, the processor considers this execution to be the first execution of PAUSE in a loop. (It also does so for the first execution of PAUSE at CPL 0 after VM entry.)

      Otherwise, the processor determines the amount of time since the most recent execution of PAUSE that was considered to be the first in a loop. If this amount of time exceeds the value of the VM-execution control field PLE_Window, a VM exit occurs.

For purposes of these computations, time is measured based on a counter that runs at the same rate as the timestamp counter (TSC).

— CPL > 0.

  • If the "PAUSE exiting" VM-execution control is 0, the PAUSE instruction executes normally.

  • If the "PAUSE exiting" VM-execution control is 1, the PAUSE instruction causes a VM exit.

The "PAUSE-loop exiting" VM-execution control is ignored if CPL > 0.

- **RDMSR.** The RDMSR instruction causes a VM exit if any of the following are true:
  — The "use MSR bitmaps" VM-execution control is 0.
  — The value of ECX is not in the ranges 00000000H – 00001FFFH and C0000000H – C0001FFFH.
  — The value of ECX is in the range 00000000H – 00001FFFH and bit *n* in read bitmap for low MSRs is 1, where *n* is the value of ECX.
  — The value of ECX is in the range C0000000H – C0001FFFH and bit *n* in read bitmap for high MSRs is 1, where *n* is the value of ECX & 00001FFFH.

  See Section 24.6.9 for details regarding how these bitmaps are identified.
- **RDPMC.** The RDPMC instruction causes a VM exit if the "RDPMC exiting" VM-execution control is 1.
- **RDRAND.** The RDRAND instruction causes a VM exit if the "RDRAND exiting" VM-execution control is 1.
- **RDSEED.** The RDSEED instruction causes a VM exit if the "RDSEED exiting" VM-execution control is 1.
- **RDTSC.** The RDTSC instruction causes a VM exit if the "RDTSC exiting" VM-execution control is 1.
- **RDTSCP.** The RDTSCP instruction causes a VM exit if the "RDTSC exiting" and "enable RDTSCP" VM-execution controls are both 1.
- **RSM.** The RSM instruction causes a VM exit if executed in system-management mode (SMM).[1]
- **VMREAD.** The VMREAD instruction causes a VM exit if any of the following are true:
  — The "VMCS shadowing" VM-execution control is 0.
  — Bits 63:15 (bits 31:15 outside 64-bit mode) of the register source operand are not all 0.
  — Bit *n* in VMREAD bitmap is 1, where *n* is the value of bits 14:0 of the register source operand. See Section 24.6.15 for details regarding how the VMREAD bitmap is identified.

  If the VMREAD instruction does not cause a VM exit, it reads from the VMCS referenced by the VMCS link pointer. See Chapter 30, "VMREAD—Read Field from Virtual-Machine Control Structure" for details of the operation of the VMREAD instruction.
- **VMWRITE.** The VMWRITE instruction causes a VM exit if any of the following are true:
  — The "VMCS shadowing" VM-execution control is 0.
  — Bits 63:15 (bits 31:15 outside 64-bit mode) of the register source operand are not all 0.
  — Bit *n* in VMWRITE bitmap is 1, where *n* is the value of bits 14:0 of the register source operand. See Section 24.6.15 for details regarding how the VMWRITE bitmap is identified.

  If the VMWRITE instruction does not cause a VM exit, it writes to the VMCS referenced by the VMCS link pointer. See Chapter 30, "VMWRITE—Write Field to Virtual-Machine Control Structure" for details of the operation of the VMWRITE instruction.
- **WBINVD.** The WBINVD instruction causes a VM exit if the "WBINVD exiting" VM-execution control is 1.
- **WRMSR.** The WRMSR instruction causes a VM exit if any of the following are true:
  — The "use MSR bitmaps" VM-execution control is 0.
  — The value of ECX is not in the ranges 00000000H – 00001FFFH and C0000000H – C0001FFFH.
  — The value of ECX is in the range 00000000H – 00001FFFH and bit *n* in write bitmap for low MSRs is 1, where *n* is the value of ECX.

1. Execution of the RSM instruction outside SMM causes an invalid-opcode exception regardless of whether the processor is in VMX operation. It also does so in VMX root operation in SMM; see Section 34.15.3.

— The value of ECX is in the range C0000000H – C0001FFFH and bit *n* in write bitmap for high MSRs is 1, where *n* is the value of ECX & 00001FFFH.

See Section 24.6.9 for details regarding how these bitmaps are identified.

- **XRSTORS.** The XRSTORS instruction causes a VM exit if the "enable XSAVES/XRSTORS" VM-execution control is 1and any bit is set in the logical-AND of the following three values: EDX:EAX, the IA32_XSS MSR, and the XSS-exiting bitmap (see Section 24.6.19).
- **XSAVES.** The XSAVES instruction causes a VM exit if the "enable XSAVES/XRSTORS" VM-execution control is 1 and any bit is set in the logical-AND of the following three values: EDX:EAX, the IA32_XSS MSR, and the XSS-exiting bitmap (see Section 24.6.19).

25.2 OTHER CAUSES OF VM EXITS

In addition to VM exits caused by instruction execution, the following events can cause VM exits:

- **Exceptions.** Exceptions (faults, traps, and aborts) cause VM exits based on the exception bitmap (see Section 24.6.3). If an exception occurs, its vector (in the range 0–31) is used to select a bit in the exception bitmap. If the bit is 1, a VM exit occurs; if the bit is 0, the exception is delivered normally through the guest IDT. This use of the exception bitmap applies also to exceptions generated by the instructions INT3, INTO, BOUND, and UD.

  Page faults (exceptions with vector 14) are specially treated. When a page fault occurs, a processor consults (1) bit 14 of the exception bitmap; (2) the error code produced with the page fault [PFEC]; (3) the page-fault error-code mask field [PFEC_MASK]; and (4) the page-fault error-code match field [PFEC_MATCH]. It checks if PFEC & PFEC_MASK = PFEC_MATCH. If there is equality, the specification of bit 14 in the exception bitmap is followed (for example, a VM exit occurs if that bit is set). If there is inequality, the meaning of that bit is reversed (for example, a VM exit occurs if that bit is clear).

  Thus, if software desires VM exits on all page faults, it can set bit 14 in the exception bitmap to 1 and set the page-fault error-code mask and match fields each to 00000000H. If software desires VM exits on no page faults, it can set bit 14 in the exception bitmap to 1, the page-fault error-code mask field to 00000000H, and the page-fault error-code match field to FFFFFFFFH.
- **Triple fault.** A VM exit occurs if the logical processor encounters an exception while attempting to call the double-fault handler and that exception itself does not cause a VM exit due to the exception bitmap. This applies to the case in which the double-fault exception was generated within VMX non-root operation, the case in which the double-fault exception was generated during event injection by VM entry, and to the case in which VM entry is injecting a double-fault exception.
- **External interrupts.** An external interrupt causes a VM exit if the "external-interrupt exiting" VM-execution control is 1. (See Section 25.6 for an exception.) Otherwise, the interrupt is delivered normally through the IDT. (If a logical processor is in the shutdown state or the wait-for-SIPI state, external interrupts are blocked. The interrupt is not delivered through the IDT and no VM exit occurs.)
- **Non-maskable interrupts (NMIs).** An NMI causes a VM exit if the "NMI exiting" VM-execution control is 1. Otherwise, it is delivered using descriptor 2 of the IDT. (If a logical processor is in the wait-for-SIPI state, NMIs are blocked. The NMI is not delivered through the IDT and no VM exit occurs.)
- **INIT signals.** INIT signals cause VM exits. A logical processor performs none of the operations normally associated with these events. Such exits do not modify register state or clear pending events as they would outside of VMX operation. (If a logical processor is in the wait-for-SIPI state, INIT signals are blocked. They do not cause VM exits in this case.)
- **Start-up IPIs (SIPIs). SIPIs cause VM exits.** If a logical processor is not in the wait-for-SIPI activity state when a SIPI arrives, no VM exit occurs and the SIPI is discarded. VM exits due to SIPIs do not perform any of the normal operations associated with those events: they do not modify register state as they would outside of VMX operation. (If a logical processor is not in the wait-for-SIPI state, SIPIs are blocked. They do not cause VM exits in this case.)
- **Task switches.** Task switches are not allowed in VMX non-root operation. Any attempt to effect a task switch in VMX non-root operation causes a VM exit. See Section 25.4.2.
- **System-management interrupts (SMIs).** If the logical processor is using the dual-monitor treatment of SMIs and system-management mode (SMM), SMIs cause SMM VM exits. See Section 34.15.2.[1]

- **VMX-preemption timer.** A VM exit occurs when the timer counts down to zero. See Section 25.5.1 for details of operation of the VMX-preemption timer.

  Debug-trap exceptions and higher priority events take priority over VM exits caused by the VMX-preemption timer. VM exits caused by the VMX-preemption timer take priority over VM exits caused by the "NMI-window exiting" VM-execution control and lower priority events.

  These VM exits wake a logical processor from the same inactive states as would a non-maskable interrupt. Specifically, they wake a logical processor from the shutdown state and from the states entered using the HLT and MWAIT instructions. These VM exits do not occur if the logical processor is in the wait-for-SIPI state.

In addition, there are controls that cause VM exits based on the readiness of guest software to receive interrupts:

- If the "interrupt-window exiting" VM-execution control is 1, a VM exit occurs before execution of any instruction if RFLAGS.IF = 1 and there is no blocking of events by STI or by MOV SS (see Table 24-3). Such a VM exit occurs immediately after VM entry if the above conditions are true (see Section 26.6.5).

  Non-maskable interrupts (NMIs) and higher priority events take priority over VM exits caused by this control. VM exits caused by this control take priority over external interrupts and lower priority events.

  These VM exits wake a logical processor from the same inactive states as would an external interrupt. Specifically, they wake a logical processor from the states entered using the HLT and MWAIT instructions. These VM exits do not occur if the logical processor is in the shutdown state or the wait-for-SIPI state.
- If the "NMI-window exiting" VM-execution control is 1, a VM exit occurs before execution of any instruction if there is no virtual-NMI blocking and there is no blocking of events by MOV SS (see Table 24-3). (A logical processor may also prevent such a VM exit if there is blocking of events by STI.) Such a VM exit occurs immediately after VM entry if the above conditions are true (see Section 26.6.6).

  VM exits caused by the VMX-preemption timer and higher priority events take priority over VM exits caused by this control. VM exits caused by this control take priority over non-maskable interrupts (NMIs) and lower priority events.

  These VM exits wake a logical processor from the same inactive states as would an NMI. Specifically, they wake a logical processor from the shutdown state and from the states entered using the HLT and MWAIT instructions. These VM exits do not occur if the logical processor is in the wait-for-SIPI state.

25.3 CHANGES TO INSTRUCTION BEHAVIOR IN VMX NON-ROOT OPERATION

The behavior of some instructions is changed in VMX non-root operation. Some of these changes are determined by the settings of certain VM-execution control fields. The following items detail such changes:[1]

- **CLTS.** Behavior of the CLTS instruction is determined by the bits in position 3 (corresponding to CR0.TS) in the CR0 guest/host mask and the CR0 read shadow:
  - If bit 3 in the CR0 guest/host mask is 0, CLTS clears CR0.TS normally (the value of bit 3 in the CR0 read shadow is irrelevant in this case), unless CR0.TS is fixed to 1 in VMX operation (see Section 23.8), in which case CLTS causes a general-protection exception.
  - If bit 3 in the CR0 guest/host mask is 1 and bit 3 in the CR0 read shadow is 0, CLTS completes but does not change the contents of CR0.TS.
  - If the bits in position 3 in the CR0 guest/host mask and the CR0 read shadow are both 1, CLTS causes a VM exit.
- **INVPCID.** Behavior of the INVPCID instruction is determined first by the setting of the "enable INVPCID" VM-execution control:
  - If the "enable INVPCID" VM-execution control is 0, INVPCID causes an invalid-opcode exception (#UD). This exception takes priority over any other exception the instruction may incur.

---

1. Under the dual-monitor treatment of SMIs and SMM, SMIs also cause SMM VM exits if they occur in VMX root operation outside SMM. If the processor is using the default treatment of SMIs and SMM, SMIs are delivered as described in Section 34.14.1.

1. Some of the items in this section refer to secondary processor-based VM-execution controls. If bit 31 of the primary processor-based VM-execution controls is 0, VMX non-root operation functions as if these controls were all 0. See Section 24.6.2.

— If the "enable INVPCID" VM-execution control is 1, treatment is based on the setting of the "INVLPG exiting" VM-execution control:

  • If the "INVLPG exiting" VM-execution control is 0, INVPCID operates normally.

  • If the "INVLPG exiting" VM-execution control is 1, INVPCID causes a VM exit.

- **IRET.** Behavior of IRET with regard to NMI blocking (see Table 24-3) is determined by the settings of the "NMI exiting" and "virtual NMIs" VM-execution controls:

  — If the "NMI exiting" VM-execution control is 0, IRET operates normally and unblocks NMIs. (If the "NMI exiting" VM-execution control is 0, the "virtual NMIs" control must be 0; see Section 26.2.1.1.)

  — If the "NMI exiting" VM-execution control is 1, IRET does not affect blocking of NMIs. If, in addition, the "virtual NMIs" VM-execution control is 1, the logical processor tracks virtual-NMI blocking. In this case, IRET removes any virtual-NMI blocking.

  The unblocking of NMIs or virtual NMIs specified above occurs even if IRET causes a fault.

- **LMSW.** Outside of VMX non-root operation, LMSW loads its source operand into CR0[3:0], but it does not clear CR0.PE if that bit is set. In VMX non-root operation, an execution of LMSW that does not cause a VM exit (see Section 25.1.3) leaves unmodified any bit in CR0[3:0] corresponding to a bit set in the CR0 guest/host mask. An attempt to set any other bit in CR0[3:0] to a value not supported in VMX operation (see Section 23.8) causes a general-protection exception. Attempts to clear CR0.PE are ignored without fault.

- **MOV from CR0.** The behavior of MOV from CR0 is determined by the CR0 guest/host mask and the CR0 read shadow. For each position corresponding to a bit clear in the CR0 guest/host mask, the destination operand is loaded with the value of the corresponding bit in CR0. For each position corresponding to a bit set in the CR0 guest/host mask, the destination operand is loaded with the value of the corresponding bit in the CR0 read shadow. Thus, if every bit is cleared in the CR0 guest/host mask, MOV from CR0 reads normally from CR0; if every bit is set in the CR0 guest/host mask, MOV from CR0 returns the value of the CR0 read shadow.

  Depending on the contents of the CR0 guest/host mask and the CR0 read shadow, bits may be set in the destination that would never be set when reading directly from CR0.

- **MOV from CR3.** If the "enable EPT" VM-execution control is 1 and an execution of MOV from CR3 does not cause a VM exit (see Section 25.1.3), the value loaded from CR3 is a guest-physical address; see Section 28.2.1.

- **MOV from CR4.** The behavior of MOV from CR4 is determined by the CR4 guest/host mask and the CR4 read shadow. For each position corresponding to a bit clear in the CR4 guest/host mask, the destination operand is loaded with the value of the corresponding bit in CR4. For each position corresponding to a bit set in the CR4 guest/host mask, the destination operand is loaded with the value of the corresponding bit in the CR4 read shadow. Thus, if every bit is cleared in the CR4 guest/host mask, MOV from CR4 reads normally from CR4; if every bit is set in the CR4 guest/host mask, MOV from CR4 returns the value of the CR4 read shadow.

  Depending on the contents of the CR4 guest/host mask and the CR4 read shadow, bits may be set in the destination that would never be set when reading directly from CR4.

- **MOV from CR8.** If the MOV from CR8 instruction does not cause a VM exit (see Section 25.1.3), its behavior is modified if the "use TPR shadow" VM-execution control is 1; see Section 29.3.

- **MOV to CR0.** An execution of MOV to CR0 that does not cause a VM exit (see Section 25.1.3) leaves unmodified any bit in CR0 corresponding to a bit set in the CR0 guest/host mask. Treatment of attempts to modify other bits in CR0 depends on the setting of the "unrestricted guest" VM-execution control:

  — If the control is 0, MOV to CR0 causes a general-protection exception if it attempts to set any bit in CR0 to a value not supported in VMX operation (see Section 23.8).

  — If the control is 1, MOV to CR0 causes a general-protection exception if it attempts to set any bit in CR0 other than bit 0 (PE) or bit 31 (PG) to a value not supported in VMX operation. It remains the case, however, that MOV to CR0 causes a general-protection exception if it would result in CR0.PE = 0 and CR0.PG = 1 or if it would result in CR0.PG = 1, CR4.PAE = 0, and IA32_EFER.LME = 1.

- **MOV to CR3.** If the "enable EPT" VM-execution control is 1 and an execution of MOV to CR3 does not cause a VM exit (see Section 25.1.3), the value loaded into CR3 is treated as a guest-physical address; see Section 28.2.1.

  — If PAE paging is not being used, the instruction does not use the guest-physical address to access memory and it does not cause it to be translated through EPT.[1]

— If PAE paging is being used, the instruction translates the guest-physical address through EPT and uses the result to load the four (4) page-directory-pointer-table entries (PDPTEs). The instruction does not use the guest-physical addresses the PDPTEs to access memory and it does not cause them to be translated through EPT.

- **MOV to CR4.** An execution of MOV to CR4 that does not cause a VM exit (see Section 25.1.3) leaves unmodified any bit in CR4 corresponding to a bit set in the CR4 guest/host mask. Such an execution causes a general-protection exception if it attempts to set any bit in CR4 (not corresponding to a bit set in the CR4 guest/host mask) to a value not supported in VMX operation (see Section 23.8).
- **MOV to CR8.** If the MOV **to** CR8 instruction does not cause a VM exit (see Section 25.1.3), its behavior is modified if the "use TPR shadow" VM-execution control is 1; see Section 29.3.
- **MWAIT.** Behavior of the MWAIT instruction (which always causes an invalid-opcode exception—#UD—if CPL > 0) is determined by the setting of the "MWAIT exiting" VM-execution control:
  - If the "MWAIT exiting" VM-execution control is 1, MWAIT causes a VM exit.
  - If the "MWAIT exiting" VM-execution control is 0, MWAIT operates normally if one of the following are true: (1) ECX[0] is 0; (2) RFLAGS.IF = 1; or both of the following are true: (a) the "interrupt-window exiting" VM-execution control is 0; and (b) the logical processor has not recognized a pending virtual interrupt (see Section 29.2.1).
  - If the "MWAIT exiting" VM-execution control is 0, ECX[0] = 1, and RFLAGS.IF = 0, MWAIT does not cause the processor to enter an implementation-dependent optimized state if either the "interrupt-window exiting" VM-execution control is 1 or the logical processor has recognized a pending virtual interrupt; instead, control passes to the instruction following the MWAIT instruction.
- **RDMSR.** Section 25.1.3 identifies when executions of the RDMSR instruction cause VM exits. If such an execution causes neither a fault due to CPL > 0 nor a VM exit, the instruction's behavior may be modified for certain values of ECX:
  - If ECX contains 10H (indicating the IA32_TIME_STAMP_COUNTER MSR), the value returned by the instruction is determined by the setting of the "use TSC offsetting" VM-execution control:
    - If the control is 0, RDMSR operates normally, loading EAX:EDX with the value of the IA32_TIME_STAMP_COUNTER MSR.
    - If the control is 1, the value returned is determined by the setting of the "use TSC scaling" VM-execution control:
      - If the control is 0, RDMSR loads EAX:EDX with the sum of the value of the IA32_TIME_STAMP_COUNTER MSR and the value of the TSC offset.
      - If the control is 1, RDMSR first computes the product of the value of the IA32_TIME_STAMP_COUNTER MSR and the value of the TSC multiplier. It then shifts the value of the product right 48 bits and loads EAX:EDX with the sum of that shifted value and the value of the TSC offset.

    The 1-setting of the "use TSC-offsetting" VM-execution control does not affect executions of RDMSR if ECX contains 6E0H (indicating the IA32_TSC_DEADLINE MSR). Such executions return the APIC-timer deadline relative to the actual timestamp counter without regard to the TSC offset.
  - If ECX is in the range 800H–8FFH (indicating an APIC MSR), instruction behavior may be modified if the "virtualize x2APIC mode" VM-execution control is 1; see Section 29.5.
- **RDPID.** Behavior of the RDPID instruction is determined first by the setting of the "enable RDTSCP" VM-execution control:
  - If the "enable RDTSCP" VM-execution control is 0, RDPID causes an invalid-opcode exception (#UD).
  - If the "enable RDTSCP" VM-execution control is 1, RDPID operates normally.
- **RDTSC.** Behavior of the RDTSC instruction is determined by the settings of the "RDTSC exiting" and "use TSC offsetting" VM-execution controls:

1. A logical processor uses PAE paging if CR0.PG = 1, CR4.PAE = 1 and IA32_EFER.LMA = 0. See Section 4.4 in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*.

— If both controls are 0, RDTSC operates normally.

— If the "RDTSC exiting" VM-execution control is 0 and the "use TSC offsetting" VM-execution control is 1, the value returned is determined by the setting of the "use TSC scaling" VM-execution control:

- If the control is 0, RDTSC loads EAX:EDX with the sum of the value of the IA32_TIME_STAMP_COUNTER MSR and the value of the TSC offset.
- If the control is 1, RDTSC first computes the product of the value of the IA32_TIME_STAMP_COUNTER MSR and the value of the TSC multiplier. It then shifts the value of the product right 48 bits and loads EAX:EDX with the sum of that shifted value and the value of the TSC offset.

— If the "RDTSC exiting" VM-execution control is 1, RDTSC causes a VM exit.

- **RDTSCP.** Behavior of the RDTSCP instruction is determined first by the setting of the "enable RDTSCP" VM-execution control:

  — If the "enable RDTSCP" VM-execution control is 0, RDTSCP causes an invalid-opcode exception (#UD). This exception takes priority over any other exception the instruction may incur.

  — If the "enable RDTSCP" VM-execution control is 1, treatment is based on the settings of the "RDTSC exiting" and "use TSC offsetting" VM-execution controls:

  - If both controls are 0, RDTSCP operates normally.
  - If the "RDTSC exiting" VM-execution control is 0 and the "use TSC offsetting" VM-execution control is 1, the value returned is determined by the setting of the "use TSC scaling" VM-execution control:

    — If the control is 0, RDTSCP loads EAX:EDX with the sum of the value of the IA32_TIME_STAMP_COUNTER MSR and the value of the TSC offset.

    — If the control is 1, RDTSCP first computes the product of the value of the IA32_TIME_STAMP_COUNTER MSR and the value of the TSC multiplier. It then shifts the value of the product right 48 bits and loads EAX:EDX with the sum of that shifted value and the value of the TSC offset.

    In either case, RDTSCP also loads ECX with the value of bits 31:0 of the IA32_TSC_AUX MSR.

  - If the "RDTSC exiting" VM-execution control is 1, RDTSCP causes a VM exit.

- **SMSW.** The behavior of SMSW is determined by the CR0 guest/host mask and the CR0 read shadow. For each position corresponding to a bit clear in the CR0 guest/host mask, the destination operand is loaded with the value of the corresponding bit in CR0. For each position corresponding to a bit set in the CR0 guest/host mask, the destination operand is loaded with the value of the corresponding bit in the CR0 read shadow. Thus, if every bit is cleared in the CR0 guest/host mask, SMSW reads normally from CR0; if every bit is set in the CR0 guest/host mask, SMSW returns the value of the CR0 read shadow.

  Note the following: (1) for any memory destination or for a 16-bit register destination, only the low 16 bits of the CR0 guest/host mask and the CR0 read shadow are used (bits 63:16 of a register destination are left unchanged); (2) for a 32-bit register destination, only the low 32 bits of the CR0 guest/host mask and the CR0 read shadow are used (bits 63:32 of the destination are cleared); and (3) depending on the contents of the CR0 guest/host mask and the CR0 read shadow, bits may be set in the destination that would never be set when reading directly from CR0.

- **WRMSR.** Section 25.1.3 identifies when executions of the WRMSR instruction cause VM exits. If such an execution neither a fault due to CPL > 0 nor a VM exit, the instruction's behavior may be modified for certain values of ECX:

  — If ECX contains 79H (indicating IA32_BIOS_UPDT_TRIG MSR), no microcode update is loaded, and control passes to the next instruction. This implies that microcode updates cannot be loaded in VMX non-root operation.

  — On processors that support Intel PT but which do not allow it to be used in VMX operation, if ECX contains 570H (indicating the IA32_RTIT_CTL MSR), the instruction causes a general-protection exception.[1]

1. Software should read the VMX capability MSR IA32_VMX_MISC to determine whether the processor allows Intel PT to be used in VMX operation (see Appendix A.6).

— If ECX contains 808H (indicating the TPR MSR), 80BH (the EOI MSR), or 83FH (self-IPI MSR), instruction behavior may modified if the "virtualize x2APIC mode" VM-execution control is 1; see Section 29.5.

- **XRSTORS.** Behavior of the XRSTORS instruction is determined first by the setting of the "enable XSAVES/XRSTORS" VM-execution control:
  — If the "enable XSAVES/XRSTORS" VM-execution control is 0, XRSTORS causes an invalid-opcode exception (#UD).
  — If the "enable XSAVES/XRSTORS" VM-execution control is 1, treatment is based on the value of the XSS-exiting bitmap (see Section 24.6.19):
    - XRSTORS causes a VM exit if any bit is set in the logical-AND of the following three values: EDX:EAX, the IA32_XSS MSR, and the XSS-exiting bitmap.
    - Otherwise, XRSTORS operates normally.
- **XSAVES.** Behavior of the XSAVES instruction is determined first by the setting of the "enable XSAVES/XRSTORS" VM-execution control:
  — If the "enable XSAVES/XRSTORS" VM-execution control is 0, XSAVES causes an invalid-opcode exception (#UD).
  — If the "enable XSAVES/XRSTORS" VM-execution control is 1, treatment is based on the value of the XSS-exiting bitmap (see Section 24.6.19):
    - XSAVES causes a VM exit if any bit is set in the logical-AND of the following three values: EDX:EAX, the IA32_XSS MSR, and the XSS-exiting bitmap.
    - Otherwise, XSAVES operates normally.

25.4 OTHER CHANGES IN VMX NON-ROOT OPERATION

Treatments of event blocking and of task switches differ in VMX non-root operation as described in the following sections.

25.4.1 Event Blocking

Event blocking is modified in VMX non-root operation as follows:

- If the "external-interrupt exiting" VM-execution control is 1, RFLAGS.IF does not control the blocking of external interrupts. In this case, an external interrupt that is not blocked for other reasons causes a VM exit (even if RFLAGS.IF = 0).
- If the "external-interrupt exiting" VM-execution control is 1, external interrupts may or may not be blocked by STI or by MOV SS (behavior is implementation-specific).
- If the "NMI exiting" VM-execution control is 1, non-maskable interrupts (NMIs) may or may not be blocked by STI or by MOV SS (behavior is implementation-specific).

25.4.2 Treatment of Task Switches

Task switches are not allowed in VMX non-root operation. Any attempt to effect a task switch in VMX non-root operation causes a VM exit. However, the following checks are performed (in the order indicated), possibly resulting in a fault, before there is any possibility of a VM exit due to task switch:

1. If a task gate is being used, appropriate checks are made on its P bit and on the proper values of the relevant privilege fields. The following cases detail the privilege checks performed:
   a. If CALL, INT *n*, or JMP accesses a task gate in IA-32e mode, a general-protection exception occurs.
   b. If CALL, INT *n*, INT3, INTO, or JMP accesses a task gate outside IA-32e mode, privilege-levels checks are performed on the task gate but, if they pass, privilege levels are not checked on the referenced task-state segment (TSS) descriptor.

c. If CALL or JMP accesses a TSS descriptor directly in IA-32e mode, a general-protection exception occurs.

d. If CALL or JMP accesses a TSS descriptor directly outside IA-32e mode, privilege levels are checked on the TSS descriptor.

e. If a non-maskable interrupt (NMI), an exception, or an external interrupt accesses a task gate in the IDT in IA-32e mode, a general-protection exception occurs.

f. If a non-maskable interrupt (NMI), an exception other than breakpoint exceptions (#BP) and overflow exceptions (#OF), or an external interrupt accesses a task gate in the IDT outside IA-32e mode, no privilege checks are performed.

g. If IRET is executed with RFLAGS.NT = 1 in IA-32e mode, a general-protection exception occurs.

h. If IRET is executed with RFLAGS.NT = 1 outside IA-32e mode, a TSS descriptor is accessed directly and no privilege checks are made.

2. Checks are made on the new TSS selector (for example, that is within GDT limits).

3. The new TSS descriptor is read. (A page fault results if a relevant GDT page is not present).

4. The TSS descriptor is checked for proper values of type (depends on type of task switch), P bit, S bit, and limit.

Only if checks 1–4 all pass (do not generate faults) might a VM exit occur. However, the ordering between a VM exit due to a task switch and a page fault resulting from accessing the old TSS or the new TSS is implementation-specific. Some processors may generate a page fault (instead of a VM exit due to a task switch) if accessing either TSS would cause a page fault. Other processors may generate a VM exit due to a task switch even if accessing either TSS would cause a page fault.

If an attempt at a task switch through a task gate in the IDT causes an exception (before generating a VM exit due to the task switch) and that exception causes a VM exit, information about the event whose delivery that accessed the task gate is recorded in the IDT-vectoring information fields and information about the exception that caused the VM exit is recorded in the VM-exit interruption-information fields. See Section 27.2. The fact that a task gate was being accessed is not recorded in the VMCS.

If an attempt at a task switch through a task gate in the IDT causes VM exit due to the task switch, information about the event whose delivery accessed the task gate is recorded in the IDT-vectoring fields of the VMCS. Since the cause of such a VM exit is a task switch and not an interruption, the valid bit for the VM-exit interruption information field is 0. See Section 27.2.

25.5 FEATURES SPECIFIC TO VMX NON-ROOT OPERATION

Some VM-execution controls support features that are specific to VMX non-root operation. These are the VMX-preemption timer (Section 25.5.1) and the monitor trap flag (Section 25.5.2), translation of guest-physical addresses (Section 25.5.3), VM functions (Section 25.5.5), and virtualization exceptions (Section 25.5.6).

25.5.1 VMX-Preemption Timer

If the last VM entry was performed with the 1-setting of "activate VMX-preemption timer" VM-execution control, the **VMX-preemption timer** counts down (from the value loaded by VM entry; see Section 26.6.4) in VMX non-root operation. When the timer counts down to zero, it stops counting down and a VM exit occurs (see Section 25.2).

The VMX-preemption timer counts down at rate proportional to that of the timestamp counter (TSC). Specifically, the timer counts down by 1 every time bit X in the TSC changes due to a TSC increment. The value of X is in the range 0–31 and can be determined by consulting the VMX capability MSR IA32_VMX_MISC (see Appendix A.6).

The VMX-preemption timer operates in the C-states C0, C1, and C2; it also operates in the shutdown and wait-for-SIPI states. If the timer counts down to zero in any state other than the wait-for SIPI state, the logical processor transitions to the C0 C-state and causes a VM exit; the timer does not cause a VM exit if it counts down to zero in the wait-for-SIPI state. The timer is not decremented in C-states deeper than C2.

Treatment of the timer in the case of system management interrupts (SMIs) and system-management mode (SMM) depends on whether the treatment of SMIs and SMM:

- If the default treatment of SMIs and SMM (see Section 34.14) is active, the VMX-preemption timer counts across an SMI to VMX non-root operation, subsequent execution in SMM, and the return from SMM via the RSM instruction. However, the timer can cause a VM exit only from VMX non-root operation. If the timer expires during SMI, in SMM, or during RSM, a timer-induced VM exit occurs immediately after RSM with its normal priority unless it is blocked based on activity state (Section 25.2).
- If the dual-monitor treatment of SMIs and SMM (see Section 34.15) is active, transitions into and out of SMM are VM exits and VM entries, respectively. The treatment of the VMX-preemption timer by those transitions is mostly the same as for ordinary VM exits and VM entries; Section 34.15.2 and Section 34.15.4 detail some differences.

25.5.2 Monitor Trap Flag

The **monitor trap flag** is a debugging feature that causes VM exits to occur on certain instruction boundaries in VMX non-root operation. Such VM exits are called **MTF VM exits**. An MTF VM exit may occur on an instruction boundary in VMX non-root operation as follows:

- If the "monitor trap flag" VM-execution control is 1 and VM entry is injecting a vectored event (see Section 26.5.1), an MTF VM exit is pending on the instruction boundary before the first instruction following the VM entry.
- If VM entry is injecting a pending MTF VM exit (see Section 26.5.2), an MTF VM exit is pending on the instruction boundary before the first instruction following the VM entry. This is the case even if the "monitor trap flag" VM-execution control is 0.
- If the "monitor trap flag" VM-execution control is 1, VM entry is not injecting an event, and a pending event (e.g., debug exception or interrupt) is delivered before an instruction can execute, an MTF VM exit is pending on the instruction boundary following delivery of the event (or any nested exception).
- Suppose that the "monitor trap flag" VM-execution control is 1, VM entry is not injecting an event, and the first instruction following VM entry is a REP-prefixed string instruction:
  - If the first iteration of the instruction causes a fault, an MTF VM exit is pending on the instruction boundary following delivery of the fault (or any nested exception).
  - If the first iteration of the instruction does not cause a fault, an MTF VM exit is pending on the instruction boundary after that iteration.
- Suppose that the "monitor trap flag" VM-execution control is 1, VM entry is not injecting an event, and the first instruction following VM entry is the XBEGIN instruction. In this case, an MTF VM exit is pending at the fallback instruction address of the XBEGIN instruction. This behavior applies regardless of whether advanced debugging of RTM transactional regions has been enabled (see Section 16.3.7, "RTM-Enabled Debugger Support," of *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*).
- Suppose that the "monitor trap flag" VM-execution control is 1, VM entry is not injecting an event, and the first instruction following VM entry is neither a REP-prefixed string instruction or the XBEGIN instruction:
  - If the instruction causes a fault, an MTF VM exit is pending on the instruction boundary following delivery of the fault (or any nested exception).[1]
  - If the instruction does not cause a fault, an MTF VM exit is pending on the instruction boundary following execution of that instruction. If the instruction is INT3 or INTO, this boundary follows delivery of any software exception. If the instruction is INT *n*, this boundary follows delivery of a software interrupt. If the instruction is HLT, the MTF VM exit will be from the HLT activity state.

No MTF VM exit occurs if another VM exit occurs before reaching the instruction boundary on which an MTF VM exit would be pending (e.g., due to an exception or triple fault).

An MTF VM exit occurs on the instruction boundary on which it is pending unless a higher priority event takes precedence or the MTF VM exit is blocked due to the activity state:

- System-management interrupts (SMIs), INIT signals, and higher priority events take priority over MTF VM exits. MTF VM exits take priority over debug-trap exceptions and lower priority events.

1. This item includes the cases of an invalid opcode exception—#UD— generated by the UD instruction and a BOUND-range exceeded exception—#BR—generated by the BOUND instruction.

- No MTF VM exit occurs if the processor is in either the shutdown activity state or wait-for-SIPI activity state. If a non-maskable interrupt subsequently takes the logical processor out of the shutdown activity state without causing a VM exit, an MTF VM exit is pending after delivery of that interrupt.

Special treatment may apply to Intel SGX instructions or if the logical processor is in enclave mode. See Section 42.2 for details.

25.5.3 Translation of Guest-Physical Addresses Using EPT

The extended page-table mechanism (EPT) is a feature that can be used to support the virtualization of physical memory. When EPT is in use, certain physical addresses are treated as guest-physical addresses and are not used to access memory directly. Instead, guest-physical addresses are translated by traversing a set of EPT paging structures to produce physical addresses that are used to access memory.

Details of the EPT mechanism are given in Section 28.2.

25.5.4 APIC Virtualization

APIC virtualization is a collection of features that can be used to support the virtualization of interrupts and the Advanced Programmable Interrupt Controller (APIC). When APIC virtualization is enabled, the processor emulates many accesses to the APIC, tracks the state of the virtual APIC, and delivers virtual interrupts — all in VMX non-root operation without a VM exit.

Details of the APIC virtualization are given in Chapter 29.

25.5.5 VM Functions

A **VM function** is an operation provided by the processor that can be invoked from VMX non-root operation without a VM exit. VM functions are enabled and configured by the settings of different fields in the VMCS. Software in VMX non-root operation invokes a VM function with the **VMFUNC** instruction; the value of EAX selects the specific VM function being invoked.

Section 25.5.5.1 explains how VM functions are enabled. Section 25.5.5.2 specifies the behavior of the VMFUNC instruction. Section 25.5.5.3 describes a specific VM function called **EPTP switching**.

25.5.5.1 Enabling VM Functions

Software enables VM functions generally by setting the "enable VM functions" VM-execution control. A specific VM function is enabled by setting the corresponding VM-function control.

Suppose, for example, that software wants to enable EPTP switching (VM function 0; see Section 24.6.14).To do so, it must set the "activate secondary controls" VM-execution control (bit 31 of the primary processor-based VM-execution controls), the "enable VM functions" VM-execution control (bit 13 of the secondary processor-based VM-execution controls) and the "EPTP switching" VM-function control (bit 0 of the VM-function controls).

25.5.5.2 General Operation of the VMFUNC Instruction

The VMFUNC instruction causes an invalid-opcode exception (#UD) if the "enable VM functions" VM-execution controls is 0[1] or the value of EAX is greater than 63 (only VM functions 0–63 can be enable). Otherwise, the instruction causes a VM exit if the bit at position EAX is 0 in the VM-function controls (the selected VM function is not enabled). If such a VM exit occurs, the basic exit reason used is 59 (3BH), indicating "VMFUNC", and the length of the VMFUNC instruction is saved into the VM-exit instruction-length field. If the instruction causes neither an invalid-opcode exception nor a VM exit due to a disabled VM function, it performs the functionality of the VM function specified by the value in EAX.

1. "Enable VM functions" is a secondary processor-based VM-execution control. If bit 31 of the primary processor-based VM-execution controls is 0, VMX non-root operation functions as if the "enable VM functions" VM-execution control were 0. See Section 24.6.2.

Individual VM functions may perform additional fault checking (e.g., one might cause a general-protection exception if CPL > 0). In addition, specific VM functions may include checks that might result in a VM exit. If such a VM exit occurs, VM-exit information is saved as described in the previous paragraph. The specification of a VM function may indicate that additional VM-exit information is provided.

The specific behavior of the EPTP-switching VM function (including checks that result in VM exits) is given in Section 25.5.5.3.

25.5.5.3 EPTP Switching

EPTP switching is VM function 0. This VM function allows software in VMX non-root operation to load a new value for the EPT pointer (EPTP), thereby establishing a different EPT paging-structure hierarchy (see Section 28.2 for details of the operation of EPT). Software is limited to selecting from a list of potential EPTP values configured in advance by software in VMX root operation.

Specifically, the value of ECX is used to select an entry from the EPTP list, the 4-KByte structure referenced by the EPTP-list address (see Section 24.6.14; because this structure contains 512 8-Byte entries, VMFUNC causes a VM exit if ECX ≥ 512). If the selected entry is a valid EPTP value (it would not cause VM entry to fail; see Section 26.2.1.1), it is stored in the EPTP field of the current VMCS and is used for subsequent accesses using guest-physical addresses. The following pseudocode provides details:

```
IF ECX ≥ 512
    THEN VM exit;
    ELSE
        tent_EPTP ← 8 bytes from EPTP-list address + 8 * ECX;
        IF tent_EPTP is not a valid EPTP value (would cause VM entry to fail if in EPTP)
            THEN VMexit;
            ELSE
                write tent_EPTP to the EPTP field in the current VMCS;
                use tent_EPTP as the new EPTP value for address translation;
                IF processor supports the 1-setting of the "EPT-violation #VE" VM-execution control
                    THEN
                        write ECX[15:0] to EPTP-index field in current VMCS;
                        use ECX[15:0] as EPTP index for subsequent EPT-violation virtualization exceptions (see Section 25.5.6.2);
                FI;
        FI;
FI;
```

Execution of the EPTP-switching VM function does not modify the state of any registers; no flags are modified.

As noted in Section 25.5.5.2, an execution of the EPTP-switching VM function that causes a VM exit (as specified above), uses the basic exit reason 59, indicating "VMFUNC". The length of the VMFUNC instruction is saved into the VM-exit instruction-length field. No additional VM-exit information is provided.

An execution of VMFUNC loads EPTP from the EPTP list (and thus does not cause a fault or VM exit) is called an **EPTP-switching VMFUNC**. After an EPTP-switching VMFUNC, control passes to the next instruction. The logical processor starts creating and using guest-physical and combined mappings associated with the new value of bits 51:12 of EPTP; the combined mappings created and used are associated with the current VPID and PCID (these are not changed by VMFUNC).[1] If the "enable VPID" VM-execution control is 0, an EPTP-switching VMFUNC invalidates combined mappings associated with VPID 0000H (for all PCIDs and for all EP4TA values, where EP4TA is the value of bits 51:12 of EPTP).

Because an EPTP-switching VMFUNC may change the translation of guest-physical addresses, it may affect use of the guest-physical address in CR3. The EPTP-switching VMFUNC cannot itself cause a VM exit due to an EPT violation or an EPT misconfiguration due to the translation of that guest-physical address through the new EPT paging structures. The following items provide details that apply if CR0.PG = 1:

1. If the "enable VPID" VM-execution control is 0, the current VPID is 0000H; if CR4.PCIDE = 0, the current PCID is 000H.

- If 32-bit paging or 4-level paging[1] is in use (either CR4.PAE = 0 or IA32_EFER.LMA = 1), the next memory access with a linear address uses the translation of the guest-physical address in CR3 through the new EPT paging structures. As a result, this access may cause a VM exit due to an EPT violation or an EPT misconfiguration encountered during that translation.
- If PAE paging is in use (CR4.PAE = 1 and IA32_EFER.LMA = 0), an EPTP-switching VMFUNC **does not** load the four page-directory-pointer-table entries (PDPTEs) from the guest-physical address in CR3. The logical processor continues to use the four guest-physical addresses already present in the PDPTEs. The guest-physical address in CR3 is not translated through the new EPT paging structures (until some operation that would load the PDPTEs).

  The EPTP-switching VMFUNC cannot itself cause a VM exit due to an EPT violation or an EPT misconfiguration encountered during the translation of a guest-physical address in any of the PDPTEs. A subsequent memory access with a linear address uses the translation of the guest-physical address in the appropriate PDPTE through the new EPT paging structures. As a result, such an access may cause a VM exit due to an EPT violation or an EPT misconfiguration encountered during that translation.

If an EPTP-switching VMFUNC establishes an EPTP value that enables accessed and dirty flags for EPT (by setting bit 6), subsequent memory accesses may fail to set those flags as specified if there has been no appropriate execution of INVEPT since the last use of an EPTP value that does not enable accessed and dirty flags for EPT (because bit 6 is clear) and that is identical to the new value on bits 51:12.

IF the processor supports the 1-setting of the "EPT-violation #VE" VM-execution control, an EPTP-switching VMFUNC loads the value in ECX[15:0] into to EPTP-index field in current VMCS. Subsequent EPT-violation virtualization exceptions will save this value into the virtualization-exception information area (see Section 25.5.6.2);

25.5.6 Virtualization Exceptions

A **virtualization exception** is a new processor exception. It uses vector 20 and is abbreviated #VE.

A virtualization exception can occur only in VMX non-root operation. Virtualization exceptions occur only with certain settings of certain VM-execution controls. Generally, these settings imply that certain conditions that would normally cause VM exits instead cause virtualization exceptions

In particular, the 1-setting of the "EPT-violation #VE" VM-execution control causes some EPT violations to generate virtualization exceptions instead of VM exits. Section 25.5.6.1 provides the details of how the processor determines whether an EPT violation causes a virtualization exception or a VM exit.

When the processor encounters a virtualization exception, it saves information about the exception to the virtualization-exception information area; see Section 25.5.6.2.

After saving virtualization-exception information, the processor delivers a virtualization exception as it would any other exception; see Section 25.5.6.3 for details.

25.5.6.1 Convertible EPT Violations

If the "EPT-violation #VE" VM-execution control is 0 (e.g., on processors that do not support this feature), EPT violations always cause VM exits. If instead the control is 1, certain EPT violations may be converted to cause virtualization exceptions instead; such EPT violations are **convertible**.

The values of certain EPT paging-structure entries determine which EPT violations are convertible. Specifically, bit 63 of certain EPT paging-structure entries may be defined to mean **suppress #VE**:

- If bits 2:0 of an EPT paging-structure entry are all 0, the entry is not **present**.[2] If the processor encounters such an entry while translating a guest-physical address, it causes an EPT violation. The EPT violation is convertible if and only if bit 63 of the entry is 0.
- If an EPT paging-structure entry is present, the following cases apply:

1. Earlier versions of this manual used the term "IA-32e paging" to identify 4-level paging.
2. If the "mode-based execute control for EPT" VM-execution control is 1, an EPT paging-structure entry is present if any of bits 2:0 **or bit 10** is 1.

— If the value of the EPT paging-structure entry is not supported, the entry is **misconfigured**. If the processor encounters such an entry while translating a guest-physical address, it causes an EPT misconfiguration (not an EPT violation). EPT misconfigurations always cause VM exits.

— If the value of the EPT paging-structure entry is supported, the following cases apply:

- If bit 7 of the entry is 1, or if the entry is an EPT PTE, the entry maps a page. If the processor uses such an entry to translate a guest-physical address, and if an access to that address causes an EPT violation, the EPT violation is convertible if and only if bit 63 of the entry is 0.
- If bit 7 of the entry is 0 and the entry is not an EPT PTE, the entry references another EPT paging structure. The processor does not use the value of bit 63 of the entry to determine whether any subsequent EPT violation is convertible.

If an access to a guest-physical address causes an EPT violation, bit 63 of exactly one of the EPT paging-structure entries used to translate that address is used to determine whether the EPT violation is convertible: either a entry that is not present (if the guest-physical address does not translate to a physical address) or an entry that maps a page (if it does).

A convertible EPT violation instead causes a virtualization exception if the following all hold:

- CR0.PE = 1;
- the logical processor is not in the process of delivering an event through the IDT; and
- the 32 bits at offset 4 in the virtualization-exception information area are all 0.

Delivery of virtualization exceptions writes the value FFFFFFFFH to offset 4 in the virtualization-exception information area (see Section 25.5.6.2). Thus, once a virtualization exception occurs, another can occur only if software clears this field.

25.5.6.2 Virtualization-Exception Information

Virtualization exceptions save data into the virtualization-exception information area (see Section 24.6.18). Table 25-1 enumerates the data saved and the format of the area.

Table 25-1. Format of the Virtualization-Exception Information Area

Byte Offset	Contents
0	The 32-bit value that would have been saved into the VMCS as an exit reason had a VM exit occurred instead of the virtualization exception. For EPT violations, this value is 48 (00000030H)
4	FFFFFFFFH
8	The 64-bit value that would have been saved into the VMCS as an exit qualification had a VM exit occurred instead of the virtualization exception
16	The 64-bit value that would have been saved into the VMCS as a guest-linear address had a VM exit occurred instead of the virtualization exception
24	The 64-bit value that would have been saved into the VMCS as a guest-physical address had a VM exit occurred instead of the virtualization exception
32	The current 16-bit value of the EPTP index VM-execution control (see Section 24.6.18 and Section 25.5.5.3)

25.5.6.3 Delivery of Virtualization Exceptions

After saving virtualization-exception information, the processor treats a virtualization exception as it does other exceptions:

- If bit 20 (#VE) is 1 in the exception bitmap in the VMCS, a virtualization exception causes a VM exit (see below). If the bit is 0, the virtualization exception is delivered using gate descriptor 20 in the IDT.

- Virtualization exceptions produce no error code. Delivery of a virtualization exception pushes no error code on the stack.
- With respect to double faults, virtualization exceptions have the same severity as page faults. If delivery of a virtualization exception encounters a nested fault that is either contributory or a page fault, a double fault (#DF) is generated. See Chapter 6, "Interrupt 8—Double Fault Exception (#DF)" in *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*.

  It is not possible for a virtualization exception to be encountered while delivering another exception (see Section 25.5.6.1).

If a virtualization exception causes a VM exit directly (because bit 20 is 1 in the exception bitmap), information about the exception is saved normally in the VM-exit interruption information field in the VMCS (see Section 27.2.2). Specifically, the event is reported as a hardware exception with vector 20 and no error code. Bit 12 of the field (NMI unblocking due to IRET) is set normally.

If a virtualization exception causes a VM exit indirectly (because bit 20 is 0 in the exception bitmap and delivery of the exception generates an event that causes a VM exit), information about the exception is saved normally in the IDT-vectoring information field in the VMCS (see Section 27.2.3). Specifically, the event is reported as a hardware exception with vector 20 and no error code.

25.6 UNRESTRICTED GUESTS

The first processors to support VMX operation require CR0.PE and CR0.PG to be 1 in VMX operation (see Section 23.8). This restriction implies that guest software cannot be run in unpaged protected mode or in real-address mode. Later processors support a VM-execution control called "unrestricted guest".[1] If this control is 1, CR0.PE and CR0.PG may be 0 in VMX non-root operation. Such processors allow guest software to run in unpaged protected mode or in real-address mode. The following items describe the behavior of such software:

- The MOV CR0 instructions does not cause a general-protection exception simply because it would set either CR0.PE and CR0.PG to 0. See Section 25.3 for details.
- A logical processor treats the values of CR0.PE and CR0.PG in VMX non-root operation just as it does outside VMX operation. Thus, if CR0.PE = 0, the processor operates as it does normally in real-address mode (for example, it uses the 16-bit **interrupt table** to deliver interrupts and exceptions). If CR0.PG = 0, the processor operates as it does normally when paging is disabled.
- Processor operation is modified by the fact that the processor is in VMX non-root operation and by the settings of the VM-execution controls just as it is in protected mode or when paging is enabled. Instructions, interrupts, and exceptions that cause VM exits in protected mode or when paging is enabled also do so in real-address mode or when paging is disabled. The following examples should be noted:
  - If CR0.PG = 0, page faults do not occur and thus cannot cause VM exits.
  - If CR0.PE = 0, invalid-TSS exceptions do not occur and thus cannot cause VM exits.
  - If CR0.PE = 0, the following instructions cause invalid-opcode exceptions and do not cause VM exits: INVEPT, INVVPID, LLDT, LTR, SLDT, STR, VMCLEAR, VMLAUNCH, VMPTRLD, VMPTRST, VMREAD, VMRESUME, VMWRITE, VMXOFF, and VMXON.
- If CR0.PG = 0, each linear address is passed directly to the EPT mechanism for translation to a physical address.[2] The guest memory type passed on to the EPT mechanism is WB (writeback).

1. "Unrestricted guest" is a secondary processor-based VM-execution control. If bit 31 of the primary processor-based VM-execution controls is 0, VMX non-root operation functions as if the "unrestricted guest" VM-execution control were 0. See Section 24.6.2.
2. As noted in Section 26.2.1.1, the "enable EPT" VM-execution control must be 1 if the "unrestricted guest" VM-execution control is 1.

19.Updates to Chapter 27, Volume 3C

Change bars show changes to Chapter 27 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3C:* System Programming Guide, Part 3.

Change to this chapter: Minor update to Section 27.1 "Architectural State Before a VM Exit".

CHAPTER 27
VM EXITS

VM exits occur in response to certain instructions and events in VMX non-root operation as detailed in Section 25.1 through Section 25.2. VM exits perform the following operations:

1. Information about the cause of the VM exit is recorded in the VM-exit information fields and VM-entry control fields are modified as described in Section 27.2.
2. Processor state is saved in the guest-state area (Section 27.3).
3. MSRs may be saved in the VM-exit MSR-store area (Section 27.4). This step is not performed for SMM VM exits that activate the dual-monitor treatment of SMIs and SMM.
4. The following may be performed in parallel and in any order (Section 27.5):
   — Processor state is loaded based in part on the host-state area and some VM-exit controls. This step is not performed for SMM VM exits that activate the dual-monitor treatment of SMIs and SMM. See Section 34.15.6 for information on how processor state is loaded by such VM exits.
   — Address-range monitoring is cleared.
5. MSRs may be loaded from the VM-exit MSR-load area (Section 27.6). This step is not performed for SMM VM exits that activate the dual-monitor treatment of SMIs and SMM.

VM exits are not logged with last-branch records, do not produce branch-trace messages, and do not update the branch-trace store.

Section 27.1 clarifies the nature of the architectural state before a VM exit begins. The steps described above are detailed in Section 27.2 through Section 27.6.

Section 34.15 describes the dual-monitor treatment of system-management interrupts (SMIs) and system-management mode (SMM). Under this treatment, ordinary transitions to SMM are replaced by VM exits to a separate SMM monitor. Called **SMM VM exits**, these are caused by the arrival of an SMI or the execution of VMCALL in VMX root operation. SMM VM exits differ from other VM exits in ways that are detailed in Section 34.15.2.

27.1 ARCHITECTURAL STATE BEFORE A VM EXIT

This section describes the architectural state that exists before a VM exit, especially for VM exits caused by events that would normally be delivered through the IDT. Note the following:

- An exception causes a VM exit **directly** if the bit corresponding to that exception is set in the exception bitmap. A non-maskable interrupt (NMI) causes a VM exit directly if the "NMI exiting" VM-execution control is 1. An external interrupt causes a VM exit directly if the "external-interrupt exiting" VM-execution control is 1. A start-up IPI (SIPI) that arrives while a logical processor is in the wait-for-SIPI activity state causes a VM exit directly. INIT signals that arrive while the processor is not in the wait-for-SIPI activity state cause VM exits directly.
- An exception, NMI, external interrupt, or software interrupt causes a VM exit **indirectly** if it does not do so directly but delivery of the event causes a nested exception, double fault, task switch, APIC access (see Section 27.4), EPT violation, EPT misconfiguration, or page-modification log-full event that causes a VM exit.
- An event **results** in a VM exit if it causes a VM exit (directly or indirectly).

The following bullets detail when architectural state is and is not updated in response to VM exits:

- If an event causes a VM exit directly, it does not update architectural state as it would have if it had it not caused the VM exit:
  — A debug exception does not update DR6, DR7.GD, or IA32_DEBUGCTL.LBR. (Information about the nature of the debug exception is saved in the exit qualification field.)
  — A page fault does not update CR2. (The linear address causing the page fault is saved in the exit-qualification field.)
  — An NMI causes subsequent NMIs to be blocked, but only after the VM exit completes.

  - An external interrupt does not acknowledge the interrupt controller and the interrupt remains pending, unless the "acknowledge interrupt on exit" VM-exit control is 1. In such a case, the interrupt controller is acknowledged and the interrupt is no longer pending.
  - The flags L0 – L3 in DR7 (bit 0, bit 2, bit 4, and bit 6) are not cleared when a task switch causes a VM exit.
  - If a task switch causes a VM exit, none of the following are modified by the task switch: old task-state segment (TSS); new TSS; old TSS descriptor; new TSS descriptor; RFLAGS.NT[1]; or the TR register.
  - No last-exception record is made if the event that would do so directly causes a VM exit.
  - If a machine-check exception causes a VM exit directly, this does not prevent machine-check MSRs from being updated. These are updated by the machine-check event itself and not the resulting machine-check exception.
  - If the logical processor is in an inactive state (see Section 24.4.2) and not executing instructions, some events may be blocked but others may return the logical processor to the active state. Unblocked events may cause VM exits.[2] If an unblocked event causes a VM exit directly, a return to the active state occurs only after the VM exit completes.[3] The VM exit generates any special bus cycle that is normally generated when the active state is entered from that activity state.

    MTF VM exits (see Section 25.5.2 and Section 26.6.8) are not blocked in the HLT activity state. If an MTF VM exit occurs in the HLT activity state, the logical processor returns to the active state only after the VM exit completes. MTF VM exits are blocked the shutdown state and the wait-for-SIPI state.

- If an event causes a VM exit indirectly, the event does update architectural state:
  - A debug exception updates DR6, DR7, and the IA32_DEBUGCTL MSR. No debug exceptions are considered pending.
  - A page fault updates CR2.
  - An NMI causes subsequent NMIs to be blocked before the VM exit commences.
  - An external interrupt acknowledges the interrupt controller and the interrupt is no longer pending.
  - If the logical processor had been in an inactive state, it enters the active state and, before the VM exit commences, generates any special bus cycle that is normally generated when the active state is entered from that activity state.
  - There is no blocking by STI or by MOV SS when the VM exit commences.
  - Processor state that is normally updated as part of delivery through the IDT (CS, RIP, SS, RSP, RFLAGS) is not modified. However, the incomplete delivery of the event may write to the stack.
  - The treatment of last-exception records is implementation dependent:
    - Some processors make a last-exception record when beginning the delivery of an event through the IDT (before it can encounter a nested exception). Such processors perform this update even if the event encounters a nested exception that causes a VM exit (including the case where nested exceptions lead to a triple fault).
    - Other processors delay making a last-exception record until event delivery has reached some event handler successfully (perhaps after one or more nested exceptions). Such processors do not update the last-exception record if a VM exit or triple fault occurs before an event handler is reached.
- If the "virtual NMIs" VM-execution control is 1, VM entry injects an NMI, and delivery of the NMI causes a nested exception, double fault, task switch, or APIC access that causes a VM exit, virtual-NMI blocking is in effect before the VM exit commences.

---

1. This chapter uses the notation RAX, RIP, RSP, RFLAGS, etc. for processor registers because most processors that support VMX operation also support Intel 64 architecture. For processors that do not support Intel 64 architecture, this notation refers to the 32-bit forms of those registers (EAX, EIP, ESP, EFLAGS, etc.). In a few places, notation such as EAX is used to refer specifically to lower 32 bits of the indicated register.
2. If a VM exit takes the processor from an inactive state resulting from execution of a specific instruction (HLT or MWAIT), the value saved for RIP by that VM exit will reference the following instruction.
3. An exception is made if the logical processor had been inactive due to execution of MWAIT; in this case, it is considered to have become active before the VM exit.

- If a VM exit results from a fault, EPT violation, EPT misconfiguration, or page-modification log-full event is encountered during execution of IRET and the "NMI exiting" VM-execution control is 0, any blocking by NMI is cleared before the VM exit commences. However, the previous state of blocking by NMI may be recorded in the exit qualification or in the VM-exit interruption-information field; see Section 27.2.1 and Section 27.2.2.
- If a VM exit results from a fault, EPT violation, EPT misconfiguration, or page-modification log-full event is encountered during execution of IRET and the "virtual NMIs" VM-execution control is 1, virtual-NMI blocking is cleared before the VM exit commences. However, the previous state of blocking by NMI may be recorded in the exit qualification or in the VM-exit interruption-information field; see Section 27.2.1 and Section 27.2.2.
- Suppose that a VM exit is caused directly by an x87 FPU Floating-Point Error (#MF) or by any of the following events if the event was unblocked due to (and given priority over) an x87 FPU Floating-Point Error: an INIT signal, an external interrupt, an NMI, an SMI; or a machine-check exception. In these cases, there is no blocking by STI or by MOV SS when the VM exit commences.
- Normally, a last-branch record may be made when an event is delivered through the IDT. However, if such an event results in a VM exit before delivery is complete, no last-branch record is made.
- If machine-check exception results in a VM exit, processor state is suspect and may result in suspect state being saved to the guest-state area. A VM monitor should consult the RIPV and EIPV bits in the IA32_MCG_STATUS MSR before resuming a guest that caused a VM exit resulting from a machine-check exception.
- If a VM exit results from a fault, APIC access (see Section 29.4), EPT violation, EPT misconfiguration, or page-modification log-full event is encountered while executing an instruction, data breakpoints due to that instruction may have been recognized and information about them may be saved in the pending debug exceptions field (see Section 27.3.4).
- The following VM exits are considered to happen after an instruction is executed:
  - VM exits resulting from debug traps (single-step, I/O breakpoints, and data breakpoints).
  - VM exits resulting from debug exceptions whose recognition was delayed by blocking by MOV SS.
  - VM exits resulting from some machine-check exceptions.
  - Trap-like VM exits due to execution of MOV to CR8 when the "CR8-load exiting" VM-execution control is 0 and the "use TPR shadow" VM-execution control is 1 (see Section 29.3). (Such VM exits can occur only from 64-bit mode and thus only on processors that support Intel 64 architecture.)
  - Trap-like VM exits due to execution of WRMSR when the "use MSR bitmaps" VM-execution control is 1; the value of ECX is in the range 800H–8FFH; and the bit corresponding to the ECX value in write bitmap for low MSRs is 0; and the "virtualize x2APIC mode" VM-execution control is 1. See Section 29.5.
  - VM exits caused by APIC-write emulation (see Section 29.4.3.2) that result from APIC accesses as part of instruction execution.

  For these VM exits, the instruction's modifications to architectural state complete before the VM exit occurs. Such modifications include those to the logical processor's interruptibility state (see Table 24-3). If there had been blocking by MOV SS, POP SS, or STI before the instruction executed, such blocking is no longer in effect.

A VM exit that occurs in enclave mode sets bit 27 of the exit-reason field and bit 4 of the guest interruptibility-state field. Before such a VM exit is delivered, an Asynchronous Enclave Exit (AEX) occurs (see Chapter 39, "Enclave Exiting Events"). An AEX modifies architectural state (Section 39.3). In particular, the processor establishes the following architectural state as indicated:

- The following bits in RFLAGS are cleared: CF, PF, AF, ZF, SF, OF, and RF.
- FS and GS are restored to the values they had prior to the most recent enclave entry.
- RIP is loaded with the AEP of interrupted enclave thread.
- RSP is loaded from the URSP field in the enclave's state-save area (SSA).

27.2 RECORDING VM-EXIT INFORMATION AND UPDATING VM-ENTRY CONTROL FIELDS

VM exits begin by recording information about the nature of and reason for the VM exit in the VM-exit information fields. Section 27.2.1 to Section 27.2.4 detail the use of these fields.

In addition to updating the VM-exit information fields, the valid bit (bit 31) is cleared in the VM-entry interruption-information field. If bit 5 of the IA32_VMX_MISC MSR (index 485H) is read as 1 (see Appendix A.6), the value of IA32_EFER.LMA is stored into the "IA-32e mode guest" VM-entry control.[1]

27.2.1 Basic VM-Exit Information

Section 24.9.1 defines the basic VM-exit information fields. The following items detail their use.

- **Exit reason.**
  - Bits 15:0 of this field contain the basic exit reason. It is loaded with a number indicating the general cause of the VM exit. Appendix C lists the numbers used and their meaning.
  - Bit 27 of this field is set to 1 if the VM exit occurred while the logical processor was in enclave mode.

    Such VM exits includes those caused by interrupts, non-maskable interrupts, system-management interrupts, INIT signals, and exceptions occurring in enclave mode as well as exceptions encountered during the delivery of such events incident to enclave mode.

    A VM exit also sets this bit if it is incident to delivery of an event injected by VM entry and the guest interruptibility-state field indicates an enclave interrupt (bit 4 of the field is 1).
  - The remainder of the field (bits 31:28 and bits 26:16) is cleared to 0 (certain SMM VM exits may set some of these bits; see Section 34.15.2.3).[2]
- **Exit qualification.** This field is saved for VM exits due to the following causes: debug exceptions; page-fault exceptions; start-up IPIs (SIPIs); system-management interrupts (SMIs) that arrive immediately after the retirement of I/O instructions; task switches; INVEPT; INVLPG; INVPCID; INVVPID; LGDT; LIDT; LLDT; LTR; SGDT; SIDT; SLDT; STR; VMCLEAR; VMPTRLD; VMPTRST; VMREAD; VMWRITE; VMXON; XRSTORS; XSAVES; control-register accesses; MOV DR; I/O instructions; MWAIT; accesses to the APIC-access page (see Section 29.4); EPT violations; EOI virtualization (see Section 29.1.4); APIC-write emulation (see Section 29.4.3.3); and page-modification log full (see Section 28.2.5). For all other VM exits, this field is cleared. The following items provide details:
  - For a debug exception, the exit qualification contains information about the debug exception. The information has the format given in Table 27-1.

Table 27-1. Exit Qualification for Debug Exceptions

Bit Position(s)	Contents
3:0	B3 – B0. When set, each of these bits indicates that the corresponding breakpoint condition was met. Any of these bits may be set even if its corresponding enabling bit in DR7 is not set.
12:4	Reserved (cleared to 0).
13	BD. When set, this bit indicates that the cause of the debug exception is "debug register access detected."
14	BS. When set, this bit indicates that the cause of the debug exception is either the execution of a single instruction (if RFLAGS.TF = 1 and IA32_DEBUGCTL.BTF = 0) or a taken branch (if RFLAGS.TF = DEBUGCTL.BTF = 1).
63:15	Reserved (cleared to 0). Bits 63:32 exist only on processors that support Intel 64 architecture.

1. Bit 5 of the IA32_VMX_MISC MSR is read as 1 on any logical processor that supports the 1-setting of the "unrestricted guest" VM-execution control.
2. Bit 31 of this field is set on certain VM-entry failures; see Section 26.7.

— For a page-fault exception, the exit qualification contains the linear address that caused the page fault. On processors that support Intel 64 architecture, bits 63:32 are cleared if the logical processor was not in 64-bit mode before the VM exit.

  If the page-fault exception occurred during execution of an instruction in enclave mode (and not during delivery of an event incident to enclave mode), bits 11:0 of the exit qualification are cleared.

— For a start-up IPI (SIPI), the exit qualification contains the SIPI vector information in bits 7:0. Bits 63:8 of the exit qualification are cleared to 0.

— For a task switch, the exit qualification contains details about the task switch, encoded as shown in Table 27-2.

— For INVLPG, the exit qualification contains the linear-address operand of the instruction.

  • On processors that support Intel 64 architecture, bits 63:32 are cleared if the logical processor was not in 64-bit mode before the VM exit.

  • If the INVLPG source operand specifies an unusable segment, the linear address specified in the exit qualification will match the linear address that the INVLPG would have used if no VM exit occurred. This address is not architecturally defined and may be implementation-specific.

Table 27-2. Exit Qualification for Task Switch

Bit Position(s)	Contents
15:0	Selector of task-state segment (TSS) to which the guest attempted to switch
29:16	Reserved (cleared to 0)
31:30	Source of task switch initiation: 0: CALL instruction 1: IRET instruction 2: JMP instruction 3: Task gate in IDT
63:32	Reserved (cleared to 0). These bits exist only on processors that support Intel 64 architecture.

— For INVEPT, INVPCID, INVVPID, LGDT, LIDT, LLDT, LTR, SGDT, SIDT, SLDT, STR, VMCLEAR, VMPTRLD, VMPTRST, VMREAD, VMWRITE, VMXON, XRSTORS, and XSAVES, the exit qualification receives the value of the instruction's displacement field, which is sign-extended to 64 bits if necessary (32 bits on processors that do not support Intel 64 architecture). If the instruction has no displacement (for example, has a register operand), zero is stored into the exit qualification.

  On processors that support Intel 64 architecture, an exception is made for RIP-relative addressing (used only in 64-bit mode). Such addressing causes an instruction to use an address that is the sum of the displacement field and the value of RIP that references the following instruction. In this case, the exit qualification is loaded with the sum of the displacement field and the appropriate RIP value.

  In all cases, bits of this field beyond the instruction's address size are undefined. For example, suppose that the address-size field in the VM-exit instruction-information field (see Section 24.9.4 and Section 27.2.4) reports an *n*-bit address size. Then bits 63:*n* (bits 31:*n* on processors that do not support Intel 64 architecture) of the instruction displacement are undefined.

— For a control-register access, the exit qualification contains information about the access and has the format given in Table 27-3.

— For MOV DR, the exit qualification contains information about the instruction and has the format given in Table 27-4.

— For an I/O instruction, the exit qualification contains information about the instruction and has the format given in Table 27-5.

— For MWAIT, the exit qualification contains a value that indicates whether address-range monitoring hardware was armed. The exit qualification is set either to 0 (if address-range monitoring hardware is not armed) or to 1 (if address-range monitoring hardware is armed).

— For an APIC-access VM exit resulting from a linear access or a guest-physical access to the APIC-access page (see Section 29.4), the exit qualification contains information about the access and has the format given in Table 27-6.[1]

If the access to the APIC-access page occurred during execution of an instruction in enclave mode (and not during delivery of an event incident to enclave mode), bits 11:0 of the exit qualification are cleared.

Such a VM exit that set bits 15:12 of the exit qualification to 0000b (data read during instruction execution) or 0001b (data write during instruction execution) set bit 12—which distinguishes data read from data write—to that which would have been stored in bit 1—W/R—of the page-fault error code had the access caused a page fault instead of an APIC-access VM exit. This implies the following:

- For an APIC-access VM exit caused by the CLFLUSH and CLFLUSHOPT instructions, the access type is "data read during instruction execution."
- For an APIC-access VM exit caused by the ENTER instruction, the access type is "data write during instruction execution."

Table 27-3. Exit Qualification for Control-Register Accesses

Bit Positions	Contents
3:0	Number of control register (0 for CLTS and LMSW). Bit 3 is always 0 on processors that do not support Intel 64 architecture as they do not support CR8.
5:4	Access type: 0 = MOV to CR 1 = MOV from CR 2 = CLTS 3 = LMSW
6	LMSW operand type: 0 = register 1 = memory For CLTS and MOV CR, cleared to 0
7	Reserved (cleared to 0)
11:8	For MOV CR, the general-purpose register: 0 = RAX 1 = RCX 2 = RDX 3 = RBX 4 = RSP 5 = RBP 6 = RSI 7 = RDI 8–15 represent R8–R15, respectively (used only on processors that support Intel 64 architecture) For CLTS and LMSW, cleared to 0
15:12	Reserved (cleared to 0)

1. The exit qualification is undefined if the access was part of the logging of a branch record or a processor-event-based-sampling (PEBS) record to the DS save area. It is recommended that software configure the paging structures so that no address in the DS save area translates to an address on the APIC-access page.

Table 27-3. Exit Qualification for Control-Register Accesses (Contd.)

Bit Positions	Contents
31:16	For LMSW, the LMSW source data For CLTS and MOV CR, cleared to 0
63:32	Reserved (cleared to 0). These bits exist only on processors that support Intel 64 architecture.

- For an APIC-access VM exit caused by the MASKMOVQ instruction or the MASKMOVDQU instruction, the access type is "data write during instruction execution."
- For an APIC-access VM exit caused by the MONITOR instruction, the access type is "data read during instruction execution."

Such a VM exit stores 1 for bit 31 for IDT-vectoring information field (see Section 27.2.3) if and only if it sets bits 15:12 of the exit qualification to 0011b (linear access during event delivery) or 1010b (guest-physical access during event delivery).

See Section 29.4.4 for further discussion of these instructions and APIC-access VM exits.

For APIC-access VM exits resulting from physical accesses to the APIC-access page (see Section 29.4.6), the exit qualification is undefined.

— For an EPT violation, the exit qualification contains information about the access causing the EPT violation and has the format given in Table 27-7.

As noted in that table, the format and meaning of the exit qualification depends on the setting of the "mode-based execute control for EPT" VM-execution control and whether the processor supports advanced VM-exit information for EPT violations.[1]

An EPT violation that occurs during as a result of execution of a read-modify-write operation sets bit 1 (data write). Whether it also sets bit 0 (data read) is implementation-specific and, for a given implementation, may differ for different kinds of read-modify-write operations.

Table 27-4. Exit Qualification for MOV DR

Bit Position(s)	Contents
2:0	Number of debug register
3	Reserved (cleared to 0)
4	Direction of access (0 = MOV to DR; 1 = MOV from DR)
7:5	Reserved (cleared to 0)
11:8	General-purpose register: 0 = RAX 1 = RCX 2 = RDX 3 = RBX 4 = RSP 5 = RBP 6 = RSI 7 = RDI 8 –15 = R8 – R15, respectively
63:12	Reserved (cleared to 0)

1. Software can determine whether advanced VM-exit information for EPT violations is supported by consulting the VMX capability MSR IA32_VMX_EPT_VPID_CAP (see Appendix A.10).

Table 27-5. Exit Qualification for I/O Instructions

Bit Position(s)	Contents
2:0	Size of access: 0 = 1-byte 1 = 2-byte 3 = 4-byte Other values not used
3	Direction of the attempted access (0 = OUT, 1 = IN)
4	String instruction (0 = not string; 1 = string)
5	REP prefixed (0 = not REP; 1 = REP)
6	Operand encoding (0 = DX, 1 = immediate)
15:7	Reserved (cleared to 0)
31:16	Port number (as specified in DX or in an immediate operand)
63:32	Reserved (cleared to 0). These bits exist only on processors that support Intel 64 architecture.

Bit 12 is undefined in any of the following cases:

- If the "NMI exiting" VM-execution control is 1 and the "virtual NMIs" VM-execution control is 0.
- If the VM exit sets the valid bit in the IDT-vectoring information field (see Section 27.2.3).

Otherwise, bit 12 is defined as follows:

- If the "virtual NMIs" VM-execution control is 0, the EPT violation was caused by a memory access as part of execution of the IRET instruction, and blocking by NMI (see Table 24-3) was in effect before execution of IRET, bit 12 is set to 1.

Table 27-6. Exit Qualification for APIC-Access VM Exits from Linear Accesses and Guest-Physical Accesses

Bit Position(s)	Contents
11:0	▪ If the APIC-access VM exit is due to a linear access, the offset of access within the APIC page. ▪ Undefined if the APIC-access VM exit is due a guest-physical access
15:12	Access type: 0 = linear access for a data read during instruction execution 1 = linear access for a data write during instruction execution 2 = linear access for an instruction fetch 3 = linear access (read or write) during event delivery 10 = guest-physical access during event delivery 15 = guest-physical access for an instruction fetch or during instruction execution Other values not used
63:16	Reserved (cleared to 0). Bits 63:32 exist only on processors that support Intel 64 architecture.

  - If the "virtual NMIs" VM-execution control is 1,the EPT violation was caused by a memory access as part of execution of the IRET instruction, and virtual-NMI blocking was in effect before execution of IRET, bit 12 is set to 1.
  - For all other relevant VM exits, bit 12 is cleared to 0.

— For VM exits caused as part of EOI virtualization (Section 29.1.4), bits 7:0 of the exit qualification are set to vector of the virtual interrupt that was dismissed by the EOI virtualization. Bits above bit 7 are cleared.

— For APIC-write VM exits (Section 29.4.3.3), bits 11:0 of the exit qualification are set to the page offset of the write access that caused the VM exit.[1] Bits above bit 11 are cleared.

— For a VM exit due to a page-modification log-full event (Section 28.2.5), only bit 12 of the exit qualification is defined, and only in some cases. It is undefined in the following cases:

  - If the "NMI exiting" VM-execution control is 1 and the "virtual NMIs" VM-execution control is 0.
  - If the VM exit sets the valid bit in the IDT-vectoring information field (see Section 27.2.3).

  Otherwise, it is defined as follows:

  - If the "virtual NMIs" VM-execution control is 0, the page-modification log-full event was caused by a memory access as part of execution of the IRET instruction, and blocking by NMI (see Table 24-3) was in effect before execution of IRET, bit 12 is set to 1.
  - If the "virtual NMIs" VM-execution control is 1,the page-modification log-full event was caused by a memory access as part of execution of the IRET instruction, and virtual-NMI blocking was in effect before execution of IRET, bit 12 is set to 1.
  - For all other relevant VM exits, bit 12 is cleared to 0.

  For these VM exits, all bits other than bit 12 are undefined.

- **Guest-linear address.** For some VM exits, this field receives a linear address that pertains to the VM exit. The field is set for different VM exits as follows:

  — VM exits due to attempts to execute LMSW with a memory operand. In these cases, this field receives the linear address of that operand. Bits 63:32 are cleared if the logical processor was not in 64-bit mode before the VM exit.

  — VM exits due to attempts to execute INS or OUTS for which the relevant segment is usable (if the relevant segment is not usable, the value is undefined). (ES is always the relevant segment for INS; for OUTS, the relevant segment is DS unless overridden by an instruction prefix.) The linear address is the base address of relevant segment plus (E)DI (for INS) or (E)SI (for OUTS). Bits 63:32 are cleared if the logical processor was not in 64-bit mode before the VM exit.

Table 27-7. Exit Qualification for EPT Violations

Bit Position(s)	Contents
0	Set if the access causing the EPT violation was a data read.[1]
1	Set if the access causing the EPT violation was a data write.[1]
2	Set if the access causing the EPT violation was an instruction fetch.
3	The logical-AND of bit 0 in the EPT paging-structure entries used to translate the guest-physical address of the access causing the EPT violation (indicates whether the guest-physical address was readable).[2]
4	The logical-AND of bit 1 in the EPT paging-structure entries used to translate the guest-physical address of the access causing the EPT violation (indicates whether the guest-physical address was writeable).

1. Execution of WRMSR with ECX = 83FH (self-IPI MSR) can lead to an APIC-write VM exit; the exit qualification for such an APIC-write VM exit is 3F0H.

Table 27-7. Exit Qualification for EPT Violations (Contd.)

Bit Position(s)	Contents
5	The logical-AND of bit 2 in the EPT paging-structure entries used to translate the guest-physical address of the access causing the EPT violation. If the "mode-based execute control for EPT" VM-execution control is 0, this indicates whether the guest-physical address was executable. If that control is 1, this indicates whether the guest-physical address was executable for supervisor-mode linear addresses.
6	If the "mode-based execute control" VM-execution control is 0, the value of this bit is undefined. If that control is 1, this bit is the logical-AND of bit 10 in the EPT paging-structures entries used to translate the guest-physical address of the access causing the EPT violation. In this case, it indicates whether the guest-physical address was executable for user-mode linear addresses.
7	Set if the guest linear-address field is valid. The guest linear-address field is valid for all EPT violations except those resulting from an attempt to load the guest PDPTEs as part of the execution of the MOV CR instruction.
8	If bit 7 is 1: ▪ Set if the access causing the EPT violation is to a guest-physical address that is the translation of a linear address. ▪ Clear if the access causing the EPT violation is to a paging-structure entry as part of a page walk or the update of an accessed or dirty bit. Reserved if bit 7 is 0 (cleared to 0).
9	If bit 7 is 1, bit 8 is 1, and the processor supports advanced VM-exit information for EPT violations,[3] this bit is 0 if the linear address is a supervisor-mode linear address and 1 if it is a user-mode linear address. (If CR0.PG = 0, the translation of every linear address is a user-mode linear address and thus this bit will be 1.) Otherwise, this bit is undefined.
10	If bit 7 is 1, bit 8 is 1, and the processor supports advanced VM-exit information for EPT violations,[3] this bit is 0 if paging translates the linear address to a read-only page and 1 if it translates to a read/write page. (If CR0.PG = 0, every linear address is read/write and thus this bit will be 1.) Otherwise, this bit is undefined.
11	If bit 7 is 1, bit 8 is 1, and the processor supports advanced VM-exit information for EPT violations,[3] this bit is 0 if paging translates the linear address to an executable page and 1 if it translates to an execute-disable page. (If CR0.PG = 0, CR4.PAE = 0, or IA32_EFER.NXE = 0, every linear address is executable and thus this bit will be 0.) Otherwise, this bit is undefined.
12	NMI unblocking due to IRET
63:13	Reserved (cleared to 0).

NOTES:

1. If accessed and dirty flags for EPT are enabled, processor accesses to guest paging-structure entries are treated as writes with regard to EPT violations (see Section 28.2.3.2). If such an access causes an EPT violation, the processor sets both bit 0 and bit 1 of the exit qualification.
2. Bits 5:3 are cleared to 0 if any of EPT paging-structure entries used to translate the guest-physical address of the access causing the EPT violation is not present (see Section 28.2.2).
3. Software can determine whether advanced VM-exit information for EPT violations is supported by consulting the VMX capability MSR IA32_VMX_EPT_VPID_CAP (see Appendix A.10).

— VM exits due to EPT violations that set bit 7 of the exit qualification (see Table 27-7; these are all EPT violations except those resulting from an attempt to load the PDPTEs as of execution of the MOV CR instruction). The linear address may translate to the guest-physical address whose access caused the EPT violation. Alternatively, translation of the linear address may reference a paging-structure entry whose access caused the EPT violation. Bits 63:32 are cleared if the logical processor was not in 64-bit mode before the VM exit.

If the EPT violation occurred during execution of an instruction in enclave mode (and not during delivery of an event incident to enclave mode), bits 11:0 of this field are cleared.

— For all other VM exits, the field is undefined.

- **Guest-physical address.** For a VM exit due to an EPT violation or an EPT misconfiguration, this field receives the guest-physical address that caused the EPT violation or EPT misconfiguration. For all other VM exits, the field is undefined.

  If the EPT violation or EPT misconfiguration occurred during execution of an instruction in enclave mode (and not during delivery of an event incident to enclave mode), bits 11:0 of this field are cleared.

27.2.2 Information for VM Exits Due to Vectored Events

Section 24.9.2 defines fields containing information for VM exits due to the following events: exceptions (including those generated by the instructions INT3, INTO, BOUND, and UD); external interrupts that occur while the "acknowledge interrupt on exit" VM-exit control is 1; and non-maskable interrupts (NMIs). Such VM exits include those that occur on an attempt at a task switch that causes an exception before generating the VM exit due to the task switch that causes the VM exit.

The following items detail the use of these fields:

- **VM-exit interruption information** (format given in Table 24-15). The following items detail how this field is established for VM exits due to these events:

  — For an exception, bits 7:0 receive the exception vector (at most 31). For an NMI, bits 7:0 are set to 2. For an external interrupt, bits 7:0 receive the vector.

  — Bits 10:8 are set to 0 (external interrupt), 2 (non-maskable interrupt), 3 (hardware exception), or 6 (software exception). Hardware exceptions comprise all exceptions except breakpoint exceptions (#BP; generated by INT3) and overflow exceptions (#OF; generated by INTO); these are software exceptions. (A #BP that occurs in enclave mode is considered a hardware exception.) BOUND-range exceeded exceptions (#BR; generated by BOUND) and invalid opcode exceptions (#UD) generated by UD are hardware exceptions.

  — Bit 11 is set to 1 if the VM exit is caused by a hardware exception that would have delivered an error code on the stack. This bit is always 0 if the VM exit occurred while the logical processor was in real-address mode (CR0.PE=0).[1] If bit 11 is set to 1, the error code is placed in the VM-exit interruption error code (see below).

  — Bit 12 is undefined in any of the following cases:

    - If the "NMI exiting" VM-execution control is 1 and the "virtual NMIs" VM-execution control is 0.
    - If the VM exit sets the valid bit in the IDT-vectoring information field (see Section 27.2.3).
    - If the VM exit is due to a double fault (the interruption type is hardware exception and the vector is 8).

    Otherwise, bit 12 is defined as follows:

    - If the "virtual NMIs" VM-execution control is 0, the VM exit is due to a fault on the IRET instruction (other than a debug exception for an instruction breakpoint), and blocking by NMI (see Table 24-3) was in effect before execution of IRET, bit 12 is set to 1.
    - If the "virtual NMIs" VM-execution control is 1, the VM exit is due to a fault on the IRET instruction (other than a debug exception for an instruction breakpoint), and virtual-NMI blocking was in effect before execution of IRET, bit 12 is set to 1.
    - For all other relevant VM exits, bit 12 is cleared to 0.[2]

  — Bits 30:13 are always set to 0.

1. If the capability MSR IA32_VMX_CR0_FIXED0 reports that CR0.PE must be 1 in VMX operation, a logical processor cannot be in real-address mode unless the "unrestricted guest" VM-execution control and bit 31 of the primary processor-based VM-execution controls are both 1.
2. The conditions imply that, if the "NMI exiting" VM-execution control is 0 or the "virtual NMIs" VM-execution control is 1, bit 12 is always cleared to 0 by VM exits due to debug exceptions.

— Bit 31 is always set to 1.

For other VM exits (including those due to external interrupts when the "acknowledge interrupt on exit" VM-exit control is 0), the field is marked invalid (by clearing bit 31) and the remainder of the field is undefined.

- VM-exit interruption error code.
  - — For VM exits that set both bit 31 (valid) and bit 11 (error code valid) in the VM-exit interruption-information field, this field receives the error code that would have been pushed on the stack had the event causing the VM exit been delivered normally through the IDT. The EXT bit is set in this field exactly when it would be set normally. For exceptions that occur during the delivery of double fault (if the IDT-vectoring information field indicates a double fault), the EXT bit is set to 1, assuming that (1) that the exception would produce an error code normally (if not incident to double-fault delivery) and (2) that the error code uses the EXT bit (not for page faults, which use a different format).
  - — For other VM exits, the value of this field is undefined.

27.2.3 Information for VM Exits During Event Delivery

Section 24.9.3 defined fields containing information for VM exits that occur while delivering an event through the IDT and as a result of any of the following cases:[1]

- A fault occurs during event delivery and causes a VM exit (because the bit associated with the fault is set to 1 in the exception bitmap).
- A task switch is invoked through a task gate in the IDT. The VM exit occurs due to the task switch only after the initial checks of the task switch pass (see Section 25.4.2).
- Event delivery causes an APIC-access VM exit (see Section 29.4).
- An EPT violation, EPT misconfiguration, or page-modification log-full event that occurs during event delivery.

These fields are used for VM exits that occur during delivery of events injected as part of VM entry (see Section 26.5.1.2).

A VM exit is not considered to occur during event delivery in any of the following circumstances:

- The original event causes the VM exit directly (for example, because the original event is a non-maskable interrupt (NMI) and the "NMI exiting" VM-execution control is 1).
- The original event results in a double-fault exception that causes the VM exit directly.
- The VM exit occurred as a result of fetching the first instruction of the handler invoked by the event delivery.
- The VM exit is caused by a triple fault.

The following items detail the use of these fields:

- IDT-vectoring information (format given in Table 24-16). The following items detail how this field is established for VM exits that occur during event delivery:
  - — If the VM exit occurred during delivery of an exception, bits 7:0 receive the exception vector (at most 31). If the VM exit occurred during delivery of an NMI, bits 7:0 are set to 2. If the VM exit occurred during delivery of an external interrupt, bits 7:0 receive the vector.
  - — Bits 10:8 are set to indicate the type of event that was being delivered when the VM exit occurred: 0 (external interrupt), 2 (non-maskable interrupt), 3 (hardware exception), 4 (software interrupt), 5 (privileged software interrupt), or 6 (software exception).

    Hardware exceptions comprise all exceptions except breakpoint exceptions (#BP; generated by INT3) and overflow exceptions (#OF; generated by INTO); these are software exceptions. (A #BP that occurs in enclave mode is considered a hardware exception.) BOUND-range exceeded exceptions (#BR; generated by BOUND) and invalid opcode exceptions (#UD) generated by UD are hardware exceptions.

    Bits 10:8 may indicate privileged software interrupt if such an event was injected as part of VM entry.

1. This includes the case in which a VM exit occurs while delivering a software interrupt (INT *n*) through the 16-bit IVT (interrupt vector table) that is used in virtual-8086 mode with virtual-machine extensions (if RFLAGS.VM = CR4.VME = 1).

— Bit 11 is set to 1 if the VM exit occurred during delivery of a hardware exception that would have delivered an error code on the stack. This bit is always 0 if the VM exit occurred while the logical processor was in real-address mode (CR0.PE=0).[1] If bit 11 is set to 1, the error code is placed in the IDT-vectoring error code (see below).

— Bit 12 is undefined.

— Bits 30:13 are always set to 0.

— Bit 31 is always set to 1.

For other VM exits, the field is marked invalid (by clearing bit 31) and the remainder of the field is undefined.

- IDT-vectoring error code.
  — For VM exits that set both bit 31 (valid) and bit 11 (error code valid) in the IDT-vectoring information field, this field receives the error code that would have been pushed on the stack by the event that was being delivered through the IDT at the time of the VM exit. The EXT bit is set in this field when it would be set normally.
  — For other VM exits, the value of this field is undefined.

27.2.4 Information for VM Exits Due to Instruction Execution

Section 24.9.4 defined fields containing information for VM exits that occur due to instruction execution. (The VM-exit instruction length is also used for VM exits that occur during the delivery of a software interrupt or software exception.) The following items detail their use.

- **VM-exit instruction length.** This field is used in the following cases:
  — For fault-like VM exits due to attempts to execute one of the following instructions that cause VM exits unconditionally (see Section 25.1.2) or based on the settings of VM-execution controls (see Section 25.1.3): CLTS, CPUID, ENCLS, GETSEC, HLT, IN, INS, INVD, INVEPT, INVLPG, INVPCID, INVVPID, LGDT, LIDT, LLDT, LMSW, LTR, MONITOR, MOV CR, MOV DR, MWAIT, OUT, OUTS, PAUSE, RDMSR, RDPMC, RDRAND, RDSEED, RDTSC, RDTSCP, RSM, SGDT, SIDT, SLDT, STR, VMCALL, VMCLEAR, VMLAUNCH, VMPTRLD, VMPTRST, VMREAD, VMRESUME, VMWRITE, VMXOFF, VMXON, WBINVD, WRMSR, XRSTORS, XSETBV, and XSAVES.[2]
  — For VM exits due to software exceptions (those generated by executions of INT3 or INTO).
  — For VM exits due to faults encountered during delivery of a software interrupt, privileged software exception, or software exception.
  — For VM exits due to attempts to effect a task switch via instruction execution. These are VM exits that produce an exit reason indicating task switch and either of the following:
    - An exit qualification indicating execution of CALL, IRET, or JMP instruction.
    - An exit qualification indicating a task gate in the IDT and an IDT-vectoring information field indicating that the task gate was encountered during delivery of a software interrupt, privileged software exception, or software exception.
  — For APIC-access VM exits and for VM exits caused by EPT violations and page-modification log-full events encountered during delivery of a software interrupt, privileged software exception, or software exception.[3]
  — For VM exits due executions of VMFUNC that fail because one of the following is true:

1. If the capability MSR IA32_VMX_CR0_FIXED0 reports that CR0.PE must be 1 in VMX operation, a logical processor cannot be in real-address mode unless the "unrestricted guest" VM-execution control and bit 31 of the primary processor-based VM-execution controls are both 1.
2. This item applies only to fault-like VM exits. It does not apply to trap-like VM exits following executions of the MOV to CR8 instruction when the "use TPR shadow" VM-execution control is 1 or to those following executions of the WRMSR instruction when the "virtualize x2APIC mode" VM-execution control is 1.
3. The VM-exit instruction-length field is not defined following APIC-access VM exits resulting from physical accesses (see Section 29.4.6) even if encountered during delivery of a software interrupt, privileged software exception, or software exception.

- EAX indicates a VM function that is not enabled (the bit at position EAX is 0 in the VM-function controls; see Section 25.5.5.2).
- EAX = 0 and either ECX ≥ 512 or the value of ECX selects an invalid tentative EPTP value (see Section 25.5.5.3).

In all the above cases, this field receives the length in bytes (1–15) of the instruction (including any instruction prefixes) whose execution led to the VM exit (see the next paragraph for one exception).

The cases of VM exits encountered during delivery of a software interrupt, privileged software exception, or software exception include those encountered during delivery of events injected as part of VM entry (see Section 26.5.1.2). If the original event was injected as part of VM entry, this field receives the value of the VM-entry instruction length.

All VM exits other than those listed in the above items leave this field undefined.

If the VM exit occurred in enclave mode, this field is cleared (none of the previous items apply).

Table 27-8. Format of the VM-Exit Instruction-Information Field as Used for INS and OUTS

Bit Position(s)	Content
6:0	Undefined.
9:7	Address size: 0: 16-bit 1: 32-bit 2: 64-bit (used only on processors that support Intel 64 architecture) Other values not used.
14:10	Undefined.
17:15	Segment register: 0: ES 1: CS 2: SS 3: DS 4: FS 5: GS Other values not used. Undefined for VM exits due to execution of INS.
31:18	Undefined.

- **VM-exit instruction information.** For VM exits due to attempts to execute INS, INVEPT, INVPCID, INVVPID, LIDT, LGDT, LLDT, LTR, OUTS, RDRAND, RDSEED, SIDT, SGDT, SLDT, STR, VMCLEAR, VMPTRLD, VMPTRST, VMREAD, VMWRITE, VMXON, XRSTORS, or XSAVES, this field receives information about the instruction that caused the VM exit. The format of the field depends on the identity of the instruction causing the VM exit:
  - For VM exits due to attempts to execute INS or OUTS, the field has the format is given in Table 27-8.[1]
  - For VM exits due to attempts to execute INVEPT, INVPCID, or INVVPID, the field has the format is given in Table 27-9.
  - For VM exits due to attempts to execute LIDT, LGDT, SIDT, or SGDT, the field has the format is given in Table 27-10.
  - For VM exits due to attempts to execute LLDT, LTR, SLDT, or STR, the field has the format is given in Table 27-11.
  - For VM exits due to attempts to execute RDRAND or RDSEED, the field has the format is given in Table 27-12.

1. The format of the field was undefined for these VM exits on the first processors to support the virtual-machine extensions. Software can determine whether the format specified in Table 27-8 is used by consulting the VMX capability MSR IA32_VMX_BASIC (see Appendix A.1).

— For VM exits due to attempts to execute VMCLEAR, VMPTRLD, VMPTRST, VMXON, XRSTORS, or XSAVES, the field has the format is given in Table 27-13.

— For VM exits due to attempts to execute VMREAD or VMWRITE, the field has the format is given in Table 27-14.

For all other VM exits, the field is undefined, unless the VM exit occurred in enclave mode, in which case the field is cleared.

- **I/O RCX, I/O RSI, I/O RDI, I/O RIP**. These fields are undefined except for SMM VM exits due to system-management interrupts (SMIs) that arrive immediately after retirement of I/O instructions. See Section 34.15.2.3. Note that, if the VM exit occurred in enclave mode, these fields are all cleared.

Table 27-9. Format of the VM-Exit Instruction-Information Field as Used for INVEPT, INVPCID, and INVVPID

Bit Position(s)	Content
1:0	Scaling: 0: no scaling 1: scale by 2 2: scale by 4 3: scale by 8 (used only on processors that support Intel 64 architecture) Undefined for instructions with no index register (bit 22 is set).
6:2	Undefined.
9:7	Address size: 0: 16-bit 1: 32-bit 2: 64-bit (used only on processors that support Intel 64 architecture) Other values not used.
10	Cleared to 0.
14:11	Undefined.
17:15	Segment register: 0: ES 1: CS 2: SS 3: DS 4: FS 5: GS Other values not used.
21:18	IndexReg: 0 = RAX 1 = RCX 2 = RDX 3 = RBX 4 = RSP 5 = RBP 6 = RSI 7 = RDI 8–15 represent R8–R15, respectively (used only on processors that support Intel 64 architecture) Undefined for instructions with no index register (bit 22 is set).
22	IndexReg invalid (0 = valid; 1 = invalid)
26:23	BaseReg (encoded as IndexReg above) Undefined for memory instructions with no base register (bit 27 is set).
27	BaseReg invalid (0 = valid; 1 = invalid)
31:28	Reg2 (same encoding as IndexReg above)

Table 27-10. Format of the VM-Exit Instruction-Information Field as Used for LIDT, LGDT, SIDT, or SGDT

Bit Position(s)	Content
1:0	Scaling: 0: no scaling 1: scale by 2 2: scale by 4 3: scale by 8 (used only on processors that support Intel 64 architecture) Undefined for instructions with no index register (bit 22 is set).
6:2	Undefined.
9:7	Address size: 0: 16-bit 1: 32-bit 2: 64-bit (used only on processors that support Intel 64 architecture) Other values not used.
10	Cleared to 0.
11	Operand size: 0: 16-bit 1: 32-bit Undefined for VM exits from 64-bit mode.
14:12	Undefined.
17:15	Segment register: 0: ES 1: CS 2: SS 3: DS 4: FS 5: GS Other values not used.
21:18	IndexReg: 0 = RAX 1 = RCX 2 = RDX 3 = RBX 4 = RSP 5 = RBP 6 = RSI 7 = RDI 8–15 represent R8–R15, respectively (used only on processors that support Intel 64 architecture) Undefined for instructions with no index register (bit 22 is set).
22	IndexReg invalid (0 = valid; 1 = invalid)
26:23	BaseReg (encoded as IndexReg above) Undefined for instructions with no base register (bit 27 is set).
27	BaseReg invalid (0 = valid; 1 = invalid)
29:28	Instruction identity: 0: SGDT 1: SIDT 2: LGDT 3: LIDT

Table 27-10. Format of the VM-Exit Instruction-Information Field as Used for LIDT, LGDT, SIDT, or SGDT (Contd.)

Bit Position(s)	Content
31:30	Undefined.

Table 27-11. Format of the VM-Exit Instruction-Information Field as Used for LLDT, LTR, SLDT, and STR

Bit Position(s)	Content
1:0	Scaling: 0: no scaling 1: scale by 2 2: scale by 4 3: scale by 8 (used only on processors that support Intel 64 architecture) Undefined for register instructions (bit 10 is set) and for memory instructions with no index register (bit 10 is clear and bit 22 is set).
2	Undefined.
6:3	Reg1: 0 = RAX 1 = RCX 2 = RDX 3 = RBX 4 = RSP 5 = RBP 6 = RSI 7 = RDI 8–15 represent R8–R15, respectively (used only on processors that support Intel 64 architecture) Undefined for memory instructions (bit 10 is clear).
9:7	Address size: 0: 16-bit 1: 32-bit 2: 64-bit (used only on processors that support Intel 64 architecture) Other values not used. Undefined for register instructions (bit 10 is set).
10	Mem/Reg (0 = memory; 1 = register).
14:11	Undefined.
17:15	Segment register: 0: ES 1: CS 2: SS 3: DS 4: FS 5: GS Other values not used. Undefined for register instructions (bit 10 is set).
21:18	IndexReg (encoded as Reg1 above) Undefined for register instructions (bit 10 is set) and for memory instructions with no index register (bit 10 is clear and bit 22 is set).
22	IndexReg invalid (0 = valid; 1 = invalid) Undefined for register instructions (bit 10 is set).
26:23	BaseReg (encoded as Reg1 above) Undefined for register instructions (bit 10 is set) and for memory instructions with no base register (bit 10 is clear and bit 27 is set).

Table 27-11. Format of the VM-Exit Instruction-Information Field as Used for LLDT, LTR, SLDT, and STR (Contd.)

Bit Position(s)	Content
27	BaseReg invalid (0 = valid; 1 = invalid) Undefined for register instructions (bit 10 is set).
29:28	Instruction identity: 0: SLDT 1: STR 2: LLDT 3: LTR
31:30	Undefined.

Table 27-12. Format of the VM-Exit Instruction-Information Field as Used for RDRAND and RDSEED

Bit Position(s)	Content
2:0	Undefined.
6:3	Destination register: 0 = RAX 1 = RCX 2 = RDX 3 = RBX 4 = RSP 5 = RBP 6 = RSI 7 = RDI 8–15 represent R8–R15, respectively (used only on processors that support Intel 64 architecture)
10:7	Undefined.
12:11	Operand size: 0: 16-bit 1: 32-bit 2: 64-bit The value 3 is not used.
31:13	Undefined.

Table 27-13. Format of the VM-Exit Instruction-Information Field as Used for VMCLEAR, VMPTRLD, VMPTRST, VMXON, XRSTORS, and XSAVES

Bit Position(s)	Content
1:0	Scaling: 0: no scaling 1: scale by 2 2: scale by 4 3: scale by 8 (used only on processors that support Intel 64 architecture) Undefined for instructions with no index register (bit 22 is set).
6:2	Undefined.
9:7	Address size: 0: 16-bit 1: 32-bit 2: 64-bit (used only on processors that support Intel 64 architecture) Other values not used.

Table 27-13. Format of the VM-Exit Instruction-Information Field as Used for VMCLEAR, VMPTRLD, VMPTRST, VMXON, XRSTORS, and XSAVES (Contd.)

Bit Position(s)	Content
10	Cleared to 0.
14:11	Undefined.
17:15	Segment register: 0: ES 1: CS 2: SS 3: DS 4: FS 5: GS Other values not used.
21:18	IndexReg: 0 = RAX 1 = RCX 2 = RDX 3 = RBX 4 = RSP 5 = RBP 6 = RSI 7 = RDI 8–15 represent R8–R15, respectively (used only on processors that support Intel 64 architecture) Undefined for instructions with no index register (bit 22 is set).
22	IndexReg invalid (0 = valid; 1 = invalid)
26:23	BaseReg (encoded as IndexReg above) Undefined for instructions with no base register (bit 27 is set).
27	BaseReg invalid (0 = valid; 1 = invalid)
31:28	Undefined.

Table 27-14. Format of the VM-Exit Instruction-Information Field as Used for VMREAD and VMWRITE

Bit Position(s)	Content
1:0	Scaling: 0: no scaling 1: scale by 2 2: scale by 4 3: scale by 8 (used only on processors that support Intel 64 architecture) Undefined for register instructions (bit 10 is set) and for memory instructions with no index register (bit 10 is clear and bit 22 is set).
2	Undefined.

Table 27-14. Format of the VM-Exit Instruction-Information Field as Used for VMREAD and VMWRITE (Contd.)

Bit Position(s)	Content
6:3	Reg1: 0 = RAX 1 = RCX 2 = RDX 3 = RBX 4 = RSP 5 = RBP 6 = RSI 7 = RDI 8–15 represent R8–R15, respectively (used only on processors that support Intel 64 architecture) Undefined for memory instructions (bit 10 is clear).
9:7	Address size: 0: 16-bit 1: 32-bit 2: 64-bit (used only on processors that support Intel 64 architecture) Other values not used. Undefined for register instructions (bit 10 is set).
10	Mem/Reg (0 = memory; 1 = register).
14:11	Undefined.
17:15	Segment register: 0: ES 1: CS 2: SS 3: DS 4: FS 5: GS Other values not used. Undefined for register instructions (bit 10 is set).
21:18	IndexReg (encoded as Reg1 above) Undefined for register instructions (bit 10 is set) and for memory instructions with no index register (bit 10 is clear and bit 22 is set).
22	IndexReg invalid (0 = valid; 1 = invalid) Undefined for register instructions (bit 10 is set).
26:23	BaseReg (encoded as Reg1 above) Undefined for register instructions (bit 10 is set) and for memory instructions with no base register (bit 10 is clear and bit 27 is set).
27	BaseReg invalid (0 = valid; 1 = invalid) Undefined for register instructions (bit 10 is set).
31:28	Reg2 (same encoding as Reg1 above)

27.3 SAVING GUEST STATE

Each field in the guest-state area of the VMCS (see Section 24.4) is written with the corresponding component of processor state. On processors that support Intel 64 architecture, the full values of each natural-width field (see Section 24.11.2) is saved regardless of the mode of the logical processor before and after the VM exit.

In general, the state saved is that which was in the logical processor at the time the VM exit commences. See Section 27.1 for a discussion of which architectural updates occur at that time.

Section 27.3.1 through Section 27.3.4 provide details for how certain components of processor state are saved. These sections reference VMCS fields that correspond to processor state. Unless otherwise stated, these references are to fields in the guest-state area.

27.3.1 Saving Control Registers, Debug Registers, and MSRs

Contents of certain control registers, debug registers, and MSRs is saved as follows:

- The contents of CR0, CR3, CR4, and the IA32_SYSENTER_CS, IA32_SYSENTER_ESP, and IA32_SYSENTER_EIP MSRs are saved into the corresponding fields. Bits 63:32 of the IA32_SYSENTER_CS MSR are not saved. On processors that do not support Intel 64 architecture, bits 63:32 of the IA32_SYSENTER_ESP and IA32_SYSENTER_EIP MSRs are not saved.
- If the "save debug controls" VM-exit control is 1, the contents of DR7 and the IA32_DEBUGCTL MSR are saved into the corresponding fields. The first processors to support the virtual-machine extensions supported only the 1-setting of this control and thus always saved data into these fields.
- If the "save IA32_PAT" VM-exit control is 1, the contents of the IA32_PAT MSR are saved into the corresponding field.
- If the "save IA32_EFER" VM-exit control is 1, the contents of the IA32_EFER MSR are saved into the corresponding field.
- If the processor supports either the 1-setting of the "load IA32_BNDCFGS" VM-entry control or that of the "clear IA32_BNDCFGS" VM-exit control, the contents of the IA32_BNDCFGS MSR are saved into the corresponding field.
- The value of the SMBASE field is undefined after all VM exits except SMM VM exits. See Section 34.15.2.

27.3.2 Saving Segment Registers and Descriptor-Table Registers

For each segment register (CS, SS, DS, ES, FS, GS, LDTR, or TR), the values saved for the base-address, segment-limit, and access rights are based on whether the register was unusable (see Section 24.4.1) before the VM exit:

- If the register was unusable, the values saved into the following fields are undefined: (1) base address; (2) segment limit; and (3) bits 7:0 and bits 15:12 in the access-rights field. The following exceptions apply:
  - CS.
    - The base-address and segment-limit fields are saved.
    - The L, D, and G bits are saved in the access-rights field.
  - SS.
    - DPL is saved in the access-rights field.
    - On processors that support Intel 64 architecture, bits 63:32 of the value saved for the base address are always zero.
  - DS and ES. On processors that support Intel 64 architecture, bits 63:32 of the values saved for the base addresses are always zero.
  - FS and GS. The base-address field is saved.
  - LDTR. The value saved for the base address is always canonical.
- If the register was not unusable, the values saved into the following fields are those which were in the register before the VM exit: (1) base address; (2) segment limit; and (3) bits 7:0 and bits 15:12 in access rights.
- Bits 31:17 and 11:8 in the access-rights field are always cleared. Bit 16 is set to 1 if and only if the segment is unusable.

The contents of the GDTR and IDTR registers are saved into the corresponding base-address and limit fields.

27.3.3 Saving RIP, RSP, and RFLAGS

The contents of the RIP, RSP, and RFLAGS registers are saved as follows:

- The value saved in the RIP field is determined by the nature and cause of the VM exit:
  - If the VM exit occurred in enclave mode, the value saved is the AEP of interrupted enclave thread (the remaining items do not apply).

  - If the VM exit occurs due to by an attempt to execute an instruction that causes VM exits unconditionally or that has been configured to cause a VM exit via the VM-execution controls, the value saved references that instruction.
  - If the VM exit is caused by an occurrence of an INIT signal, a start-up IPI (SIPI), or system-management interrupt (SMI), the value saved is that which was in RIP before the event occurred.
  - If the VM exit occurs due to the 1-setting of either the "interrupt-window exiting" VM-execution control or the "NMI-window exiting" VM-execution control, the value saved is that which would be in the register had the VM exit not occurred.
  - If the VM exit is due to an external interrupt, non-maskable interrupt (NMI), or hardware exception (as defined in Section 27.2.2), the value saved is the return pointer that would have been saved (either on the stack had the event been delivered through a trap or interrupt gate,[1] or into the old task-state segment had the event been delivered through a task gate).
  - If the VM exit is due to a triple fault, the value saved is the return pointer that would have been saved (either on the stack had the event been delivered through a trap or interrupt gate, or into the old task-state segment had the event been delivered through a task gate) had delivery of the double fault not encountered the nested exception that caused the triple fault.
  - If the VM exit is due to a software exception (due to an execution of INT3 or INTO), the value saved references the INT3 or INTO instruction that caused that exception.
  - Suppose that the VM exit is due to a task switch that was caused by execution of CALL, IRET, or JMP or by execution of a software interrupt (INT *n*) or software exception (due to execution of INT3 or INTO) that encountered a task gate in the IDT. The value saved references the instruction that caused the task switch (CALL, IRET, JMP, INT *n*, INT3, or INTO).
  - Suppose that the VM exit is due to a task switch that was caused by a task gate in the IDT that was encountered for any reason except the direct access by a software interrupt or software exception. The value saved is that which would have been saved in the old task-state segment had the task switch completed normally.
  - If the VM exit is due to an execution of MOV to CR8 or WRMSR that reduced the value of bits 7:4 of VTPR (see Section 29.1.1) below that of TPR threshold VM-execution control field (see Section 29.1.2), the value saved references the instruction following the MOV to CR8 or WRMSR.
  - If the VM exit was caused by APIC-write emulation (see Section 29.4.3.2) that results from an APIC access as part of instruction execution, the value saved references the instruction following the one whose execution caused the APIC-write emulation.
- The contents of the RSP register are saved into the RSP field.
- With the exception of the resume flag (RF; bit 16), the contents of the RFLAGS register is saved into the RFLAGS field. RFLAGS.RF is saved as follows:
  - If the VM exit occurred in enclave mode, the value saved is 0 (the remaining items do not apply).
  - If the VM exit is caused directly by an event that would normally be delivered through the IDT, the value saved is that which would appear in the saved RFLAGS image (either that which would be saved on the stack had the event been delivered through a trap or interrupt gate[2] or into the old task-state segment had the event been delivered through a task gate) had the event been delivered through the IDT. See below for VM exits due to task switches caused by task gates in the IDT.
  - If the VM exit is caused by a triple fault, the value saved is that which the logical processor would have in RF in the RFLAGS register had the triple fault taken the logical processor to the shutdown state.
  - If the VM exit is caused by a task switch (including one caused by a task gate in the IDT), the value saved is that which would have been saved in the RFLAGS image in the old task-state segment (TSS) had the task switch completed normally without exception.

1. The reference here is to the full value of RIP before any truncation that would occur had the stack width been only 32 bits or 16 bits.
2. The reference here is to the full value of RFLAGS before any truncation that would occur had the stack width been only 32 bits or 16 bits.

— If the VM exit is caused by an attempt to execute an instruction that unconditionally causes VM exits or one that was configured to do with a VM-execution control, the value saved is 0.[1]

— For APIC-access VM exits and for VM exits caused by EPT violations, EPT misconfigurations, and page-modification log-full events, the value saved depends on whether the VM exit occurred during delivery of an event through the IDT:

  • If the VM exit stored 0 for bit 31 for IDT-vectoring information field (because the VM exit did not occur during delivery of an event through the IDT; see Section 27.2.3), the value saved is 1.

  • If the VM exit stored 1 for bit 31 for IDT-vectoring information field (because the VM exit did occur during delivery of an event through the IDT), the value saved is the value that would have appeared in the saved RFLAGS image had the event been delivered through the IDT (see above).

— For all other VM exits, the value saved is the value RFLAGS.RF had before the VM exit occurred.

27.3.4 Saving Non-Register State

Information corresponding to guest non-register state is saved as follows:

- The activity-state field is saved with the logical processor's activity state before the VM exit.[2] See Section 27.1 for details of how events leading to a VM exit may affect the activity state.
- The interruptibility-state field is saved to reflect the logical processor's interruptibility before the VM exit.
  — See Section 27.1 for details of how events leading to a VM exit may affect this state.
  — VM exits that end outside system-management mode (SMM) save bit 2 (blocking by SMI) as 0 regardless of the state of such blocking before the VM exit.
  — Bit 3 (blocking by NMI) is treated specially if the "virtual NMIs" VM-execution control is 1. In this case, the value saved for this field does not indicate the blocking of NMIs but rather the state of virtual-NMI blocking.
  — Bit 4 (enclave interruption) is set to 1 if the VM exit occurred while the logical processor was in enclave mode.

    Such VM exits includes those caused by interrupts, non-maskable interrupts, system-management interrupts, INIT signals, and exceptions occurring in enclave mode as well as exceptions encountered during the delivery of such events incident to enclave mode.

    A VM exit that is incident to delivery of an event injected by VM entry leaves this bit unmodified.
- The pending debug exceptions field is saved as clear for all VM exits except the following:
  — A VM exit caused by an INIT signal, a machine-check exception, or a system-management interrupt (SMI).
  — A VM exit with basic exit reason "TPR below threshold",[3] "virtualized EOI", "APIC write", or "monitor trap flag."
  — VM exits that are not caused by debug exceptions and that occur while there is MOV-SS blocking of debug exceptions.

  For VM exits that do not clear the field, the value saved is determined as follows:
  — Each of bits 3:0 may be set if it corresponds to a matched breakpoint. This may be true even if the corresponding breakpoint is not enabled in DR7.
  — Suppose that a VM exit is due to an INIT signal, a machine-check exception, or an SMI; or that a VM exit has basic exit reason "TPR below threshold" or "monitor trap flag." In this case, the value saved sets bits corresponding to the causes of any debug exceptions that were pending at the time of the VM exit.

---

1. This is true even if RFLAGS.RF was 1 before the instruction was executed. If, in response to such a VM exit, a VM monitor re-enters the guest to re-execute the instruction that caused the VM exit (for example, after clearing the VM-execution control that caused the VM exit), the instruction may encounter a code breakpoint that has already been processed. A VM monitor can avoid this by setting the guest value of RFLAGS.RF to 1 before resuming guest software.
2. If this activity state was an inactive state resulting from execution of a specific instruction (HLT or MWAIT), the value saved for RIP by that VM exit will reference the following instruction.
3. This item includes VM exits that occur as a result of certain VM entries (Section 26.6.7).

If the VM exit occurs immediately after VM entry, the value saved may match that which was loaded on VM entry (see Section 26.6.3). Otherwise, the following items apply:

- Bit 12 (enabled breakpoint) is set to 1 in any of the following cases:
  - If there was at least one matched data or I/O breakpoint that was enabled in DR7.
  - If it had been set on VM entry, causing there to be valid pending debug exceptions (see Section 26.6.3) and the VM exit occurred before those exceptions were either delivered or lost.
  - If the XBEGIN instruction was executed immediately before the VM exit and advanced debugging of RTM transactional regions had been enabled (see Section 16.3.7, "RTM-Enabled Debugger Support," of *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*). (This does not apply to VM exits with basic exit reason "monitor trap flag.")

  In other cases, bit 12 is cleared to 0.
- Bit 14 (BS) is set if RFLAGS.TF = 1 in either of the following cases:
  - IA32_DEBUGCTL.BTF = 0 and the cause of a pending debug exception was the execution of a single instruction.
  - IA32_DEBUGCTL.BTF = 1 and the cause of a pending debug exception was a taken branch.
- Bit 16 (RTM) is set if a debug exception (#DB) or a breakpoint exception (#BP) occurred inside an RTM region while advanced debugging of RTM transactional regions had been enabled. (This does not apply to VM exits with basic exit reason "monitor trap flag.")

— Suppose that a VM exit is due to another reason (but not a debug exception) and occurs while there is MOV-SS blocking of debug exceptions. In this case, the value saved sets bits corresponding to the causes of any debug exceptions that were pending at the time of the VM exit. If the VM exit occurs immediately after VM entry (no instructions were executed in VMX non-root operation), the value saved may match that which was loaded on VM entry (see Section 26.6.3). Otherwise, the following items apply:

- Bit 12 (enabled breakpoint) is set to 1 if there was at least one matched data or I/O breakpoint that was enabled in DR7. Bit 12 is also set if it had been set on VM entry, causing there to be valid pending debug exceptions (see Section 26.6.3) and the VM exit occurred before those exceptions were either delivered or lost. In other cases, bit 12 is cleared to 0.
- The setting of bit 14 (BS) is implementation-specific. However, it is not set if RFLAGS.TF = 0 or IA32_DEBUGCTL.BTF = 1.

— The reserved bits in the field are cleared.

- If the "save VMX-preemption timer value" VM-exit control is 1, the value of timer is saved into the VMX-preemption timer-value field. This is the value loaded from this field on VM entry as subsequently decremented (see Section 25.5.1). VM exits due to timer expiration save the value 0. Other VM exits may also save the value 0 if the timer expired during VM exit. (If the "save VMX-preemption timer value" VM-exit control is 0, VM exit does not modify the value of the VMX-preemption timer-value field.)
- If the logical processor supports the 1-setting of the "enable EPT" VM-execution control, values are saved into the four (4) PDPTE fields as follows:
  - If the "enable EPT" VM-execution control is 1 and the logical processor was using PAE paging at the time of the VM exit, the PDPTE values currently in use are saved:[1]
    - The values saved into bits 11:9 of each of the fields is undefined.
    - If the value saved into one of the fields has bit 0 (present) clear, the value saved into bits 63:1 of that field is undefined. That value need not correspond to the value that was loaded by VM entry or to any value that might have been loaded in VMX non-root operation.
    - If the value saved into one of the fields has bit 0 (present) set, the value saved into bits 63:12 of the field is a guest-physical address.

1. A logical processor uses PAE paging if CR0.PG = 1, CR4.PAE = 1 and IA32_EFER.LMA = 0. See Section 4.4 in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*. "Enable EPT" is a secondary processor-based VM-execution control. If bit 31 of the primary processor-based VM-execution controls is 0, VM exit functions as if the "enable EPT" VM-execution control were 0. See Section 24.6.2.

— If the "enable EPT" VM-execution control is 0 or the logical processor was not using PAE paging at the time of the VM exit, the values saved are undefined.

27.4 SAVING MSRS

After processor state is saved to the guest-state area, values of MSRs may be stored into the VM-exit MSR-store area (see Section 24.7.2). Specifically each entry in that area (up to the number specified in the VM-exit MSR-store count) is processed in order by storing the value of the MSR indexed by bits 31:0 (as they would be read by RDMSR) into bits 127:64. Processing of an entry fails in either of the following cases:

- The value of bits 31:8 is 000008H, meaning that the indexed MSR is one that allows access to an APIC register when the local APIC is in x2APIC mode.
- The value of bits 31:0 indicates an MSR that can be read only in system-management mode (SMM) and the VM exit will not end in SMM. (IA32_SMBASE is an MSR that can be read only in SMM.)
- The value of bits 31:0 indicates an MSR that cannot be saved on VM exits for model-specific reasons. A processor may prevent certain MSRs (based on the value of bits 31:0) from being stored on VM exits, even if they can normally be read by RDMSR. Such model-specific behavior is documented in Chapter 2, "Model-Specific Registers (MSRs)" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 4*.
- Bits 63:32 of the entry are not all 0.
- An attempt to read the MSR indexed by bits 31:0 would cause a general-protection exception if executed via RDMSR with CPL = 0.

A VMX abort occurs if processing fails for any entry. See Section 27.7.

27.5 LOADING HOST STATE

Processor state is updated on VM exits in the following ways:

- Some state is loaded from or otherwise determined by the contents of the host-state area.
- Some state is determined by VM-exit controls.
- Some state is established in the same way on every VM exit.
- The page-directory pointers are loaded based on the values of certain control registers.

This loading may be performed in any order.

On processors that support Intel 64 architecture, the full values of each 64-bit field loaded (for example, the base address for GDTR) is loaded regardless of the mode of the logical processor before and after the VM exit.

The loading of host state is detailed in Section 27.5.1 to Section 27.5.5. These sections reference VMCS fields that correspond to processor state. Unless otherwise stated, these references are to fields in the host-state area.

A logical processor is in IA-32e mode after a VM exit only if the "host address-space size" VM-exit control is 1. If the logical processor was in IA-32e mode before the VM exit and this control is 0, a VMX abort occurs. See Section 27.7.

In addition to loading host state, VM exits clear address-range monitoring (Section 27.5.6).

After the state loading described in this section, VM exits may load MSRs from the VM-exit MSR-load area (see Section 27.6). This loading occurs only after the state loading described in this section.

27.5.1 Loading Host Control Registers, Debug Registers, MSRs

VM exits load new values for controls registers, debug registers, and some MSRs:

- CR0, CR3, and CR4 are loaded from the CR0 field, the CR3 field, and the CR4 field, respectively, with the following exceptions:
  — The following bits are not modified:

  - For CR0, ET, CD, NW; bits 63:32 (on processors that support Intel 64 architecture), 28:19, 17, and 15:6; and any bits that are fixed in VMX operation (see Section 23.8).[1]
  - For CR3, bits 63:52 and bits in the range 51:32 beyond the processor's physical-address width (they are cleared to 0).[2] (This item applies only to processors that support Intel 64 architecture.)
  - For CR4, any bits that are fixed in VMX operation (see Section 23.8).
- CR4.PAE is set to 1 if the "host address-space size" VM-exit control is 1.
- CR4.PCIDE is set to 0 if the "host address-space size" VM-exit control is 0.

- DR7 is set to 400H.
- The following MSRs are established as follows:
  - The IA32_DEBUGCTL MSR is cleared to 00000000_00000000H.
  - The IA32_SYSENTER_CS MSR is loaded from the IA32_SYSENTER_CS field. Since that field has only 32 bits, bits 63:32 of the MSR are cleared to 0.
  - IA32_SYSENTER_ESP MSR and IA32_SYSENTER_EIP MSR are loaded from the IA32_SYSENTER_ESP field and the IA32_SYSENTER_EIP field, respectively.

    If the processor does not support the Intel 64 architecture, these fields have only 32 bits; bits 63:32 of the MSRs are cleared to 0.

    If the processor does support the Intel 64 architecture and the processor supports N < 64 linear-address bits, each of bits 63:N is set to the value of bit N–1.[3]
  - The following steps are performed on processors that support Intel 64 architecture:
    - The MSRs FS.base and GS.base are loaded from the base-address fields for FS and GS, respectively (see Section 27.5.2).
    - The LMA and LME bits in the IA32_EFER MSR are each loaded with the setting of the "host address-space size" VM-exit control.
  - If the "load IA32_PERF_GLOBAL_CTRL" VM-exit control is 1, the IA32_PERF_GLOBAL_CTRL MSR is loaded from the IA32_PERF_GLOBAL_CTRL field. Bits that are reserved in that MSR are maintained with their reserved values.
  - If the "load IA32_PAT" VM-exit control is 1, the IA32_PAT MSR is loaded from the IA32_PAT field. Bits that are reserved in that MSR are maintained with their reserved values.
  - If the "load IA32_EFER" VM-exit control is 1, the IA32_EFER MSR is loaded from the IA32_EFER field. Bits that are reserved in that MSR are maintained with their reserved values.
  - If the "clear IA32_BNDCFGS" VM-exit control is 1, the IA32_BNDCFGS MSR is cleared to 00000000_00000000H; otherwise, it is not modified.

  With the exception of FS.base and GS.base, any of these MSRs is subsequently overwritten if it appears in the VM-exit MSR-load area. See Section 27.6.

27.5.2 Loading Host Segment and Descriptor-Table Registers

Each of the registers CS, SS, DS, ES, FS, GS, and TR is loaded as follows (see below for the treatment of LDTR):

- The selector is loaded from the selector field. The segment is unusable if its selector is loaded with zero. The checks specified Section 26.3.1.2 limit the selector values that may be loaded. In particular, CS and TR are never loaded with zero and are thus never unusable. SS can be loaded with zero only on processors that

1. Bits 28:19, 17, and 15:6 of CR0 and CR0.ET are unchanged by executions of MOV to CR0. CR0.ET is always 1 and the other bits are always 0.
2. Software can determine a processor's physical-address width by executing CPUID with 80000008H in EAX. The physical-address width is returned in bits 7:0 of EAX.
3. Software can determine the number N by executing CPUID with 80000008H in EAX. The number of linear-address bits supported is returned in bits 15:8 of EAX.

support Intel 64 architecture and only if the VM exit is to 64-bit mode (64-bit mode allows use of segments marked unusable).

- The base address is set as follows:
  - CS. Cleared to zero.
  - SS, DS, and ES. Undefined if the segment is unusable; otherwise, cleared to zero.
  - FS and GS. Undefined (but, on processors that support Intel 64 architecture, canonical) if the segment is unusable and the VM exit is not to 64-bit mode; otherwise, loaded from the base-address field.

    If the processor supports the Intel 64 architecture and the processor supports N < 64 linear-address bits, each of bits 63:N is set to the value of bit N–1.[1] The values loaded for base addresses for FS and GS are also manifest in the FS.base and GS.base MSRs.
  - TR. Loaded from the host-state area. If the processor supports the Intel 64 architecture and the processor supports N < 64 linear-address bits, each of bits 63:N is set to the value of bit N–1.
- The segment limit is set as follows:
  - CS. Set to FFFFFFFFH (corresponding to a descriptor limit of FFFFFH and a G-bit setting of 1).
  - SS, DS, ES, FS, and GS. Undefined if the segment is unusable; otherwise, set to FFFFFFFFH.
  - TR. Set to 00000067H.
- The type field and S bit are set as follows:
  - CS. Type set to 11 and S set to 1 (execute/read, accessed, non-conforming code segment).
  - SS, DS, ES, FS, and GS. Undefined if the segment is unusable; otherwise, type set to 3 and S set to 1 (read/write, accessed, expand-up data segment).
  - TR. Type set to 11 and S set to 0 (busy 32-bit task-state segment).
- The DPL is set as follows:
  - CS, SS, and TR. Set to 0. The current privilege level (CPL) will be 0 after the VM exit completes.
  - DS, ES, FS, and GS. Undefined if the segment is unusable; otherwise, set to 0.
- The P bit is set as follows:
  - CS, TR. Set to 1.
  - SS, DS, ES, FS, and GS. Undefined if the segment is unusable; otherwise, set to 1.
- On processors that support Intel 64 architecture, CS.L is loaded with the setting of the "host address-space size" VM-exit control. Because the value of this control is also loaded into IA32_EFER.LMA (see Section 27.5.1), no VM exit is ever to compatibility mode (which requires IA32_EFER.LMA = 1 and CS.L = 0).
- D/B.
  - CS. Loaded with the inverse of the setting of the "host address-space size" VM-exit control. For example, if that control is 0, indicating a 32-bit guest, CS.D/B is set to 1.
  - SS. Set to 1.
  - DS, ES, FS, and GS. Undefined if the segment is unusable; otherwise, set to 1.
  - TR. Set to 0.
- G.
  - CS. Set to 1.
  - SS, DS, ES, FS, and GS. Undefined if the segment is unusable; otherwise, set to 1.
  - TR. Set to 0.

1. Software can determine the number N by executing CPUID with 80000008H in EAX. The number of linear-address bits supported is returned in bits 15:8 of EAX.

The host-state area does not contain a selector field for LDTR. LDTR is established as follows on all VM exits: the selector is cleared to 0000H, the segment is marked unusable and is otherwise undefined (although the base address is always canonical).

The base addresses for GDTR and IDTR are loaded from the GDTR base-address field and the IDTR base-address field, respectively. If the processor supports the Intel 64 architecture and the processor supports N < 64 linear-address bits, each of bits 63:N of each base address is set to the value of bit N–1 of that base address. The GDTR and IDTR limits are each set to FFFFH.

27.5.3 Loading Host RIP, RSP, and RFLAGS

RIP and RSP are loaded from the RIP field and the RSP field, respectively. RFLAGS is cleared, except bit 1, which is always set.

27.5.4 Checking and Loading Host Page-Directory-Pointer-Table Entries

If CR0.PG = 1, CR4.PAE = 1, and IA32_EFER.LMA = 0, the logical processor uses **PAE paging**. See Section 4.4 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*.[1] When in PAE paging is in use, the physical address in CR3 references a table of **page-directory-pointer-table entries** (PDPTEs). A MOV to CR3 when PAE paging is in use checks the validity of the PDPTEs and, if they are valid, loads them into the processor (into internal, non-architectural registers).

A VM exit is to a VMM that uses PAE paging if (1) bit 5 (corresponding to CR4.PAE) is set in the CR4 field in the host-state area of the VMCS; and (2) the "host address-space size" VM-exit control is 0. Such a VM exit may check the validity of the PDPTEs referenced by the CR3 field in the host-state area of the VMCS. Such a VM exit must check their validity if either (1) PAE paging was not in use before the VM exit; or (2) the value of CR3 is changing as a result of the VM exit. A VM exit to a VMM that does not use PAE paging must not check the validity of the PDPTEs.

A VM exit that checks the validity of the PDPTEs uses the same checks that are used when CR3 is loaded with MOV to CR3 when PAE paging is in use. If MOV to CR3 would cause a general-protection exception due to the PDPTEs that would be loaded (e.g., because a reserved bit is set), a VMX abort occurs (see Section 27.7). If a VM exit to a VMM that uses PAE does not cause a VMX abort, the PDPTEs are loaded into the processor as would MOV to CR3, using the value of CR3 being load by the VM exit.

27.5.5 Updating Non-Register State

VM exits affect the non-register state of a logical processor as follows:

- A logical processor is always in the active state after a VM exit.
- Event blocking is affected as follows:
  - — There is no blocking by STI or by MOV SS after a VM exit.
  - — VM exits caused directly by non-maskable interrupts (NMIs) cause blocking by NMI (see Table 24-3). Other VM exits do not affect blocking by NMI. (See Section 27.1 for the case in which an NMI causes a VM exit indirectly.)
- There are no pending debug exceptions after a VM exit.

Section 28.3 describes how the VMX architecture controls how a logical processor manages information in the TLBs and paging-structure caches. The following items detail how VM exits invalidate cached mappings:

- If the "enable VPID" VM-execution control is 0, the logical processor invalidates linear mappings and combined mappings associated with VPID 0000H (for all PCIDs); combined mappings for VPID 0000H are invalidated for all EP4TA values (EP4TA is the value of bits 51:12 of EPTP).

1. On processors that support Intel 64 architecture, the physical-address extension may support more than 36 physical-address bits. Software can determine a processor's physical-address width by executing CPUID with 80000008H in EAX. The physical-address width is returned in bits 7:0 of EAX.

- VM exits are not required to invalidate any guest-physical mappings, nor are they required to invalidate any linear mappings or combined mappings if the "enable VPID" VM-execution control is 1.

27.5.6 Clearing Address-Range Monitoring

The Intel 64 and IA-32 architectures allow software to monitor a specified address range using the MONITOR and MWAIT instructions. See Section 8.10.4 in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*. VM exits clear any address-range monitoring that may be in effect.

27.6 LOADING MSRS

VM exits may load MSRs from the VM-exit MSR-load area (see Section 24.7.2). Specifically each entry in that area (up to the number specified in the VM-exit MSR-load count) is processed in order by loading the MSR indexed by bits 31:0 with the contents of bits 127:64 as they would be written by WRMSR.

Processing of an entry fails in any of the following cases:

- The value of bits 31:0 is either C0000100H (the IA32_FS_BASE MSR) or C0000101H (the IA32_GS_BASE MSR).
- The value of bits 31:8 is 000008H, meaning that the indexed MSR is one that allows access to an APIC register when the local APIC is in x2APIC mode.
- The value of bits 31:0 indicates an MSR that can be written only in system-management mode (SMM) and the VM exit will not end in SMM. (IA32_SMM_MONITOR_CTL is an MSR that can be written only in SMM.)
- The value of bits 31:0 indicates an MSR that cannot be loaded on VM exits for model-specific reasons. A processor may prevent loading of certain MSRs even if they can normally be written by WRMSR. Such model-specific behavior is documented in Chapter 2, "Model-Specific Registers (MSRs)" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 4*.
- Bits 63:32 are not all 0.
- An attempt to write bits 127:64 to the MSR indexed by bits 31:0 of the entry would cause a general-protection exception if executed via WRMSR with CPL = 0.[1]

If processing fails for any entry, a VMX abort occurs. See Section 27.7.

If any MSR is being loaded in such a way that would architecturally require a TLB flush, the TLBs are updated so that, after VM exit, the logical processor does not use any translations that were cached before the transition.

27.7 VMX ABORTS

A problem encountered during a VM exit leads to a **VMX abort**. A VMX abort takes a logical processor into a shut-down state as described below.

A VMX abort does not modify the VMCS data in the VMCS region of any active VMCS. The contents of these data are thus suspect after the VMX abort.

On a VMX abort, a logical processor saves a nonzero 32-bit VMX-abort indicator field at byte offset 4 in the VMCS region of the VMCS whose misconfiguration caused the failure (see Section 24.2). The following values are used:

1. There was a failure in saving guest MSRs (see Section 27.4).
2. Host checking of the page-directory-pointer-table entries (PDPTEs) failed (see Section 27.5.4).
3. The current VMCS has been corrupted (through writes to the corresponding VMCS region) in such a way that the logical processor cannot complete the VM exit properly.

---

1. Note the following about processors that support Intel 64 architecture. If CR0.PG = 1, WRMSR to the IA32_EFER MSR causes a general-protection exception if it would modify the LME bit. Since CR0.PG is always 1 in VMX operation, the IA32_EFER MSR should not be included in the VM-exit MSR-load area for the purpose of modifying the LME bit.

4. There was a failure on loading host MSRs (see Section 27.6).
5. There was a machine-check event during VM exit (see Section 27.8).
6. The logical processor was in IA-32e mode before the VM exit and the "host address-space size" VM-entry control was 0 (see Section 27.5).

Some of these causes correspond to failures during the loading of state from the host-state area. Because the loading of such state may be done in any order (see Section 27.5) a VM exit that might lead to a VMX abort for multiple reasons (for example, the current VMCS may be corrupt and the host PDPTEs might not be properly configured). In such cases, the VMX-abort indicator could correspond to any one of those reasons.

A logical processor never reads the VMX-abort indicator in a VMCS region and writes it only with one of the non-zero values mentioned above. The VMX-abort indicator allows software on one logical processor to diagnose the VMX-abort on another. For this reason, it is recommended that software running in VMX root operation zero the VMX-abort indicator in the VMCS region of any VMCS that it uses.

After saving the VMX-abort indicator, operation of a logical processor experiencing a VMX abort depends on whether the logical processor is in SMX operation:[1]

- If the logical processor is in SMX operation, an Intel® TXT shutdown condition occurs. The error code used is 000DH, indicating "VMX abort." See *Intel® Trusted Execution Technology Measured Launched Environment Programming Guide*.
- If the logical processor is outside SMX operation, it issues a special bus cycle (to notify the chipset) and enters the **VMX-abort shutdown state**. RESET is the only event that wakes a logical processor from the VMX-abort shutdown state. The following events do not affect a logical processor in this state: machine-check events; INIT signals; external interrupts; non-maskable interrupts (NMIs); start-up IPIs (SIPIs); and system-management interrupts (SMIs).

27.8 MACHINE-CHECK EVENTS DURING VM EXIT

If a machine-check event occurs during VM exit, one of the following occurs:

- The machine-check event is handled as if it occurred before the VM exit:
  - If CR4.MCE = 0, operation of the logical processor depends on whether the logical processor is in SMX operation:[2]
    - If the logical processor is in SMX operation, an Intel® TXT shutdown condition occurs. The error code used is 000CH, indicating "unrecoverable machine-check condition."
    - If the logical processor is outside SMX operation, it goes to the shutdown state.
  - If CR4.MCE = 1, a machine-check exception (#MC) is generated:
    - If bit 18 (#MC) of the exception bitmap is 0, the exception is delivered through the guest IDT.
    - If bit 18 of the exception bitmap is 1, the exception causes a VM exit.
- The machine-check event is handled after VM exit completes:
  - If the VM exit ends with CR4.MCE = 0, operation of the logical processor depends on whether the logical processor is in SMX operation:
    - If the logical processor is in SMX operation, an Intel® TXT shutdown condition occurs with error code 000CH (unrecoverable machine-check condition).

1. A logical processor is in SMX operation if GETSEC[SEXIT] has not been executed since the last execution of GETSEC[SENTER]. A logical processor is outside SMX operation if GETSEC[SENTER] has not been executed or if GETSEC[SEXIT] was executed after the last execution of GETSEC[SENTER]. See Chapter 6, "Safer Mode Extensions Reference," in *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2B*.
2. A logical processor is in SMX operation if GETSEC[SEXIT] has not been executed since the last execution of GETSEC[SENTER]. A logical processor is outside SMX operation if GETSEC[SENTER] has not been executed or if GETSEC[SEXIT] was executed after the last execution of GETSEC[SENTER]. See Chapter 6, "Safer Mode Extensions Reference," in *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2B*.

    - If the logical processor is outside SMX operation, it goes to the shutdown state.
  - If the VM exit ends with CR4.MCE = 1, a machine-check exception (#MC) is delivered through the host IDT.
- A VMX abort is generated (see Section 27.7). The logical processor blocks events as done normally in VMX abort. The VMX abort indicator is 5, for "machine-check event during VM exit."

The first option is not used if the machine-check event occurs after any host state has been loaded. The second option is used only if VM entry is able to load all host state.

20.Updates to Chapter 35, Volume 3C

Change bars show changes to Chapter 35 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3C:* System Programming Guide, Part 3.

Changes to chapter: Updates to Sections 35.2.8.2 “Virtual-Machine Extensions (VMX)”, 35.3.7 “Decoder Synchronization (PSB+)”, 35.3.8 “Internal Buffer Overflow” and 35.8.3.2 “Estimating TSC within Intel PT”.

CHAPTER 35
INTEL® PROCESSOR TRACE

35.1 OVERVIEW

Intel® Processor Trace (**Intel PT**) is an extension of Intel® Architecture that captures information about software execution using dedicated hardware facilities that cause only minimal performance perturbation to the software being traced. This information is collected in **data packets**. The initial implementations of Intel PT offer **control flow tracing**, which generates a variety of packets to be processed by a software decoder. The packets include timing, program flow information (e.g. branch targets, branch taken/not taken indications) and program-induced mode related information (e.g. Intel TSX state transitions, CR3 changes). These packets may be buffered internally before being sent to the memory subsystem or other output mechanism available in the platform. Debug software can process the trace data and reconstruct the program flow.

Later generations include additional trace sources, including software trace instrumentation using PTWRITE, and Power Event tracing.

35.1.1 Features and Capabilities

Intel PT's control flow trace generates a variety of packets that, when combined with the binaries of a program by a post-processing tool, can be used to produce an exact execution trace. The packets record flow information such as instruction pointers (IP), indirect branch targets, and directions of conditional branches within contiguous code regions (basic blocks).

Intel PT can also be configured to log software-generated packets using PTWRITE, and packets describing processor power management events.

In addition, the packets record other contextual, timing, and bookkeeping information that enables both functional and performance debugging of applications. Intel PT has several control and filtering capabilities available to customize the tracing information collected and to append other processor state and timing information to enable debugging. For example, there are modes that allow packets to be filtered based on the current privilege level (CPL) or the value of CR3.

Configuration of the packet generation and filtering capabilities are programmed via a set of MSRs. The MSRs generally follow the naming convention of IA32_RTIT_*. The capability provided by these configuration MSRs are enumerated by CPUID, see Section 35.3. Details of the MSRs for configuring Intel PT are described in Section 35.2.7.

35.1.1.1 Packet Summary

After a tracing tool has enabled and configured the appropriate MSRs, the processor will collect and generate trace information in the following categories of packets (for more details on the packets, see Section 35.4):

- Packets about basic information on program execution: These include:
  - Packet Stream Boundary (**PSB**) packets: PSB packets act as 'heartbeats' that are generated at regular intervals (e.g., every 4K trace packet bytes). These packets allow the packet decoder to find the packet boundaries within the output data stream; a PSB packet should be the first packet that a decoder looks for when beginning to decode a trace.
  - Paging Information Packet (**PIP**): PIPs record modifications made to the CR3 register. This information, along with information from the operating system on the CR3 value of each process, allows the debugger to attribute linear addresses to their correct application source.
  - Time-Stamp Counter (**TSC**) packets: TSC packets aid in tracking wall-clock time, and contain some portion of the software-visible time-stamp counter.
  - Core Bus Ratio (**CBR**) packets: CBR packets contain the core:bus clock ratio.

  - Overflow (**OVF**) packets: OVF packets are sent when the processor experiences an internal buffer overflow, resulting in packets being dropped. This packet notifies the decoder of the loss and can help the decoder to respond to this situation.
- Packets about control flow information:
  - Taken Not-Taken (**TNT**) packets: TNT packets track the "direction" of direct conditional branches (taken or not taken).
  - Target IP (**TIP**) packets: TIP packets record the target IP of indirect branches, exceptions, interrupts, and other branches or events. These packets can contain the IP, although that IP value may be compressed by eliminating upper bytes that match the last IP. There are various types of TIP packets; they are covered in more detail in Section 35.4.2.2.
  - Flow Update Packets (**FUP**): FUPs provide the source IP addresses for asynchronous events (interrupt and exceptions), as well as other cases where the source address cannot be determined from the binary.
  - **MODE** packets: These packets provide the decoder with important processor execution information so that it can properly interpret the dis-assembled binary and trace log. MODE packets have a variety of formats that indicate details such as the execution mode (16-bit, 32-bit, or 64-bit).
- Packets inserted by software:
  - PTWRITE (PTW) packets: includes the value of the operand passed to the PTWRITE instruction (see "PTWRITE - Write Data to a Processor Trace Packet" in *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2B*).
- Packets about processor power management events:
  - MWAIT packets: Indicate successful completion of an MWAIT operation to a C-state deeper than C0.0.
  - Power State Entry (PWRE) packets: Indicate entry to a C-state deeper than C0.0.
  - Power State Exit (PWRX) packets: Indicate exit from a C-state deeper than C0.0, returning to C0.
  - Execution Stopped (EXSTOP) packets: Indicate that software execution has stopped, due to events such as P-state change, C-state change, or thermal throttling.

35.2 INTEL® PROCESSOR TRACE OPERATIONAL MODEL

This section describes the overall Intel Processor Trace mechanism and the essential concepts relevant to how it operates.

35.2.1 Change of Flow Instruction (COFI) Tracing

A basic program block is a section of code where no jumps or branches occur. The instruction pointers (IPs) in this block of code need not be traced, as the processor will execute them from start to end without redirecting code flow. Instructions such as branches, and events such as exceptions or interrupts, can change the program flow. These instructions and events that change program flow are called Change of Flow Instructions (COFI). There are three categories of COFI:

- Direct transfer COFI.
- Indirect transfer COFI.
- Far transfer COFI.

The following subsections describe the COFI events that result in trace packet generation. Table 35-1 lists branch instruction by COFI types. For detailed description of specific instructions, see *Intel® 64 and IA-32 Architectures Software Developer's Manual*.

Table 35-1. COFI Type for Branch Instructions

COFI Type	Instructions
Conditional Branch	JA, JAE, JB, JBE, JC, JCXZ< JECXZ, JRCXZ, JE, JG, JGE, JL, JLE, JNA, JNAE, JNB, JNBE, JNC, JNE, JNG, JNGE, JNL, JNLE, JNO, JNP, JNS, JNZ, JO, JP, JPE, JPO, JS, JZ, LOOP, LOOPE, LOOPNE, LOOPNZ, LOOPZ

Table 35-1. COFI Type for Branch Instructions

COFI Type	Instructions
Unconditional Direct Branch	JMP (E9 xx, EB xx), CALL (E8 xx)
Indirect Branch	JMP (FF /4), CALL (FF /2)
Near Ret	RET (C3, C2 xx)
Far Transfers	INT3, INTn, INTO, IRET, IRETD, IRETQ, JMP (EA xx, FF /5), CALL (9A xx, FF /3), RET (CB, CA xx), SYSCALL, SYSRET, SYSENTER, SYSEXIT, VMLAUNCH, VMRESUME

35.2.1.1 Direct Transfer COFI

Direct Transfer COFI are relative branches. This means that their target is an IP whose offset from the current IP is embedded in the instruction bytes. It is not necessary to indicate target of these instructions in the trace output since it can be obtained through the source disassembly. Conditional branches need to indicate only whether the branch is taken or not. Unconditional branches do not need any recording in the trace output. There are two sub-categories:

- **Conditional Branch (Jcc, J*CXZ) and LOOP**

  To track this type of instruction, the processor encodes a single bit (taken or not taken — TNT) to indicate the program flow after the instruction.

  Jcc, J*CXZ, and LOOP can be traced with TNT bits. To improve the trace packet output efficiency, the processor will compact several TNT bits into a single packet.

- **Unconditional Direct Jumps**

  There is no trace output required for direct unconditional jumps (like JMP near relative or CALL near relative) since they can be directly inferred from the application assembly. Direct unconditional jumps do not generate a TNT bit or a Target IP packet, though TIP.PGD and TIP.PGE packets can be generated by unconditional direct jumps that toggle Intel PT enables (see Section 35.2.5).

35.2.1.2 Indirect Transfer COFI

Indirect transfer instructions involve updating the IP from a register or memory location. Since the register or memory contents can vary at any time during execution, there is no way to know the target of the indirect transfer until the register or memory contents are read. As a result, the disassembled code is not sufficient to determine the target of this type of COFI. Therefore, tracing hardware must send out the destination IP in the trace packet for debug software to determine the target address of the COFI. Note that this IP may be a linear or effective address (see Section 35.3.1.1).

An indirect transfer instruction generates a Target IP Packet (TIP) that contains the target address of the branch. There are two sub-categories:

- **Near JMP Indirect and Near Call Indirect**

  As previously mentioned, the target of an indirect COFI resides in the contents of either a register or memory location. Therefore, the processor must generate a packet that includes this target address to allow the decoder to determine the program flow.

- **Near RET**

  When a CALL instruction executes, it pushes onto the stack the address of the next instruction following the CALL. Upon completion of the call procedure, the RET instruction is often used to pop the return address off of the call stack and redirect code flow back to the instruction following the CALL.

  A RET instruction simply transfers program flow to the address it popped off the stack. Because a called procedure may change the return address on the stack before executing the RET instruction, debug software can be misled if it assumes that code flow will return to the instruction following the last CALL. Therefore, even for near RET, a Target IP Packet may be sent.

  — **RET Compression**

    A special case is applied if the target of the RET is consistent with what would be expected from tracking the CALL stack. If it is assured that the decoder has seen the corresponding CALL (with "corresponding" defined

as the CALL with matching stack depth), and the RET target is the instruction after that CALL, the RET target may be "compressed". In this case, only a single TNT bit of "taken" is generated instead of a Target IP Packet. To ensure that the decoder will not be confused in cases of RET compression, only RETs that correspond to CALLs which have been seen since the last PSB packet may be compressed in a given logical processor. For details, see "Indirect Transfer Compression for Returns (RET)" in Section 35.4.2.2.

35.2.1.3 Far Transfer COFI

All operations that change the instruction pointer and are not near jumps are "far transfers". This includes exceptions, interrupts, traps, TSX aborts, and instructions that do far transfers.

All far transfers will produce a Target IP (TIP) packet, which provides the destination IP address. For those far transfers that cannot be inferred from the binary source (e.g., asynchronous events such as exceptions and interrupts), the TIP will be preceded by a Flow Update packet (FUP), which provides the source IP address at which the event was taken. Table 35-23 indicates exactly which IP will be included in the FUP generated by a far transfer.

35.2.2 Software Trace Instrumentation with PTWRITE

PTWRITE provides a mechanism by which software can instrument the Intel PT trace. PTWRITE is a ring3-accessible instruction that can be passed a register or memory variable, see "PTWRITE - Write Data to a Processor Trace Packet" in *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2B* for details. The contents of that variable will be used as the payload for the PTW packet (see Table 35-40 "PTW Packet Definition"), inserted at the time of PTWRITE retirement, assuming PTWRITE is enabled and all other filtering conditions are met. Decode and analysis software will then be able to determine the meaning of the PTWRITE packet based on the IP of the associated PTWRITE instruction.

PTWRITE is enabled via IA32_RTIT_CTL.PTWEn[12] (see Table 35-6). Optionally, the user can use IA32_RTIT_CTL.FUPonPTW[5] to enable PTW packets to be followed by FUP packets containing the IP of the associated PTWRITE instruction.

35.2.3 Power Event Tracing

Power Event Trace is a capability that exposes core- and thread-level sleep state and power down transition information. When this capability is enabled, the trace will expose information about:

- Scenarios where software execution stops.
  - Due to sleep state entry, frequency change, or other powerdown.
  - Includes the IP, when in the tracing context.
- The requested and resolved hardware thread C-state.
  - Including indication of hardware autonomous C-state entry.
- The last and deepest core C-state achieved during a sleep session.
- The reason for C-state wake.

This information is in addition to the bus ratio (CBR) information provided by default after any powerdown, and the timing information (TSC, TMA, MTC, CYC) provided during or after a powerdown state.

Power Event Trace is enabled via IA32_RTIT_CTL.PwrEvtEn[4].

35.2.4 Trace Filtering

Intel Processor Trace provides filtering capabilities, by which the debug/profile tool can control what code is traced.

35.2.4.1 Filtering by Current Privilege Level (CPL)

Intel PT provides the ability to configure a logical processor to generate trace packets only when CPL = 0, when CPL > 0, or regardless of CPL.

CPL filtering ensures that no IPs or other architectural state information associated with the filtered CPL can be seen in the log. For example, if the processor is configured to trace only when CPL > 0, and software executes SYSCALL (changing the CPL to 0), the destination IP of the SYSCALL will be suppressed from the generated packet (see the discussion of TIP.PGD in Section 35.4.2.5).

It should be noted that CPL is always 0 in real-address mode and that CPL is always 3 in virtual-8086 mode. To trace code in these modes, filtering should be configured accordingly.

When software is executing in a non-enabled CPL, ContextEn is cleared. See Section 35.2.5.1 for details.

35.2.4.2 Filtering by CR3

Intel PT supports a CR3-filtering mechanism by which the generation of packets containing architectural states can be enabled or disabled based on the value of CR3. A debugger can use CR3 filtering to trace only a single application without context switching the state of the RTIT MSRs. For the reconstruction of traces from software with multiple threads, debug software may wish to context-switch for the state of the RTIT MSRs (if the operating system does not provide context-switch support) to separate the output for the different threads (see Section 35.3.5, “Context Switch Consideration”).

To trace for only a single CR3 value, software can write that value to the IA32_RTIT_CR3_MATCH MSR, and set IA32_RTIT_CTL.CR3Filter. When CR3 value does not match IA32_RTIT_CR3_MATCH and IA32_RTIT_CTL.CR3Filter is 1, ContextEn is forced to 0, and packets containing architectural states will not be generated. Some other packets can be generated when ContextEn is 0; see Section 35.2.5.3 for details. When CR3 does match IA32_RTIT_CR3_MATCH (or when IA32_RTIT_CTL.CR3Filter is 0), CR3 filtering does not force ContextEn to 0 (although it could be 0 due to other filters or modes).

CR3 matches IA32_RTIT_CR3_MATCH if the two registers are identical for bits 63:12, or 63:5 when in PAE paging mode; the lower 5 bits of CR3 and IA32_RTIT_CR3_MATCH are ignored. CR3 filtering is independent of the value of CR0.PG.

When CR3 filtering is in use, PIP packets may still be seen in the log if the processor is configured to trace when CPL = 0 (IA32_RTIT_CTL.OS = 1). If not, no PIP packets will be seen.

35.2.4.3 Filtering by IP

Trace packet generation with configurable filtering by IP is supported if CPUID.(EAX=14H, ECX=0):EBX[bit 2] = 1. Intel PT can be configured to enable the generation of packets containing architectural states only when the processor is executing code within certain IP ranges. If the IP is outside of these ranges, generation of some packets is blocked.

IP filtering is enabled using the ADDRn_CFG fields in the IA32_RTIT_CTL MSR (Section 35.2.7.2), where the digit 'n' is a zero-based number that selects which address range is being configured. Each ADDRn_CFG field configures the use of the register pair IA32_RTIT_ADDRn_A and IA32_RTIT_ADDRn_B (Section 35.2.7.5). IA32_RTIT_ADDRn_A defines the base and IA32_RTIT_ADDRn_B specifies the limit of the range in which tracing is enabled. Thus each range, referred to as the ADDRn range, is defined by [IA32_RTIT_ADDRn_A. IA32_RTIT_ADDRn_B]. There can be multiple such ranges, software can query CPUID (Section 35.3.1) for the number of ranges supported on a processor.

Default behavior (ADDRn_CFG=0) defines no IP filter range, meaning FilterEn is always set. In this case code at any IP can be traced, though other filters, such as CR3 or CPL, could limit tracing. When ADDRn_CFG is set to enable IP filtering (see Section 35.3.1), tracing will commence when a taken branch or event is seen whose target address is in the ADDRn range.

While inside a tracing region and with FilterEn is set, leaving the tracing region may only be detected once a taken branch or event with a target outside the range is retired. If an ADDRn range is entered or exited by executing the next sequential instruction, rather than by a control flow transfer, FilterEn may not toggle immediately. See Section 35.2.5.5 for more details on FilterEn.

Note that these address range base and limit values are inclusive, such that the range includes the first and last instruction whose first instruction byte is in the ADDRn range.

Depending upon processor implementation, IP filtering may be based on linear or effective address. This can cause different behavior between implementations if CSbase is not equal to zero or in real mode. See Section 35.3.1.1 for details. Software can query CPUID to determine filters are based on linear or effective address (Section 35.3.1).

Note that some packets, such as MTC (Section 35.3.7) and other timing packets, do not depend on FilterEn. For details on which packets depend on FilterEn, and hence are impacted by IP filtering, see Section 35.4.1.

TraceStop

The ADDRn ranges can also be configured to cause tracing to be disabled upon entry to the specified region. This is intended for cases where unexpected code is executed, and the user wishes to immediately stop generating packets in order to avoid overwriting previously written packets.

The TraceStop mechanism works much the same way that IP filtering does, and uses the same address comparison logic. The TraceStop region base and limit values are programmed into one or more ADDRn ranges, but IA32_RTIT_CTL.ADDRn_CFG is configured with the TraceStop encoding. Like FilterEn, TraceStop is detected when a taken branch or event lands in a TraceStop region.

Further, TraceStop requires that TriggerEn=1 at the beginning of the branch/event, and ContextEn=1 upon completion of the branch/event. When this happens, the CPU will set IA32_RTIT_STATUS.Stopped, thereby clearing TriggerEn and hence disabling packet generation. This may generate a TIP.PGD packet with the target IP of the branch or event that entered the TraceStop region. Finally, a TraceStop packet will be inserted, to indicate that the condition was hit.

If a TraceStop condition is encountered during buffer overflow (Section 35.3.8), it will not be dropped, but will instead be signaled once the overflow has resolved.

Note that a TraceStop event does not guarantee that all internally buffered packets are flushed out of internal buffers. To ensure that this has occurred, the user should clear TraceEn.

To resume tracing after a TraceStop event, the user must first disable Intel PT by clearing IA32_RTIT_CTL.TraceEn before the IA32_RTIT_STATUS.Stopped bit can be cleared. At this point Intel PT can be reconfigured, and tracing resumed.

Note that the IA32_RTIT_STATUS.Stopped bit can also be set using the ToPA STOP bit. See Section 35.2.6.2.

IP Filtering Example

The following table gives an example of IP filtering behavior. Assume that IA32_RTIT_ADDRn_A = the IP of RangeBase, and that IA32_RTIT_ADDRn_B = the IP of RangeLimit, while IA32_RTIT_CTL.ADDRn_CFG = 0x1 (enable ADDRn range as a FilterEn range).

Table 35-2. IP Filtering Packet Example

Code Flow	Packets

`Bar:`
`  jmp RangeBase // jump into filter range`
`RangeBase:`
`  jcc Foo // not taken`
`  add eax, 1`
`Foo:`
`  jmp RangeLimit+1 // jump out of filter range`
`RangeLimit:`
`  nop`
`  jcc Bar`
	TIP.PGE(RangeBase)
	TNT(0)
	TIP.PGD(RangeLimit+1)

IP Filtering and TraceStop

It is possible for the user to configure IP filter range(s) and TraceStop range(s) that overlap. In this case, code executing in the non-overlapping portion of either range will behave as would be expected from that range. Code executing in the overlapping range will get TraceStop behavior.

35.2.5 Packet Generation Enable Controls

Intel Processor Trace includes a variety of controls that determine whether a packet is generated. In general, most packets are sent only if Packet Enable (**PacketEn**) is set. PacketEn is an internal state maintained in hardware in

response to software configurable enable controls, PacketEn is not visible to software directly. The relationship of PacketEn to the software-visible controls in the configuration MSRs is described in this section.

35.2.5.1 Packet Enable (PacketEn)

When PacketEn is set, the processor is in the mode that Intel PT is monitoring and all packets can be generated to log what is being executed. PacketEn is composed of other states according to this relationship:

```
PacketEn ← TriggerEn AND ContextEn AND FilterEn AND BranchEn
```

These constituent controls are detailed in the following subsections.

PacketEn ultimately determines when the processor is tracing. When PacketEn is set, all control flow packets are enabled. When PacketEn is clear, no control flow packets are generated, though other packets (timing and bookkeeping packets) may still be sent. See Section 35.2.6 for details of PacketEn and packet generation.

Note that, on processors that do not support IP filtering (i.e., CPUID.(EAX=14H, ECX=0):EBX.IPFILT_WRSTPRSV[bit 2] = 0), FilterEn is treated as always set.

35.2.5.2 Trigger Enable (TriggerEn)

Trigger Enable (**TriggerEn**) is the primary indicator that trace packet generation is active. TriggerEn is set when IA32_RTIT_CTL.TraceEn is set, and cleared by any of the following conditions:

- TraceEn is cleared by software.
- A TraceStop condition is encountered and IA32_RTIT_STATUS.Stopped is set.
- IA32_RTIT_STATUS.Error is set due to an operational error (see Section 35.3.9).

Software can discover the current TriggerEn value by reading the IA32_RTIT_STATUS.TriggerEn bit. When TriggerEn is clear, tracing is inactive and no packets are generated.

35.2.5.3 Context Enable (ContextEn)

Context Enable (**ContextEn**) indicates whether the processor is in the state or mode that software configured hardware to trace. For example, if execution with CPL = 0 code is not being traced (IA32_RTIT_CTL.OS = 0), then ContextEn will be 0 when the processor is in CPL0.

Software can discover the current ContextEn value by reading the IA32_RTIT_STATUS.ContextEn bit. ContextEn is defined as follows:

```
ContextEn = !((IA32_RTIT_CTL.OS = 0 AND CPL = 0) OR
(IA32_RTIT_CTL.USER = 0 AND CPL > 0) OR (IS_IN_A_PRODUCTION_ENCLAVE[1]) OR
(IA32_RTIT_CTL.CR3Filter = 1 AND IA32_RTIT_CR3_MATCH does not match CR3)
```

If the clearing of ContextEn causes PacketEn to be cleared, a Packet Generation Disable (TIP.PGD) packet is generated, but its IP payload is suppressed. If the setting of ContextEn causes PacketEn to be set, a Packet Generation Enable (TIP.PGE) packet is generated.

When ContextEn is 0, control flow packets (TNT, FUP, TIP.*, MODE.*) are not generated, and no Linear Instruction Pointers (LIPs) are exposed. However, some packets, such as MTC and PSB (see Section 35.4.2.16 and Section 35.4.2.17), may still be generated while ContextEn is 0. For details of which packets are generated only when ContextEn is set, see Section 35.4.1.

The processor does not update ContextEn when TriggerEn = 0.

The value of ContextEn will toggle only when TriggerEn = 1.

35.2.5.4 Branch Enable (BranchEn)

This value is based purely on the IA32_RTIT_CTL.BranchEn value. If **BranchEn** is not set, then relevant COFI packets (TNT, TIP*, FUP, MODE.*) are suppressed. Other packets related to timing (TSC, TMA, MTC, CYC), as well

1. Trace packets generation is disabled in a production enclave, see Section 35.2.8.3. See *Intel® Software Guard Extensions Programming Reference* about differences between a production enclave and a debug enclave.

as PSB, will be generated normally regardless. Further, PIP and VMCS continue to be generated, as indicators of what software is running.

35.2.5.5 Filter Enable (FilterEn)

Filter Enable indicates that the Instruction Pointer (IP) is within the range of IPs that Intel PT is configured to watch. Software can get the state of Filter Enable by a RDMSR of IA32_RTIT_STATUS.FilterEn. For details on configuration and use of IP filtering, see Section 35.2.4.3.

On clearing of FilterEn that also clears PacketEn, a Packet Generation Disable (TIP.PGD) will be generated, but unlike the ContextEn case, the IP payload may not be suppressed. For direct, unconditional branches, as well as for indirect branches (including RETs), the PGD generated by leaving the tracing region and clearing FilterEn will contain the target IP. This means that IPs from outside the configured range can be exposed in the trace, as long as they are within context.

When FilterEn is 0, control flow packets are not generated (e.g., TNT, TIP). However, some packets, such as PIP, MTC, and PSB, may still be generated while FilterEn is clear. For details on packet enable dependencies, see Section 35.4.1.

After TraceEn is set, FilterEn is set to 1 at all times if there is no IP filter range configured by software (IA32_RTIT_CTL.ADDRn_CFG != 1, for all n), or if the processor does not support IP filtering (i.e., CPUID.(EAX=14H, ECX=0):EBX.IPFILT_WRSTPRSV[bit 2] = 0). FilterEn will toggle only when TraceEn=1 and ContextEn=1, and when at least one range is configured for IP filtering.

35.2.6 Trace Output

Intel PT output should be viewed independently from trace content and filtering mechanisms. The options available for trace output can vary across processor generations and platforms.

Trace output is written out using one of the following output schemes, as configured by the ToPA and FabricEn bit fields of IA32_RTIT_CTL (see Section 35.2.7.2):

- A single, contiguous region of physical address space.
- A collection of variable-sized regions of physical memory. These regions are linked together by tables of pointers to those regions, referred to as Table of Physical Addresses (**ToPA**). The trace output stores bypass the caches and the TLBs, but are not serializing. This is intended to minimize the performance impact of the output.
- A platform-specific trace transport subsystem.

Regardless of the output scheme chosen, Intel PT stores bypass the processor caches by default. This ensures that they don't consume precious cache space, but they do not have the serializing aspects associated with un-cacheable (UC) stores. Software should avoid using MTRRs to mark any portion of the Intel PT output region as UC, as this may override the behavior described above and force Intel PT stores to UC, thereby incurring severe performance impact.

There is no guarantee that a packet will be written to memory or other trace endpoint after some fixed number of cycles after a packet-producing instruction executes. The only way to assure that all packets generated have reached their endpoint is to clear TraceEn and follow that with a store, fence, or serializing instruction; doing so ensures that all buffered packets are flushed out of the processor.

35.2.6.1 Single Range Output

When IA32_RTIT_CTL.ToPA and IA32_RTIT_CTL.FabricEn bits are clear, trace packet output is sent to a single, contiguous memory (or MMIO if DRAM is not available) range defined by a base address in IA32_RTIT_OUTPUT_BASE (Section 35.2.7.7) and mask value in IA32_RTIT_OUTPUT_MASK_PTRS (Section 35.2.7.8). The current write pointer in this range is also stored in IA32_RTIT_OUTPUT_MASK_PTRS. This output range is circular, meaning that when the writes wrap around the end of the buffer they begin again at the base address.

This output method is best suited for cases where Intel PT output is either:

- Configured to be directed to a sufficiently large contiguous region of DRAM.

- Configured to go to an MMIO debug port, in order to route Intel PT output to a platform-specific trace endpoint (e.g., JTAG). In this scenario, a specific range of addresses is written in a circular manner, and SoC will intercept these writes and direct them to the proper device. Repeated writes to the same address do not overwrite each other, but are accumulated by the debugger, and hence no data is lost by the circular nature of the buffer.

The processor will determine the address to which to write the next trace packet output byte as follows:

```
OutputBase[63:0] ← IA32_RTIT_OUTPUT_BASE[63:0]
OutputMask[63:0] ← ZeroExtend64(IA32_RTIT_OUTPUT_MASK_PTRS[31:0])
OutputOffset[63:0] ← ZeroExtend64(IA32_RTIT_OUTPUT_MASK_PTRS[63:32])
trace_store_phys_addr ← (OutputBase & ~OutputMask) + (OutputOffset & OutputMask)
```

Single-Range Output Errors

If the output base and mask are not properly configured by software, an operational error (see Section 35.3.9) will be signaled, and tracing disabled. Error scenarios with single-range output are:

- Mask value is non-contiguous.

  IA32_RTIT_OUTPUT_MASK_PTRS.MaskOrTablePointer value has a 0 in a less significant bit position than the most significant bit containing a 1.

- Base address and Mask are mis-aligned, and have overlapping bits set.

  IA32_RTIT_OUTPUT_BASE && IA32_RTIT_OUTPUT_MASK_PTRS.MaskOrTableOffset > 0.

- Illegal Output Offset

  IA32_RTIT_OUTPUT_MASK_PTRS.OutputOffset is greater than the mask value (IA32_RTIT_OUTPUT_MASK_PTRS.MaskOrTableOffset).

Also note that errors can be signaled due to trace packet output overlapping with restricted memory, see Section 35.2.6.4.

35.2.6.2 Table of Physical Addresses (ToPA)

When IA32_RTIT_CTL.ToPA is set and IA32_RTIT_CTL.FabricEn is clear, the ToPA output mechanism is utilized. The ToPA mechanism uses a linked list of tables; see Figure 35-1 for an illustrative example. Each entry in the table contains some attribute bits, a pointer to an output region, and the size of the region. The last entry in the table may hold a pointer to the next table. This pointer can either point to the top of the current table (for circular array) or to the base of another table. The table size is not fixed, since the link to the next table can exist at any entry.

The processor treats the various output regions referenced by the ToPA table(s) as a unified buffer. This means that a single packet may span the boundary between one output region and the next.

The ToPA mechanism is controlled by three values maintained by the processor:

- **proc_trace_table_base.**
  This is the physical address of the base of the current ToPA table. When tracing is enabled, the processor loads this value from the IA32_RTIT_OUTPUT_BASE MSR. While tracing is enabled, the processor updates the IA32_RTIT_OUTPUT_BASE MSR with changes to proc_trace_table_base, but these updates may not be synchronous to software execution. When tracing is disabled, the processor ensures that the MSR contains the latest value of proc_trace_table_base.

- **proc_trace_table_offset.**
  This indicates the entry of the current table that is currently in use. (This entry contains the address of the current output region.) When tracing is enabled, the processor loads this value from bits 31:7 (MaskOrTableOffset) of the IA32_RTIT_OUTPUT_MASK_PTRS. While tracing is enabled, the processor updates IA32_RTIT_OUTPUT_MASK_PTRS.MaskOrTableOffset with changes to proc_trace_table_offset, but these updates may not be synchronous to software execution. When tracing is disabled, the processor ensures that the MSR contains the latest value of proc_trace_table_offset.

- **proc_trace_output_offset.**
  This a pointer into the current output region and indicates the location of the next write. When tracing is enabled, the processor loads this value from bits 63:32 (OutputOffset) of the IA32_RTIT_OUTPUT_MASK_PTRS. While tracing is enabled, the processor updates

IA32_RTIT_OUTPUT_MASK_PTRS.OutputOffset with changes to proc_trace_output_offset, but these updates may not be synchronous to software execution. When tracing is disabled, the processor ensures that the MSR contains the latest value of proc_trace_output_offset.

Figure 35-1 provides an illustration (not to scale) of the table and associated pointers.

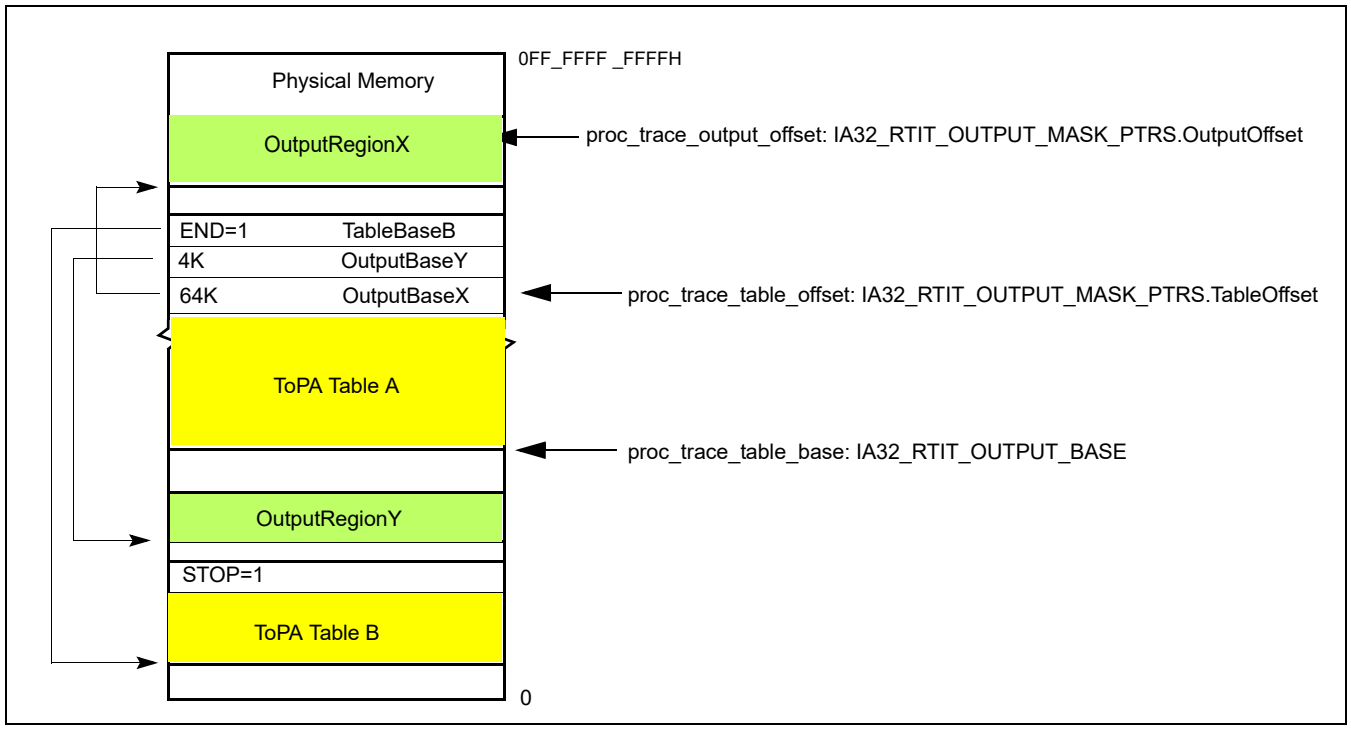


Figure 35-1. ToPA Memory Illustration

With the ToPA mechanism, the processor writes packets to the current output region (identified by proc_trace_table_base and the proc_trace_table_offset). The offset within that region to which the next byte will be written is identified by proc_trace_output_offset. When that region is filled with packet output (thus proc_trace_output_offset = RegionSize–1), proc_trace_table_offset is moved to the next ToPA entry, proc_trace_output_offset is set to 0, and packet writes begin filling the new output region specified by proc_trace_table_offset.

As packets are written out, each store derives its physical address as follows:

```
trace_store_phys_addr ← Base address from current ToPA table entry +
proc_trace_output_offset
```

Eventually, the regions represented by all entries in the table may become full, and the final entry of the table is reached. An entry can be identified as the final entry because it has either the END or STOP attribute. The END attribute indicates that the address in the entry does not point to another output region, but rather to another ToPA table. The STOP attribute indicates that tracing will be disabled once the corresponding region is filled. See Section 35.2.6.2 for details on STOP.

When an END entry is reached, the processor loads proc_trace_table_base with the base address held in this END entry, thereby moving the current table pointer to this new table. The proc_trace_table_offset is reset to 0, as is the proc_trace_output_offset, and packet writes will resume at the base address indicated in the first entry.

If the table has no STOP or END entry, and trace-packet generation remains enabled, eventually the maximum table size will be reached (proc_trace_table_offset = 01FFFFFH). In this case, the proc_trace_table_offset and proc_trace_output_offset are reset to 0 (wrapping back to the beginning of the current table) once the last output region is filled.

It is important to note that processor updates to the IA32_RTIT_OUTPUT_BASE and IA32_RTIT_OUTPUT_MASK_PTRS MSRs are asynchronous to instruction execution. Thus, reads of these MSRs

while Intel PT is enabled may return stale values. Like all IA32_RTIT_* MSRs, the values of these MSRs should not be trusted or saved unless trace packet generation is first disabled by clearing IA32_RTIT_CTL.TraceEn. This ensures that he output MSR values account for all packets generated to that point, after which the output MSR values will be frozen until tracing resumes. [1]

The processor may cache internally any number of entries from the current table or from tables that it references (directly or indirectly). If tracing is enabled, the processor may ignore or delay detection of modifications to these tables. To ensure that table changes are detected by the processor in a predictable manner, software should clear TraceEn before modifying the current table (or tables that it references) and only then re-enable packet generation.

Single Output Region ToPA Implementation

The first processor generation to implement Intel PT supports only ToPA configurations with a single ToPA entry followed by an END entry that points back to the first entry (creating one circular output buffer). Such processors enumerate CPUID.(EAX=14H,ECX=0):ECX.MENTRY[bit 1] = 0 and CPUID.(EAX=14H,ECX=0):ECX.TOPAOUT[bit 0] = 1.

If CPUID.(EAX=14H,ECX=0):ECX.MENTRY[bit 1] = 0, ToPA tables can hold only one output entry, which must be followed by an END=1 entry which points back to the base of the table. Hence only one contiguous block can be used as output.

The lone output entry can have INT or STOP set, but nonetheless must be followed by an END entry as described above. Note that, if INT=1, the PMI will actually be delivered before the region is filled.

ToPA Table Entry Format

The format of ToPA table entries is shown in Figure 35-2. The size of the address field is determined by the processor's physical-address width (MAXPHYADDR) in bits, as reported in CPUID.80000008H:EAX[7:0].

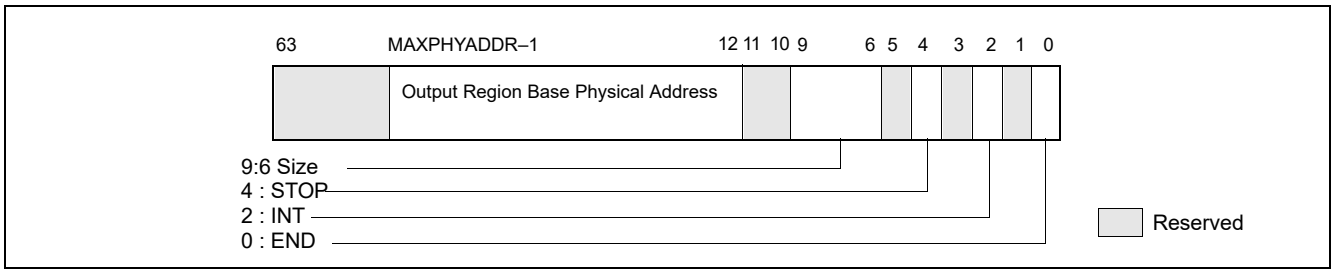


Figure 35-2. Layout of ToPA Table Entry

Table 35-3 describes the details of the ToPA table entry fields. If reserved bits are set to 1, an error is signaled.

Table 35-3. ToPA Table Entry Fields

ToPA Entry Field	Description
Output Region Base Physical Address	If END=0, this is the base physical address of the output region specified by this entry. Note that all regions must be aligned based on their size. Thus a 2M region must have bits 20:12 clear. If the region is not properly aligned, an operational error will be signaled when the entry is reached. If END=1, this is the 4K-aligned base physical address of the next ToPA table (which may be the base of the current table, or the first table in the linked list if a circular buffer is desired). If the processor supports only a single ToPA output region (see above), this address must be the value currently in the IA32_RTIT_OUTPUT_BASE MSR.
Size	Indicates the size of the associated output region. Encodings are: 0: 4K, 1: 8K, 2: 16K, 3: 32K, 4: 64K, 5: 128K, 6: 256K, 7: 512K, 8: 1M, 9: 2M, 10: 4M, 11: 8M, 12: 16M, 13: 32M, 14: 64M, 15: 128M This field is ignored if END=1.

1. Although WRMSR is a serializing instruction, the execution of WRMSR that forces packet writes by clearing TraceEn does not itself cause these writes to be globally observed.

Table 35-3. ToPA Table Entry Fields (Contd.)

ToPA Entry Field	Description
STOP	When the output region indicated by this entry is filled, software should disable packet generation. This will be accomplished by setting IA32_RTIT_STATUS.Stopped, which clears TriggerEn. This bit must be 0 if END=1; otherwise it is treated as reserved bit violation (see ToPA Errors).
INT	When the output region indicated by this entry is filled, signal Perfmon LVT interrupt. Note that if both INT and STOP are set in the same entry, the STOP will happen before the INT. Thus the interrupt handler should expect that the IA32_RTIT_STATUS.Stopped bit will be set, and will need to be reset before tracing can be resumed. This bit must be 0 if END=1; otherwise it is treated as reserved bit violation (see ToPA Errors).
END	If set, indicates that this is an END entry, and thus the address field points to a table base rather than an output region base. If END=1, INT and STOP must be set to 0; otherwise it is treated as reserved bit violation (see ToPA Errors). The Size field is ignored in this case. If the processor supports only a single ToPA output region (see above), END must be set in the second table entry.

ToPA STOP

Each ToPA entry has a STOP bit. If this bit is set, the processor will set the IA32_RTIT_STATUS.Stopped bit when the corresponding trace output region is filled. This will clear TriggerEn and thereby cease packet generation. See Section 35.2.7.4 for details on IA32_RTIT_STATUS.Stopped. This sequence is known as "ToPA Stop".

No TIP.PGD packet will be seen in the output when the ToPA stop occurs, since the disable happens only when the region is already full. When this occurs, output ceases after the last byte of the region is filled, which may mean that a packet is cut off in the middle. Any packets remaining in internal buffers are lost and cannot be recovered.

When ToPA stop occurs, the IA32_RTIT_OUTPUT_BASE MSR will hold the base address of the table whose entry had STOP=1. IA32_RTIT_OUTPUT_MASK_PTRS.MaskOrTableOffset will hold the index value for that entry, and the IA32_RTIT_OUTPUT_MASK_PTRS.OutputOffset should be set to the size of the region.

Note that this means the offset pointer is pointing to the next byte after the end of the region, a configuration that would produce an operational error if the configuration remained when tracing is re-enabled with IA32_RTIT_STATUS.Stopped cleared.

ToPA PMI

Each ToPA entry has an INT bit. If this bit is set, the processor will signal a performance-monitoring interrupt (PMI) when the corresponding trace output region is filled. This interrupt is not precise, and it is thus likely that writes to the next region will occur by the time the interrupt is taken.

The following steps should be taken to configure this interrupt:

1. Enable PMI via the LVT Performance Monitor register (at MMIO offset 340H in xAPIC mode; via MSR 834H in x2APIC mode). See *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3B* for more details on this register. For ToPA PMI, set all fields to 0, save for the interrupt vector, which can be selected by software.
2. Set up an interrupt handler to service the interrupt vector that a ToPA PMI can raise.
3. Set the interrupt flag by executing STI.
4. Set the INT bit in the ToPA entry of interest and enable packet generation, using the ToPA output option. Thus, TraceEn=ToPA=1 in the IA32_RTIT_CTL MSR.

Once the INT region has been filled with packet output data, the interrupt will be signaled. This PMI can be distinguished from others by checking bit 55 (Trace_ToPA_PMI) of the IA32_PERF_GLOBAL_STATUS MSR (MSR 38EH). Once the ToPA PMI handler has serviced the relevant buffer, writing 1 to bit 55 of the MSR at 390H (IA32_GLOBAL_STATUS_RESET) clears IA32_PERF_GLOBAL_STATUS.Trace_ToPA_PMI.

Intel PT is not frozen on PMI, and thus the interrupt handler will be traced (though filtering can prevent this). The Freeze_Perfmon_on_PMI and Freeze_LBRs_on_PMI settings in IA32_DEBUGCTL will be applied on ToPA PMI just as on other PMIs, and hence Perfmon counters are frozen.

Assuming the PMI handler wishes to read any buffered packets for persistent output, or wishes to modify any Intel PT MSRs, software should first disable packet generation by clearing TraceEn. This ensures that all buffered packets are written to memory and avoids tracing of the PMI handler. The configuration MSRs can then be used to determine where tracing has stopped. If packet generation is disabled by the handler, it should then be manually re-enabled before the IRET if continued tracing is desired.

In rare cases, it may be possible to trigger a second ToPA PMI before the first is handled. This can happen if another ToPA region with INT=1 is filled before, or shortly after, the first PMI is taken, perhaps due to EFLAGS.IF being cleared for an extended period of time. This can manifest in two ways: either the second PMI is triggered before the first is taken, and hence only one PMI is taken, or the second is triggered after the first is taken, and thus will be taken when the handler for the first completes. Software can minimize the likelihood of the second case by clearing TraceEn at the beginning of the PMI handler. Further, it can detect such cases by then checking the Interrupt Request Register (IRR) for PMI pending, and checking the ToPA table base and off-set pointers (in IA32_RTIT_OUTPUT_BASE and IA32_RTIT_OUTPUT_MASK_PTRS) to see if multiple entries with INT=1 have been filled.

ToPA PMI and Single Output Region ToPA Implementation

A processor that supports only a single ToPA output region implementation (such that only one output region is supported; see above) will attempt to signal a ToPA PMI interrupt before the output wraps and overwrites the top of the buffer. To support this functionality, the PMI handler should disable packet generation as soon as possible.

Due to PMI skid, it is possible that, in rare cases, the wrap will have occurred before the PMI is delivered. Software can avoid this by setting the STOP bit in the ToPA entry (see Table 35-3); this will disable tracing once the region is filled, and no wrap will occur. This approach has the downside of disabling packet generation so that some of the instructions that led up to the PMI will not be traced. If the PMI skid is significant enough to cause the region to fill and tracing to be disabled, the PMI handler will need to clear the IA32_RTIT_STATUS.Stopped indication before tracing can resume.

ToPA PMI and XSAVES/XRSTORS State Handling

In some cases the ToPA PMI may be taken after completion of an XSAVES instruction that switches Intel PT state, and in such cases any modification of Intel PT MSRs within the PMI handler will not persist when the saved Intel PT context is later restored with XRSTORS. To account for such a scenario, it is recommended that the Intel PT output configuration be modified by altering the ToPA tables themselves, rather than the Intel PT output MSRs.

Table 35-4 depicts a recommended PMI handler algorithm for managing multi-region ToPA output and handling ToPA PMIs that may arrive between XSAVES and XRSTORS. This algorithm is flexible to allow software to choose between adding entries to the current ToPA table, adding a new ToPA table, or using the current ToPA table as a circular buffer. It assumes that the ToPA entry that triggers the PMI is not the last entry in the table, which is the recommended treatment.

Table 35-4. Algorithm to Manage Intel PT ToPA PMI and XSAVES/XRSTORS

Pseudo Code Flow

```
IF (IA32_PERF_GLOBAL_STATUS.ToPA)
    Save IA32_RTIT_CTL value;
    IF ( IA32_RTIT_CTL.TraceEN )
        Disable Intel PT by clearing TraceEn;
    FI;
    IF ( there is space available to grow the current ToPA table )
        Add one or more ToPA entries after the last entry in the ToPA table;
        Point new ToPA entry address field(s) to new output region base(s);
    ELSE
        Modify an upcoming ToPA entry in the current table to have END=1;
        IF (output should transition to a new ToPA table )
            Point the address of the "END=1" entry of the current table to the new table base;
        ELSE
            /* Continue to use the current ToPA table, make a circular. */
            Point the address of the "END=1"l entry to the base of the current table;
            Modify the ToPA entry address fields for filled output regions to point to new, unused output regions;
            /* Filled regions are those with index in the range of 0 to (IA32_RTIT_MASK_PTRS.MaskOrTableOffset -1). */
        FI;
      FI;
      Restore saved IA32_RTIT_CTL.value;
FI;
```

ToPA Errors

When a malformed ToPA entry is found, an **operation error** results (see Section 35.3.9). A malformed entry can be any of the following:

1. **ToPA entry reserved bit violation**.
   This describes cases where a bit marked as reserved in Section 35.2.6.2 above is set to 1.
2. **ToPA alignment violation**.
   This includes cases where illegal ToPA entry base address bits are set to 1:
   a. ToPA table base address is not 4KB-aligned. The table base can be from a WRMSR to IA32_RTIT_OUTPUT_BASE, or from a ToPA entry with END=1.
   b. ToPA entry base address is not aligned to the ToPA entry size (e.g., a 2MB region with base address[20:12] not equal to 0).
   c. ToPA entry base address sets upper physical address bits not supported by the processor.
3. **Illegal ToPA Output Offset** (if IA32_RTIT_STATUS.Stopped=0).
   IA32_RTIT_OUTPUT_MASK_PTRS.OutputOffset is greater than or equal to the size of the current ToPA output region size.
4. **ToPA rules violations**.
   These are similar to ToPA entry reserved bit violations; they are cases when a ToPA entry is encountered with illegal field combinations. They include the following:
   a. Setting the STOP or INT bit on an entry with END=1.
   b. Setting the END bit in entry 0 of a ToPA table.
   c. On processors that support only a single ToPA entry (see above), two additional illegal settings apply:
      i) ToPA table entry 1 with END=0.
      ii) ToPA table entry 1 with base address not matching the table base.

In all cases, the error will be logged by setting IA32_RTIT_STATUS.Error, thereby disabling tracing when the problematic ToPA entry is reached (when proc_trace_table_offset points to the entry containing the error). Any packet bytes that are internally buffered when the error is detected may be lost.

Note that operational errors may also be signaled due to attempts to access restricted memory. See Section 35.2.6.4 for details.

A tracing software have a range of flexibility using ToPA to manage the interaction of Intel PT with application buffers, see Section 35.5.

35.2.6.3 Trace Transport Subsystem

When IA32_RTIT_CTL.FabricEn is set, the IA32_RTIT_CTL.ToPA bit is ignored, and trace output is written to the trace transport subsystem. The endpoints of this transport are platform-specific, and details of configuration options should refer to the specific platform documentation. The FabricEn bit is available to be set if CPUID(EAX=14H,ECX=0):EBX[bit 3] = 1.

35.2.6.4 Restricted Memory Access

Packet output cannot be directed to any regions of memory that are restricted by the platform. In particular, all memory accesses on behalf of packet output are checked against the SMRR regions. If there is any overlap with these regions, trace data collection will not function properly. Exact processor behavior is implementation-dependent; Table 35-5 summarizes several scenarios.

Table 35-5. Behavior on Restricted Memory Access

Scenario	Description
ToPA output region overlaps with SMRR	Stores to the restricted memory region will be dropped, and that packet data will be lost. Any attempt to read from that restricted region will return all 1s. The processor also may signal an error (Section 35.3.9) and disable tracing when the output pointer reaches the restricted region. If packet generation remains enabled, then packet output may continue once stores are no longer directed to restricted memory (on wrap, or if the output region is larger than the restricted memory region).
ToPA table overlaps with SMRR	The processor will signal an error (Section 35.3.9) and disable tracing when the ToPA read pointer (IA32_RTIT_OUTPUT_BASE + (proc_trace_table_offset « 3)) enters the restricted region.

It should also be noted that packet output should not be routed to the 4KB APIC MMIO region, as defined by the IA32_APIC_BASE MSR. For details about the APIC, refer to *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*. No error is signaled for this case.

Modifications to Restricted Memory Regions

It is recommended that software disable packet generation before modifying the SMRRs to change the scope of the SMRR regions. This is because the processor reserves the right to cache any number of ToPA table entries internally, after checking them against restricted memory ranges. Once cached, the entries will not be checked again, meaning one could potentially route packet output to a newly restricted region. Software can ensure that any cached entries are written to memory by clearing IA32_RTIT_CTL.TraceEn.

35.2.7 Enabling and Configuration MSRs

35.2.7.1 General Considerations

Trace packet generation is enabled and configured by a collection of model-specific registers (MSRs), which are detailed below. Some notes on the configuration MSR behavior:

- If Intel Processor Trace is not supported by the processor (see Section 35.3.1), RDMSR or WRMSR of the IA32_RTIT_* MSRs will cause #GP.
- A WRMSR to any of these configuration MSRs that begins and ends with IA32_RTIT_CTL.TraceEn set will #GP fault. Packet generation must be disabled before the configuration MSRs can be changed.

Note: Software may write the same value back to IA32_RTIT_CTL without #GP, even if TraceEn=1.

- All configuration MSRs for Intel PT are duplicated per logical processor
- For each configuration MSR, any MSR write that attempts to change bits marked reserved, or utilize encodings marked reserved, will cause a #GP fault.
- All configuration MSRs for Intel PT are cleared on a cold RESET.
  - If CPUID.(EAX=14H, ECX=0):EBX.IPFILT_WRSTPRSV[bit 2] = 1, only the TraceEn bit is cleared on warm RESET; though this may have the impact of clearing other bits in IA32_RTIT_STATUS. Other MSR values of the trace configuration MSRs are preserved on warm RESET.
- The semantics of MSR writes to trace configuration MSRs in this chapter generally apply to explicit WRMSR to these registers, using VM-exit or VM-entry MSR load list to these MSRs, XRSTORS with requested feature bit map including XSAVE map component of state_8 (corresponding to IA32_XSS[bit 8]), and the write to IA32_RTIT_CTL.TraceEn by XSAVES (Section 35.3.5.2).

35.2.7.2 IA32_RTIT_CTL MSR

IA32_RTIT_CTL, at address 570H, is the primary enable and control MSR for trace packet generation. Bit positions are listed in Table 35-6.

Table 35-6. IA32_RTIT_CTL MSR

Position	Bit Name	At Reset	Bit Description
0	TraceEn	0	If 1, enables tracing; else tracing is disabled if 0. When this bit transitions from 1 to 0, all buffered packets are flushed out of internal buffers. A further store, fence, or architecturally serializing instruction may be required to ensure that packet data can be observed at the trace endpoint. See Section 35.2.7.3 for details of enabling and disabling packet generation. Note that the processor will clear this bit on #SMI (Section) and warm reset. Other MSR bits of IA32_RTIT_CTL (and other trace configuration MSRs) are not impacted by these events.
1	CYCEn	0	0: Disables CYC Packet (see Section 35.4.2.14). 1: Enables CYC Packet. This bit is reserved if CPUID.(EAX=14H, ECX=0):EBX.CPSB_CAM[bit 1] = 0.
2	OS	0	0: Packet generation is disabled when CPL = 0. 1: Packet generation may be enabled when CPL = 0.
3	User	0	0: Packet generation is disabled when CPL > 0. 1: Packet generation may be enabled when CPL > 0.
4	PwrEvtEn	0	0: Power Event Trace packets are disabled. 1: Power Event Trace packets are enabled (see Section 35.2.3, "Power Event Tracing").
5	FUPonPTW	0	0: PTW packets are not followed by FUPs. 1: PTW packets are followed by FUPs.
6	FabricEn	0	0: Trace output is directed to the memory subsystem, mechanism depends on IA32_RTIT_CTL.ToPA. 1: Trace output is directed to the trace transport subsystem, IA32_RTIT_CTL.ToPA is ignored. This bit is reserved if CPUID.(EAX=14H, ECX=0):ECX[bit 3] = 0.
7	CR3Filter	0	0: Disables CR3 filtering. 1: Enables CR3 filtering.

Table 35-6. IA32_RTIT_CTL MSR (Contd.)

Position	Bit Name	At Reset	Bit Description
8	ToPA	0	0: Single-range output scheme enabled if CPUID.(EAX=14H, ECX=0):ECX.SNGLRGNOUT[bit 2] = 1 and IA32_RTIT_CTL.FabricEn=0. 1: ToPA output scheme enabled (see Section 35.2.6.2) if CPUID.(EAX=14H, ECX=0):ECX.TOPA[bit 0] = 1, and IA32_RTIT_CTL.FabricEn=0. Note: WRMSR to IA32_RTIT_CTL that sets TraceEn but clears this bit and FabricEn would cause #GP, if CPUID.(EAX=14H, ECX=0):ECX.SNGLRGNOUT[bit 2] = 0. WRMSR to IA32_RTIT_CTL that sets this bit causes #GP, if CPUID.(EAX=14H, ECX=0):ECX.TOPA[bit 0] = 0.
9	MTCEn	0	0: Disables MTC Packet (see Section 35.4.2.16). 1: Enables MTC Packet. This bit is reserved if CPUID.(EAX=14H, ECX=0):EBX.MTC[bit 3] = 0.
10	TSCEn	0	0: Disable TSC packets. 1: Enable TSC packets (see Section 35.4.2.11).
11	DisRETC	0	0: Enable RET compression. 1: Disable RET compression (see Section 35.2.1.2).
12	PTWEn	0	0: PTWRITE packet generation disabled. 1: PTWRITE packet generation enabled (see Table 35-40 "PTW Packet Definition").
13	BranchEn	0	0: Disable COFI-based packets. 1: Enable COFI-based packets: FUP, TIP, TIP.PGE, TIP.PGD, TNT, MODE.Exec, MODE.TSX. see Section 35.2.6 for details on BranchEn.
17:14	MTCFreq	0	Defines MTC packet Frequency, which is based on the core crystal clock, or Always Running Timer (ART). MTC will be sent each time the selected ART bit toggles. The following Encodings are defined: 0: ART(0), 1: ART(1), 2: ART(2), 3: ART(3), 4: ART(4), 5: ART(5), 6: ART(6), 7: ART(7), 8: ART(8), 9: ART(9), 10: ART(10), 11: ART(11), 12: ART(12), 13: ART(13), 14: ART(14), 15: ART(15) Software must use CPUID to query the supported encodings in the processor, see Section 35.3.1. Use of unsupported encodings will result in a #GP fault. This field is reserved if CPUID.(EAX=14H, ECX=0):EBX.MTC[bit 3] = 0.
18	Reserved	0	Must be 0.
22:19	CycThresh	0	CYC packet threshold, see Section 35.3.6 for details. CYC packets will be sent with the first eligible packet after N cycles have passed since the last CYC packet. If CycThresh is 0 then N=0, otherwise N is defined as $2^{(CycThresh-1)}$. The following Encodings are defined: 0: 0, 1: 1, 2: 2, 3: 4, 4: 8, 5: 16, 6: 32, 7: 64, 8: 128, 9: 256, 10: 512, 11: 1024, 12: 2048, 13: 4096, 14: 8192, 15: 16384 Software must use CPUID to query the supported encodings in the processor, see Section 35.3.1. Use of unsupported encodings will result in a #GP fault. This field is reserved if CPUID.(EAX=14H, ECX=0):EBX.CPSB_CAM[bit 1] = 0.
23	Reserved	0	Must be 0.

Table 35-6. IA32_RTIT_CTL MSR (Contd.)

Position	Bit Name	At Reset	Bit Description
27:24	PSBFreq	0	Indicates the frequency of PSB packets. PSB packet frequency is based on the number of Intel PT packet bytes output, so this field allows the user to determine the increment of IA32_IA32_RTIT_STATUS.PacketByteCnt that should cause a PSB to be generated. Note that PSB insertion is not precise, but the average output bytes per PSB should approximate the SW selected period. The following Encodings are defined: 0: 2K, 1: 4K, 2: 8K, 3: 16K, 4: 32K, 5: 64K, 6: 128K, 7: 256K, 8: 512K, 9: 1M, 10: 2M, 11: 4M, 12: 8M, 13: 16M, 14: 32M, 15: 64M Software must use CPUID to query the supported encodings in the processor, see Section 35.3.1. Use of unsupported encodings will result in a #GP fault. This field is reserved if CPUID.(EAX=14H, ECX=0):EBX.CPSB_CAM[bit 1] = 0.
31:28	Reserved	0	Must be 0.
35:32	ADDR0_CFG	0	Configures the base/limit register pair IA32_RTIT_ADDR0_A/B based on the following encodings: 0: ADDR0 range unused. 1: The [IA32_RTIT_ADDR0_A..IA32_RTIT_ADDR0_B] range defines a FilterEn range. FilterEn will only be set when the IP is within this range, though other FilterEn ranges can additionally be used. See Section 35.2.4.3 for details on IP filtering. 2: The [IA32_RTIT_ADDR0_A..IA32_RTIT_ADDR0_B] range defines a TraceStop range. TraceStop will be asserted if code branches into this range. See 4.2.8 for details on TraceStop. 3..15: Reserved (#GP). This field is reserved if CPUID.(EAX=14H, ECX=1):EBX.RANGECNT[2:0] >= 0.
39:36	ADDR1_CFG	0	Configures the base/limit register pair IA32_RTIT_ADDR1_A/B based on the following encodings: 0: ADDR1 range unused. 1: The [IA32_RTIT_ADDR1_A..IA32_RTIT_ADDR1_B] range defines a FilterEn range. FilterEn will only be set when the IP is within this range, though other FilterEn ranges can additionally be used. See Section 35.2.4.3 for details on IP filtering. 2: The [IA32_RTIT_ADDR1_A..IA32_RTIT_ADDR1_B] range defines a TraceStop range. TraceStop will be asserted if code branches into this range. See Section 35.4.2.10 for details on TraceStop. 3..15: Reserved (#GP). This field is reserved if CPUID.(EAX=14H, ECX=1):EBX.RANGECNT[2:0] < 2.
43:40	ADDR2_CFG	0	Configures the base/limit register pair IA32_RTIT_ADDR2_A/B based on the following encodings: 0: ADDR2 range unused. 1: The [IA32_RTIT_ADDR2_A..IA32_RTIT_ADDR2_B] range defines a FilterEn range. FilterEn will only be set when the IP is within this range, though other FilterEn ranges can additionally be used. See Section 35.2.4.3 for details on IP filtering. 2: The [IA32_RTIT_ADDR2_A..IA32_RTIT_ADDR2_B] range defines a TraceStop range. TraceStop will be asserted if code branches into this range. See Section 35.4.2.10 for details on TraceStop. 3..15: Reserved (#GP). This field is reserved if CPUID.(EAX=14H, ECX=1):EBX.RANGECNT[2:0] < 3.

Table 35-6. IA32_RTIT_CTL MSR (Contd.)

Position	Bit Name	At Reset	Bit Description
47:44	ADDR3_CFG	0	Configures the base/limit register pair IA32_RTIT_ADDR3_A/B based on the following encodings: 0: ADDR3 range unused. 1: The [IA32_RTIT_ADDR3_A..IA32_RTIT_ADDR3_B] range defines a FilterEn range. FilterEn will only be set when the IP is within this range, though other FilterEn ranges can additionally be used. See Section 35.2.4.3 for details on IP filtering. 2: The [IA32_RTIT_ADDR3_A..IA32_RTIT_ADDR3_B] range defines a TraceStop range. TraceStop will be asserted if code branches into this range. See Section 35.4.2.10 for details on TraceStop. 3..15: Reserved (#GP). This field is reserved if CPUID.(EAX=14H, ECX=1):EBX.RANGECNT[2:0] < 4.
59:48	Reserved	0	Reserved only for future trace content enables, or address filtering configuration enables. Must be 0.
63:60	Reserved	0	Must be 0.

35.2.7.3 Enabling and Disabling Packet Generation with TraceEn

When TraceEn transitions from 0 to 1, Intel Processor Trace is enabled, and a series of packets may be generated. These packets help ensure that the decoder is aware of the state of the processor when the trace begins, and that it can keep track of any timing or state changes that may have occurred while packet generation was disabled. A full PSB+ (see Section 35.4.2.17) will be generated if IA32_RTIT_STATUS.PacketByteCnt=0, and may be generated in other cases as well. Otherwise, timing packets will be generated, including TSC, TMA, and CBR (see Section 35.4.2).

In addition to the packets discussed above, if and when PacketEn (Section 35.2.5.1) transitions from 0 to 1 (which may happen immediately, depending on filtering settings), a TIP.PGE packet (Section 35.4.2.3) will be generated.

When TraceEn is set, the processor may read ToPA entries from memory and cache them internally. For this reason, software should disable packet generation before making modifications to the ToPA tables (or changing the configuration of restricted memory regions). See Section 35.7 for more details of packets that may be generated with modifications to TraceEn.

Disabling Packet Generation

Clearing TraceEn causes any packet data buffered within the logical processor to be flushed out, after which the output MSRs (IA32_RTIT_OUTPUT_BASE and IA32_RTIT_OUTPUT_MASK_PTRS) will have stable values. When output is directed to memory, a store, fence, or architecturally serializing instruction may be required to ensure that the packet data is globally observed. No special packets are generated by disabling packet generation, though a TIP.PGD may result if PacketEn=1 at the time of disable.

Other Writes to IA32_RTIT_CTL

Any attempt to modify IA32_RTIT_CTL while TraceEn is set will result in a general-protection fault (#GP) unless the same write also clears TraceEn. However, writes to IA32_RTIT_CTL that do not modify any bits will not cause a #GP, even if TraceEn remains set.

35.2.7.4 IA32_RTIT_STATUS MSR

The IA32_RTIT_STATUS MSR is readable and writable by software, but some bits (ContextEn, TriggerEn) are read-only and cannot be directly modified. The WRMSR instruction ignores these bits in the source operand (attempts to modify these bits are ignored and do not cause WRMSR to fault).

This MSR can only be written when IA32_RTIT_CTL.TraceEn is 0; otherwise WRMSR causes a general-protection fault (#GP). The processor does not modify the value of this MSR while TraceEn is 0 (software can modify it with WRMSR).

Table 35-7. IA32_RTIT_STATUS MSR

Position	Bit Name	At Reset	Bit Description
0	FilterEn	0	This bit is written by the processor, and indicates that tracing is allowed for the current IP, see Section 35.2.5.5. Writes are ignored.
1	ContextEn	0	The processor sets this bit to indicate that tracing is allowed for the current context. See Section 35.2.5.3. Writes are ignored.
2	TriggerEn	0	The processor sets this bit to indicate that tracing is enabled. See Section 35.2.5.2. Writes are ignored.
3	Reserved	0	Must be 0.
4	Error	0	The processor sets this bit to indicate that an operational error has been encountered. When this bit is set, TriggerEn is cleared to 0 and packet generation is disabled. For details, see "ToPA Errors" in Section 35.2.6.2. When TraceEn is cleared, software can write this bit. Once it is set, only software can clear it. It is not recommended that software ever set this bit, except in cases where it is restoring a prior saved state.
5	Stopped	0	The processor sets this bit to indicate that a ToPA Stop condition has been encountered. When this bit is set, TriggerEn is cleared to 0 and packet generation is disabled. For details, see "ToPA STOP" in Section 35.2.6.2. When TraceEn is cleared, software can write this bit. Once it is set, only software can clear it. It is not recommended that software ever set this bit, except in cases where it is restoring a prior saved state.
31:6	Reserved	0	Must be 0.
48:32	PacketByteCnt	0	This field is written by the processor, and holds a count of packet bytes that have been sent out. The processor also uses this field to determine when the next PSB packet should be inserted. Note that the processor may clear or modify this field at any time while IA32_RTIT_CTL.TraceEn=1. It will have a stable value when IA32_RTIT_CTL.TraceEn=0. See Section 35.4.2.17 for details.
63:49	Reserved	0	Must be 0.

35.2.7.5 IA32_RTIT_ADDRn_A and IA32_RTIT_ADDRn_B MSRs

The role of the IA32_RTIT_ADDRn_A/B register pairs, for each n, is determined by the corresponding ADDRn_CFG fields in IA32_RTIT_CTL (see Section 35.2.7.2). The number of these register pairs is enumerated by CPUID.(EAX=14H, ECX=1):EAX.RANGECNT[2:0].

- Processors that enumerate support for 1 range support:

  IA32_RTIT_ADDR0_A, IA32_RTIT_ADDR0_B

- Processors that enumerate support for 2 ranges support:

  IA32_RTIT_ADDR0_A, IA32_RTIT_ADDR0_B

  IA32_RTIT_ADDR1_A, IA32_RTIT_ADDR1_B
- Processors that enumerate support for 3 ranges support:

  IA32_RTIT_ADDR0_A, IA32_RTIT_ADDR0_B

  IA32_RTIT_ADDR1_A, IA32_RTIT_ADDR1_B

  IA32_RTIT_ADDR2_A, IA32_RTIT_ADDR2_B
- Processors that enumerate support for 4 ranges support:

  IA32_RTIT_ADDR0_A, IA32_RTIT_ADDR0_B

  IA32_RTIT_ADDR1_A, IA32_RTIT_ADDR1_B

  IA32_RTIT_ADDR2_A, IA32_RTIT_ADDR2_B

  IA32_RTIT_ADDR3_A, IA32_RTIT_ADDR3_B

Each register has a single 64-bit field that holds a linear address value. Writes must ensure that the address is properly sign-extended, otherwise a #GP fault will result.

35.2.7.6 IA32_RTIT_CR3_MATCH MSR

The IA32_RTIT_CR3_MATCH register is compared against CR3 when IA32_RTIT_CTL.CR3Filter is 1. Bits 63:5 hold the CR3 address value to match, bits 4:0 are reserved to 0. For more details on CR3 filtering and the treatment of this register, see Section 35.2.4.2.

This MSR can be written only when IA32_RTIT_CTL.TraceEn is 0; otherwise WRMSR causes a general-protection fault (#GP). IA32_RTIT_CR3_MATCH[4:0] are reserved and must be 0; an attempt to set those bits using WRMSR causes a #GP.

35.2.7.7 IA32_RTIT_OUTPUT_BASE MSR

This MSR is used to configure the trace output destination, when output is directed to memory (IA32_RTIT_CTL.FabricEn = 0). The size of the address field is determined by the maximum physical address width (MAXPHYADDR), as reported by CPUID.80000008H:EAX[7:0].

When the ToPA output scheme is used, the processor may update this MSR when packet generation is enabled, and those updates are asynchronous to instruction execution. Therefore, the values in this MSR should be considered unreliable unless packet generation is disabled (IA32_RTIT_CTL.TraceEn = 0).

Accesses to this MSR are supported only if Intel PT output to memory is supported, hence when either CPUID.(EAX=14H, ECX=0):ECX[bit 0] or CPUID.(EAX=14H, ECX=0):ECX[bit 2] are set. Otherwise WRMSR or RDMSR cause a general-protection fault (#GP). If supported, this MSR can be written only when IA32_RTIT_CTL.TraceEn is 0; otherwise WRMSR causes a general-protection fault (#GP).

Table 35-8. IA32_RTIT_OUTPUT_BASE MSR

Position	Bit Name	At Reset	Bit Description
6:0	Reserved	0	Must be 0.
MAXPHYADDR-1:7	BasePhysAddr	0	The base physical address. How this address is used depends on the value of IA32_RTIT_CTL.ToPA: 0: This is the base physical address of a single, contiguous physical output region. This could be mapped to DRAM or to MMIO, depending on the value. The base address should be aligned with the size of the region, such that none of the 1s in the mask value(Section 35.2.7.8) overlap with 1s in the base address. If the base is not aligned, an operational error will result (see Section 35.3.9). 1: The base physical address of the current ToPA table. The address must be 4K aligned. Writing an address in which bits 11:7 are non-zero will not cause a #GP, but an operational error will be signaled once TraceEn is set. See "ToPA Errors" in Section 35.2.6.2 as well as Section 35.3.9.
63:MAXPHYADDR	Reserved	0	Must be 0.

35.2.7.8 IA32_RTIT_OUTPUT_MASK_PTRS MSR

This MSR holds any mask or pointer values needed to indicate where the next byte of trace output should be written. The meaning of the values held in this MSR depend on whether the ToPA output mechanism is in use. See Section 35.2.6.2 for details.

The processor updates this MSR while when packet generation is enabled, and those updates are asynchronous to instruction execution. Therefore, the values in this MSR should be considered unreliable unless packet generation is disabled (IA32_RTIT_CTL.TraceEn = 0).

Accesses to this MSR are supported only if Intel PT output to memory is supported, hence when either CPUID.(EAX=14H, ECX=0):ECX[bit 0] or CPUID.(EAX=14H, ECX=0):ECX[bit 2] are set. Otherwise WRMSR or RDMSR cause a general-protection fault (#GP). If supported, this MSR can be written only when IA32_RTIT_CTL.TraceEn is 0; otherwise WRMSR causes a general-protection fault (#GP).

Table 35-9. IA32_RTIT_OUTPUT_MASK_PTRS MSR

Position	Bit Name	At Reset	Bit Description
6:0	LowerMask	7FH	Forced to 1, writes are ignored.
31:7	MaskOrTableOffset	0	The use of this field depends on the value of IA32_RTIT_CTL.ToPA: 0: This field holds bits 31:7 of the mask value for the single, contiguous physical output region. The size of this field indicates that regions can be of size 128B up to 4GB. This value (combined with the lower 7 bits, which are reserved to 1) will be ANDed with the OutputOffset field to determine the next write address. All 1s in this field should be consecutive and starting at bit 7, otherwise the region will not be contiguous, and an operational error (Section 35.3.9) will be signaled when TraceEn is set. 1: This field holds bits 27:3 of the offset pointer into the current ToPA table. This value can be added to the IA32_RTIT_OUTPUT_BASE value to produce a pointer to the current ToPA table entry, which itself is a pointer to the current output region. In this scenario, the lower 7 reserved bits are ignored. This field supports tables up to 256 MBytes in size.

Table 35-9. IA32_RTIT_OUTPUT_MASK_PTRS MSR (Contd.)

Position	Bit Name	At Reset	Bit Description
63:32	OutputOffset	0	The use of this field depends on the value of IA32_RTIT_CTL.ToPA: 0: This is bits 31:0 of the offset pointer into the single, contiguous physical output region. This value will be added to the IA32_RTIT_OUTPUT_BASE value to form the physical address at which the next byte of packet output data will be written. This value must be less than or equal to the MaskOrTableOffset field, otherwise an operational error (Section 35.3.9) will be signaled when TraceEn is set. 1: This field holds bits 31:0 of the offset pointer into the current ToPA output region. This value will be added to the output region base field, found in the current ToPA table entry, to form the physical address at which the next byte of trace output data will be written. This value must be less than the ToPA entry size, otherwise an operational error (Section 35.3.9) will be signaled when TraceEn is set.

35.2.8 Interaction of Intel® Processor Trace and Other Processor Features

35.2.8.1 Intel® Transactional Synchronization Extensions (Intel® TSX)

The operation of Intel TSX is described in Chapter 14 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*. For tracing purpose, packet generation does not distinguish between hardware lock elision (HLE) and restricted transactional memory (RTM), but speculative execution does have impacts on the trace output. Specifically, packets are generated as instructions complete, even for instructions in a transactional region that is later aborted. For this reason, debugging software will need indication of the beginning and end of a transactional region; this will allow software to understand when instructions are part of a transactional region and whether that region has been committed.

To enable this, TSX information is included in a MODE packet leaf. The mode bits in the leaf are:

- **InTX**: Set to 1 on an TSX transaction begin, and cleared on transaction commit or abort.
- **TXAbort**: Set to 1 only when InTX transitions from 1 to 0 on an abort. Cleared otherwise.

If BranchEn=1, this MODE packet will be sent each time the transaction status changes. See Table 35-10 for details.

Table 35-10. TSX Packet Scenarios

TSX Event	Instruction	Packets
Transaction Begin	Either XBEGIN or XACQUIRE lock (the latter if executed transactionally)	MODE(TXAbort=0, InTX=1), FUP(CurrentIP)
Transaction Commit	Either XEND or XRELEASE lock, if transactional execution ends. This happens only on the outermost commit	MODE(TXAbort=0, InTX=0), FUP(CurrentIP)
Transaction Abort	XABORT or other transactional abort	MODE(TXAbort=1, InTX=0), FUP(CurrentIP), TIP(TargetIP)
Other	One of the following: ▪ Nested XBEGIN or XACQUIRE lock ▪ An outer XACQUIRE lock that doesn't begin a transaction (InTX not set) ▪ Non-outermost XEND or XRELEASE lock	None. No change to TSX mode bits for these cases.

The CurrentIP listed above is the IP of the associated instruction. The TargetIP is the IP of the next instruction to be executed; for HLE, this is the XACQUIRE lock; for RTM, this is the fallback handler.

Intel PT stores are non-transactional, and thus packet writes are not rolled back on TSX abort.

TSX and IP Filtering

A complication with tracking transactions is handling transactions that start or end outside of the tracing region. Transactions can't span across a change in ContextEn, because CPL changes and CR3 changes each cause aborts. But a transaction can start within the IP filter region and end outside it.

To assist the decoder handling this situation, MODE.TSX packets can be sent even if FilterEn=0, though there will be no FUP attached. Instead, they will merely serve to indicate to the decoder when transactions are active and when they are not. When tracing resumes (due to PacketEn=1), the last MODE.TSX preceding the TIP.PGE will indicate the current transaction status.

System Management Mode (SMM)

SMM code has special privileges that non-SMM code does not have. Intel Processor Trace can be used to trace SMM code, but special care is taken to ensure that SMM handler context is not exposed in any non-SMM trace collection. Additionally, packet output from tracing non-SMM code cannot be written into memory space that is either protected by SMRR or used by the SMM handler.

SMM is entered via a system management interrupt (SMI). SMI delivery saves the value of IA32_RTIT_CTL.TraceEn into SMRAM and then clears it, thereby disabling packet generation.

The saving and clearing of IA32_RTIT_CTL.TraceEn ensures two things:

1. All internally buffered packet data is flushed before entering SMM (see Section 35.2.7.2).
2. Packet generation ceases before entering SMM, so any tracing that was configured outside SMM does not continue into SMM. No SMM instruction pointers or other state will be exposed in the non-SMM trace.

When the RSM instruction is executed to return from SMM, the TraceEn value that was saved by SMI delivery is restored, allowing tracing to be resumed. As is done any time packet generation is enabled, ContextEn is re-evaluated, based on the values of CPL, CR3, etc., established by RSM.

Like other interrupts, delivery of an SMI produces a FUP containing the IP of the next instruction to execute. By toggling TraceEn, SMI and RSM can produce TIP.PGD and TIP.PGE packets, respectively, indicating that tracing was disabled or re-enabled. See Table 35.7 for more information about packets entering and leaving SMM.

Although #SMI and RSM change CR3, PIP packets are not generated in these cases. With #SMI tracing is disabled before the CR3 change; with RSM TraceEn is restored after CR3 is written.

TraceEn must be cleared before executing RSM, otherwise it will cause a shutdown. Further, on processors that restrict use of Intel PT with LBRs (see Section 35.3.1.2), any RSM that results in enabling of both will cause a shutdown.

Intel PT can support tracing of System Transfer Monitor operating in SMM, see Section 35.6.

35.2.8.2 Virtual-Machine Extensions (VMX)

Initial implementations of Intel Processor Trace do not support tracing in VMX operation. Such processors indicate this by returning 0 for IA32_VMX_MISC[bit 14]. On these processors, execution of the VMXON instruction clears IA32_RTIT_CTL.TraceEn and any attempt to write IA32_RTIT_CTL causes a general-protection exception (#GP).

Processors that support Intel Processor Trace in VMX operation return 1 for IA32_VMX_MISC[bit 14]. Details of tracing in VMX operation are described in Section 35.5.

35.2.8.3 Intel Software Guard Extensions (SGX)

SGX provides an application with ability to instantiate a protective container (an enclave) with confidentiality and integrity (see *Intel® Software Guard Extensions Programming Reference*). On a processor with both Intel PT and SGX enabled, when executing code within a production enclave, no control flow packets are produced by Intel PT. Enclave entry will clear ContextEn, thereby blocking control flow packet generation. A TIP.PGD packet will be generated if PacketEn=1 at the time of the entry.

Upon enclave exit, ContextEn will no longer be forced to 0. If other enables are set at the time, a TIP.PGE may be generated to indicate that tracing is resumed.

During the enclave execution, Intel PT remains enabled, and periodic or timing packets such as PSB, TSC, MTC, or CBR can still be generated. No IPs or other architectural state will be exposed.

For packet generation examples on enclave entry or exit, see Section 35.7.

Debug Enclaves

SGX allows an enclave to be configured with relaxed protection of confidentiality for debug purposes, see *Intel® Software Guard Extensions Programming Reference*. In a debug enclave, Intel PT continues to function normally. Specifically, ContextEn is not impacted by enclave entry or exit. Hence the generation of ContextEn-dependent packets within a debug enclave is allowed.

35.2.8.4 SENTER/ENTERACCS and ACM

GETSEC[SENTER] and GETSEC[ENTERACCS] instructions clear TraceEn, and it is not restored when those instruction complete. SENTER also causes TraceEn to be cleared on other logical processors when they rendezvous and enter the SENTER sleep state. In these two cases, the disabling of packet generation is not guaranteed to flush internally buffered packets. Some packets may be dropped.

When executing an authenticated code module (ACM), packet generation is silently disabled during ACRAM setup. TraceEn will be cleared, but no TIP.PGD packet is generated. After completion of the module, the TraceEn value will be restored. There will be no TIP.PGE packet, but timing packets, like TSC and CBR, may be produced.

35.2.8.5 Intel® Memory Protection Extensions (Intel® MPX)

Bounds exceptions (#BR) caused by Intel MPX are treated like other exceptions, producing FUP and TIP packets that indicate the source and destination IPs.

35.3 CONFIGURATION AND PROGRAMMING GUIDELINE

35.3.1 Detection of Intel Processor Trace and Capability Enumeration

Processor support for Intel Processor Trace is indicated by CPUID.(EAX=07H,ECX=0H):EBX[bit 25] = 1. CPUID function 14H is dedicated to enumerate the resource and capability of processors that report CPUID.(EAX=07H,ECX=0H):EBX[bit 25] = 1. Different processor generations may have architecturally-defined variation in capabilities. Table 35-11 describes details of the enumerable capabilities that software must use across generations of processors that support Intel Processor Trace.

Table 35-11. CPUID Leaf 14H Enumeration of Intel Processor Trace Capabilities

CPUID.(EAX=14H,ECX=0)		Name	Description Behavior
Register	Bits		
EAX	31:0	Maximum valid sub-leaf Index	Specifies the index of the maximum valid sub-leaf for this CPUID leaf
EBX	0	CR3 Filtering Support	1: Indicates that IA32_RTIT_CTL.CR3Filter can be set to 1, and that IA32_RTIT_CR3_MATCH MSR can be accessed. See Section 35.2.7. 0: Indicates that writes that set IA32_RTIT_CTL.CR3Filter to 1, or any access to IA32_RTIT_CR3_MATCH, will #GP fault.
	1	Configurable PSB and Cycle-Accurate Mode Supported	1: (a) IA32_RTIT_CTL.PSBFreq can be set to a non-zero value, in order to select the preferred PSB frequency (see below for allowed values). (b) IA32_RTIT_STATUS.PacketByteCnt can be set to a non-zero value, and will be incremented by the processor when tracing to indicate progress towards the next PSB. If trace packet generation is enabled by setting TraceEn, a PSB will only be generated if PacketByteCnt=0. (c) IA32_RTIT_CTL.CYCEn can be set to 1 to enable Cycle-Accurate Mode. See Section 35.2.7. 0: (a) Any attempt to set IA32_RTIT_CTL.PSBFreq, to set IA32_RTIT_CTL.CYCEn, or write a non-zero value to IA32_RTIT_STATUS.PacketByteCnt any access to IA32_RTIT_CR3_MATCH, will #GP fault. (b) If trace packet generation is enabled by setting TraceEn, a PSB is always generated. (c) Any attempt to set IA32_RTIT_CTL.CYCEn will #GP fault.
	2	IP Filtering and TraceStop supported, and Preserve Intel PT MSRs across warm reset	1: (a) IA32_RTIT_CTL provides at one or more ADDRn_CFG field to configure the corresponding address range MSRs for IP Filtering or IP TraceStop. Each ADDRn_CFG field accepts a value in the range of 0:2 inclusive. The number of ADDRn_CFG fields is reported by CPUID.(EAX=14H, ECX=1):EAX.RANGECNT[2:0]. (b) At least one register pair IA32_RTIT_ADDRn_A and IA32_RTIT_ADDRn_B are provided to configure address ranges for IP filtering or IP TraceStop. (c) On warm reset, all Intel PT MSRs will retain their pre-reset values, though IA32_RTIT_CTL.TraceEn will be cleared. The Intel PT MSRs are listed in Section 35.2.7. 0: (a) An Attempt to write IA32_RTIT_CTL.ADDRn_CFG with non-zero encoding values will cause #GP. (b) Any access to IA32_RTIT_ADDRn_A and IA32_RTIT_ADDRn_B, will #GP fault. (c) On warm reset, all Intel PT MSRs will be cleared.
	3	MTC Supported	1: IA32_RTIT_CTL.MTCEn can be set to 1, and MTC packets will be generated. See Section 35.2.7. 0: An attempt to set IA32_RTIT_CTL.MTCEn or IA32_RTIT_CTL.MTCFreq to a non-zero value will #GP fault.
	4	PTWRITE Supported	1: Writes can set IA32_RTIT_CTL[12] (PTWEn) and IA32_RTIT_CTL[5] (FUPonPTW), and PTWRITE can generate packets. 0: Writes that set IA32_RTIT_CTL[12] or IA32_RTIT_CTL[5] will #GP, and PTWRITE will #UD fault.
	5	Power Event Trace Supported	1: Writes can set IA32_RTIT_CTL[4] (PwrEvtEn), enabling Power Event Trace packet generation. 0: Writes that set IA32_RTIT_CTL[4] will #GP.
	31:6	Reserved	

Table 35-11. CPUID Leaf 14H Enumeration of Intel Processor Trace Capabilities (Contd.)

CPUID.(EAX=14H,ECX=0)		Name	Description Behavior
Register	Bits		
ECX	0	ToPA Output Supported	1: Tracing can be enabled with IA32_RTIT_CTL.ToPA = 1, hence utilizing the ToPA output scheme (Section 35.2.6.2) IA32_RTIT_OUTPUT_BASE and IA32_RTIT_OUTPUT_MASK_PTRS MSRs can be accessed. 0: Unless CPUID.(EAX=14H, ECX=0):ECX.SNGLRNGOUT[bit 2] = 1. writes to IA32_RTIT_OUTPUT_BASE or IA32_RTIT_OUTPUT_MASK_PTRS. MSRs will #GP fault.
	1	ToPA Tables Allow Multiple Output Entries	1: ToPA tables can hold any number of output entries, up to the maximum allowed by the MaskOrTableOffset field of IA32_RTIT_OUTPUT_MASK_PTRS. 0: ToPA tables can hold only one output entry, which must be followed by an END=1 entry which points back to the base of the table. Further, ToPA PMIs will be delivered before the region is filled. See ToPA PMI in Section 35.2.6.2. If there is more than one output entry before the END entry, or if the END entry has the wrong base address, an operational error will be signaled (see "ToPA Errors" in Section 35.2.6.2).
	2	Single-Range Output Supported	1: Enabling tracing (TraceEn=1) with IA32_RTIT_CTL.ToPA=0 is supported. 0: Unless CPUID.(EAX=14H, ECX=0):ECX.TOPAOUT[bit 0] = 1. writes to IA32_RTIT_OUTPUT_BASE or IA32_RTIT_OUTPUT_MASK_PTRS. MSRs will #GP fault.
	3	Output to Trace Transport Subsystem Supported	1: Setting IA32_RTIT_CTL.FabricEn to 1 is supported. 0: IA32_RTIT_CTL.FabricEn is reserved. Write 1 to IA32_RTIT_CTL.FabricEn will #GP fault.
	30:4	Reserved	
	31	IP Payloads are LIP	1: Generated packets which contain IP payloads have LIP values, which include the CS base component. 0: Generated packets which contain IP payloads have RIP values, which are the offset from CS base.
EDX	31:0	Reserved	

If CPUID.(EAX=14H, ECX=0):EAX reports a non-zero value, additional capabilities of Intel Processor Trace are described in the sub-leaves of CPUID leaf 14H.

Table 35-12. CPUID Leaf 14H, sub-leaf 1H Enumeration of Intel Processor Trace Capabilities

CPUID.(EAX=14H,ECX=1)		Name	Description Behavior
Register	Bits		
EAX	2:0	Number of Address Ranges	A non-zero value specifies the number ADDRn_CFG field supported in IA32_RTIT_CTL and the number of register pair IA32_RTIT_ADDRn_A/IA32_RTIT_ADDRn_B supported for IP filtering and IP TraceStop. **NOTE:** Currently, no processors support more than 4 address ranges.
	15:3	Reserved	
	31:16	Bitmap of supported MTC Period Encodings	The non-zero bit positions indicate the map of supported encoding values for the IA32_RTIT_CTL.MTCFreq field. This applies only if CPUID.(EAX=14H, ECX=0):EBX.MTC[bit 3] = 1 (MTC Packet generation is supported), otherwise the MTCFreq field is reserved to 0. Each bit position in this field represents 1 encoding value in the 4-bit MTCFreq field (ie, bit 0 is associated with encoding value 0). For each bit: 1: MTCFreq can be assigned the associated encoding value. 0: MTCFreq cannot be assigned to the associated encoding value. A write to IA32_RTIT_CTLMTCFreq with unsupported encoding will cause #GP fault.
EBX	15:0	Bitmap of supported Cycle Threshold values	The non-zero bit positions indicate the map of supported encoding for the IA32_RTIT_CTL.CycThresh field. This applies only if CPUID.(EAX=14H, ECX=0):EBX.CPSB_CAM[bit 1] = 1 (Cycle-Accurate Mode is Supported), otherwise the CycThresh field is reserved to 0. See Section 35.2.7. Each bit position in this field represents 1 encoding value in the 4-bit CycThresh field (ie, bit 0 is associated with encoding value 0). For each bit: 1: CycThresh can be assigned the associated encoding value. 0: CycThresh cannot be assigned to the associated encoding value. A write to CycThresh with unsupported encoding will cause #GP fault.
	31:16	Bitmap of supported Configurable PSB Frequency encoding	The non-zero bit positions indicate the map of supported encoding for the IA32_RTIT_CTL.PSBFreq field. This applies only if CPUID.(EAX=14H, ECX=0):EBX.CPSB_CAM[bit 1] = 1 (Configurable PSB is supported), otherwise the PSBFreq field is reserved to 0. See Section 35.2.7. Each bit position in this field represents 1 encoding value in the 4-bit PSBFreq field (ie, bit 0 is associated with encoding value 0). For each bit: 1: PSBFreq can be assigned the associated encoding value. 0: PSBFreq cannot be assigned to the associated encoding value. A write to PSBFreq with unsupported encoding will cause #GP fault.
ECX	31:0	Reserved	
EDX	31:0	Reserved	

35.3.1.1 Packet Decoding of RIP versus LIP

FUP, TIP, TIP.PGE, and TIP.PGE packets can contain an instruction pointer (IP) payload. On some processor generations, this payload will be an effective address (RIP), while on others this will be a linear address (LIP). In the former case, the payload is the offset from the current CS base address, while in the latter it is the sum of the offset and the CS base address (Note that in real mode, the CS base address is the value of CS<<4, while in protected mode the CS base address is the base linear address of the segment indicated by the CS register.). Which IP type is in use is indicated by enumeration (see CPUID.(EAX=14H, ECX=0):ECX.LIP[bit 31] in Table 35-11).

For software that executes while the CS base address is 0 (including all software executing in 64-bit mode), the difference is indistinguishable. A trace decoder must account for cases where the CS base address is not 0 and the resolved LIP will not be evident in a trace generated on a CPU that enumerates use of RIP. This is likely to cause problems when attempting to link the trace with the associated binaries.

Note that IP comparison logic, for IP filtering and TraceStop range calculation, is based on the same IP type as these IP packets. For processors that output RIP, the IP comparison mechanism is also based on RIP, and hence on those processors RIP values should be written to IA32_RTIT_ADDRn_[AB] MSRs. This can produce differing behavior if the same trace configuration setting is run on processors reporting different IP types, i.e. CPUID.(EAX=14H, ECX=0):ECX.LIP[bit 31]. Care should be taken to check CPUID when configuring IP filters.

35.3.1.2 Model Specific Capability Restrictions

Some processor generations impose restrictions that prevent use of LBRs/BTS/BTM/LERs when software has enabled tracing with Intel Processor Trace. On these processors, when TraceEn is set, updates of LBR, BTS, BTM, LERs are suspended but the states of the corresponding IA32_DEBUGCTL control fields remained unchanged as if it were still enabled. When TraceEn is cleared, the LBR array is reset, and LBR/BTS/BTM/LERs updates will resume. Further, reads of these registers will return 0, and writes will be dropped.

The list of MSRs whose updates/accesses are restricted follows.

- MSR_LASTBRANCH_x_TO_IP, MSR_LASTBRANCH_x_FROM_IP, MSR_LBR_INFO_x, MSR_LASTBRANCH_TOS
- MSR_LER_FROM_IP, MSR_LER_TO_IP
- MSR_LBR_SELECT

For processor with CPUID DisplayFamily_DisplayModel signature of 06_3DH, 06_47H, 06_4EH, 06_4FH, 06_56H and 06_5EH, the use of Intel PT and LBRs are mutually exclusive.

35.3.2 Enabling and Configuration of Trace Packet Generation

To configure trace packets, enable packet generation, and capture packets, software starts with using CPUID instruction to detect its feature flag, CPUID.(EAX=07H,ECX=0H):EBX[bit 25] = 1; followed by enumerating the capabilities described in Section 35.3.1.

Based on the capability queried from Section 35.3.1, software must configure a number of model-specific registers. This section describes programming considerations related to those MSRs.

35.3.2.1 Enabling Packet Generation

When configuring and enabling packet generation, the IA32_RTIT_CTL MSR should be written after any other Intel PT MSRs have been written, since writes to the other configuration MSRs cause a general-protection fault (#GP) if TraceEn = 1. If a prior trace collection context is not being restored, then software should first clear IA32_RTIT_STATUS. This is important since the Stopped, and Error fields are writable; clearing the MSR clears any values that may have persisted from prior trace packet collection contexts. See Section 35.2.7.2 for details of packets generated by setting TraceEn to 1.

If setting TraceEn to 1 causes an operational error (see Section 35.3.9), there may be a delay after the WRMSR completes before the error is signaled in the IA32_RTIT_STATUS MSR.

While packet generation is enabled, the values of some configuration MSRs (e.g., IA32_RTIT_STATUS and IA32_RTIT_OUTPUT_*) are transient, and reads may return values that are out of date. Only after packet generation is disabled (by clearing TraceEn) do reads of these MSRs return reliable values.

35.3.2.2 Disabling Packet Generation

After disabling packet generation by clearing IA32_RTIT_CTL, it is advisable to read the IA32_RTIT_STATUS MSR (Section 35.2.7.4):

- If the Error bit is set, an operational error was encountered, and the trace is most likely compromised. Software should check the source of the error (by examining the output MSR values), correct the source of the problem, and then attempt to gather the trace again. For details on operational errors, see Section 35.3.9. Software should clear IA32_RTIT_STATUS.Error before re-enabling packet generation.
- If the Stopped bit is set, software execution encountered an IP TraceStop (see Section 35.2.4.3) or the ToPA Stop condition (see "ToPA STOP" in Section 35.2.6.2) before packet generation was disabled.

35.3.3 Flushing Trace Output

Packets are first buffered internally and then written out asynchronously. To collect packet output for post-processing, a collector needs first to ensure that all packet data has been flushed from internal buffers. Software can ensure this by stopping packet generation by clearing IA32_RTIT_CTL.TraceEn (see "Disabling Packet Generation" in Section 35.2.7.2).

When software clears IA32_RTIT_CTL.TraceEn to flush out internally buffered packets, the logical processor issues an SFENCE operation which ensures that WC trace output stores will be ordered with respect to the next store, or serializing operation. A subsequent read from the same logical processor will see the flushed trace data, while a read from another logical processor should be preceded by a store, fence, or architecturally serializing operation on the tracing logical processor.

When the flush operations complete, the IA32_RTIT_OUTPUT_* MSR values indicate where the trace ended. While TraceEn is set, these MSRs may hold stale values. Further, if a ToPA region with INT=1 is filled, meaning a ToPA PMI has been triggered, IA32_PERF_GLOBAL_STATUS.Trace_ToPA_PMI[55] will be set by the time the flush completes.

35.3.4 Warm Reset

The MSRs software uses to program Intel Processor Trace are cleared after a power-on RESET (or cold RESET). On a warm RESET, the contents of those MSRs can retain their values from before the warm RESET with the exception that IA32_RTIT_CTL.TraceEn will be cleared (which may have the side effect of clearing some bits in IA32_RTIT_STATUS).

35.3.5 Context Switch Consideration

To facilitate construction of instruction execution traces at the granularity of a software process or thread context, software can save and restore the states of the trace configuration MSRs across the process or thread context switch boundary. The principle is the same as saving and restoring the typical architectural processor states across context switches.

35.3.5.1 Manual Trace Configuration Context Switch

The configuration can be saved and restored through a sequence of instructions of RDMSR, management of MSR content and WRMSR. To stop tracing and to ensure that all configuration MSRs contain stable values, software must clear IA32_RTIT_CTL.TraceEn before reading any other trace configuration MSRs. The recommended method for saving trace configuration context manually follows:

1. RDMSR IA32_RTIT_CTL, save value to memory
2. WRMSR IA32_RTIT_CTL with saved value from RDMSR above and TraceEn cleared
3. RDMSR all other configuration MSRs whose values had changed from previous saved value, save changed values to memory

When restoring the trace configuration context, IA32_RTIT_CTL should be restored last:

1. Read saved configuration MSR values, aside from IA32_RTIT_CTL, from memory, and restore them with WRMSR
2. Read saved IA32_RTIT_CTL value from memory, and restore with WRMSR.

35.3.5.2 Trace Configuration Context Switch Using XSAVES/XRSTORS

On processors whose XSAVE feature set supports XSAVES and XRSTORS, the Trace configuration state can be saved using XSAVES and restored by XRSTORS, in conjunction with the bit field associated with supervisory state component in IA32_XSS. See Chapter 13, "Managing State Using the XSAVE Feature Set" of *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*.

The layout of the trace configuration component state in the XSAVE area is shown in Table 35-13.[1]

Table 35-13. Memory Layout of the Trace Configuration State Component

Offset within Component Area	Field	Offset within Component Area	Field
0H	IA32_RTIT_CTL	08H	IA32_RTIT_OUTPUT_BASE
10H	IA32_RTIT_OUTPUT_MASK_PTRS	18H	IA32_RTIT_STATUS
20H	IA32_RTIT_CR3_MATCH	28H	IA32_RTIT_ADDR0_A
30H	IA32_RTIT_ADDR0_B	38H	IA32_RTIT_ADDR1_A
40H	IA32_RTIT_ADDR1_B	48H-End	Reserved

The IA32_XSS MSR is zero coming out of RESET. Once IA32_XSS[bit 8] is set, system software operating at CPL= 0 can use XSAVES/XRSTORS with the appropriate requested-feature bitmap (RFBM) to manage supervisor state components in the XSAVE map. See Chapter 13, "Managing State Using the XSAVE Feature Set" of *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*.

35.3.6 Cycle-Accurate Mode

Intel PT can be run in a cycle-accurate mode which enables CYC packets (see Section 35.4.2.14) that provide low-level information in the processor core clock domain. This cycle counter data in CYC packets can be used to compute IPC (Instructions Per Cycle), or to track wall-clock time on a fine-grain level.

To enable cycle-accurate mode packet generation, software should set IA32_RTIT_CTL.CYCEn=1. It is recommended that software also set TSCEn=1 anytime cycle-accurate mode is in use. With this, all CYC-eligible packets will be preceded by a CYC packet, the payload of which indicates the number of core clock cycles since the last CYC packet. In cases where multiple CYC-eligible packets are generated in a single cycle, only a single CYC will be generated before the CYC-eligible packets, otherwise each CYC-eligible packet will be preceded by its own CYC. The CYC-eligible packets are:

- TNT, TIP, TIP.PGE, TIP.PGD, MODE.EXEC, MODE.TSX, PIP, VMCS, OVF, MTC, TSC, PTWRITE, EXSTOP

TSC packets are generated when there is insufficient information to reconstruct wall-clock time, due to tracing being disabled (TriggerEn=0), or power down scenarios like a transition to a deep-sleep MWAIT C-state. In this case, the CYC that is generated along with the TSC will indicate the number of cycles actively tracing (those powered up, with TriggerEn=1) executed between the last CYC packet and the TSC packet. And hence the amount of time spent while tracing is inactive can be inferred from the difference in time between that expected based on the CYC value, and the actual time indicated by the TSC.

Additional CYC packets may be sent stand-alone, so that the processor can ensure that the decoder is aware of the number of cycles that have passed before the internal hardware counter wraps, or is reset due to other micro-architectural condition. There is no guarantee at what intervals these standalone CYC packets will be sent, except that they will be sent before the wrap occurs. An illustration is given below.

1. Table 35-13 documents support for the MSRs defining address ranges 0 and 1. Processors that provide XSAVE support for Intel Processor Trace support only those address ranges.

Example 35-1. An Illustrative CYC Packet Example

Time (cycles)	Instruction Snapshot	Generated Packets	Comment
x	call %eax	CYC(?), TIP	?Elapsed cycles from the previous CYC unknown
x + 2	call %ebx	CYC(2), TIP	1 byte CYC packet; 2 cycles elapsed from the previous CYC
x + 8	jnz Foo (not taken)	CYC(6)	1 byte CYC packet
x + 9	ret (compressed)		
x + 12	jnz Bar (taken)		
x + 16	ret (uncompressed)	TNT, CYC(8), TIP	1 byte CYC packet
x + 4111		CYC(4095)	2 byte CYC packet
x + 12305		CYC(8194)	3 byte CYC packet
x + 16332	mov cr3, %ebx	CYC(4027), PIP	2 byte CYC packet

35.3.6.1 Cycle Counter

The cycle counter is implemented in hardware (independent of the time stamp counter or performance monitoring counters), and is a simple incrementing counter that does not saturate, but rather wraps. The size of the counter is implementation specific.

The cycle counter is reset to zero any time that TriggerEn is cleared, and when a CYC packet is sent. The cycle counter will continue to count when ContextEn or FilterEn are cleared, and cycle packets will still be generated. It will not count during sleep states that result in Intel PT logic being powered-down, but will count up to the point where clocks are disabled, and resume counting once they are re-enabled.

35.3.6.2 Cycle Packet Semantics

Cycle-accurate mode adheres to the following protocol:

- All packets that precede a CYC packet represent instructions or events that took place before the CYC time.
- All packets that follow a CYC packet represent instructions or events that took place at the same time as, or after, the CYC time.
- The CYC-eligible packet that immediately follows a CYC packet represents an instruction or event that took place at the same time as the CYC time.

These items above give the decoder a means to apply CYC packets to a specific instruction in the assembly stream. Most packets represent a single instruction or event, and hence the CYC packet that precedes each of those packets represents the retirement time of that instruction or event. In the case of TNT packets, up to 6 conditional branches and/or compressed RETs may be contained in the packet. In this case, the preceding CYC packet provides the retirement time of the first branch in the packet. It is possible that multiple branches retired in the same cycle as that first branch in the TNT, but the protocol will not make that obvious. Also note that a MTC packet could be generated in the same cycle as the first JCC in the TNT packet. In this case, the CYC would precede both the MTC and the TNT, and apply to both.

Note that there are times when the cycle counter will stop counting, though cycle-accurate mode is enabled. After any such scenario, a CYC packet followed by TSC packet will be sent. See Section 35.8.3.2 to understand how to interpret the payload values

Multi-packet Instructions or Events

Some operations, such as interrupts or task switches, generate multiple packets. In these cases, multiple CYC packets may be sent for the operation, preceding each CYC-eligible packet in the operation. An example, using a task switch on a software interrupt, is shown below.

Example 35-2. An Example of CYC in the Presence of Multi-Packet Operations

Time (cycles)	Instruction Snapshot	Generated Packets
x	jnz Foo (not taken)	CYC(?),
x + 2	ret (compressed)	
x + 8	jnz Bar (taken)	
x + 9	jmp %eax	TNT, CYC(9), TIP
x + 12	jnz Bar (not taken)	CYC(3)
x + 32	int3 (task gate)	TNT, FUP, CYC(10), PIP, CYC(20), MODE.Exec, TIP

35.3.6.3 Cycle Thresholds

Software can opt to reduce the frequency of cycle packets, a trade-off to save bandwidth and intrusion at the expense of precision. This is done by utilizing a cycle threshold (see Section 35.2.7.2).

IA32_RTIT_CTL.CycThresh indicates to the processor the minimum number of cycles that must pass before the next CYC packet should be sent. If this value is 0, no threshold is used, and CYC packets can be sent every cycle in which a CYC-eligible packet is generated. If this value is greater than 0, the hardware will wait until the associated number of cycles have passed since the last CYC packet before sending another. CPUID provides the threshold options for CycThresh, see Section 35.3.1.

Note that the cycle threshold does not dictate how frequently a CYC packet will be posted, it merely assigns the maximum frequency. If the cycle threshold is 16, a CYC packet can be posted no more frequently than every 16 cycles. However, once that threshold of 16 cycles has passed, it still requires a new CYC-eligible packet to be generated before a CYC will be inserted. Table 35-14 illustrates the threshold behavior.

Table 35-14. An Illustrative CYC Packet Example

Time (cycles)	Instruction Snapshot	Threshold			
		0	16	32	64
x	jmp %eax	CYC, TIP	CYC, TIP	CYC, TIP	CYC, TIP
x + 9	call %ebx	CYC, TIP	TIP	TIP	TIP
x + 15	call %ecx	CYC, TIP	TIP	TIP	TIP
x + 30	jmp %edx	CYC, TIP	CYC, TIP	TIP	TIP
x + 38	mov cr3, %eax	CYC, PIP	PIP	CYC, PIP	PIP
x + 46	jmp [%eax]	CYC, TIP	CYC, TIP	TIP	TIP
x + 64	call %edx	CYC, TIP	CYC, TIP	TIP	CYC,TIP
x + 71	jmp %edx	CYC, TIP	TIP	CYC,TIP	TIP

35.3.7 Decoder Synchronization (PSB+)

The PSB packet (Section 35.4.2.17) serves as a synchronization point for a trace-packet decoder. It is a pattern in the trace log for which the decoder can quickly scan to align packet boundaries. No legal packet combination can result in such a byte sequence. As such, it serves as the starting point for packet decode. To decode a trace log properly, the decoder needs more than simply to be aligned: it needs to know some state and potentially some timing information as well. The decoder should never need to retain any information (e.g., LastIP, call stack, compound packet event) across a PSB; all compound packet events will be completed before a PSB, and any compression state will be reset.

When a PSB packet is generated, it is followed by a PSBEND packet (Section 35.4.2.18). One or more packets may be generated in between those two packets, and these inform the decoder of the current state of the processor. These packets, known collectively as PSB+, should be interpreted as "status only", since they do not imply any change of state at the time of the PSB, nor are they associated directly with any instruction or event. Thus, the

normal binding and ordering rules that apply to these packets outside of PSB+ can be ignored when these packets are between a PSB and PSBEND. They inform the decoder of the state of the processor at the time of the PSB.

PSB+ can include:

- Timestamp (TSC), if IA32_RTIT_CTL.TSCEn=1.
- Timestamp-MTC Align (TMA), if IA32_RTIT_CTL.TSCEn=1 && IA32_RTIT_CTL.MTCEn=1.
- Paging Info Packet (PIP), if ContextEn=1 and IA32_RTIT_CTL.OS=1. The non-root bit (NR) is set if the logical processor is in VMX non-root operation and the "conceal VMX non-root operation from Intel PT". VM-execution control is 0.
- VMCS packet, if either the logical is in VMX root operation or the logical processor is in VMX non-root operation and the "conceal VMX non-root operation from Intel PT" VM-execution control is 0.
- Core Bus Ratio (CBR).
- MODE.TSX, if ContextEn=1 and BranchEn = 1.
- MODE.Exec, if PacketEn=1.
- Flow Update Packet (FUP), if PacketEn=1.

PSB is generated only when TriggerEn=1; hence PSB+ has the same dependencies. The ordering of packets within PSB+ is not fixed. Timing packets such as CYC and MTC may be generated between PSB and PSBEND, and their meanings are the same as outside PSB+.

A PSB+ can be lost in some scenarios. If IA32_RTIT_STATUS.TriggerEn is cleared just as the PSB threshold is reached, the PSB+ may not be generated. TriggerEn can be cleared by a WRMSR that clears IA32_RTIT_CTL.TraceEn, a VM-exit that clears IA32_RTIT_CTL.TraceEn, an #SMI, or any time that either IA32_RTIT_STATUS.Stopped is set (e.g., by a TraceStop or ToPA stop condition) or IA32_RTIT_STATUS.Error is set (e.g., by an Intel PT output error).

Note that an overflow can occur during PSB+, and this could cause the PSBEND packet to be lost. For this reason, the OVF packet should also be viewed as terminating PSB+.

35.3.8 Internal Buffer Overflow

In the rare circumstances when new packets need to be generated but the processor's dedicated internal buffers are all full, an "internal buffer overflow" occurs. On such an overflow packet generation ceases (as packets would need to enter the processor's internal buffer) until the overflow resolves. Once resolved, packet generation resumes.

When the buffer overflow is cleared, an OVF packet (Section 35.4.2.16) is generated, and the processor ensures that packets which follow the OVF are not compressed (IP compression or RET compression) against packets that were lost.

If IA32_RTIT_CTL.BranchEn = 1, the OVF packet will be followed by a FUP if the overflow resolves while PacketEn=1. If the overflow resolves while PacketEn = 0 no packet is generated, but a TIP.PGE will naturally be generated later, once PacketEn = 1. The payload of the FUP or TIP.PGE will be the Current IP of the first instruction upon which tracing resumes after the overflow is cleared. If the overflow resolves while PacketEn=1, only timing packets may come between the OVF and the FUP. If the overflow resolves while PacketEn=0, any other packets that are not dependent on PacketEn may come between the OVF and the TIP.PGE.

35.3.8.1 Overflow Impact on Enables

The address comparisons to ADDRn ranges, for IP filtering and TraceStop (Section 35.2.4.3), continue during a buffer overflow, and TriggerEn, ContextEn, and FilterEn may change during a buffer overflow. Like other packets, however, any TIP.PGE or TIP.PGD packets that would have been generated will be lost. Further, IA32_RTIT_STATUS.PacketByteCnt will not increment, since it is only incremented when packets are generated.

If a TraceStop event occurs during the buffer overflow, IA32_RTIT_STATUS.Stopped will still be set, tracing will cease as a result. However, the TraceStop packet, and any TIP.PGD that result from the TraceStop, may be dropped.

35.3.8.2 Overflow Impact on Timing Packets

Any timing packets that are generated during a buffer overflow will be dropped. If only a few MTC packets are dropped, a decoder should be able to detect this by noticing that the time value in the first MTC packet after the buffer overflow incremented by more than one. If the buffer overflow lasted long enough that 256 MTC packets are lost (and thus the MTC packet 'wraps' its 8-bit CTC value), then the decoder may be unable to properly understand the trace. This is not an expected scenario. No CYC packets are generated during overflow, even if the cycle counter wraps.

Note that, if cycle-accurate mode is enabled, the OVF packet will generate a CYC packet. Because the cycle counter counts during overflows, this CYC packet can provide the duration of the overflow. However, there is a risk that the cycle counter wrapped during the overflow, which could render this CYC misleading.

35.3.9 Operational Errors

Errors are detected as a result of packet output configuration problems, which can include output alignment issues, ToPA reserved bit violations, or overlapping packet output with restricted memory. See "ToPA Errors" in Section 35.2.6.2 for details on ToPA errors, and Section 35.2.6.4 for details on restricted memory errors. Operational errors are only detected and signaled when TraceEn=1.

When an operational error is detected, tracing is disabled and the error is logged. Specifically, IA32_RTIT_STATUS.Error is set, which will cause IA32_RTIT_STATUS.TriggerEn to be 0. This will disable generation of all packets. Some causes of operational errors may lead to packet bytes being dropped.

It should be noted that the timing of error detection may not be predictable. Errors are signaled when the processor encounters the problematic configuration. This could be as soon as packet generation is enabled but could also be later when the problematic entry or field needs to be used.

Once an error is signaled, software should disable packet generation by clearing TraceEn, diagnose and fix the error condition, and clear IA32_RTIT_STATUS.Error. At this point, packet generation can be re-enabled.

35.4 TRACE PACKETS AND DATA TYPES

This section details the data packets generated by Intel Processor Trace. It is useful for developers writing the interpretation code that will decode the data packets and apply it to the traced source code.

35.4.1 Packet Relationships and Ordering

This section introduces the concept of packet "binding", which involves determining the IP in a binary disassembly at which the change indicated by a given packet applies. Some packets have the associated IP as the payload (FUP, TIP), while for others the decoder need only search for the next instance of a particular instruction (or instructions) to bind the packet (TNT). However, in many cases, the decoder will need to consider the relationship between packets, and to use this packet context to determine how to bind the packet.

Section 35.4.2 below provides detailed descriptions of the packets, including how packets bind to IPs in the disassembly, to other packets, or to nothing at all. Many packets listed are simple to bind, because they are generated in only a few scenarios. Those that require more consideration are typically part of "compound packet events", such as interrupts, exceptions, and some instructions, where multiple packets are generated by a single operation (instruction or event). These compound packet events frequently begin with a FUP to indicate the source address (if it is not clear from the disassembly), and are concluded by a TIP or TIP.PGD packet that indicates the destination address (if one is provided). In this scenario, the FUP is said to be "coupled" with the TIP packet.

Other packets could be in between the coupled FUP and TIP packet. Timing packets, such as TSC, MTC, CYC, or CBR, could arrive at any time, and hence could intercede in a compound packet event. If an operation changes CR3 or the processor's mode of execution, a state update packet (i.e., PIP or MODE) is generated. The state changes indicated by these intermediate packets should be applied at the IP of the TIP* packet. A summary of compound packet events is provided in Table 35-15; see Section 35.4.2 for more per-packet details and Section 35.7 for more detailed packet generation examples.

Table 35-15. Compound Packet Event Summary

Event Type	Beginning	Middle	End	Comment
Unconditional, uncompressed control-flow transfer	FUP or none	Any combination of PIP, VMCS, MODE.Exec, or none	TIP or TIP.PGD	FUP only for asynchronous events. Order of middle packets may vary. PIP/VMCS/MODE only if the operation modifies the state tracked by these respective packets.
TSX Update	MODE.TSX, and (FUP or none)	None	TIP, TIP.PGD, or none	FUP TIP/TIP.PGD only for TSX abort cases.
Overflow	OVF	PSB, PSBEND, or none	FUP or TIP.PGE	FUP if overflow resolves while ContextEn=1, else TIP.PGE.

35.4.2 Packet Definitions

The following description of packet definitions are in tabular format. Figure 35-3 explains how to interpret them. Packet bits listed as "RSVD" are not guaranteed to be 0.

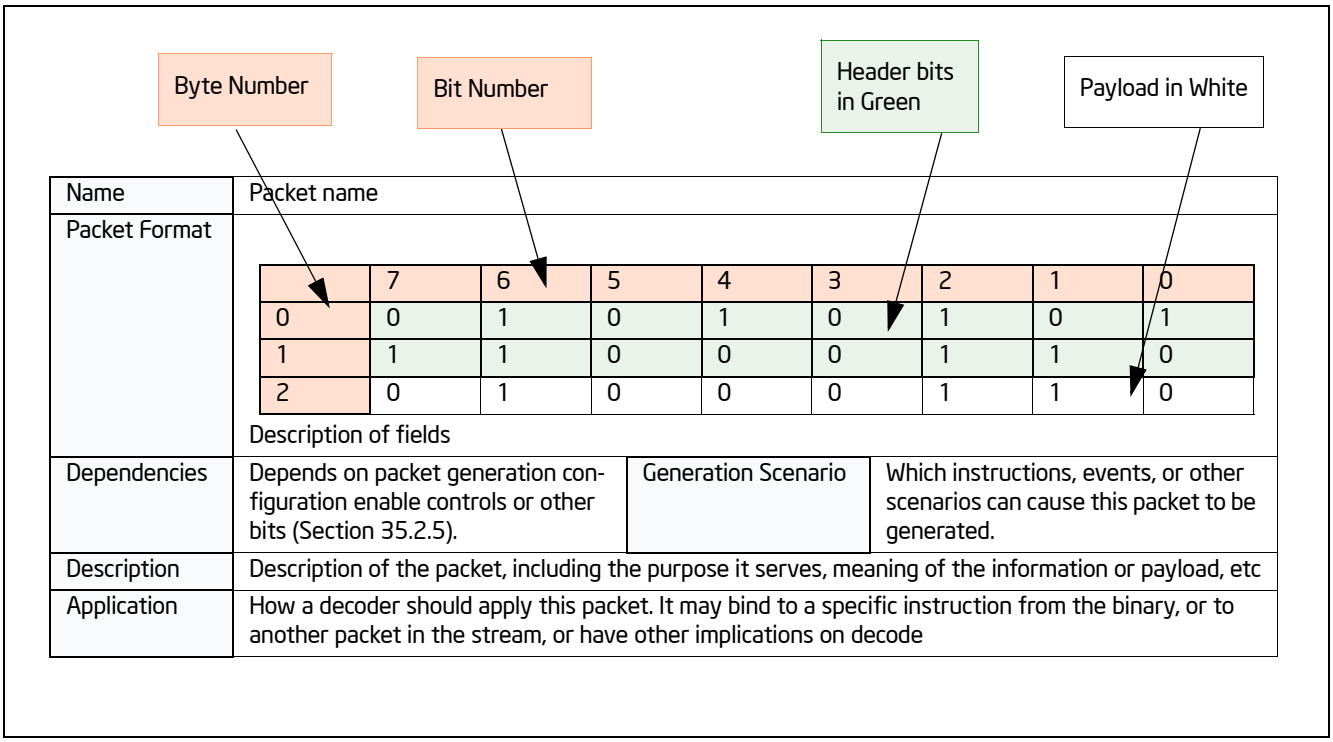


Figure 35-3. Interpreting Tabular Definition of Packet Format

35.4.2.1 Taken/Not-taken (TNT) Packet

Table 35-16. TNT Packet Definition

Name	Taken/Not-taken (TNT) Packet								
Packet Format									
	7	6	5	4	3	2	1	0	
0	1	B_1	B_2	B_3	B_4	B_5	B_6	0	Short TNT

Table 35-16. TNT Packet Definition (Contd.)

<table>
<tr><td></td><td>B1...BN represent the last N conditional branch or compressed RET (Section 35.4.2.2) results, such that B1 is oldest and BN is youngest. The short TNT packet can contain from 1 to 6 TNT bits. The long TNT packet can contain up from 1 to 47 TNT bits.</td></tr>
<tr><td></td><td>
<table>
<tr><td></td><td>7</td><td>6</td><td>5</td><td>4</td><td>3</td><td>2</td><td>1</td><td>0</td><td></td></tr>
<tr><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>1</td><td>0</td><td>Long TNT</td></tr>
<tr><td>1</td><td>1</td><td>0</td><td>1</td><td>0</td><td>0</td><td>0</td><td>1</td><td>1</td><td></td></tr>
<tr><td>2</td><td>B_{40}</td><td>B_{41}</td><td>B_{42}</td><td>B_{43}</td><td>B_{44}</td><td>B_{45}</td><td>B_{46}</td><td>B_{47}</td><td></td></tr>
<tr><td>3</td><td>B_{32}</td><td>B_{33}</td><td>B_{34}</td><td>B_{35}</td><td>B_{36}</td><td>B_{37}</td><td>B_{38}</td><td>B_{39}</td><td></td></tr>
<tr><td>4</td><td>B_{24}</td><td>B_{25}</td><td>B_{26}</td><td>B_{27}</td><td>B_{28}</td><td>B_{29}</td><td>B_{30}</td><td>B_{31}</td><td></td></tr>
<tr><td>5</td><td>B_{16}</td><td>B_{17}</td><td>B_{18}</td><td>B_{19}</td><td>B_{20}</td><td>B_{21}</td><td>B_{22}</td><td>B_{23}</td><td></td></tr>
<tr><td>6</td><td>B_{8}</td><td>B_{9}</td><td>B_{10}</td><td>B_{11}</td><td>B_{12}</td><td>B_{13}</td><td>B_{14}</td><td>B_{15}</td><td></td></tr>
<tr><td>7</td><td>1</td><td>B_{1}</td><td>B_{2}</td><td>B_{3}</td><td>B_{4}</td><td>B_{5}</td><td>B_{6}</td><td>B_{7}</td><td></td></tr>
</table>
</td></tr>
<tr><td></td><td>Irrespective of how many TNT bits is in a packet, the last valid TNT bit is followed by a trailing 1, or Stop bit, as shown above. If the TNT packet is not full (fewer than 6 TNT bits for the Short TNT, or fewer than 47 TNT bits for the Long TNT), the Stop bit moves up, and the trailing bits of the packet are filled with 0s. Examples of these “partial TNTs” are shown below.
<table>
<tr><td></td><td>7</td><td>6</td><td>5</td><td>4</td><td>3</td><td>2</td><td>1</td><td>0</td><td></td></tr>
<tr><td>0</td><td>0</td><td>0</td><td>1</td><td>B_{1}</td><td>B_{2}</td><td>B_{3}</td><td>B_{4}</td><td>0</td><td>Short TNT</td></tr>
</table>
</td></tr>
<tr><td></td><td>
<table>
<tr><td></td><td>7</td><td>6</td><td>5</td><td>4</td><td>3</td><td>2</td><td>1</td><td>0</td><td></td></tr>
<tr><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>1</td><td>0</td><td>Long TNT</td></tr>
<tr><td>1</td><td>1</td><td>0</td><td>1</td><td>0</td><td>0</td><td>0</td><td>1</td><td>1</td><td></td></tr>
<tr><td>2</td><td>B_{24}</td><td>B_{25}</td><td>B_{26}</td><td>B_{27}</td><td>B_{28}</td><td>B_{29}</td><td>B_{30}</td><td>B_{31}</td><td></td></tr>
<tr><td>3</td><td>B_{16}</td><td>B_{17}</td><td>B_{18}</td><td>B_{19}</td><td>B_{20}</td><td>B_{21}</td><td>B_{22}</td><td>B_{23}</td><td></td></tr>
<tr><td>4</td><td>B_{8}</td><td>B_{9}</td><td>B_{10}</td><td>B_{11}</td><td>B_{12}</td><td>B_{13}</td><td>B_{14}</td><td>B_{15}</td><td></td></tr>
<tr><td>5</td><td>1</td><td>B_{1}</td><td>B_{2}</td><td>B_{3}</td><td>B_{4}</td><td>B_{5}</td><td>B_{6}</td><td>B_{7}</td><td></td></tr>
<tr><td>6</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td></td></tr>
<tr><td>7</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td></td></tr>
</table>
</td></tr>
<tr><td>Dependencies</td><td>PacketEn</td><td>Generation Scenario</td><td>On a conditional branch or compressed RET, if it fills the TNT. Also, partial TNTs may be generated at any time, as a result of other packets being generated,
or certain micro-architectural conditions occurring, before the TNT is full.</td></tr>
<tr><td>Description</td><td colspan="3">Provides the taken/not-taken results for the last 1–N conditional branches (Jcc, J*CXZ, or LOOP) or compressed RETs (Section 35.4.2.2). The TNT payload bits should be interpreted as follows:
▪ 1 indicates a taken conditional branch, or a compressed RET
▪ 0 indicates a not-taken conditional branch</td></tr>
<tr><td>Application</td><td colspan="3">Each valid payload bit (that is, bits between the header bits and the trailing Stop bit) applies to an upcoming conditional branch or RET instruction. Once a decoder consumes a TNT packet with N valid payload bits, these bits should be applied to (and hence provide the destination for) the next N conditional branches or RETs</td></tr>
</table>

35.4.2.2 Target IP (TIP) Packet

Table 35-17. IP Packet Definition

<table>
<tr><td>Name</td><td colspan="4">Target IP (TIP) Packet</td></tr>
<tr><td>Packet Format</td><td colspan="4">
<table>
<tr><th></th><th>7</th><th>6</th><th>5</th><th>4</th><th>3</th><th>2</th><th>1</th><th>0</th></tr>
<tr><td>0</td><td colspan="3">IPBytes</td><td>0</td><td>1</td><td>1</td><td>0</td><td>1</td></tr>
<tr><td>1</td><td colspan="8">TargetIP[7:0]</td></tr>
<tr><td>2</td><td colspan="8">TargetIP[15:8]</td></tr>
<tr><td>3</td><td colspan="8">TargetIP[23:16]</td></tr>
<tr><td>4</td><td colspan="8">TargetIP[31:24]</td></tr>
<tr><td>5</td><td colspan="8">TargetIP[39:32]</td></tr>
<tr><td>6</td><td colspan="8">TargetIP[47:40]</td></tr>
<tr><td>7</td><td colspan="8">TargetIP[55:48]</td></tr>
<tr><td>8</td><td colspan="8">TargetIP[63:56]</td></tr>
</table>
</td></tr>
<tr><td>Dependencies</td><td>PacketEn</td><td>Generation Scenario</td><td colspan="2">Indirect branch (including un-compressed RET), far branch, interrupt, exception, INIT, SIPI, (VM exit, VM entry,)[1] TSX abort, (EENTER, EEXIT, ERESUME, AEX)[2].</td></tr>
<tr><td>Description</td><td colspan="4">Provides the target for some control flow transfers</td></tr>
<tr><td>Application</td><td colspan="4">Anytime a TIP is encountered, it indicates that control was transferred to the IP provided in the payload.</td></tr>
<tr><td></td><td colspan="4">The source of this control flow change, and hence the IP or instruction to which it binds, depends on the packets that precede the TIP. If a TIP is encountered and all preceding packets have already been bound, then the TIP will apply to the upcoming indirect branch, far branch, or VMRESUME. However, if there was a preceding FUP that remains unbound, it will bind to the TIP. Here, the TIP provides the target of an asynchronous event or TSX abort that occurred at the IP given in the FUP payload. Note that there may be other packets, in addition to the FUP, which will bind to the TIP packet. See the packet application descriptions for other packets for details.</td></tr>
</table>

NOTES:

1. If IA32_VMX_MISC[bit 14] reports 1.
2. In a debug enclave.

IP Compression

The IP payload in a TIP. FUP, TIP.PGE, or TIP.PGD packet can vary in size, based on the mode of execution, and the use of IP compression. IP compression is an optional compression technique the processor may choose to employ to reduce bandwidth. With IP compression, the IP to be represented in the payload is compared with the last IP sent out, via any of FUP, TIP, TIP.PGE, or TIP.PGD. If that previous IP had the same upper (most significant) address bytes, those matching bytes may be suppressed in the current packet. The processor maintains an internal state of the "Last IP" that was encoded in trace packets, thus the decoder will need to keep track of the "Last IP" state in software, to match fidelity with packets generated by hardware. "Last IP" is initialized to zero, hence if the first IP in the trace may be compressed if the upper bytes are zeroes.

The "IPBytes" field of the IP packets (FUP, TIP, TIP.PGE, TIP.PGD) serves to indicate how many bytes of payload are provided, and how the decoder should fill in any suppressed bytes. The algorithm for reconstructing the IP for a TIP/FUP packet is shown in the table below.

Table 35-18. FUP/TIP IP Reconstruction

IPBytes	Uncompressed IP Value							
	63:56	55:48	47:40	39:32	31:24	23:16	15:8	7:0
000b	None, IP is out of context							
001b	Last IP[63:16]						IP Payload[15:0]	
010b	Last IP[63:32]				IP Payload[31:0]			
011b	IP Payload[47] extended		IP Payload[47:0]					
100b	Last IP [63:48]		IP Payload[47:0]					
101b	Reserved							
110b	IP Payload[63:0]							
111b	Reserved							

The processor-internal Last IP state is guaranteed to be reset to zero when a PSB is sent out. This means that the IP that follows the PSB with either be un-compressed (011b or 110b, see Table 35-18), or compressed against zero.

At times, "IPbytes" will have a value of 0. As shown above, this does not mean that the IP payload matches the full address of the last IP, but rather that the IP for this packet was suppressed. This is used for cases where the IP that applies to the packet is out of context. An example is the TIP.PGD sent on a SYSCALL, when tracing only USR code. In that case, no TargetIP will be included in the packet, since that would expose an instruction point at CPL = 0. When the IP payload is suppressed in this manner, Last IP is not cleared, and instead refers to the last IP packet with a non-zero IPBytes field.

On processors that support a maximum linear address size of 32 bits, IP payloads may never exceed 32 bits (IPBytes <= 010b).

Indirect Transfer Compression for Returns (RET)

In addition to IP compression, TIP packets for near return (RET) instructions can also be compressed. If the RET target matches the next IP of the corresponding CALL, then the TIP packet is unneeded, since the decoder can deduce the target IP by maintaining a CALL/RET stack of its own.

A CALL/RET stack can be maintained by the decoder by doing the following:

1. Allocate space to store 64 RET targets.
2. For near CALLs, push the Next IP onto the stack. Once the stack is full, new CALLs will force the oldest entry off the end of the stack, such that only the youngest 64 entries are stored. Note that this excludes zero-length CALLs, which are direct near CALLs with displacement zero (to the next IP). These CALLs typically don't have matching RETs.
3. For near RETs, pop the top (youngest) entry off the stack. This will be the target of the RET.

In cases where the RET is compressed, the target is guaranteed to match the value produced in 2) above. If the target is not compressed, a TIP packet will be generated with the RET target, which may differ from 2).

The hardware ensure that packets read by the decoder will always have seen the CALL that corresponds to any compressed RET. The processor will never compress a RET across a PSB, a buffer overflow, or scenario where PacketEn=0. This means that a RET whose corresponding CALL executed while PacketEn=0, or before the last PSB, etc., will not be compressed.

If the CALL/RET stack is manipulated or corrupted by software, and thereby causes a RET to transfer control to a target that is inconsistent with the CALL/RET stack, then the RET will not be compressed, and will produce a TIP packet. This can happen, for example, if software executes a PUSH instruction to push a target onto the stack, and a later RET uses this target.

When a RET is compressed, a Taken indication is added to the TNT buffer. Because it sends no TIP packet, it also does not update the internal Last IP value, and thus the decoder should treat it the same way. If the RET is not compressed, it will generate a TIP packet (just like when RET compression is disabled, via IA32_RTIT_CTL.DisRETC). For processors that employ deferred TIPs (Section 35.4.2.3), an uncompressed RET will not be deferred, and hence will force out any accumulated TNTs or TIPs. This serves to avoid ambiguity, and make

clear to the decoder whether the near RET was compressed, and hence a bit in the in-progress TNT should be consumed, or uncompressed, in which case there will be no in-progress TNT and thus a TIP should be consumed.

Note that in the unlikely case that a RET executes in a different execution mode than the associated CALL, the decoder will need to model the same behavior with its CALL stack. For instance, if a CALL executes in 64-bit mode, a 64-bit IP value will be pushed onto the software stack. If the corresponding RET executes in 32-bit mode, then only the lower 32 target bits will be popped off of the stack, which may mean that the RET does not go to the CALL's Next IP. This is architecturally correct behavior, and this RET could be compressed, thus the decoder should match this behavior

35.4.2.3 Deferred TIPs

The processor may opt to defer sending out the TNT when TIPs are generated. Thus, rather than sending a partial TNT followed by a TIP, both packets will be deferred while the TNT accumulates more Jcc/RET results. Any number of TIP packets may be accumulated this way, such that only once the TNT is filled, or once another packet (e.g., FUP) is generated, the TNT will be sent, followed by all the deferred TIP packets, and finally terminated by the other packet(s) that forced out the TNT and TIP packets. Generation of many other packets (see list below) will force out the TNT and any accumulated TIP packets. This is an optional optimization in hardware to reduce the bandwidth consumption, and hence the performance impact, incurred by tracing.

Table 35-19. TNT Examples with Deferred TIPs

Code Flow	Packets, Non-Deferred TIPS	Packets, Deferred TIPS
0x1000 cmp %rcx, 0 0x1004 jnz Foo // not-taken 0x1008 jmp %rdx	TNT(0b0), TIP(0x1308)	
0x1308 cmp %rcx, 1 0x130c jnz Bar // not-taken 0x1310 cmp %rcx, 2 0x1314 jnz Baz // taken 0x1500 cmp %eax, 7 0x1504 jg Exit // not-taken 0x1508 jmp %r15	TNT(0b010), TIP(0x1100)	
0x1100 cmp %rbx, 1 0x1104 jg Start // not-taken 0x1108 add %rcx, %eax 0x110c ... // **an asynchronous Interrupt arrives** INThandler: 0xcc00 pop %rdx	TNT(0b0), FUP(0x110c), TIP(0xcc00)	TNT(0b00100), TIP(0x1308), TIP(0x1100), FUP(0x110c), TIP(0xcc00)

35.4.2.4 Packet Generation Enable (TIP.PGE)

Table 35-20. TIP.PGE Packet Definition

<table>
<tr><td>Name</td><td colspan="4">Target IP - Packet Generation Enable (TIP.PGE)</td></tr>
<tr><td>Packet Format</td><td colspan="4">
<table>
<tr><th></th><th>7</th><th>6</th><th>5</th><th>4</th><th>3</th><th>2</th><th>1</th><th>0</th></tr>
<tr><td>0</td><td colspan="3">IPBytes</td><td>1</td><td>0</td><td>0</td><td>0</td><td>1</td></tr>
<tr><td>1</td><td colspan="8">TargetIP[7:0]</td></tr>
<tr><td>2</td><td colspan="8">TargetIP[15:8]</td></tr>
<tr><td>3</td><td colspan="8">TargetIP[23:16]</td></tr>
<tr><td>4</td><td colspan="8">TargetIP[31:24]</td></tr>
<tr><td>5</td><td colspan="8">TargetIP[39:32]</td></tr>
<tr><td>6</td><td colspan="8">TargetIP[47:40]</td></tr>
<tr><td>7</td><td colspan="8">TargetIP[55:48]</td></tr>
<tr><td>8</td><td colspan="8">TargetIP[63:56]</td></tr>
</table>
</td></tr>
<tr><td>Dependencies</td><td>PacketEn transitions to 1</td><td>Generation Scenario</td><td colspan="2">Any branch instruction, control flow transfer, or MOV CR3 that sets PacketEn, a WRMSR that enables packet generation and sets PacketEn</td></tr>
<tr><td>Description</td><td colspan="4">Indicates that PacketEn has transitioned to 1. It provides the IP at which the tracing begins.
This can occur due to any of the enables that comprise PacketEn transitioning from 0 to 1, as long as all the others are asserted. Examples:
▪ TriggerEn: This is set on software write to set IA32_RTIT_CTL.TraceEn as long as the Stopped and Error bits in IA32_RTIT_STATUS are clear. The IP payload will be the Next IP of the WRMSR.
▪ FilterEn: This is set when software jumps into the tracing region. This region is defined by enabling IP filtering in IA32_RTIT_CTL.ADDRn_CFG, and defining the range in IA32_RTIT_ADDRn_[AB], see. Section 35.2.4.3. The IP payload will be the target of the branch.
▪ ContextEn: This is set on a CPL change, a CR3 write or any other means of changing ContextEn. The IP payload will be the Next IP of the instruction that changes context if it is not a branch, otherwise it will be the target of the branch.</td></tr>
<tr><td>Application</td><td colspan="4">TIP.PGE packets bind to the instruction at the IP given in the payload.</td></tr>
</table>

35.4.2.5 Packet Generation Disable (TIP.PGD)

Table 35-21. TIP.PGD Packet Definition

<table>
<tr><td>Name</td><td colspan="4">Target IP - Packet Generation Disable (TIP.PGD)</td></tr>
<tr><td>Packet Format</td><td colspan="4">
<table>
<tr><th></th><th>7</th><th>6</th><th>5</th><th>4</th><th>3</th><th>2</th><th>1</th><th>0</th></tr>
<tr><td>0</td><td colspan="3">IPBytes</td><td>0</td><td>0</td><td>0</td><td>0</td><td>1</td></tr>
<tr><td>1</td><td colspan="8">TargetIP[7:0]</td></tr>
<tr><td>2</td><td colspan="8">TargetIP[15:8]</td></tr>
<tr><td>3</td><td colspan="8">TargetIP[23:16]</td></tr>
<tr><td>4</td><td colspan="8">TargetIP[31:24]</td></tr>
<tr><td>5</td><td colspan="8">TargetIP[39:32]</td></tr>
<tr><td>6</td><td colspan="8">TargetIP[47:40]</td></tr>
<tr><td>7</td><td colspan="8">TargetIP[55:48]</td></tr>
<tr><td>8</td><td colspan="8">TargetIP[63:56]</td></tr>
</table>
</td></tr>
<tr><td>Dependencies</td><td>PacketEn transitions to 0</td><td>Generation Scenario</td><td colspan="2">Any branch instruction, control flow transfer, or MOV CR3 that clears PacketEn, a WRMSR that disables packet generation and clears PacketEn</td></tr>
<tr><td>Description</td><td colspan="4">Indicates that PacketEn has transitioned to 0. It will include the IP at which the tracing ends, unless ContextEn= 0 or TraceEn=0 at the conclusion of the instruction or event that cleared PacketEn.
PacketEn can be cleared due to any of the enables that comprise PacketEn transitioning from 1 to 0. Examples:
▪ TriggerEn: This is cleared on software write to clear IA32_RTIT_CTL.TraceEn, or when IA32_RTIT_STATUS.Stopped is set, or on operational error. The IP payload will be suppressed in this case, and the “IPBytes” field will have the value 0.
▪ FilterEn: This is set when software jumps out of the tracing region. This region is defined by enabling IP filtering in IA32_RTIT_CTL.ADDRn_CFG, and defining the range in IA32_RTIT_ADDRn_[AB], see. Section 35.2.4.3. The IP payload will depend on the type of the branch. For conditional branches, the payload is suppressed (IPBytes = 0), and in this case the destination can be inferred from the disassembly. For any other type of branch, the IP payload will be the target of the branch.
▪ ContextEn: This can happen on a CPL change, a CR3 write or any other means of changing ContextEn. See Section 35.2.4.3 for details. In this case, when ContextEn is cleared, there will be no IP payload. The “IPBytes” field will have value 0.
Note that, in cases where a branch that would normally produce a TIP packet (i.e., far transfer, indirect branch, interrupt, etc) or TNT update (conditional branch or compressed RT) causes PacketEn to transition from 1 to 0, the TIP or TNI bit will be replaced with TIP.PGD. The payload of the TIP.PGD will be the target of the branch, unless the result of the instruction causes TraceEn or ContextEn to be cleared (ie, SYSCALL when IA32_RTIT_CTL.OS=0, In the case where a conditional branch clears FilterEn and hence PacketEn, there will be no TNT bit for this branch, replaced instead by the TIP.PGD.</td></tr>
<tr><td>Application</td><td colspan="4">TIP.PGD can be produced by any branch instructions, as well as some non-branch instructions, that clear PacketEn. When produced by a branch, it replaces any TIP or TNT update that the branch would normally produce.
In cases where there is an unbound FUP preceding the TIP.PGD, then the TIP.PGD is part of compound operation (i.e., asynchronous event or TSX abort) which cleared PacketEn. For most such cases, the TIP.PGD is simply replacing a TIP, and should be treated the same way. The TIP.PGD may or may not have an IP payload, depending on whether the operation cleared ContextEn.
If there is not an associated FUP, the binding will depend on whether there is an IP payload. If there is an IP payload, then the TIP.PGD should be applied to either the next direct branch whose target matches the TIP.PGD payload, or the next branch that would normally generate a TIP or TNT packet. If there is no IP payload, then the TIP.PGD should apply to the next branch or MOV CR3 instruction.</td></tr>
</table>

35.4.2.6 Flow Update (FUP) Packet

Table 35-22. FUP Packet Definition

<table>
<tr><td>Name</td><td colspan="4">Flow Update (FUP) Packet</td></tr>
<tr><td>Packet Format</td><td colspan="4">
<table>
<tr><th></th><th>7</th><th>6</th><th>5</th><th>4</th><th>3</th><th>2</th><th>1</th><th>0</th></tr>
<tr><td>0</td><td colspan="3">IPBytes</td><td>1</td><td>1</td><td>1</td><td>0</td><td>1</td></tr>
<tr><td>1</td><td colspan="8">IP[7:0]</td></tr>
<tr><td>2</td><td colspan="8">IP[15:8]</td></tr>
<tr><td>3</td><td colspan="8">IP[23:16]</td></tr>
<tr><td>4</td><td colspan="8">IP[31:24]</td></tr>
<tr><td>5</td><td colspan="8">IP[39:32]</td></tr>
<tr><td>6</td><td colspan="8">IP[47:40]</td></tr>
<tr><td>7</td><td colspan="8">IP[55:48]</td></tr>
<tr><td>8</td><td colspan="8">IP[63:56]</td></tr>
</table>
</td></tr>
<tr><td>Dependencies</td><td>TriggerEn & ContextEn. (Typically depends on BranchEn and FilterEn as well, see Section 35.2.4 for details.)</td><td>Generation Scenario</td><td colspan="2">Asynchronous Events (interrupts, exceptions, INIT, SIPI, SMI, VM exit[1], #MC), XBEGIN, XEND, XABORT, XACQUIRE, XRELEASE, (EENTRY, EEXIT, ERESUME, EEE, AEX,)[2], INT 0, INT 3, INT n, a WRMSR that disables packet generation.</td></tr>
<tr><td>Description</td><td colspan="4">Provides the source address for asynchronous events, and some other instructions. Is never sent alone, always sent with an associated TIP or MODE packet, and potentially others.</td></tr>
<tr><td>Application</td><td colspan="4">FUP packets provide the IP to which they bind. However, they are never standalone, but are coupled with other packets.
In TSX cases, the FUP is immediately preceded by a MODE.TSX, which binds to the same IP. A TIP will follow only in the case of TSX aborts, see Section 35.4.2.8 for details.
Otherwise, FUPs are part of compound packet events (see Section 35.4.1). In these compound cases, the FUP provides the source IP for an instruction or event, while a following TIP (or TIP.PGD) uop will provide any destination IP. Other packets may be included in the compound event between the FUP and TIP.</td></tr>
</table>

NOTES:

1. If IA32_VMX_MISC[bit 14] reports 1.
2. If Intel Software Guard Extensions is supported.

FUP IP Payload

Flow Update Packet gives the source address of an instruction when it is needed. In general, branch instructions do not need a FUP, because the source address is clear from the disassembly. For asynchronous events, however, the source address cannot be inferred from the source, and hence a FUP will be sent. Table 35-23 illustrates cases where FUPs are sent, and which IP can be expected in those cases.

Table 35-23. FUP Cases and IP Payload

Event	Flow Update IP	Comment
External Interrupt, NMI/SMI, Traps, Machine Check (trap-like), INIT/SIPI	Address of next instruction (Next IP) that would have been executed	Functionally, this matches the LBR FROM field value and also the EIP value which is saved onto the stack.
Exceptions/Faults, Machine check (fault-like)	Address of the instruction which took the exception/fault (Current IP)	This matches the similar functionality of LBR FROM field value and also the EIP value which is saved onto the stack.
Software Interrupt	Address of the software interrupt instruction (Current IP)	This matches the similar functionality of LBR FROM field value, but does not match the EIP value which is saved onto the stack (Next Linear Instruction Pointer - NLIP).
EENTER, EEXIT, ERESUME, Enclave Exiting Event (EEE), AEX[1]	Current IP of the instruction	This matches the LBR FROM field value and also the EIP value which is saved onto the stack.
XACQUIRE	Address of the X* instruction	
XRELEASE, XBEGIN, XEND, XABORT, other transactional abort	Current IP	
#SMI	IP that is saved into SMRAM	
WRMSR that clears TraceEn	Current IP	

NOTES:

1. Information on EENTER, EEXIT, ERESUME, EEE, Asynchronous Enclave eXit (AEX) can be found in *Intel® Software Guard Extensions Programming Reference*.

On a canonical fault due to sequentially fetching an instruction in non-canonical space (as opposed to jumping to non-canonical space), the IP of the fault (and thus the payload of the FUP) will be a non-canonical address. This is consistent with what is pushed on the stack for such faulting cases.

If there are post-commit task switch faults, the IP value of the FUP will be the original IP when the task switch started. This is the same value as would be seen in the LBR_FROM field. But it is a different value as is saved on the stack or VMCS.

35.4.2.7 Paging Information (PIP) Packet

Table 35-24. PIP Packet Definition

<table>
<tr><td>Name</td><td colspan="4">Paging Information (PIP) Packet</td></tr>
<tr><td>Packet Format</td><td colspan="4">
<table>
<tr><th></th><th>7</th><th>6</th><th>5</th><th>4</th><th>3</th><th>2</th><th>1</th><th>0</th></tr>
<tr><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>1</td><td>0</td></tr>
<tr><td>1</td><td>0</td><td>1</td><td>0</td><td>0</td><td>0</td><td>0</td><td>1</td><td>1</td></tr>
<tr><td>2</td><td colspan="7">CR3[11:5] or 0</td><td>RSVD/NR</td></tr>
<tr><td>3</td><td colspan="8">CR3[19:12]</td></tr>
<tr><td>4</td><td colspan="8">CR3[27:20]</td></tr>
<tr><td>5</td><td colspan="8">CR3[35:28]</td></tr>
<tr><td>6</td><td colspan="8">CR3[43:36]</td></tr>
<tr><td>7</td><td colspan="8">CR3[51:44]</td></tr>
</table>
</td></tr>
<tr><td>Dependencies</td><td>TriggerEn && ContextEn && IA32_RTIT_CTL.OS</td><td>Generation Scenario</td><td colspan="2">MOV CR3, Task switch, INIT, SIPI, PSB+;
If IA32_VMX_MISC[bit 14] reports 1: VM exit, VM entry</td></tr>
<tr><td>Description</td><td colspan="4">The CR3 payload shown includes only the address portion of the CR3 value. For PAE paging, CR3[11:5] are thus included. For other page modes (32-bit and 4-level paging[1]), these bits are 0.
This packet holds the CR3 address value. It will be generated on operations that modify CR3:
▪ MOV CR3 operation
▪ Task Switch
▪ INIT and SIPI
▪ VM exit and VM entry, if appropriate controls in the VMCS are clear (see Section 35.5.1)
PIPs are not generated, despite changes to CR3, on SMI and RSM. This is due to the special behavior on these operations, see Section for details. Note that, for some cases of task switch where CR3 is not modified, no PIP will be produced.
The purpose of the PIP is to indicate to the decoder which application is running, so that it can apply the proper binaries to the linear addresses that are being traced.
The PIP packet contains the new CR3 value when CR3 is written.
PIPs generated by VM entries set the NR bit. PIPs generated in VMX non-root operation set the NR bit if the “conceal VMX non-root operation from Intel PT” VM-execution control is 0. All other PIPs clear the NR bit.</td></tr>
<tr><td>Application</td><td colspan="4">The purpose of the PIP packet is to help the decoder uniquely identify what software is running at any given time. When a PIP is encountered, a decoder should do the following:
1) If there was a prior unbound FUP (that is, a FUP not preceded by a packet such as MODE.TSX that consumes it, and it hence pairs with a TIP that has not yet been seen), then this PIP is part of a compound packet event (Section 35.4.1). Find the ending TIP and apply the new CR3/NR values to the TIP payload IP.
2) Otherwise, look for the next MOV CR3, far branch, or VMRESUME/VMLAUNCH in the disassembly, and apply the new CR3 to the next (or target) IP.
For examples of the packets generated by these flows, see Section 35.7.</td></tr>
</table>

NOTES:

1. Earlier versions of this manual used the term “IA-32e paging” to identify 4-level paging.

35.4.2.8 MODE Packets

MODE packets keep the decoder informed of various processor modes about which it needs to know in order to properly manage the packet output, or to properly disassemble the associated binaries. MODE packets include a header and a mode byte, as shown below.

Table 35-25. General Form of MODE Packets

	7	6	5	4	3	2	1	0
0	1	0	0	1	1	0	0	1
1	Leaf ID			Mode				

The MODE Leaf ID indicates which set of mode bits are held in the lower bits.

MODE.Exec Packet

Table 35-26. MODE.Exec Packet Definition

<table>
<tr><td>Name</td><td colspan="4">MODE.Exec Packet</td></tr>
<tr><td>Packet Format</td><td colspan="4">
<table>
<tr><td></td><td>7</td><td>6</td><td>5</td><td>4</td><td>3</td><td>2</td><td>1</td><td>0</td></tr>
<tr><td>0</td><td>1</td><td>0</td><td>0</td><td>1</td><td>1</td><td>0</td><td>0</td><td>1</td></tr>
<tr><td>1</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>CS.D</td><td>(CS.L & LMA)</td></tr>
</table>
</td></tr>
<tr><td>Dependencies</td><td>PacketEn</td><td>Generation Scenario</td><td colspan="2">Far branch, interrupt, exception, (VM exit, VM entry,)[1] if the mode changes. PSB+, and any scenario that can generate a TIP.PGE, such that the mode may have changed since the last MODE.Exec.</td></tr>
<tr><td>Description</td><td colspan="4">Indicates whether software is in 16, 32, or 64-bit mode, by providing the CS.D and (CS.L & IA32_EFER.LMA) values. Essential for the decoder to properly disassemble the associated binary.

<table>
<tr><td>CS.D</td><td>(CS.L & IA32_EFER.LMA)</td><td>Addressing Mode</td></tr>
<tr><td>1</td><td>1</td><td>N/A</td></tr>
<tr><td>0</td><td>1</td><td>64-bit mode</td></tr>
<tr><td>1</td><td>0</td><td>32-bit mode</td></tr>
<tr><td>0</td><td>0</td><td>16-bit mode</td></tr>
</table>

MODE.Exec is sent at the time of a mode change, if PacketEn=1 at the time, or when tracing resumes, if necessary. In the former case, the MODE.Exec packet is generated along with other packets that result from the far transfer operation that changes the mode. In cases where the mode changes while PacketEn=0, the processor will send out a MODE.Exec along with the TIP.PGE when tracing resumes. The processor may opt to suppress the MODE.Exec when tracing resumes if the mode matches that from the last MODE.Exec packet, if there was no PSB in between.</td></tr>
<tr><td>Application</td><td colspan="4">MODE.Exec always immediately precedes a TIP or TIP.PGE. The mode change applies to the IP address in the payload of the next TIP or TIP.PGE.</td></tr>
</table>

NOTES:

1. If IA32_VMX_MISC[bit 14] reports 1.

MODE.TSX Packet

Table 35-27. MODE.TSX Packet Definition

<table>
<tr><td>Name</td><td colspan="4">MODE.TSX Packet</td></tr>
<tr><td>Packet Format</td><td colspan="4">
<table>
<tr><td></td><td>7</td><td>6</td><td>5</td><td>4</td><td>3</td><td>2</td><td>1</td><td>0</td></tr>
<tr><td>0</td><td>1</td><td>0</td><td>0</td><td>1</td><td>1</td><td>0</td><td>0</td><td>1</td></tr>
<tr><td>1</td><td>0</td><td>0</td><td>1</td><td>0</td><td>0</td><td>0</td><td>TXAbort</td><td>InTX</td></tr>
</table>
</td></tr>
<tr><td>Dependencies</td><td>TriggerEn and ContextEn</td><td>Generation Scenario</td><td colspan="2">XBEGIN, XEND, XABORT, XACQUIRE, XRELEASE, if INTX changes, Asynchronous TSX Abort, PSB+</td></tr>
<tr><td>Description</td><td colspan="4">Indicates when a TSX transaction (either HLE or RTM) begins, commits, or aborts. Instructions executed transactionally will be “rolled back” if the transaction is aborted.
<table>
<tr><td>TXAbort</td><td>InTX</td><td>Implication</td></tr>
<tr><td>1</td><td>1</td><td>N/A</td></tr>
<tr><td>0</td><td>1</td><td>Transaction begins, or executing transactionally</td></tr>
<tr><td>1</td><td>0</td><td>Transaction aborted</td></tr>
<tr><td>0</td><td>0</td><td>Transaction committed, or not executing transactionally</td></tr>
</table>
</td></tr>
<tr><td>Application</td><td colspan="4">If PacketEn=1, MODE.TSX always immediately precedes a FUP. If the TXAbort bit is zero, then the mode change applies to the IP address in the payload of the FUP. If TXAbort=1, then the FUP will be followed by a TIP, and the mode change will apply to the IP address in the payload of the TIP.
MODE.TSX packets may be generated when PacketEn=0, due to FilterEn=0. In this case, only the last MODE.TSX generated before TIP.PGE need be applied.</td></tr>
</table>

35.4.2.9 TraceStop Packet

Table 35-28. TraceStop Packet Definition

Name	TraceStop Packet		
Packet Format			

	7	6	5	4	3	2	1	0
0	0	0	0	0	0	0	1	0
1	1	0	0	0	0	0	1	1

Dependencies	TriggerEn && ContextEn	Generation Scenario	Taken branch with target in TraceStop IP region, MOV CR3 in TraceStop IP region, or WRMSR that sets TraceEn in TraceStop IP region.
Description	Indicates when software has entered a user-configured TraceStop region. When the IP matches a TraceStop range while ContextEn and TriggerEn are set, a TraceStop action occurs. This disables tracing by setting IA32_RTIT_STATUS.Stopped, thereby clearing TriggerEn, and causes a TraceStop packet to be generated. The TraceStop action also forces FilterEn to 0. Note that TraceStop may not force a flush of internally buffered packets, and thus trace packet generation should still be manually disabled by clearing IA32_RTIT_CTL.TraceEn before examining output. See Section 35.2.4.3 for more details.		
Application	.If TraceStop follows a TIP.PGD (before the next TIP.PGE), then it was triggered either by the instruction that cleared PacketEn, or it was triggered by some later instruction that executed while FilterEn=0. In either case, the TraceStop can be applied at the IP of the TIP.PGD (if any). If TraceStop follows a TIP.PGE (before the next TIP.PGD), it should be applied at the last known IP.		

35.4.2.10 Core:Bus Ratio (CBR) Packet

Table 35-29. CBR Packet Definition

Name	Core:Bus Ratio (CBR) Packet		
Packet Format			

	7	6	5	4	3	2	1	0
0	0	0	0	0	0	0	1	0
1	0	0	0	0	0	0	1	1
2	Core:Bus Ratio							
3	Reserved							

Dependencies	TriggerEn	Generation Scenario	After any frequency change, on C-state wake up, PSB+, and after enabling trace packet generation.
Description	Indicates the core:bus ratio of the processor core. Useful for correlating wall-clock time and cycle time.		
Application	All packets following the CBR represent instructions that executed with the new core:bus ratio, while all preceding packets (aside from timing packets) represent instructions that executed with the prior ratio. There is not a precise IP provided, to which to bind the CBR packet.		

35.4.2.11 Timestamp Counter (TSC) Packet

Table 35-30. TSC Packet Definition

<table>
<tr><td>Name</td><td colspan="4">Timestamp Counter (TSC) Packet</td></tr>
<tr><td>Packet Format</td><td colspan="4">
<table>
<tr><td></td><td>7</td><td>6</td><td>5</td><td>4</td><td>3</td><td>2</td><td>1</td><td>0</td></tr>
<tr><td>0</td><td>0</td><td>0</td><td>0</td><td>1</td><td>1</td><td>0</td><td>0</td><td>1</td></tr>
<tr><td>1</td><td colspan="8">SW TSC[7:0]</td></tr>
<tr><td>2</td><td colspan="8">SW TSC[15:8]</td></tr>
<tr><td>3</td><td colspan="8">SW TSC[23:16]</td></tr>
<tr><td>4</td><td colspan="8">SW TSC[31:24]</td></tr>
<tr><td>5</td><td colspan="8">SW TSC[39:32]</td></tr>
<tr><td>6</td><td colspan="8">SW TSC[47:40]</td></tr>
<tr><td>7</td><td colspan="8">SW TSC[55:48]</td></tr>
</table>
</td></tr>
<tr><td>Dependencies</td><td>IA32_RTIT_CTL.TSCEn && TriggerEn</td><td>Generation Scenario</td><td colspan="2">Sent after any event that causes the processor clocks or Intel PT timing packets (such as MTC or CYC) to stop, This may include P-state changes, wake from C-state, or clock modulation. Also on transition of TraceEn from 0 to 1.</td></tr>
<tr><td>Description</td><td colspan="4">When enabled by software, a TSC packet provides the lower 7 bytes of the current TSC value, as returned by the RDTSC instruction. This may be useful for tracking wall-clock time, and synchronizing the packets in the log with other timestamped logs.</td></tr>
<tr><td>Application</td><td colspan="4">TSC packet provides a wall-clock proxy of the event which generated it (packet generation enable, sleep state wake, etc). In all cases, TSC does not precisely indicate the time of any control flow packets; however, all preceding packets represent instructions that executed before the indicated TSC time, and all subsequent packets represent instructions that executed after it. There is not a precise IP to which to bind the TSC packet.</td></tr>
</table>

35.4.2.12 Mini Time Counter (MTC) Packet

Table 35-31. MTC Packet Definition

<table>
<tr><td>Name</td><td colspan="3">Mini time Counter (MTC) Packet</td></tr>
<tr><td>Packet Format</td><td colspan="3">
<table>
<tr><td></td><td>7</td><td>6</td><td>5</td><td>4</td><td>3</td><td>2</td><td>1</td><td>0</td></tr>
<tr><td>0</td><td>0</td><td>1</td><td>0</td><td>1</td><td>1</td><td>0</td><td>0</td><td>1</td></tr>
<tr><td>1</td><td colspan="8">CTC[N+7:N]</td></tr>
</table>
</td></tr>
<tr><td>Dependencies</td><td>IA32_RTIT_CTL.MTCEn && TriggerEn</td><td>Generation Scenario</td><td>Periodic, based on the core crystal clock, or Always Running Timer (ART).</td></tr>
<tr><td>Description</td><td colspan="3">When enabled by software, an MTC packet provides a periodic indication of wall-clock time. The 8-bit CTC (Common Timestamp Copy) payload value is set to (ART >> N) & 0xFF. The frequency of the ART is related to the Maximum Non-Turbo frequency, and the ratio can be determined from CPUID leaf 15H, as described in Section 35.8.3.
Software can select the threshold N, which determines the MTC frequency by setting the IA32_RTIT_CTL.MTCFreq field (see Section 35.2.7.2) to a supported value using the lookup enumerated by CPUID (see Section 35.3.1).
See Section 35.8.3 for details on how to use the MTC payload to track TSC time.
MTC provides 8 bits from the ART, starting with the bit selected by MTCFreq to dictate the frequency of the packet. Whenever that 8-bit range being watched changes, an MTC packet will be sent out with the new value of that 8-bit range. This allows the decoder to keep track of how much wall-clock time has elapsed since the last TSC packet was sent, by keeping track of how many MTC packets were sent and what their value was. The decoder can infer the truncated bits, CTC[N-1:0], are 0 at the time of the MTC packet.
There are cases in which MTC packet can be dropped, due to overflow or other micro-architectural conditions. The decoder should be able to recover from such cases by checking the 8-bit payload of the next MTC packet, to determine how many MTC packets were dropped. It is not expected that >256 consecutive MTC packets should ever be dropped.</td></tr>
<tr><td>Application</td><td colspan="3">MTC does not precisely indicate the time of any other packet, nor does it bind to any IP. However, all preceding packets represent instructions or events that executed before the indicated ART time, and all subsequent packets represent instructions that executed after, or at the same time as, the ART time.</td></tr>
</table>

35.4.2.13 TSC/MTC Alignment (TMA) Packet

Table 35-32. TMA Packet Definition

<table>
<tr><td>Name</td><td colspan="4">TSC/MTC Alignment (TMA) Packet</td></tr>
<tr><td>Packet Format</td><td colspan="4">
<table>
<tr><td></td><td>7</td><td>6</td><td>5</td><td>4</td><td>3</td><td>2</td><td>1</td><td>0</td></tr>
<tr><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>1</td><td>0</td></tr>
<tr><td>1</td><td>0</td><td>1</td><td>1</td><td>1</td><td>0</td><td>0</td><td>1</td><td>1</td></tr>
<tr><td>2</td><td colspan="8">CTC[7:0]</td></tr>
<tr><td>3</td><td colspan="8">CTC[15:8]</td></tr>
<tr><td>4</td><td colspan="7">Reserved</td><td>0</td></tr>
<tr><td>5</td><td colspan="8">FastCounter[7:0]</td></tr>
<tr><td>6</td><td colspan="7">Reserved</td><td>FC[8]</td></tr>
</table>
</td></tr>
<tr><td>Dependencies</td><td>IA32_RTIT_CTL.MTCEn && IA32_RTIT_CTL.TSCEn && TriggerEn</td><td>Generation Scenario</td><td colspan="2">Sent with any TSC packet.</td></tr>
<tr><td>Description</td><td colspan="4">The TMA packet serves to provide the information needed to allow the decoder to correlate MTC packets with TSC packets. With this packet, when a MTC packet is encountered, the decoder can determine how many timestamp counter ticks have passed since the last TSC or MTC packet. See Section 35.8.3.2 for details on how to make this calculation.</td></tr>
<tr><td>Application</td><td colspan="4">TMA is always sent immediately following a TSC packet, and the payload values are consistent with the TSC payload value. Thus the application of TMA matches that of TSC.</td></tr>
</table>

35.4.2.14 Cycle Count Packet (CYC) Packet

Table 35-33. Cycle Count Packet Definition

<table>
<tr><td>Name</td><td colspan="4">Cycle Count (CYC) Packet</td></tr>
<tr><td>Packet Format</td><td colspan="4">
<table>
<tr><th></th><th>7</th><th>6</th><th>5</th><th>4</th><th>3</th><th>2</th><th>1</th><th>0</th></tr>
<tr><td>0</td><td colspan="5">Cycle Counter[4:0]</td><td>Exp</td><td>1</td><td>1</td></tr>
<tr><td>1</td><td colspan="7">Cycle Counter[11:5]</td><td>Exp</td></tr>
<tr><td>2</td><td colspan="7">Cycle Counter[18:12]</td><td>Exp</td></tr>
<tr><td>...</td><td colspan="8">... (if Exp = 1 in the previous byte)</td></tr>
</table>
</td></tr>
<tr><td>Dependencies</td><td>IA32_RTIT_CTL.CYCEn && TriggerEn</td><td>Generation Scenario</td><td colspan="2">Can be sent at any time, though a maximum of one CYC packet is sent per core clock cycle. See Section 35.3.6 for CYC-eligible packets.</td></tr>
<tr><td>Description</td><td colspan="4">The Cycle Counter field increments at the same rate as the processor core clock ticks, but with a variable length format (using a trailing EXP bit field) and a range-capped byte length.
If the CYC value is less than 32, a 1-byte CYC will be generated, with Exp=0. If the CYC value is between 32 and 4095 inclusive, a 2-byte CYC will be generated, with byte 0 Exp=1 and byte 1 Exp=0. And so on.
CYC provides the number of core clocks that have passed since the last CYC packet. CYC can be configured to be sent in every cycle in which an eligible packet is generated, or software can opt to use a threshold to limit the number of CYC packets, at the expense of some precision. These settings are configured using the IA32_RTIT_CTL.CycThresh field (see Section 35.2.7.2). For details on Cycle-Accurate Mode, IPC calculation, etc, see Section 35.3.6.
When CycThresh=0, and hence no threshold is in use, then a CYC packet will be generated in any cycle in which any CYC-eligible packet is generated. The CYC packet will precede the other packets generated in the cycle, and provides the precise cycle time of the packets that follow.
In addition to these CYC packets generated with other packets, CYC packets can be sent stand-alone. These packets serve simply to update the decoder with the number of cycles passed, and are used to ensure that a wrap of the processor's internal cycle counter doesn't cause cycle information to be lost. These stand-alone CYC packets do not indicate the cycle time of any other packet or operation, and will be followed by another CYC packet before any other CYC-eligible packet is seen.
When CycThresh>0, CYC packets are generated only after a minimum number of cycles have passed since the last CYC packet. Once this threshold has passed, the behavior above resumes, where CYC will either be sent in the next cycle that produces other CYC-eligible packets, or could be sent stand-alone.
When using CYC thresholds, only the cycle time of the operation (instruction or event) that generates the CYC packet is truly known. Other operations simply have their execution time bounded: they completed at or after the last CYC time, and before the next CYC time.</td></tr>
<tr><td>Application</td><td colspan="4">CYC provides the offset cycle time (since the last CYC packet) for the CYC-eligible packet that follows. If another CYC is encountered before the next CYC-eligible packet, the cycle values should be accumulated and applied to the next CYC-eligible packet.
If a CYC packet is generated by a TNT, note that the cycle time provided by the CYC packet applies to the first branch in the TNT packet.</td></tr>
</table>

35.4.2.15 VMCS Packet

Table 35-34. VMCS Packet Definition

<table>
<tr><td>Name</td><td colspan="4">VMCS Packet</td></tr>
<tr><td>Packet Format</td><td colspan="4">
<table>
<tr><td></td><td>7</td><td>6</td><td>5</td><td>4</td><td>3</td><td>2</td><td>1</td><td>0</td></tr>
<tr><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>1</td><td>0</td></tr>
<tr><td>1</td><td>1</td><td>1</td><td>0</td><td>0</td><td>1</td><td>0</td><td>0</td><td>0</td></tr>
<tr><td>2</td><td colspan="8">VMCS Base Address [19:12]</td></tr>
<tr><td>3</td><td colspan="8">VMCS Base Address [27:20]</td></tr>
<tr><td>4</td><td colspan="8">VMCS Base Address [35:28]</td></tr>
<tr><td>5</td><td colspan="8">VMCS Base Address [43:36]</td></tr>
<tr><td>6</td><td colspan="8">VMCS Base Address [51:44]</td></tr>
</table>
</td></tr>
<tr><td>Dependencies</td><td>TriggerEn && ContextEn; Also in VMX operation.</td><td>Generation Scenario</td><td colspan="2">Generated on successful VMPTRLD, and optionally on SMM VM exits and VM entries that return from SMM (see Section 35.5).</td></tr>
<tr><td>Description</td><td colspan="4">The VMCS packet provides an address related to a VMCS pointer for a decoder to determine the transition of code contexts:
• On a successful VMPTRLD (i.e., a VMPTRLD that doesn't fault, fail, or VM exit), the VMCS packet contains the address of the current working VMCS pointer of the logical processor that will execute a VM guest context.
• On SMM VM exits, the VMCS packet provides the STM VMCS base address (SMM Transfer VMCS pointer), if VMCS-based controls are clear (see Section 35.5.1). See Section 35.6 on tracing inside and outside STM.
• On VM entries that return from SMM, the VMCS packet provides the current working VMCS pointer of the guest VM (see Section 35.6), if VMCS-based controls are clear (see Section 35.5.1). Root versus Non-Root operation can be distinguished from the PIP.NR bit.
If a VMCS packet is generated before a VMCS has been loaded, or after it has been cleared, the base address value will be all 1s.
VMCS packets will not be seen on processors with IA32_VMX_MISC[bit 14]=0, as these processors do not allow TraceEn to be set in VMX operation.</td></tr>
<tr><td>Application</td><td colspan="4">The purpose of the VMCS packet is to help the decoder uniquely identify changes in the executing software context in situations that CR3 may not be unique.
When a VMCS is encountered, a decoder should do the following:
▪ If there was a prior unbound FUP (that is, a FUP not preceded by a packet such as MODE.TSX that consumes it, and it hence pairs with a TIP that has not yet been seen), then this VMCS is part of a compound packet event (Section 35.4.1). Find the ending TIP and apply the new VMCS base pointer value to the TIP payload IP.
▪ Otherwise, look for the next VMPTRLD, VMRESUME, or VMLAUNCH in the disassembly, and apply the new VMCS base pointer on the next VM entry.
For examples of the packets generated by these flows, see Section 35.7.</td></tr>
</table>

35.4.2.16 Overflow (OVF) Packet

Table 35-35. OVF Packet Definition

Name	Overflow (OVF) Packet		
Packet Format			
Dependencies	TriggerEn	Generation Scenario	On resolution of internal buffer overflow
Description	OVF simply indicates to the decoder that an internal buffer overflow occurred, and packets were likely lost. If BranchEN= 1, OVF is followed by a FUP or TIP.PGE which will provide the IP at which packet generation resumes. See Section 35.3.8.		
Application	When an OVF packet is encountered, the decoder should skip to the IP given in the subsequent FUP or TIP.PGE. The cycle counter for the CYC packet will be reset at the time the OVF packet is sent. Software should reset its call stack depth on overflow, since no RET compression is allowed across an overflow. Similarly, any IP compression that follows the OVF is guaranteed to use as a reference LastIP the IP payload of an IP packet that preceded the overflow.		

	7	6	5	4	3	2	1	0
0	0	0	0	0	0	0	1	0
1	1	1	1	1	0	0	1	1

35.4.2.17 Packet Stream Boundary (PSB) Packet

Table 35-36. PSB Packet Definition

Name	Packet Stream Boundary (PSB) Packet
Packet Format	

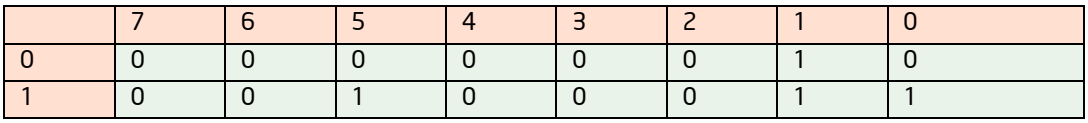
	7	6	5	4	3	2	1	0
0	0	0	0	0	0	0	1	0
1	1	0	0	0	0	0	1	0
2	0	0	0	0	0	0	1	0
3	1	0	0	0	0	0	1	0
4	0	0	0	0	0	0	1	0
5	1	0	0	0	0	0	1	0
6	0	0	0	0	0	0	1	0
7	1	0	0	0	0	0	1	0
8	0	0	0	0	0	0	1	0
9	1	0	0	0	0	0	1	0
10	0	0	0	0	0	0	1	0
11	1	0	0	0	0	0	1	0
12	0	0	0	0	0	0	1	0
13	1	0	0	0	0	0	1	0
14	0	0	0	0	0	0	1	0
15	1	0	0	0	0	0	1	0

Table 35-36. PSB Packet Definition (Contd.)

Dependencies	TriggerEn	Generation Scenario	Periodic, based on the number of output bytes generated while tracing. PSB is sent when IA32_RTIT_STATUS.PacketByteCnt=0, and each time it crosses the software selected threshold after that. May be sent for other micro-architectural conditions as well.
Description	PSB is a unique pattern in the packet output log, and hence serves as a sync point for the decoder. It is a pattern that the decoder can search for in order to get aligned on packet boundaries. This packet is periodic, based on the number of output bytes, as indicated by IA32_RTIT_STATUS.PacketByteCnt. The period is chosen by software, via IA32_RTIT_CTL.PSBFreq (see Section 35.2.7.2). Note, however, that the PSB period is not precise, it simply reflects the average number of output bytes that should pass between PSBs. The processor will make a best effort to insert PSB as quickly after the selected threshold is reached as possible. The processor also may send extra PSB packets for some micro-architectural conditions. PSB also serves as the leading packet for a set of "status-only" packets collectively known as PSB+ (Section 35.3.7).		
Application	When a PSB is seen, the decoder should interpret all following packets as "status only", until either a PSBEND or OVF packet is encountered. "Status only" implies that the binding and ordering rules to which these packets normally adhere are ignored, and the state they carry can instead be applied to the IP payload in the FUP packet that is included.		

35.4.2.18 PSBEND Packet

Table 35-37. PSBEND Packet Definition

Name	PSBEND Packet		
Packet Format			
Dependencies	TriggerEn	Generation Scenario	Always follows PSB packet, separated by PSB+ packets
Description	PSBEND is simply a terminator for the series of "status only" (PSB+) packets that follow PSB (Section 35.3.7).		
Application	When a PSBEND packet is seen, the decoder should cease to treat packets as "status only".		

35.4.2.19 Maintenance (MNT) Packet

Table 35-38. MNT Packet Definition

<table>
<tr><td>Name</td><td colspan="4">Maintenance (MNT) Packet</td></tr>
<tr><td>Packet Format</td><td colspan="4">
<table>
<tr><th></th><th>7</th><th>6</th><th>5</th><th>4</th><th>3</th><th>2</th><th>1</th><th>0</th></tr>
<tr><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>1</td><td>0</td></tr>
<tr><td>1</td><td>1</td><td>1</td><td>0</td><td>0</td><td>0</td><td>0</td><td>1</td><td>1</td></tr>
<tr><td>2</td><td>1</td><td>0</td><td>0</td><td>0</td><td>1</td><td>0</td><td>0</td><td>0</td></tr>
<tr><td>3</td><td colspan="8">Payload[7:0]</td></tr>
<tr><td>4</td><td colspan="8">Payload[15:8]</td></tr>
<tr><td>5</td><td colspan="8">Payload[23:16]</td></tr>
<tr><td>6</td><td colspan="8">Payload[31:24]</td></tr>
<tr><td>7</td><td colspan="8">Payload[39:32]</td></tr>
<tr><td>8</td><td colspan="8">Payload[47:40]</td></tr>
<tr><td>9</td><td colspan="8">Payload[55:48]</td></tr>
<tr><td>10</td><td colspan="8">Payload[63:56]</td></tr>
</table>
</td></tr>
<tr><td>Dependencies</td><td>TriggerEn</td><td>Generation Scenario</td><td colspan="2">Implementation specific.</td></tr>
<tr><td>Description</td><td colspan="4">This packet is generated by hardware, the payload meaning is model-specific.</td></tr>
<tr><td>Application</td><td colspan="4">Unless a decoder has been extended for a particular family/model/stepping to interpret MNT packet payloads, this packet should simply be ignored. It does not bind to any IP.</td></tr>
</table>

35.4.2.20 PAD Packet

Table 35-39. PAD Packet Definition

<table>
<tr><td>Name</td><td colspan="4">PAD Packet</td></tr>
<tr><td>Packet Format</td><td colspan="4">
<table>
<tr><th></th><th>7</th><th>6</th><th>5</th><th>4</th><th>3</th><th>2</th><th>1</th><th>0</th></tr>
<tr><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td></tr>
</table>
</td></tr>
<tr><td>Dependencies</td><td>TriggerEn</td><td>Generation Scenario</td><td colspan="2">Implementation specific</td></tr>
<tr><td>Description</td><td colspan="4">PAD is simply a NOP packet. Processor implementations may choose to add pad packets to improve packet alignment or for implementation-specific reasons.</td></tr>
<tr><td>Application</td><td colspan="4">Ignore PAD packets.</td></tr>
</table>

35.4.2.21 PTWRITE Packet

Table 35-40. PTW Packet Definition

<table>
<tr><td>Name</td><td colspan="3">PTW Packet</td></tr>
<tr><td>Packet Format</td><td colspan="3">
<table>
<tr><td></td><td>7</td><td>6</td><td>5</td><td>4</td><td>3</td><td>2</td><td>1</td><td>0</td></tr>
<tr><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>1</td><td>0</td></tr>
<tr><td>1</td><td>IP</td><td colspan="2">PayloadBytes</td><td>1</td><td>0</td><td>0</td><td>1</td><td>0</td></tr>
<tr><td>2</td><td colspan="8">Payload[7:0]</td></tr>
<tr><td>3</td><td colspan="8">Payload[15:8]</td></tr>
<tr><td>4</td><td colspan="8">Payload[23:16]</td></tr>
<tr><td>5</td><td colspan="8">Payload[31:24]</td></tr>
<tr><td>6</td><td colspan="8">Payload[39:32]</td></tr>
<tr><td>7</td><td colspan="8">Payload[47:40]</td></tr>
<tr><td>8</td><td colspan="8">Payload[55:48]</td></tr>
<tr><td>9</td><td colspan="8">Payload[63:56]</td></tr>
</table>
</td></tr>
<tr><td></td><td colspan="3">The PayloadBytes field indicates the number of bytes of payload that follow the header bytes. Encodings are as follows:
<table>
<tr><th>PayloadBytes</th><th>Bytes of Payload</th></tr>
<tr><td>‘00</td><td>4</td></tr>
<tr><td>‘01</td><td>8</td></tr>
<tr><td>‘10</td><td>Reserved</td></tr>
<tr><td>‘11</td><td>Reserved</td></tr>
</table>
IP bit indicates if a FUP, whose payload will be the IP of the PTWRITE instruction, will follow.</td></tr>
<tr><td>Dependencies</td><td>TriggerEn & ContextEn & FilterEn & PTWEn</td><td>Generation Scenario</td><td>PTWRITE Instruction</td></tr>
<tr><td>Description</td><td colspan="3">Contains the value held in the PTWRITE operand.
This packet is CYC-eligible, and hence will generate a CYC packet if IA32_RTIT_CTL.CYCEn=1 and any CYC Threshold has been reached.</td></tr>
<tr><td>Application</td><td colspan="3">Binds to the associated PTWRITE instruction. The IP of the PTWRITE will be provided by a following FUP, when PTW.IP=1.</td></tr>
</table>

35.4.2.22 Execution Stop (EXSTOP) Packet

Table 35-41. EXSTOP Packet Definition

<table>
<tr><td>Name</td><td colspan="3">EXSTOP Packet</td></tr>
<tr><td>Packet Format</td><td colspan="3">
<table>
<tr><td></td><td>7</td><td>6</td><td>5</td><td>4</td><td>3</td><td>2</td><td>1</td><td>0</td></tr>
<tr><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>1</td><td>0</td></tr>
<tr><td>1</td><td>IP</td><td>1</td><td>1</td><td>0</td><td>0</td><td>0</td><td>1</td><td>0</td></tr>
</table>
IP bit indicates if a FUP will follow.
</td></tr>
<tr><td>Dependencies</td><td>TriggerEn & PwrEvtEn</td><td>Generation Scenario</td><td>C-state entry, P-state change, or other processor clock power-down. Includes :
▪ Entry to C-state deeper than C0.0
▪ TM1/2
▪ STPCLK#
▪ Frequency change due to IA32_CLOCK_MODULATION, Turbo</td></tr>
<tr><td>Description</td><td colspan="3">This packet indicates that software execution has stopped due to processor clock powerdown. Later packets will indicate when execution resumes.
If EXSTOP is generated while ContextEn is set, the IP bit will be set, and EXSTOP will be followed by a FUP packet containing the IP at which execution stopped. More precisely, this will be the IP of the oldest instruction that has not yet completed.
This packet is CYC-eligible, and hence will generate a CYC packet if IA32_RTIT_CTL.CYCEn=1 and any CYC Threshold has been reached.</td></tr>
<tr><td>Application</td><td colspan="3">If a FUP follows EXSTOP (hence IP bit set), the EXSTOP can be bound to the FUP IP. Otherwise the IP is not known. Time of powerdown can be inferred from the preceding CYC, if CYCEn=1. Combined with the TSC at the time of wake (if TSCEn=1), this can be used to determine the duration of the powerdown.</td></tr>
</table>

35.4.2.23 MWAIT Packet

Table 35-42. MWAIT Packet Definition

<table>
<tr><td>Name</td><td colspan="4">MWAIT Packet</td></tr>
<tr><td>Packet Format</td><td colspan="4">
<table>
<tr><td></td><td>7</td><td>6</td><td>5</td><td>4</td><td>3</td><td>2</td><td>1</td><td>0</td></tr>
<tr><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>1</td><td>0</td></tr>
<tr><td>1</td><td>1</td><td>1</td><td>0</td><td>0</td><td>0</td><td>0</td><td>1</td><td>0</td></tr>
<tr><td>2</td><td colspan="8">MWAIT Hints[7:0]</td></tr>
<tr><td>3</td><td colspan="8">Reserved</td></tr>
<tr><td>4</td><td colspan="8">Reserved</td></tr>
<tr><td>5</td><td colspan="8">Reserved</td></tr>
<tr><td>6</td><td colspan="6">Reserved</td><td colspan="2">EXT[1:0]</td></tr>
<tr><td>7</td><td colspan="8">Reserved</td></tr>
<tr><td>8</td><td colspan="8">Reserved</td></tr>
<tr><td>9</td><td colspan="8">Reserved</td></tr>
</table>
</td></tr>
<tr><td>Dependencies</td><td>TriggerEn & PwrEvtEn & Contex-tEn</td><td>Generation Scenario</td><td colspan="2">MWAIT instruction, or I/O redirection to MWAIT, that complete without fault or VMexit.</td></tr>
<tr><td>Description</td><td colspan="4">Indicates that an MWAIT operation to C-state deeper than C0.0 completed. The MWAIT hints and extensions passed in by software are exposed in the payload.
This packet is CYC-eligible, and hence will generate a CYC packet if IA32_RTIT_CTL.CYCEn=1 and any CYC Threshold has been reached.</td></tr>
<tr><td>Application</td><td colspan="4">The MWAIT packet should bind to the IP of the next FUP, which will be the IP of the instruction that caused the MWAIT. This FUP will be shared with EXSTOP.</td></tr>
</table>

35.4.2.24 Power Entry (PWRE) Packet

Table 35-43. PWRE Packet Definition

<table>
<tr><td>Name</td><td colspan="4">PWRE Packet</td></tr>
<tr><td>Packet Format</td><td colspan="4">
<table>
<tr><td></td><td>7</td><td>6</td><td>5</td><td>4</td><td>3</td><td>2</td><td>1</td><td>0</td></tr>
<tr><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>1</td><td>0</td></tr>
<tr><td>1</td><td>0</td><td>0</td><td>1</td><td>0</td><td>0</td><td>0</td><td>1</td><td>0</td></tr>
<tr><td>2</td><td>HW</td><td colspan="7">Reserved</td></tr>
<tr><td>3</td><td colspan="4">Resolved Thread C-State</td><td colspan="4">Resolved Thread Sub C-State</td></tr>
</table>
</td></tr>
<tr><td>Dependencies</td><td>TriggerEn & PwrEvtEn</td><td>Generation Scenario</td><td colspan="2">Transition to a C-state deeper than C0.0.</td></tr>
<tr><td>Description</td><td colspan="4">Indicates processor entry to the resolved thread C-state and sub C-state indicated. The processor will remain in this C-state until either another PWRE indicates the processor has moved to a C-state deeper than C0.0, or a PWRX packet indicates a return to C0.
Note that some CPUs may allow MWAIT to request a deeper C-state than is supported by the core. These deeper C-states may have platform-level implications that differentiate them. However, the PWRE packet will provide only the resolved thread C-state, which will not exceed that supported by the core.
If the C-state entry was initiated by hardware, rather than a direct software request (such as MWAIT, HLT, or shut-down), the HW bit will be set to indicate this. Hardware Duty Cycling (see Section 14.5, “Hardware Duty Cycling (HDC)” in the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3B) is an example of such a case.</td></tr>
<tr><td>Application</td><td colspan="4">When transitioning from C0.0 to a deeper C-state, the PWRE packet will be followed by an EXSTOP. If that EXSTOP packet has the IP bit set, then the following FUP will provide the IP at which the C-state entry occurred. Subsequent PWRE packets generated before the next PWRX should bind to the same IP.</td></tr>
</table>

35.4.2.25 Power Exit (PWRX) Packet

Table 35-44. PWRX Packet Definition

<table>
<tr><td>Name</td><td colspan="4">PWRX Packet</td></tr>
<tr><td>Packet Format</td><td colspan="4">
<table>
<tr><td></td><td>7</td><td>6</td><td>5</td><td>4</td><td>3</td><td>2</td><td>1</td><td>0</td></tr>
<tr><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>1</td><td>0</td></tr>
<tr><td>1</td><td>1</td><td>0</td><td>1</td><td>0</td><td>0</td><td>0</td><td>1</td><td>0</td></tr>
<tr><td>2</td><td colspan="4">Last Core C-State</td><td colspan="4">Deepest Core C-State</td></tr>
<tr><td>3</td><td colspan="4">Reserved</td><td colspan="4">Wake Reason</td></tr>
<tr><td>4</td><td colspan="8">Reserved</td></tr>
<tr><td>5</td><td colspan="8">Reserved</td></tr>
<tr><td>6</td><td colspan="8">Reserved</td></tr>
</table>
</td></tr>
<tr><td>Dependencies</td><td>TriggerEn & PwrEvtEn</td><td>Generation Scenario</td><td colspan="2">Transition from a C-state deeper than C0.0 to C0.</td></tr>
<tr><td>Description</td><td colspan="4">Indicates processor return to thread C0 from a C-state deeper than C0.0.
The Last Core C-State field provides the MWAIT encoding for the core C-state at the time of the wake. The Deepest Core C-State provides the MWAIT encoding for the deepest core C-state achieved during the sleep session, or since leaving thread C0. MWAIT encodings for C-states can be found in Table 4-11 in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2B. Note that these values reflect only the core C-state, and hence will not exceed the maximum supported core C-state, even if deeper C-states can be requested.
The Wake Reason field is one-hot, encoded as follows:
<table>
<tr><th>Bit</th><th>Field</th><th>Meaning</th></tr>
<tr><td>0</td><td>Interrupt</td><td>Wake due to external interrupt received.</td></tr>
<tr><td>1</td><td>Reserved</td><td></td></tr>
<tr><td>2</td><td>Store to Monitored Address</td><td>Wake due to store to monitored address.</td></tr>
<tr><td>3</td><td>HW Wake</td><td>Wake due to hardware autonomous condition, such as HDC.</td></tr>
</table>
</td></tr>
<tr><td>Application</td><td colspan="4">PWRX will always apply to the same IP as the PWRE. The time of wake can be discerned from (optional) timing packets that precede PWRX.</td></tr>
</table>

35.5 TRACING IN VMX OPERATION

On processors that IA32_VMX_MISC[bit 14] reports 1, TraceEn can be set in VMX operation. A series of mechanisms exist to allow the VMM to configure tracing based on the desired trace domain, and on the consumer of the trace output. The VMM can configure specific VM-execution controls to control what virtualization-specific data are included within the trace packets (see Section 35.5.1 for details). MSR save and load lists can be employed by the VMM to restrict tracing to the desired context (see Section 35.5.2 for details). These configuration options are summarized in Table 35-45. Table 35-45 covers common Intel PT usages while SMIs are handled by the default SMM treatment. Tracing with SMM Transfer Monitor is described in Section 35.6.

Table 35-45. Common Usages of Intel PT and VMX

Target Domain	Output Consumer	Virtualize Output	Configure VMCS Controls	TraceEN Configuration	Save/Restore MSR states of Trace Configuration
System-Wide (VMM + VMs)	Host	NA	Default Setting (no suppression)	WRMSR or XRSTORS by Host	NA
VMM Only	Intel PT Aware VMM	NA	Enable suppression	MSR load list to disable tracing in VM, enable tracing on VM exits	NA
VM Only	Intel PT Aware VMM	NA	Enable suppression	MSR load list to enable tracing in VM, disable tracing on VM exits	NA
Intel PT Aware Guest(s)	Per Guest	VMM adds trace output virtualization	Enable suppression	MSR load list to enable tracing in VM, disable tracing on VM exits	VMM Update guest state on XRSTORS-exiting VM exits

35.5.1 VMX-Specific Packets and VMCS Controls

In all of the usages of VMX and Intel PT, the decoder in the host or VMM context can identify the occurrences of VMX transitions with the aid of VMX-specific packets. Packets relevant to VMX fall into the follow two kinds:

- VMCS Packet: The VMX transitions of individual VM can be distinguished by a decoder using the base address field in a VMCS packet. The base address field stores the VMCS pointer address of a successful VMPTRLD. A VMCS packet is sent on a successful execution of VMPTRLD. See Section 35.4.2.15 for details.
- NonRoot (NR) bit field in PIP packet: PIP packets are generated with each VM entry/exit. The NR bit in a PIP packet is set when in VMX non-Root operation. Thus a transition of the NR bit from 0 to 1 indicates the occurrence of a VM entry, and a transition of 1 to 0 indicates the occurrence of a VM exit.

Processors with IA32_VMX_MISC[bit 14]= 1 also provides VMCS controls that a VMM can configure to prevent VMX-specific information from leaking across virtualization boundaries.

Table 35-46. VMCS Controls For Intel Processor Trace

Name	Type	Bit Position	Value	Behavior
Conceal VMX non-root operation from Intel PT	VM-execution control	19	0	PIPs generated in VM non-root operation will set the PIP.NR bit. PSB+ in VMX non-root operation will include the VMCS packet, to ensure that the decoder knows which guest is currently in use.
			1	PIPs generated in VMX non-root operation will not set the PIP.NR bit. PSB+ in VMX non-root operation will not include the VMCS packet.
Conceal VM exits from Intel PT	VM-exit control	24	0	PIPs are generated on VM exit, with NonRoot=0. On VM exit to SMM, VMCS packets are additionally generated.
			1	No PIP is generated on VM exit, and no VMCS packet is generated on VM exit to SMM.
Conceal VM entries from Intel PT	VM-entry control	17	0	PIPs are generated on VM entry, with NonRoot=1 if the destination of the VM entry is VMX non-root operation. On VM entry to SMM, VMCS packets are additionally generated.
			1	No PIP is generated on VM entry, and no VMCS packet is generated on VM entry to SMM.

The default setting for the VMCS controls that interacts with Intel PT is to enable all VMX-specific packet information. The scenarios that would use the default setting also do not require the VMM to use MSR load list to manage the configuration of turning-on/off of trace packet generation across VM exits.

If IA32_VMX_MISC[bit 14] reports 0, any attempt to set the VMCS control bits in Table 35-46 will result in a failure on guest entry.

35.5.2 Managing Trace Packet Generation Across VMX Transitions

In tracing scenarios that collect packets for both VMX root operation and VMX non-root operation, a host executive can manage the MSRs associated with trace packet generation directly. The states of these MSRs need not be modified using MSR load list or MSR save list across VMX transitions.

For tracing scenarios that collect only packets within either VMX root operation or VMX non-root operation, the VMM can use the MSR load list and/or MSR save list to toggle IA32_RTIT_CTL.TraceEn.

35.5.2.1 System-Wide Tracing

When a host or VMM configures Intel PT to collect trace packets of the entire system, it can leave the VMCS controls clear to allow VMX-specific packets to provide information across VMX transitions. MSR load list is not used across VM exits or VM entries, nor is VM-exit MSR save list.

The decoder will desire to identify the occurrence of VMX transitions. The packets of interests to a decoder are shown in Table 35-47.

Table 35-47. Packets on VMX Transitions (System-Wide Tracing)

Event	Packets	Description
VM exit	FUP(GuestIP)	The FUP indicates at which point in the guest flow the VM exit occurred. This is important, since VM exit can be an asynchronous event. The IP will match that written into the VMCS.
	PIP(HostCR3, NR=0)	The PIP packet provides the new host CR3 value, as well as indication that the logical processor is entering VMX root operation. This allows the decoder to identify the change of executing context from guest to host and load the appropriate set of binaries to continue decode.
	TIP(HostIP)	The TIP indicates the destination IP, the IP of the first instruction to be executed in VMX root operation. Note, this packet could be preceded by a MODE.Exec packet (Section 35.4.2.8). This is generated only in cases where CS.D or (CS.L & EFER.LMA) change during the transition.
VM entry	PIP(GuestCR3, NR=1)	The PIP packet provides the new guest CR3 value, as well as indication that the logical processor is entering VMX non-root operation. This allows the decoder to identify the change of executing context from host to guest and load the appropriate set of binaries to continue decode.
	TIP(GuestIP)	The TIP indicates the destination IP, the IP of the first instruction to be executed in VMX non-root operation. This should match the IP value read out from the VMCS. Note, this packet could be preceded by a MODE.Exec packet (Section 35.4.2.8). This is generated only in cases where CS.D or (CS.L & EFER.LMA) change during the transition.

Since the packet suppression controls are cleared, the VMCS packet will be included in all PSB+ for this usage scenario. Thus the decoder can distinguish the execution context of different VMs. Additionally, it will be generated on VMPTRLD. Thus the decoder can distinguish the execution context of different VMs.

When the host VMM configures a system to collect trace packets in this scenario, it should emulate CPUID to report CPUID.(EAX=07H, ECX=0):EBX[bit 26] with 0 to guests, indicating to guests that Intel PT is not available.

VMX TSC Manipulation

The TSC packets generated while in VMX non-root operation will include any changes resulting from the use of a VMM's use of the TSC offsetting or TSC scaling VMCS control (see Chapter 25, "VMX Non-Root Operation"). In this system-wide usage model, the decoder may need to account for the effect of per-VM adjustments in the TSC packets generated in VMX non-root operation and the absence of TSC adjustments in TSC packets generated in VMX root operation. The VMM can supply this information to the decoder.

35.5.2.2 Host-Only Tracing

When trace packets in VMX non-root operation are not desired, the VMM can use VM-entry MSR load list with IA32_RTIT_CTL.TraceEn=0 to disable trace packet generation in guests, set IA32_RTIT_CTL.TraceEn=1 via VM-exit MSR load list.

When tracing only the host, the decoder does not need information about the guests, the VMCS controls for suppressing VMX-specific packets can be set to reduce the packets generated. VMCS packets will still be generated on successful VMPTRLD and in PSB+ generated in the Host, but these will be unused by the decoder.

The packets of interests to a decoder when trace packets are collected for host-only tracing are shown in Table 35-48.

Table 35-48. Packets on VMX Transitions (Host-Only Tracing)

Event	Packets	Description
VM exit	TIP.PGE(HostIP)	The TIP.PGE indicates that trace packet generation is enabled and gives the IP of the first instruction to be executed in VMX root operation. Note, this packet could be preceded by a MODE.Exec packet (Section 35.4.2.8). This is generated only in cases where CS.D or (CS.L & EFER.LMA) change during the transition.
VM entry	TIP.PGD()	The TIP indicates that trace packet generation was disabled. This ensure that all buffered packets are flushed out.

35.5.2.3 Guest-Only Tracing

A VMM can configure trace packet generation while in non-root operation for guests executing normally. This is accomplished by utilizing the MSR load lists across VM exit and VM entry to confine trace packet generation to stay within the guest environment.

For this usage, the VM-entry MSR load list is programmed to turn on trace packet generation. The VM-exit MSR load list is used to clear TraceEn=0 to disable trace packet generation in the host. Further, if it is preferred that the guest packet stream contain no indication that execution was in VMX non-root operation, the VMM should set the VMCS controls described in Table 35-46.

35.5.2.4 Virtualization of Guest Output Packet Streams

Each Intel PT aware guest OS can produce one or more output packet streams to destination addresses specified as guest physical address (GPA) using context-switched IA32_RTIT_OUTPUT_BASE within the guest. The processor generates trace packets to the platform physical address specified in IA32_RTIT_OUTPUT_BASE, and those specified in the ToPA tables. Thus, a VMM that supports Intel PT aware guest OS may wish to virtualize the output configurations of IA32_RTIT_OUTPUT_BASE and ToPA for each trace configuration state of all the guests.

35.5.2.5 Emulation of Intel PT Traced State

If a VMM emulates an element of processor state by taking a VM exit on reads and/or writes to that piece of state, and the state element impacts Intel PT packet generation or values, it may be incumbent upon the VMM to insert or modify the output trace data.

If a VM exit is taken on a guest write to CR3 (including “MOV CR3” as well as task switches), the PIP packet normally generated on the CR3 write will be missing.

To avoid decoder confusion when the guest trace is decoded, the VMM should emulate the missing PIP by writing it into the guest output buffer. If the guest CR3 value is manipulated, the VMM may also need to manipulate the IA32_RTIT_CR3_MATCH value, in order to ensure the trace behavior matches the guest's expectation.

Similarly, if a VMM emulates the TSC value by taking a VM exit on RDTSC, the TSC packets generated in the trace may mismatch the TSC values returned by the VMM on RDTSC. To ensure that the trace can be properly aligned with software logs based on RDTSC, the VMM should either make corresponding modifications to the TSC packet values in the guest trace, or use mechanisms such as TSC offsetting or TSC scaling in place of exiting.

35.5.2.6 TSC Scaling

When TSC scaling is enabled for a guest using Intel PT, the VMM should ensure that the value of Maximum Non-Turbo Ratio[15:8] in MSR_PLATFORM_INFO (MSR 0CEH) and the TSC/”core crystal clock” ratio (EBX/EAX) in CPUID leaf 15H are set in a manner consistent with the resulting TSC rate that will be visible to the VM. This will allow the decoder to properly apply TSC packets, MTC packets (based on the core crystal clock or ART, whose frequency is indicated by CPUID leaf 15H), and CBR packets (which indicate the ratio of the processor frequency to the Max

Non-Turbo frequency). Absent this, or separate indication of the scaling factor, the decoder will be unable to properly track time in the trace. See Section 35.8.3 for details on tracking time within an Intel PT trace.

35.5.2.7 Failed VM Entry

The packets generated by a failed VM entry depend both on the VMCS configuration, as well as on the type of failure. The results to expect are summarized in the table below. Note that packets in *italics* may or may not be generated, depending on implementation choice, and the point of failure.

Table 35-49. Packets on a Failed VM Entry

Usage Model	Entry Configuration	Early Failure (fall through to Next IP)	Late Failure (VM exit)
System-Wide	No MSR load list	TIP (NextIP)	*PIP(Guest CR3, NR=1), TraceEn 0->1 Packets (See Section 35.2.7.3),* PIP(HostCR3, NR=0), TIP(HostIP)
VMM Only	MSR load list disables TraceEn	TIP (NextIP)	*TraceEn 0->1 Packets (See Section 35.2.7.3),* TIP(HostIP)
VM Only	MSR load list Enables TraceEn	None	None

35.5.2.8 VMX Abort

VMX abort conditions take the processor into a shutdown state. On a VM exit that leads to VMX abort, some packets (FUP, PIP) may be generated, but any expected TIP, TIP.PGE, or TIP.PGD may be dropped.

35.6 TRACING AND SMM TRANSFER MONITOR (STM)

SMM Transfer Monitor is a VMM that operates inside SMM while in VMX root operation. An STM operates in conjunction with an executive monitor. The latter operates outside SMM and in VMX root operation. Transitions from the executive monitor or its VMs to the STM are called SMM VM exits. The STM returns from SMM via a VM entry to the VM in VMX non-root operation or the executive monitor in VMX root operation.

Intel PT supports tracing in an STM similar to tracing support for VMX operation as described above in Section 35.7. As a result, on a SMM VM exit resulting from #SMI, TraceEn is not saved and then cleared. Software can save the state of the trace configuration MSRs and clear TraceEn using the MSR load/save lists.

35.7 PACKET GENERATION SCENARIOS

Table 35-50 and Table 35-52 illustrate the packets generated in various scenarios. In the heading row, PacketEn is abbreviated as PktEn, ContextEn as CntxEn. Note that this assumes that TraceEn=1 in IA32_RTIT_CTL, while TriggerEn=1 and Error=0 in IA32_RTIT_STATUS, unless otherwise specified. Entries that do not matter in packet generation are marked "D.C." Packets followed by a "?" imply that these packets depend on additional factors, which are listed in the "Other Dependencies" column.

In Table 35-50, PktEn is evaluated based on TiggerEn & ContextEn & FilterEn & BranchEn.

Table 35-50. Packet Generation under Different Enable Conditions

Case	Operation	PktEn Before	PktEn After	CntxEn After	Other Dependencies	Packets Output
1a	Normal non-jump operation	0	0	D.C.		None
1b	Normal non-jump operation	1	1	1		None
2a	WRMSR/XRSTORS/RSM that changes TraceEn 0 -> 1, with PacketByteCnt >0	0	0	D.C.	*TSC if TSCEn=1; *TMA if TSCEn=MTCEn=1	TSC?, TMA?, CBR

Table 35-50. Packet Generation under Different Enable Conditions (Contd.)

Case	Operation	PktEn Before	PktEn After	CntxEn After	Other Dependencies	Packets Output
2b	WRMSR/XRSTORS/RSM that changes TraceEn 0 -> 1, with PacketByteCnt =0	0	0	D.C.	*TSC if TSCEn=1; *TMA if TSCEn=MTCEn=1	PSB, PSBEND (see Section 35.4.2.17)
2d	WRMSR/XRSTORS/RSM that changes TraceEn 0 -> 1, with PacketByteCnt >0	0	1	1	TSC if TSCEn=1; TMA if TSCEn=MTCEn=1	TSC?, TMA?, CBR, MODE.Exec, TIP.PGE(NLIP)
2e	WRMSR/XRSTORS/RSM that changes TraceEn 0 -> 1, with PacketByteCnt =0	0	1	1		MODE.Exec, TIP.PGE(NLIP), PSB, PSBEND (see Section 35.4.2.8, 35.4.2.7, 35.4.2.13,35.4.2.15, 35.4.2.17)
3a	WRMSR that changes TraceEn 1 -> 0	0	0	D.C.		None
3b	WRMSR that changes TraceEn 1 -> 0	1	0	D.C.		FUP(CLIP), TIP.PGD()
5a	MOV to CR3	0	0	0		None
5f	MOV to CR3	0	0	1	TraceStop if executed in a TraceStop region	PIP(NewCR3,NR?), TraceStop?
5b	MOV to CR3	0	1	1	*PIP.NR=1 if not in root operation, and "Conceal VMX non-root operation from Intel PT" execution control = 0 *MODE.Exec if the mode has changed since the last MODE.Exec, or if no MODE.Exec since last PSB	PIP(NewCR3, NR?), MODE.Exec?, TIP.PGE(NLIP)
5c	MOV to CR3	1	0	0		TIP.PGD()
5e	MOV to CR3	1	0	1	*PIP.NR=1 if not in root operation, and "Conceal VMX non-root operation from Intel PT" execution control = 0 *TraceStop if executed in a TraceStop region	PIP(NewCR3, NR?), TIP.PGE(NLIP), TraceStop?
5d	MOV to CR3	1	1	1	*PIP.NR=1 if not in root operation, and "Conceal VMX non-root operation from Intel PT" execution control = 0	PIP(NewCR3, NR?)
6a	Unconditional direct near jump	0	0	D.C.		None
6b	Unconditional direct near jump	1	0	1	TraceStop if BLIP is in a TraceStop region	TIP.PGD(BLIP), TraceStop?
6c	Unconditional direct near jump	0	1	1	MODE.Exec if the mode has changed since the last MODE.Exec, or if no MODE.Exec since last PSB	MODE.Exec?, TIP.PGE(BLIP)
6d	Unconditional direct near jump	1	1	1		None
7a	Conditional taken jump or compressed RET that does not fill up the internal TNT buffer	0	0	D.C.		None

Table 35-50. Packet Generation under Different Enable Conditions (Contd.)

Case	Operation	PktEn Before	PktEn After	CntxEn After	Other Dependencies	Packets Output
7b	Conditional taken jump or compressed RET	0	1	1	MODE.Exec if the mode has changed since the last MODE.Exec, or if no MODE.Exec since last PSB	MODE.Exec?, TIP.PGE(BLIP)
7e	Conditional taken jump or compressed RET, with empty TNT buffer	1	0	1	TraceStop if BLIP is in a TraceStop region	TIP.PGD(), TraceStop?
7f	Conditional taken jump or compressed RET, with non-empty TNT buffer	1	0	1	TraceStop if BLIP is in a TraceStop region	TNT, TIP.PGD(), TraceStop?
7d	Conditional taken jump or compressed RET that fills up the internal TNT buffer	1	1	1		TNT
8a	Conditional non-taken jump	0	0	D.C.		None
8d	Conditional not-taken jump that fills up the internal TNT buffer	1	1	1		TNT
9a	Near indirect jump (JMP, CALL, or uncompressed RET)	0	0	D.C.		None
9b	Near indirect jump (JMP, CALL, or uncompressed RET)	0	1	1	MODE.Exec if the mode has changed since the last MODE.Exec, or if no MODE.Exec since last PSB	MODE.Exec?, TIP.PGE(BLIP)
9c	Near indirect jump (JMP, CALL, or uncompressed RET)	1	0	1	TraceStop if BLIP is in a TraceStop region	TIP.PGD(BLIP), TraceStop?
9d	Near indirect jump (JMP, CALL, or uncompressed RET)	1	1	1		TIP(BLIP)
10a	Far Branch (CALL/JMP/RET)	0	0	0		None
10f	Far Branch (CALL/JMP/RET)	0	0	1	*PIP if CR3 is updated (i.e., task switch), and OS=1; *PIP.NR=1 if destination is not root operation, and "Conceal VMX non-root operation from Intel PT" execution control = 0; *TraceStop if BLIP is in a TraceStop region	PIP(new CR3, NR?), TraceStop?
10b	Far Branch (CALL/JMP/RET)	0	1	1	*PIP if CR3 is updated (i.e., task switch), and OS=1; *PIP.NR=1 if destination is not root operation, and "Conceal VMX non-root operation from Intel PT" execution control = 0; *MODE.Exec if the mode has changed since the last MODE.Exec, or if no MODE.Exec since last PSB	PIP(new CR3, NR?), MODE.Exec?, TIP.PGE(BLIP)
10c	Far Branch (CALL/JMP/RET)	1	0	0		TIP.PGD()

Table 35-50. Packet Generation under Different Enable Conditions (Contd.)

Case	Operation	PktEn Before	PktEn After	CntxEn After	Other Dependencies	Packets Output
10d	Far Branch (CALL/JMP/RET)	1	0	1	*PIP if CR3 is updated (i.e., task switch), and OS=1; *PIP.NR=1 if destination is not root operation, and "Conceal VMX non-root operation from Intel PT" execution control = 0; *TraceStop if BLIP is in a TraceStop region	PIP(new CR3, NR?), TIP.PGD(BLIP), TraceStop?
10e	Far Branch (CALL/JMP/RET)	1	1	1	*PIP if CR3 is updated (i.e., task switch), and OS=1 *PIP.NR=1 if destination is not root operation, and "Conceal VMX non-root operation from Intel PT" execution control = 0; * MODE.Exec if the operation changes CS.L/D or IA32_EFER.LMA	PIP(NewCR3, NR?)?, MODE.Exec?, TIP(BLIP)
11a	HW Interrupt	0	0	0		None
11f	HW Interrupt	0	0	1	*PIP if CR3 is updated (i.e., task switch), and OS=1; *PIP.NR=1 if destination is not root operation, and "Conceal VMX non-root operation from Intel PT" execution control = 0; *TraceStop if BLIP is in a TraceStop region	PIP(new CR3, NR?), TraceStop?
11b	HW Interrupt	0	1	1	*PIP if CR3 is updated (i.e., task switch), and OS=1; *PIP.NR=1 if destination is not root operation, and "Conceal VMX non-root operation from Intel PT" execution control = 0; * MODE.Exec if the mode has changed since the last MODE.Exec, or if no MODE.Exec since last PSB	PIP(new CR3, NR?), MODE.Exec?, TIP.PGE(BLIP)
11c	HW Interrupt	1	0	0		FUP(NLIP), TIP.PGD()
11d	HW Interrupt	1	0	1	* PIP if CR3 is updated (i.e., task switch), and OS=1 *PIP.NR=1 if destination is not root operation, and "Conceal VMX non-root operation from Intel PT" execution control = 0; *TraceStop if BLIP is in a TraceStop region	FUP(NLIP), PIP(NewCR3, NR?)?, TIP.PGD(BLIP), TraceStop

Table 35-50. Packet Generation under Different Enable Conditions (Contd.)

Case	Operation	PktEn Before	PktEn After	CntxEn After	Other Dependencies	Packets Output
11e	HW Interrupt	1	1	1	* PIP if CR3 is updated (i.e., task switch), and OS=1 *PIP.NR=1 if destination is not root operation, and "Conceal VMX non-root operation from Intel PT" execution control = 0; * MODE.Exec if the operation changes CS.L/D or IA32_EFER.LMA	FUP(NLIP), PIP(NewCR3, NR?)?, MODE.Exec?, TIP(BLIP)
12a	SW Interrupt	0	0	0		None
12f	SW Interrupt	0	0	1	* PIP if CR3 is updated (i.e., task switch), and OS=1 *PIP.NR=1 if destination is not root operation, and "Conceal VMX non-root operation from Intel PT" execution control = 0; *TraceStop if BLIP is in a TraceStop region	PIP(NewCR3, NR?)?, TraceStop?
12b	SW Interrupt	0	1	1	* PIP if CR3 is updated (i.e., task switch), and OS=1 *PIP.NR=1 if destination is not root operation, and "Conceal VMX non-root operation from Intel PT" execution control = 0; *MODE.Exec if the mode has changed since the last MODE.Exec, or if no MODE.Exec since last PSB	PIP(NewCR3, NR?)?, MODE.Exec?, TIP.PGE(BLIP)
12c	SW Interrupt	1	0	0		FUP(CLIP), TIP.PGD()
12d	SW Interrupt	1	0	1	* PIP if CR3 is updated (i.e., task switch), and OS=1 *PIP.NR=1 if destination is not root operation, and "Conceal VMX non-root operation from Intel PT" execution control = 0; *TraceStop if BLIP is in a TraceStop region	FUP(CLIP), PIP(NewCR3, NR?)?, TIP.PGD(BLIP), TraceStop?
12e	SW Interrupt	1	1	1	* PIP if CR3 is updated (i.e., task switch), and OS=1 *PIP.NR=1 if destination is not root operation, and "Conceal VMX non-root operation from Intel PT" execution control = 0; * MODE.Exec if the operation changes CS.L/D or IA32_EFER.LMA	FUP(CLIP), PIP(NewCR3, NR?)?, MODE.Exec?, TIP(BLIP)
13a	Exception/Fault	0	0	0		None

Table 35-50. Packet Generation under Different Enable Conditions (Contd.)

Case	Operation	PktEn Before	PktEn After	CntxEn After	Other Dependencies	Packets Output
13f	Exception/Fault	0	0	1	* PIP if CR3 is updated (i.e., task switch), and OS=1 *PIP.NR=1 if destination is not root operation, and "Conceal VMX non-root operation from Intel PT" execution control = 0; *TraceStop if BLIP is in a TraceStop region	PIP(NewCR3, NR?)?, TraceStop?
13b	Exception/Fault	0	1	1	* PIP if CR3 is updated (i.e., task switch), and OS=1 *PIP.NR=1 if destination is not root operation, and "Conceal VMX non-root operation from Intel PT" execution control = 0; *MODE.Exec if the mode has changed since the last MODE.Exec, or if no MODE.Exec since last PSB	PIP(NewCR3, NR?)?, MODE.Exec?, TIP.PGE(BLIP)
13c	Exception/Fault	1	0	0		FUP(CLIP), TIP.PGD()
13d	Exception/Fault	1	0	1	* PIP if CR3 is updated (i.e., task switch), and OS=1 *PIP.NR=1 if destination is not root operation, and "Conceal VMX non-root operation from Intel PT" execution control = 0; *TraceStop if BLIP is in a TraceStop region	FUP(CLIP), PIP(NewCR3, NR?)?, TIP.PGD(BLIP), TraceStop?
13e	Exception/Fault	1	1	1	* PIP if CR3 is updated (i.e., task switch), and OS=1 *PIP.NR=1 if destination is not root operation, and "Conceal VMX non-root operation from Intel PT" execution control = 0; * MODE.Exec if the operation changes CS.L/D or IA32_EFER.LMA	FUP(CLIP), PIP(NewCR3, NR?)?, MODE.Exec?, TIP(BLIP)
14a	SMI (TraceEn cleared)	0	0	D.C.		None
14b	SMI (TraceEn cleared)	1	0	0		FUP(SMRAM,LIP), TIP.PGD()
14f	SMI (TraceEn cleared)	1	0	1		NA
14c	SMI (TraceEn cleared)	1	1	1		NA
15a	RSM, TraceEn restored to 0	0	0	0		None
15b	RSM, TraceEn restored to 1	0	0	D.C.		See WRMSR cases for packets on enable

Table 35-50. Packet Generation under Different Enable Conditions (Contd.)

Case	Operation	PktEn Before	PktEn After	CntxEn After	Other Dependencies	Packets Output
15c	RSM, TraceEn restored to 1	0	1	1		See WRMSR cases for packets on enable. FUP/TIP.PGE IP is SMRAM.LIP
15e	RSM (TraceEn=1, goes to shutdown)	1	0	0		None
15f	RSM (TraceEn=1, goes to shutdown)	1	0	1		None
15d	RSM (TraceEn=1, goes to shutdown)	1	1	1		None
16i	Vmext	0	0	0		None
16a	Vmext	0	0	1	*PIP if OF=1, and "Conceal VM exits from Intel PT" execution control = 0; *TraceStop if VMCSh.LIP is in a TraceStop region	PIP(HostCR3, NR=0)?, TraceStop?
16b	VM exit, MSR list sets TraceEn=1	0	0	0		See WRMSR cases for packets on enable. FUP IP is VMCSh.LIP
16c	VM exit, MSR list sets TraceEn=1	0	1	1		See WRMSR cases for packets on enable. FUP/TIP.PGE IP is VMCSh.LIP
16e	VM exit	0	1	1	*PIP if OF=1, and "Conceal VM exits from Intel PT" execution control = 0; *MODE.Exec if the value is different, since last TIP.PGD	PIP(HostCR3, NR=0)?, MODE.Exec?, TIP.PGE(VMCSh.LIP)
16f	VM exit, MSR list clears TraceEn=0	1	0	0	*PIP if OF=1, and "Conceal VM exits from Intel PT" execution control = 0;	FUP(VMCSg.LIP), PIP(HostCR3, NR=0)?, TIP.PGD
16j	VM exit, ContextEN 1->0	1	0	0		FUP(VMCSg.LIP), TIP.PGD
16g	VM exit	1	0	1	*PIP if OF=1, and "Conceal VM exits from Intel PT" execution control = 0; *TraceStop if VMCSh.LIP is in a TraceStop region	FUP(VMCSg.LIP), PIP(HostCR3, NR=0)?, TIP.PGD(VMCSh.LIP), TraceStop?
16h	VM exit	1	1	1	*PIP if OF=1, and "Conceal VM exits from Intel PT" execution control = 0; *MODE.Exec if the value is different, since last TIP.PGD	FUP(VMCSg.LIP), PIP(HostCR3, NR=0)?, MODE.Exec, TIP(VMCSh.LIP)
17a	VM entry	0	0	0		None
17b	VM entry	0	0	1	*PIP if OF=1, and "Conceal VM entries from Intel PT" execution control = 0; *TraceStop if VMCSg.LIP is in a TraceStop region	PIP(GuestCR3, NR=1)?, TraceStop?
17c	VM entry, MSR load list sets TraceEn=1	0	0	1		See WRMSR cases for packets on enable. FUP IP is VMCSg.LIP

Table 35-50. Packet Generation under Different Enable Conditions (Contd.)

Case	Operation	PktEn Before	PktEn After	CntxEn After	Other Dependencies	Packets Output
17d	VM entry, MSR load list sets TraceEn=1	0	1	1		See WRMSR cases for packets on enable. FUP/TIP.PGE IP is VMCSg.LIP
17f	VM entry, FilterEN 0->1	0	1	1	*PIP if OF=1, and "Conceal VM entries from Intel PT" execution control = 0; *MODE.Exec if the value is different, since last TIP.PGD	PIP(GuestCR3, NR=1)?, MODE.Exec?, TIP.PGE(VMCSg.LIP)
17j	VM entry, ContextEN 0->1	0	1	1	*MODE.Exec if the value is different, since last TIP.PGD	MODE.Exec, TIP.PGE(VMCSg.LIP)
17g	VM entry, MSR list clears TraceEn=0	1	0	0	*PIP if OF=1, and "Conceal VM entries from Intel PT" execution control = 0;	PIP(GuestCR3, NR=1)?, TIP.PGD
17h	VM entry	1	0	1	*PIP if OF=1, and "Conceal VM entries from Intel PT" execution control = 0; *TraceStop if VMCSg.LIP is in a TraceStop region	PIP(GuestCR3, NR=1)?, TIP.PGD(VMCSg.LIP), TraceStop?
17i	VM entry	1	1	1	*PIP if OF=1, and "Conceal VM entries from Intel PT" execution control = 0; *MODE.Exec if the value is different, since last TIP.PGD	PIP(GuestCR3, NR=1)?, MODE.Exec, TIP(VMCSg.LIP)
20a	EENTER/ERESUME to non-debug enclave	0	0	0		None
20c	EENTER/ERESUME to non-debug enclave	1	0	0		FUP(CLIP), TIP.PGD()
21a	EEXIT from non-debug enclave	0	0	D.C.		None
21b	EEXIT from non-debug enclave	0	1	1	*MODE.Exec if the value is different, since last TIP.PGD	MODE.Exec?, TIP.PGE(BLIP)
22a	AEX/EEE from non-debug enclave	0	0	D.C.		None
22b	AEX/EEE from non-debug enclave	0	1	1	*MODE.Exec if the value is different, since last TIP.PGD	MODE.Exec?, TIP.PGE(AEP.LIP)
23a	EENTER/ERESUME to debug enclave	0	0	D.C.		None
23b	EENTER/ERESUME to debug enclave	0	1	1	*MODE.Exec if the value is different, since last TIP.PGD	MODE.Exec?, TIP.PGE(BLIP)
23c	EENTER/ERESUME to debug enclave	1	0	0		FUP(CLIP), TIP.PGD()
23d	EENTER/ERESUME to debug enclave	0	0	1	*TraceStop if BLIP is in a TraceStop region	FUP(CLIP), TIP.PGD(BLIP), TraceStop?
23e	EENTER/ERESUME to debug enclave	1	1	1		FUP(CLIP), TIP(BLIP)
24f	EEXIT from debug enclave	0	0	D.C.		None
24b	EEXIT from debug enclave	0	1	1	*MODE.Exec if the value is different, since last TIP.PGD	MODE.Exec?, TIP.PGE(BLIP)
24d	EEXIT from debug enclave	1	0	1	*TraceStop if BLIP is in a TraceStop region	FUP(CLIP), TIP.PGD(BLIP), TraceStop?

Table 35-50. Packet Generation under Different Enable Conditions (Contd.)

Case	Operation	PktEn Before	PktEn After	CntxEn After	Other Dependencies	Packets Output
24e	EEXIT from debug enclave	1	1	1		FUP(CLIP), TIP(BLIP)
25a	AEX/EEE from debug enclave	0	0	D.C.		None
25b	AEX/EEE from debug enclave	0	1	1	*MODE.Exec if the value is different, since last TIP.PGD	MODE.Exec?, TIP.PGE(AEP.LIP)
25d	AEX/EEE from debug enclave	1	0	1	*For AEX, FUP IP could be NLIP, for trap-like events	FUP(CLIP), TIP.PGD(AEP.LIP)
25e	AEX/EEE from debug enclave	1	1	1	*MODE.Exec if the value is different, since last TIP.PGD *For AEX, FUP IP could be NLIP, for trap-like events	FUP(CLIP), MODE.Exec?, TIP(AEP.LIP)
26a	XBEGIN/XACQUIRE	0	0	D.C.		None
26d	XBEGIN/XACQUIRE that does not set InTX	1	1	1		None
26e	XBEGIN/XACQUIRE that sets InTX	1	1	1		MODE(InTX=1, TXAbort=0), FUP(CLIP)
27a	XEND/XRELEASE	0	0	D.C.		None
27d	XEND/XRELEASE that does not clear InTX	1	1	1		None
27e	XEND/XRELEASE that clears InTX	1	1	1		MODE(InTX=0, TXAbort=0), FUP(CLIP)
28a	XABORT(Async XAbort, or other)	0	0	0		None
28e	XABORT(Async XAbort, or other)	0	0	1	*TraceStop if BLIP is in a TraceStop region	MODE(InTX=0, TXAbort=1), TraceStop?
28b	XABORT(Async XAbort, or other)	0	1	1		MODE(InTX=0, TXAbort=1), TIP.PGE(BLIP)
28c	XABORT(Async XAbort, or other)	1	0	1	*TraceStop if BLIP is in a TraceStop region	MODE(InTX=0, TXAbort=1), TIP.PGD (BLIP), TraceStop?
28d	XABORT(Async XAbort, or other)	1	1	1		MODE(InTX=0, TXAbort=1), FUP(CLIP), TIP(BLIP)
30a	INIT (BSP)	0	0	0		None
30b	INIT (BSP)	0	0	1	*TraceStop if RESET.LIP is in a TraceStop region	BIP(0), TraceStop?
30c	INIT (BSP)	0	1	1	* MODE.Exec if the value is different, since last TIP.PGD	MODE.Exec?, PIP(0), TIP.PGE(ResetLIP)
30d	INIT (BSP)	1	0	0		FUP(NLIP), TIP.PGD()
30e	INIT (BSP)	1	0	1	* PIP if OS=1 *TraceStop if RESET.LIP is in a TraceStop region	FUP(NLIP), PIP(0), TIP.PGD, TraceStop?

Table 35-50. Packet Generation under Different Enable Conditions (Contd.)

Case	Operation	PktEn Before	PktEn After	CntxEn After	Other Dependencies	Packets Output
30f	INIT (BSP)	1	1	1	* MODE.Exec if the mode has changed since the last MODE.Exec, or if no MODE.Exec since last PSB * PIP if OS=1	FUP(NLIP), PIP(0)?, MODE.Exec?, TIP(ResetLIP)
31a	INIT (AP, goes to wait-for-SIPI)	0	D.C.	D.C.		None
31b	INIT (AP, goes to wait-for-SIPI)	1	D.C.	D.C.	* PIP if OS=1	FUP(NLIP), PIP(0)
32a	SIPI	0	0	0		None
32c	SIPI	0	1	1	* MODE.Exec if the mode has changed since the last MODE.Exec, or if no MODE.Exec since last PSB	MODE.Exec?, TIP.PGE(SIPI-LIP)
32d	SIPI	1	0	0		TIP.PGD
32e	SIPI	1	0	1	*TraceStop if SIPI LIP is in a TraceStop region	TIP.PGD(SIPILIP); TraceS-top?
32f	SIPI	1	1	1	* MODE.Exec if the mode has changed since the last MODE.Exec, or if no MODE.Exec since last PSB	MODE.Exec?, TIP(SIPILIP)
33a	MWAIT (to C0)	D.C.	D.C.	D.C.		None
33b	MWAIT (to higher-numbered C-State, packet sent on wake)	D.C.	D.C.	D.C.	*TSC if TSCEn=1 *TMA if TSCEn=MTCEn=1	TSC?, TMA?, CBR

In Table 35-52, PktEn is evaluated based on (TiggerEn & ContextEn & FilterEn & BranchEn & PwrEvtEn).

Table 35-51. PwrEvtEn and PTWEn Packet Generation under Different Enable Conditions

Case	Operation	PktEn Before	PktEn After	CntxEn After	Other Dependencies	Packets Output
16.1	MWAIT or I/O redir to MWAIT, gets #UD or #GP fault	dc	dc	dc		None
16.2	MWAIT or I/O redir to MWAIT, VM exits	dc	dc	dc		See VM exit examples (16[a-z] in Table 35-50) for BranchEn packets.
16.3	MWAIT or I/O redir to MWAIT, requests C0, or monitor not armed, or VMX virtual-interrupt delivery	dc	dc	dc		None
16.4a	MWAIT(X) or I/O redir to MWAIT, goes to C-state Y (Y>0)	dc	0	0		PWRE(Cx), EXSTOP
16.4b	MWAIT(X) or I/O redir to MWAIT, goes to C-state Y (Y>0)	dc	dc	1		MWAIT(Cy), PWRE(Cx), EXSTOP(IP), FUP(CLIP)
16.5a	MWAIT(X) or I/O redir to MWAIT, Pending event after resolving to go to C-state Y (Y>0)	dc	0	0	* TSC if TSCEn=1 * TMA if TSCEn=MTCEn=1	PWRE(Cx), EXSTOP, TSC?, TMA?, CBR, PWRX(LCC, DCC, 0)
16.5b	MWAIT(X) or I/O redir to MWAIT, Pending event after resolving to go to C-state Y (Y>0)	dc	dc	1	* TSC if TSCEn=1 * TMA if TSCEn=MTCEn=1	PWRE(Cx), EXSTOP(IP), FUP(CLIP), TSC?, TMA?, CBR, PWRX(LCC, DCC, 0)

Table 35-51. PwrEvtEn and PTWEn Packet Generation under Different Enable Conditions (Contd.)

Case	Operation	PktEn Before	PktEn After	CntxEn After	Other Dependencies	Packets Output
16.6a	MWAIT(5) or I/O redir to MWAIT, other thread(s) in core in C0/C1	dc	0	0		PWRE(C1), EXSTOP
16.6b	MWAIT(5) or I/O redir to MWAIT, other thread(s) in core in C0/C1	dc	dc	1		MWAIT(5), PWRE(C1), EXSTOP(IP), FUP(CLIP)
16.9a	HLT, Triple-fault shutdown, #MC with CR4.MCE=0, RSM to Cx (x>0)	dc	0	0		PWRE(C1), EXSTOP
16.9b	HLT, Triple-fault shutdown, #MC with CR4.MCE=1, RSM to Cx (x>0)	dc	dc			PWRE(C1), EXSTOP(IP), FUP(CLIP)
16.10a	VMX abort	dc	0	0		See "VMX Abort" (cases 16* and 18* in Table 35-50) for BranchEn packets that precede PWRE(C1), EXSTOP
16.10b	VMX abort	dc	dc	1		See "VMX Abort" (cases 16* and 18* in Table 35-50) for BranchEn packets that precede PWRE(C1), EXSTOP(IP), FUP(CLIP)
16.11a	RSM to Shutdown	dc	0	0		See "RSM to Shutdown" (cases 15[def] in Table 35-50) for BranchEn packets that precede PWRE(C1), EXSTOP
16.11b	RSM to Shutdown	dc	dc	1		See "RSM to Shutdown" (cases 15[def] in Table 35-50) for BranchEn packets that precede PWRE(C1), EXSTOP(IP), FUP(CLIP)
16.12a	INIT (BSP)	dc	0	0		See "INIT (BSP)" (cases 30[a-z] in Table 35-50) for packets that BranchEn precede PWRE(C1), EXSTOP
16.12b	INIT (BSP)	dc	dc	1		See "INIT (BSP)" (cases 30[a-z] in Table 35-50) for packets that BranchEn precede PWRE(C1), EXSTOP(IP), FUP(NLIP)

Table 35-51. PwrEvtEn and PTWEn Packet Generation under Different Enable Conditions (Contd.)

Case	Operation	PktEn Before	PktEn After	CntxEn After	Other Dependencies	Packets Output
16.13a	INIT (AP, goes to Wait-for-SIPI)	dc	0	0		See "INIT (AP, goes to Wait-for-SIPI)" (cases 31[a-z] in Table 35-50) for BranchEn packets that precede PWRE(C1), EXSTOP
16.13b	INIT (AP, goes to Wait-for-SIPI)	dc	dc	1		See "INIT (AP, goes to Wait-for-SIPI)" (cases 31[a-z] in Table 35-50) for BranchEn packets that precede PWRE(C1), EXSTOP(IP), FUP(NLIP)
16.14a	Hardware Duty Cycling (HDC)	dc	0	0	* TSC if TSCEn=1 * TMA if TSCEn=MTCEn=1	PWRE(HW, C6), EXSTOP, TSC?, TMA?, CBR, PWRX(CC6, CC6, 0x8)
16.14b	Hardware Duty Cycling (HDC)	dc	dc	1	* TSC if TSCEn=1 * TMA if TSCEn=MTCEn=1	PWRE(HW, C6), EXSTOP(IP), FUP(NLIP), TSC?, TMA?, CBR, PWRX(CC6, CC6, 0x8)
16.15a	VM entry to HLT or Shutdown	dc	0	0		See "VM entry" (cases 17[a-z] in Table 35-50) for BranchEn packets that precede. PWRE(C1), EXSTOP
16.15b	VM entry to HLT or Shutdown	dc	dc	1		See "VM entry" (cases 17[a-z] in Table 35-50) for BranchEn packets that precede. PWRE(C1), EXSTOP(IP), FUP(CLIP)
16.16a	EIST in C0, S1/TM1/TM2, or STPCLK#	dc	0	0	* TSC if TSCEn=1 * TMA if TSCEn=MTCEn=1	EXSTOP, TSC?, TMA?, CBR
16.16b	EIST in C0, S1/TM1/TM2, or STPCLK#	dc	dc	1	* TSC if TSCEn=1 * TMA if TSCEn=MTCEn=1	EXSTOP(IP), FUP(NLIP), TSC?, TMA?, CBR
16.17	EIST in Cx (x>0)	dc	dc	dc		None
16.18	INTR during Cx (x>0)	dc	dc	dc	* TSC if TSCEn=1 * TMA if TSCEn=MTCEn=1	TSC?, TMA?, CBR, PWRX(LCC, DCC, 0x1) See "HW Interrupt" (cases 11[a-z] in Table 35-50) for BranchEn packets that follow.

Table 35-51. PwrEvtEn and PTWEn Packet Generation under Different Enable Conditions (Contd.)

Case	Operation	PktEn Before	PktEn After	CntxEn After	Other Dependencies	Packets Output
16.18	SMI during Cx (x>0)	dc	dc	dc	* TSC if TSCEn=1 * TMA if TSCEn=MTCEn=1	TSC?, TMA?, CBR, PWRX(LCC, DCC, 0) See "HW Interrupt" (cases 14[a-z] in Table 35-50) for BranchEn packets that follow.
16.19	NMI during Cx (x>0)	dc	dc	dc	* TSC if TSCEn=1 * TMA if TSCEn=MTCEn=1	TSC?, TMA?, CBR, PWRX(LCC, DCC, 0) See "HW Interrupt" (cases 11[a-z] in Table 35-50) for BranchEn packets that follow.
16.2	Store to monitored address during Cx (x>0)	dc	dc	dc	* TSC if TSCEn=1 * TMA if TSCEn=MTCEn=1	TSC?, TMA?, CBR, PWRX(LCC, DCC, 0x4)
16.22	#MC, IERR, TSC deadline timer expiration, or APIC counter under-flow during Cx (x>0)	dc	dc	dc	* TSC if TSCEn=1 * TMA if TSCEn=MTCEn=1	TSC?, TMA?, CBR, PWRX(LCC, DCC, 0)

In Table 35-52, PktEn is evaluated based on (TiggerEn & ContextEn & FilterEn & BranchEn & PTWEn).

Table 35-52. PwrEvtEn and PTWEn Packet Generation under Different Enable Conditions

Case	Operation	PktEn Before	PktEn After	CntxEn After	Other Dependencies	Packets Output
16.24a	PTWRITE rm32/64, no fault	dc	dc	dc		None
16.24b	PTWRITE rm32/64, no fault	dc	0	0		None
16.24d	PTWRITE rm32, no fault	dc	1	1	* FUP, IP=1 if FUPonPTW=1	PTW(IP=1?, 4B, rm32_value), FUP(CLIP)?
16.24e	PTWRITE rm64, no fault	dc	1	1	* FUP, IP=1 if FUPonPTW=1	PTW(IP=1?, 8B, rm64_value), FUP(CLIP)?
16.25a	PTWRITE mem32/64, fault	dc	dc	dc		See "Exception/fault" (cases 13[a-z] in Table 35-50) for BranchEn packets.

35.8 SOFTWARE CONSIDERATIONS

35.8.1 Tracing SMM Code

Nothing prevents an SMM handler from configuring and enabling packet generation for its own use. As described in Section , SMI will always clear TraceEn, so the SMM handler would have to set TraceEn in order to enable tracing. There are some unique aspects and guidelines involved with tracing SMM code, which follows:

1. SMM should save away the existing values of any configuration MSRs that SMM intends to modify for tracing. This will allow the non-SMM tracing context to be restored before RSM.
2. It is recommended that SMM wait until it sets CSbase to 0 before enabling packet generation, to avoid possible LIP vs RIP confusion.
3. Packet output cannot be directed to SMRR memory, even while tracing in SMM.
4. Before performing RSM, SMM should take care to restore modified configuration MSRs to the values they had immediately after #SMI. This involves first disabling packet generation by clearing TraceEn, then restoring any other configuration MSRs that were modified.
5. RSM
   - Software must ensure that TraceEn=0 at the time of RSM. Tracing RSM is not a supported usage model, and the packets generated by RSM are undefined.
   - For processors on which Intel PT and LBR use are mutually exclusive (see Section 35.3.1.2), any RSM during which TraceEn is restored to 1 will suspend any LBR or BTS logging.

35.8.2 Cooperative Transition of Multiple Trace Collection Agents

A third-party trace-collection tool should take into consideration the fact that it may be deployed on a processor that supports Intel PT but may run under any operating system.

In such a deployment scenario, Intel recommends that tool agents follow similar principles of cooperative transition of single-use hardware resources, similar to how performance monitoring tools handle performance monitoring hardware:

- Respect the "in-use" ownership of an agent who already configured the trace configuration MSRs, see architectural MSRs with the prefix "IA32_RTIT_" in Chapter 2, "Model-Specific Registers (MSRs)" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 4*, where "in-use" can be determined by reading the "enable bits" in the configuration MSRs.

- Relinquish ownership of the trace configuration MSRs by clearing the "enabled bits" of those configuration MSRs.

35.8.3 Tracking Time

This section describes the relationships of several clock counters whose update frequencies reside in different domains that feed into the timing packets. To track time, the decoder also needs to know the regularity or irregularity of the occurrences of various timing packets that store those clock counters.

Intel PT provides time information for three different but related domains:

- Processor timestamp counter

  This counter increments at the max non-turbo or P1 frequency, and its value is returned on a RDTSC. Its frequency is fixed. The TSC packet holds the lower 7 bytes of the timestamp counter value. The TSC packet occurs occasionally and are much less frequent than the frequency of the time stamp counter. The timestamp counter will continue to increment when the processor is in deep C-States, with the exception of processors reporting CPUID.80000007H:EDX.InvariantTSC[bit 8] =0.
- Core crystal clock

  The ratio of the core crystal clock to timestamp counter frequency is known as P, and can calculating CPUID.15H:EBX[31:0] / CPUID.15H:EAX[31:0]. The frequency of the core crystal clock is fixed and lower than that of the timestamp counter. The periodic MTC packet is generated based on software-selected multiples of the crystal clock frequency. The MTC packet is expected to occur more frequently than the TSC packet.
- Processor core clock

  The processor core clock frequency can vary due to P-state and thermal conditions. The CYC packet provides elapsed time as measured in processor core clock cycles relative to the last CYC packet.

A decoder can use all or some combination of these packets to track time at different resolutions throughout the trace packets.

35.8.3.1 Time Domain Relationships

The three domains are related by the following formula:

```
TimeStampValue = (CoreCrystalClockValue * P) + AdjustedProcessorCycles + Software_Offset;
```

The CoreCrystalClockValue can provide the coarse-grained component of the TSC value. P, or the TSC/"core crystal clock" ratio, can be derived from CPUID leaf 15H, as described in Section 35.8.3.

The AdjustedProcessorCycles component provides the fine-grained distance from the rising edge of the last core crystal clock. Specifically, it is a cycle count in the same frequency as the timestamp counter from the last crystal clock rising edge. The value is adjusted based on the ratio of the processor core clock frequency to the Maximum Non-Turbo (or P1) frequency.

The Software_Offsets component includes software offsets that are factored into the timestamp value, such as IA32_TSC_ADJUST.

35.8.3.2 Estimating TSC within Intel PT

For many usages, it may be useful to have an estimated timestamp value for all points in the trace. The formula provided in Section 35.8.3.1 above provides the framework for how such an estimate can be calculated from the various timing packets present in the trace.

The TSC packet provides the precise timestamp value at the time it is generated; however, TSC packets are infrequent, and estimates of the current timestamp value based purely on TSC packets are likely to be very inaccurate for this reason. In order to get more precise timing information between TSC packets, CYC packets and/or MTC packets should be enabled.

MTC packets provide incremental updates of the CoreCrystalClockValue. On processors that support CPUID leaf 15H, the frequency of the timestamp counter and the core crystal clock is fixed, thus MTC packets provide a means to update the running timestamp estimate. Between two MTC packets A and B, the number of crystal clock cycles passed is calculated from the 8-bit payloads of respective MTC packets:

(CTC_B - CTC_A), where CTC_i = MTC_i[15:8] << IA32_RTIT_CTL.MTCFreq and i = A, B.

The time from a TSC packet to the subsequent MTC packet can be calculated using the TMA packet that follows the TSC packet. The TMA packet provides both the crystal clock value (lower 16 bits, in the CTC field) and the AdjustedProcessorCycles value (in the FastCounter field) that can be used in the calculation of the corresponding core crystal clock value of the TSC packet.

When the next MTC after a pair of TSC/TMA is seen, the number of crystal clocks passed since the TSC packet can be calculated by subtracting the TMA.CTC value from the time indicated by the MTC_{Next} packet by

CTC_{Delta}[15:0] = (CTC_{Next}[15:0] - TMA.CTC[15:0]), where CTC_{Next} = $MTC_{Payload}$ << IA32_RTIT_CTL.MTCFreq.

The TMA.FastCounter field provides the fractional component of the TSC packet into the next crystal clock cycle.

CYC packets can provide further precision of an estimated timestamp value to many non-timing packets, by providing an indication of the time passed between other timing packets (MTCs or TSCs).

When enabled, CYC packets are sent preceding each CYC-eligible packet, and provide the number of processor core clock cycles that have passed since the last CYC packet. Thus between MTCs and TSCs, the accumulated CYC values can be used to estimate the adjusted_processor_cycles component of the timestamp value. The accumulated CPU cycles will have to be adjusted to account for the difference in frequency between the processor core clock and the P1 frequency. The necessary adjustment can be estimated using the core:bus ratio value given in the CBR packet, by multiplying the accumulated cycle count value by $P1/CBR_{payload}$.

Note that stand-alone TSC packets (that is, TSC packets that are not a part of a PSB+) are typically generated only when generation of other timing packets (MTCs and CYCs) has ceased for a period of time. Example scenarios include when Intel PT is re-enabled, or on wake after a sleep state. Thus any calculation of ART or cycle time leading up to a TSC packet will likely result in a discrepancy, which the TSC packet serves to correct.

A greater level of precision may be achieved by calculating the CPU clock frequency, see Section 35.8.3.4 below for a method to do so using Intel PT packets.

CYCs can be used to estimate time between TSCs even without MTCs, though this will likely result in a reduction in estimated TSC precision.

35.8.3.3 VMX TSC Manipulation

When software executes in non-Root operation, additional offset and scaling factors may be applied to the TSC value. These are optional, but may be enabled via VMCS controls on a per-VM basis. See Chapter 25, “VMX Non-Root Operation” for details on VMX TSC offsetting and TSC scaling.

Like the value returned by RDTSC, TSC packets will include these adjustments, but other timing packets (such as MTC, CYC, and CBR) are not impacted. In order to use the algorithm above to estimate the TSC value when TSC scaling is in use, it will be necessary for software to account for the scaling factor. See Section 35.5.2.6 for details.

35.8.3.4 Calculating Frequency with Intel PT

Because Intel PT can provide both wall-clock time and processor clock cycle time, it can be used to measure the processor core clock frequency. Either TSC or MTC packets can be used to track the wall-clock time. By using CYC packets to count the number of processor core cycles that pass in between a pair of wall-clock time packets, the ratio between processor core clock frequency and TSC frequency can be derived. If the P1 frequency is known, it can be applied to determine the CPU frequency. See Section 35.8.3.1 above for details on the relationship between TSC, MTC, and CYC.

21. Updates to Chapter 36, Volume 3D

Change bars show changes to Chapter 36 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3D:* System Programming Guide, Part 4.

Change to this chapter: Minor typo corrections to Section 36.1 “Overview”, Section 36.3 “Enclave Life Cycle”, and Section 36.6 “Enclave Instructions and Intel® SGX”.

CHAPTER 36
INTRODUCTION TO INTEL® SOFTWARE GUARD EXTENSIONS

36.1 OVERVIEW

Intel® Software Guard Extensions (Intel® SGX) is a set of instructions and mechanisms for memory accesses added to Intel® Architecture processors. Intel SGX can encompass two collections of instruction extensions, referred to as SGX1 and SGX2, see Table 36-1 and Table 36-2. The SGX1 extensions allow an application to instantiate a protected container, referred to as an enclave. An enclave is a protected area in the application's address space (see Figure 36-1), which provides confidentiality and integrity even in the presence of privileged malware. Accesses to the enclave memory area from any software not resident in the enclave are prevented. The SGX2 extensions allow additional flexibility in runtime management of enclave resources and thread execution within an enclave.

Chapter 37 covers main concepts, objects and data structure formats that interact within the Intel SGX architecture. Chapter 38 covers operational aspects ranging from preparing an enclave, transferring control to enclave code, and programming considerations for the enclave code and system software providing support for enclave execution. Chapter 39 describes the behavior of Asynchronous Enclave Exit (AEX) caused by events while executing enclave code. Chapter 40 covers the syntax and operational details of the instruction and associated leaf functions available in Intel SGX. Chapter 41 describes interaction of various aspects of IA32 and Intel® 64 architectures with Intel SGX. Chapter 42 covers Intel SGX support for application debug, profiling and performance monitoring.

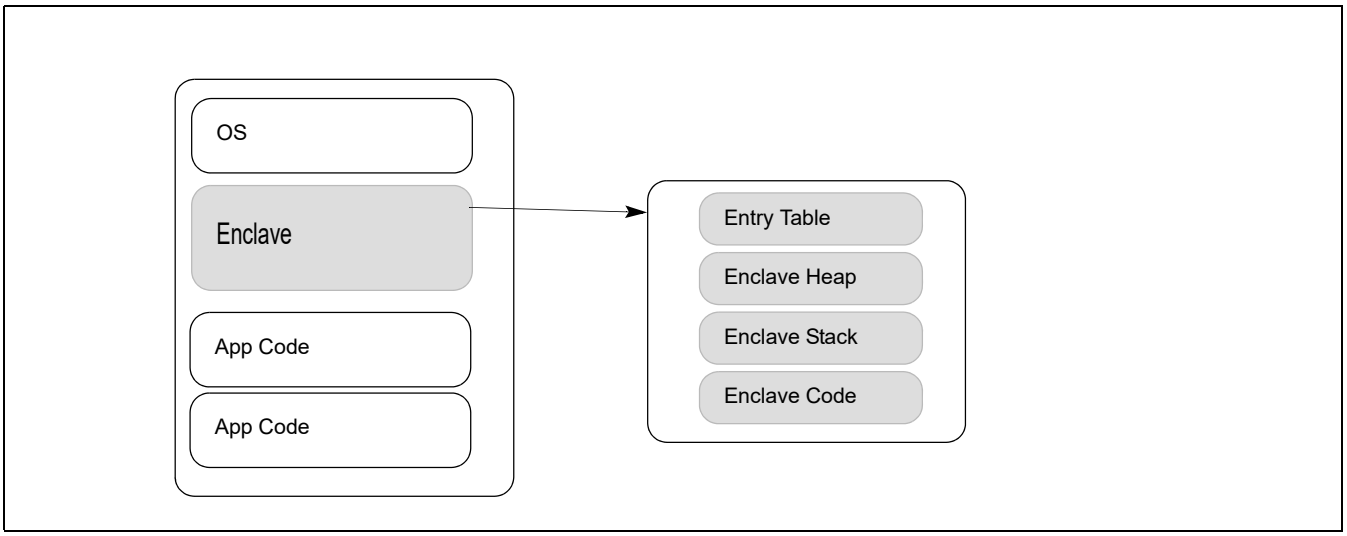


Figure 36-1. An Enclave Within the Application's Virtual Address Space

36.2 ENCLAVE INTERACTION AND PROTECTION

Intel SGX allows the protected portion of an application to be distributed in the clear. Before the enclave is built, the enclave code and data are free for inspection and analysis. The protected portion is loaded into an enclave where its code and data is measured. Once the application's protected portion of the code and data are loaded into an enclave, it is protected against external software access. An enclave can prove its identity to a remote party and provide the necessary building-blocks for secure provisioning of keys and credentials. The application can also request an enclave-specific and platform-specific key that it can use to protect keys and data that it wishes to store outside the enclave.[1]

1. For additional information, see white papers on Intel SGX at http://software.intel.com/en-us/intel-isa-extensions.

Intel SGX introduces two significant capabilities to the Intel Architecture. First is the change in enclave memory access semantics. The second is protection of the address mappings of the application.

36.3 ENCLAVE LIFE CYCLE

Enclave memory management is divided into two parts: address space allocation and memory commitment. Address space allocation is the specification of the range of logical addresses that the enclave may use. This range is called the ELRANGE. No actual resources are committed to this region. Memory commitment is the assignment of actual memory resources (as pages) within the allocated address space. This two-phase technique allows flexibility for enclaves to control their memory usage and to adjust dynamically without overusing memory resources when enclave needs are low. Commitment adds physical pages to the enclave. An operating system may support separate allocate and commit operations.

During enclave creation, code and data for an enclave are loaded from a clear-text source, i.e. from non-enclave memory.

Untrusted application code starts using an initialized enclave typically by using the EENTER leaf function provided by Intel SGX to transfer control to the enclave code residing in the protected Enclave Page Cache (EPC). The enclave code returns to the caller via the EEXIT leaf function. Upon enclave entry, control is transferred by hardware to software inside the enclave. The software inside the enclave switches the stack pointer to one inside the enclave. When returning back from the enclave, the software swaps back the stack pointer then executes the EEXIT leaf function.

On processors that support the SGX2 extensions, an enclave writer may add memory to an enclave using the SGX2 instruction set, after the enclave is built and running. These instructions allow adding additional memory resources to the enclave for use in such areas as the heap. In addition, SGX2 instructions allow the enclave to add new threads to the enclave. The SGX2 features provide additional capabilities to the software model without changing the security properties of the Intel SGX architecture.

Calling an external procedure from an enclave could be done using the EEXIT leaf function. Software would use EEXIT and a software convention between the trusted section and the untrusted section.

An active enclave consumes resources from the Enclave Page Cache (EPC, see Section 36.5). Intel SGX provides the EREMOVE instruction that an EPC manager can use to reclaim EPC pages committed to an enclave. The EPC manager uses EREMOVE on every enclave page when the enclave is torn down. After successful execution of EREMOVE the EPC page is available for allocation to another enclave.

36.4 DATA STRUCTURES AND ENCLAVE OPERATION

There are 2 main data structures associated with operating an enclave, the SGX Enclave Control Structure (SECS, see Section 37.7) and the Thread Control Structure (TCS, see Section 37.8).

There is one SECS for each enclave. The SECS contains meta-data about the enclave which is used by the hardware and cannot be directly accessed by software. Included in the SECS is a field that stores the enclave build measurement value. This field, MRENCLAVE, is initialized by the ECREATE instruction and updated by every EADD and EEXTEND. It is locked by EINIT.

Every enclave contains one or more TCS structures. The TCS contains meta-data used by the hardware to save and restore thread specific information when entering/exiting the enclave. There is one field, FLAGS, that may be accessed by software. This field can only be accessed by debug enclaves. The flag bit, DBGOPTIN, allows to single step into the thread associated with the TCS. (see Section 37.8.1)

The SECS is created when ECREATE (see Table 36-1) is executed. The TCS can be created using the EADD instruction or the SGX2 instructions (see Table 36-2).

36.5 ENCLAVE PAGE CACHE

The Enclave Page Cache (EPC) is the secure storage used to store enclave pages when they are a part of an executing enclave. For an EPC page, hardware performs additional access control checks to restrict access to the page. After the current page access checks and translations are performed, the hardware checks that the EPC page

is accessible to the program currently executing. Generally an EPC page is only accessed by the owner of the executing enclave or an instruction which is setting up an EPC page

The EPC is divided into EPC pages. An EPC page is 4KB in size and always aligned on a 4KB boundary.

Pages in the EPC can either be valid or invalid. Every valid page in the EPC belongs to one enclave instance. Each enclave instance has an EPC page that holds its SECS. The security metadata for each EPC page is held in an internal micro-architectural structure called Enclave Page Cache Map (EPCM, see Section 36.5.1).

The EPC is managed by privileged software. Intel SGX provides a set of instructions for adding and removing content to and from the EPC. The EPC may be configured by BIOS at boot time. On implementations in which EPC memory is part of system DRAM, the contents of the EPC are protected by an encryption engine.

36.5.1 Enclave Page Cache Map (EPCM)

The EPCM is a secure structure used by the processor to track the contents of the EPC. The EPCM holds one entry for each page in the EPC. The format of the EPCM is micro-architectural, and consequently is implementation dependent. However, the EPCM contains the following architectural information:

- The status of EPC page with respect to validity and accessibility.
- An SECS identifier (see Section 37.19) of the enclave to which the page belongs.
- The type of page: regular, SECS, TCS or VA.
- The linear address through which the enclave is allowed to access the page.
- The specified read/write/execute permissions on that page.

The EPCM structure is used by the CPU in the address-translation flow to enforce access-control on the EPC pages. The EPCM structure is described in Table 37-27, and the conceptual access-control flow is described in Section 37.5.

The EPCM entries are managed by the processor as part of various instruction flows.

36.6 ENCLAVE INSTRUCTIONS AND INTEL® SGX

The enclave instructions available with Intel SGX are organized as leaf functions under two instruction mnemonics: ENCLS (ring 0) and ENCLU (ring 3). Each leaf function uses EAX to specify the leaf function index, and may require additional implicit input registers as parameters. The use of EAX is implied implicitly by the ENCLS and ENCLU instructions, ModR/M byte encoding is not used with ENCLS and ENCLU. The use of additional registers does not use ModR/M encoding and is implied implicitly by the respective leaf function index.

Each leaf function index is also associated with a unique, leaf-specific mnemonic. A long-form expression of Intel SGX instruction takes the form of ENCLx[LEAF_MNEMONIC], where 'x' is either 'S' or 'U'. The long-form expression provides clear association of the privilege-level requirement of a given "leaf mnemonic". For simplicity, the unique "Leaf_Mnemonic" name is used (omitting the ENCLx for convenience) throughout in this document.

Details of individual SGX leaf functions are described in Chapter 40. Table 36-1 provides a summary of the instruction leaves that are available in the initial implementation of Intel SGX, which is introduced in the 6th generation Intel Core processors. Table 36-2 summarizes enhancement of Intel SGX for future Intel processors.

Table 36-1. Supervisor and User Mode Enclave Instruction Leaf Functions in Long-Form of SGX1

Supervisor Instruction	Description	User Instruction	Description
ENCLS[EADD]	Add an EPC page to an enclave.	ENCLU[EENTER]	Enter an enclave.
ENCLS[EBLOCK]	Block an EPC page.	ENCLU[EEXIT]	Exit an enclave.
ENCLS[ECREATE]	Create an enclave.	ENCLU[EGETKEY]	Create a cryptographic key.
ENCLS[EDBGRD]	Read data from a debug enclave by debugger.	ENCLU[EREPORT]	Create a cryptographic report.
ENCLS[EDBGWR]	Write data into a debug enclave by debugger.	ENCLU[ERESUME]	Re-enter an enclave.

Table 36-1. Supervisor and User Mode Enclave Instruction Leaf Functions in Long-Form of SGX1

Supervisor Instruction	Description	User Instruction	Description
ENCLS[EEXTEND]	Extend EPC page measurement.		
ENCLS[EINIT]	Initialize an enclave.		
ENCLS[ELDB]	Load an EPC page in blocked state.		
ENCLS[ELDU]	Load an EPC page in unblocked state.		
ENCLS[EPA]	Add an EPC page to create a version array.		
ENCLS[EREMOVE]	Remove an EPC page from an enclave.		
ENCLS[ETRACK]	Activate EBLOCK checks.		
ENCLS[EWB]	Write back/invalidate an EPC page.		

Table 36-2. Supervisor and User Mode Enclave Instruction Leaf Functions in Long-Form of SGX2

Supervisor Instruction	Description	User Instruction	Description
ENCLS[EAUG]	Allocate EPC page to an existing enclave.	ENCLU[EACCEPT]	Accept EPC page into the enclave.
ENCLS[EMODPR]	Restrict page permissions.	ENCLU[EMODPE]	Enhance page permissions.
ENCLS[EMODT]	Modify EPC page type.	ENCLU[EACCEPTCOPY]	Copy contents to an augmented EPC page and accept the EPC page into the enclave.

36.7 DISCOVERING SUPPORT FOR INTEL® SGX AND ENABLING ENCLAVE INSTRUCTIONS

Detection of support of Intel SGX and enumeration of available and enabled Intel SGX resources are queried using the CPUID instruction. The enumeration interface comprises the following:

- Processor support of Intel SGX is enumerated by a feature flag in CPUID leaf 07H: CPUID.(EAX=07H, ECX=0H):EBX.SGX[bit 2]. If CPUID.(EAX=07H, ECX=0H):EBX.SGX = 1, the processor has support for Intel SGX, and requires opt-in enabling by BIOS via IA32_FEATURE_CONTROL MSR.

  If CPUID.(EAX=07H, ECX=0H):EBX.SGX = 1, CPUID will report via the available sub-leaves of CPUID.(EAX=12H) on available and/or configured Intel SGX resources.
- The available and configured Intel SGX resources enumerated by the sub-leaves of CPUID.(EAX=12H) depend on the state of BIOS configuration.

36.7.1 Intel® SGX Opt-In Configuration

On processors that support Intel SGX, IA32_FEATURE_CONTROL provides the SGX_ENABLE field (bit 18). Before system software can configure and enable Intel SGX resources, BIOS is required to set IA32_FEATURE_CONTROL.SGX_ENABLE = 1 to opt-in the use of Intel SGX by system software.

The semantics of setting SGX_ENABLE follows the rules of IA32_FEATURE_CONTROL.LOCK (bit 0). Software is considered to have opted into Intel SGX if and only if IA32_FEATURE_CONTROL.SGX_ENABLE and IA32_FEATURE_CONTROL.LOCK are set to 1. The setting of IA32_FEATURE_CONTROL.SGX_ENABLE (bit 18) is not reflected by CPUID.

Table 36-3. Intel® SGX Opt-in and Enabling Behavior

CPUID.(07H,0H):EBX.SGX	CPUID.(12H)	FEATURE_CONTROL.LOCK	FEATURE_CONTROL.SGX_ENABLE	Enclave Instruction
0	Invalid	X	X	#UD
1	Valid*	X	X	#UD**

Table 36-3. Intel® SGX Opt-in and Enabling Behavior

CPUID.(07H,0H):EBX.SGX	CPUID.(12H)	FEATURE_CONTROL.LOCK	FEATURE_CONTROL.SGX_ENABLE	Enclave Instruction
1	Valid*	0	X	#GP
1	Valid*	1	0	#GP
1	Valid*	1	1	Available (see Table 36-4 for details of SGX1 and SGX2).
* Leaf 12H enumeration results are dependent on enablement. ** See list of conditions in the #UD section of the reference pages of ENCLS and ENCLU				

36.7.2 Intel® SGX Resource Enumeration Leaves

If CPUID.(EAX=07H, ECX=0H):EBX.SGX = 1, the processor also supports querying CPUID with EAX=12H on Intel SGX resource capability and configuration. The number of available sub-leaves in leaf 12H depends on the Opt-in and system software configuration. Information returned by CPUID.12H is thread specific; software should not assume that if Intel SGX instructions are supported on one hardware thread, they are also supported elsewhere.

A properly configured processor exposes Intel SGX functionality with CPUID.EAX=12H reporting valid information (non-zero content) in three or more sub-leaves, see Table 36-4.

- CPUID.(EAX=12H, ECX=0H) enumerates Intel SGX capability, including enclave instruction opcode support.
- CPUID.(EAX=12H, ECX=1H) enumerates Intel SGX capability of processor state configuration and enclave configuration in the SECS structure (see Table 37-3).
- CPUID.(EAX=12H, ECX >1) enumerates available EPC resources.

Table 36-4. CPUID Leaf 12H, Sub-Leaf 0 Enumeration of Intel® SGX Capabilities

CPUID.(EAX=12H,ECX=0)		Description Behavior
Register	Bits	
EAX	0	SGX1: If 1, indicates leaf functions of SGX1 instruction listed in Table 36-1 are supported.
	1	SGX2: If 1, indicates leaf functions of SGX2 instruction listed in Table 36-2 are supported.
	31:2	Reserved (0)
EBX	31:0	MISCSELECT: Reports the bit vector of supported extended features that can be written to the MISC region of the SSA.
ECX	31:0	Reserved (0).
EDX	7:0	MaxEnclaveSize_Not64: the maximum supported enclave size is 2^(EDX[7:0]) bytes when not in 64-bit mode.
	15:8	MaxEnclaveSize_64: the maximum supported enclave size is 2^(EDX[15:8]) bytes when operating in 64-bit mode.
	31:16	Reserved (0).

Table 36-5. CPUID Leaf 12H, Sub-Leaf 1 Enumeration of Intel® SGX Capabilities

CPUID.(EAX=12H,ECX=1)		Description Behavior
Register	Bits	
EAX	31:0	Report the valid bits of SECS.ATTRIBUTES[31:0] that software can set with ECREATE. SECS.ATTRIBUTES[n] can be set to 1 using ECREATE only if EAX[n] is 1, where n < 32.
EBX	31:0	Report the valid bits of SECS.ATTRIBUTES[63:32] that software can set with ECREATE. SECS.ATTRIBUTES[n+32] can be set to 1 using ECREATE only if EBX[n] is 1, where n < 32.

Table 36-5. CPUID Leaf 12H, Sub-Leaf 1 Enumeration of Intel® SGX Capabilities

CPUID.(EAX=12H,ECX=1)		Description Behavior
Register	Bits	
ECX	31:0	Report the valid bits of SECS.ATTRIBUTES[95:64] that software can set with ECREATE. SECS.ATTRIBUTES[n+64] can be set to 1 using ECREATE only if ECX[n] is 1, where n < 32.
EDX	31:0	Report the valid bits of SECS.ATTRIBUTES[127:96] that software can set with ECREATE. SECS.ATTRIBUTES[n+96] can be set to 1 using ECREATE only if EDX[n] is 1, where n < 32.

On processors that support Intel SGX1 and SGX2, CPUID leaf 12H sub-leaf 2 report physical memory resources available for use with Intel SGX. These physical memory sections are typically allocated by BIOS as **Processor Reserved Memory**, and available to the OS to manage as EPC.

To enumerate how many EPC sections are available to the EPC manager, software can enumerate CPUID leaf 12H with sub-leaf index starting from 2, and decode the sub-leaf-type encoding (returned in EAX[3:0]) until the sub-leaf type is invalid. All invalid sub-leaves of CPUID leaf 12H return EAX/EBX/ECX/EDX with 0.

Table 36-6. CPUID Leaf 12H, Sub-Leaf Index 2 or Higher Enumeration of Intel® SGX Resources

CPUID.(EAX=12H,ECX > 1)		Description Behavior
Register	Bits	
EAX	3:0	0000b: This sub-leaf is invalid; EDX:ECX:EBX:EAX return 0. 0001b: This sub-leaf enumerates an EPC section. EBX:EAX and EDX:ECX provide information on the Enclave Page Cache (EPC) section. All other encoding are reserved.
	11:4	Reserved (enumerate 0).
	31:12	If EAX[3:0] = 0001b, these are bits 31:12 of the physical address of the base of the EPC section.
EBX	19:0	If EAX[3:0] = 0001b, these are bits 51:32 of the physical address of the base of the EPC section.
	31:20	Reserved.
ECX	3: 0	If EAX[3:0] 0000b, then all bits of the EDX:ECX pair are enumerated as 0. If EAX[3:0] 0001b, then this section has confidentiality and integrity protection. All other encoding are reserved.
	11:4	Reserved (enumerate 0).
	31:12	If EAX[3:0] = 0001b, these are bits 31:12 of the size of the corresponding EPC section within the Processor Reserved Memory.
EDX	19: 0	If EAX[3:0] = 0001b, these are bits 51:32 of the size of the corresponding EPC section within the Processor Reserved Memory.
	31:20	Reserved.

22.Updates to Chapter 37, Volume 3D

Change bars show changes to Chapter 37 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3D:* System Programming Guide, Part 4.

Changes to this chapter: Minor typo corrections to Section 37.7.1 “Attributes”, Section 37.14 “EINIT Token Structure (EINITTOKEN)”, and Section 37.17.1 “KEY REQUEST KeyNames”.

CHAPTER 37
ENCLAVE ACCESS CONTROL AND DATA STRUCTURES

37.1 OVERVIEW OF ENCLAVE EXECUTION ENVIRONMENT

When an enclave is created, it has a range of linear addresses that the processor applies enhanced access control. This ranged is called the ELRANGE (see Section 36.3). When an enclave generates a memory access, the existing IA32 segmentation and paging architecture are applied. Additionally, linear addresses inside the ELRANGE must map to an EPC page otherwise when an enclave attempts to access that linear address a fault is generated.

The EPC pages need not be physically contiguous. System software allocates EPC pages to various enclaves. Enclaves must abide by OS/VMM imposed segmentation and paging policies. OS/VMM-managed page tables and extended page tables provide address translation for the enclave pages. Hardware requires that these pages are properly mapped to EPC (any failure generates an exception).

Enclave entry must happen through specific enclave instructions:

- ENCLU[EENTER], ENCLU[ERESUME].

Enclave exit must happen through specific enclave instructions or events:

- ENCLU[EEXIT], Asynchronous Enclave Exit (AEX).

Attempts to execute, read, or write to linear addresses mapped to EPC pages when not inside an enclave will result in the processor altering the access to preserve the confidentiality and integrity of the enclave. The exact behavior may be different between implementations. As an example a read of an enclave page may result in the return of all one's or return of cyphertext of the cache line. Writing to an enclave page may result in a dropped write or a machine check at a later time. The processor will provide the protections as described in Section 37.4 and Section 37.5 on such accesses.

37.2 TERMINOLOGY

A memory access to the ELRANGE and initiated by an instruction executed by an enclave is called a Direct Enclave Access (Direct EA).

Memory accesses initiated by certain Intel® SGX instruction leaf functions such as ECREATE, EADD, EDBGRD, EDBGWR, ELDU/ELDB, EWB, EREMOVE, EENTER, and ERESUME to EPC pages are called Indirect Enclave Accesses (Indirect EA). Table 37-1 lists additional details of the indirect EA of SGX1 and SGX2 extensions.

Direct EAs and Indirect EAs together are called Enclave Accesses (EAs).

Any memory access that is not an Enclave Access is called a non-enclave access.

37.3 ACCESS-CONTROL REQUIREMENTS

Enclave accesses have the following access-control attributes:

- All memory accesses must conform to segmentation and paging protection mechanisms.
- Code fetches from inside an enclave to a linear address outside that enclave result in a #GP(0) exception.
- Non-enclave accesses to EPC memory result in undefined behavior. EPC memory is protected as described in Section 37.4 and Section 37.5 on such accesses.
- EPC pages of page types PT_REG, PT_TCS and PT_TRIM must be mapped to ELRANGE at the linear address specified when the EPC page was allocated to the enclave using ENCLS[EADD] or ENCLS[EAUG] leaf functions. Enclave accesses through other linear address result in a #PF with the PFEC.SGX bit set.
- Direct EAs to any EPC pages must conform to the currently defined security attributes for that EPC page in the EPCM. These attributes may be defined at enclave creation time (EADD) or when the enclave sets them using SGX2 instructions. The failure of these checks results in a #PF with the PFEC.SGX bit set.

— Target page must belong to the currently executing enclave.
— Data may be written to an EPC page if the EPCM allow write access.
— Data may be read from an EPC page if the EPCM allow read access.
— Instruction fetches from an EPC page are allowed if the EPCM allows execute access.
— Target page must not have a restricted page type[1] (PT_SECS, PT_TCS, PT_VA, or PT_TRIM).
— The EPC page must not be BLOCKED.
— The EPC page must not be PENDING.
— The EPC page must not be MODIFIED.

37.4 SEGMENT-BASED ACCESS CONTROL

Intel SGX architecture does not modify the segment checks performed by a logical processor. All memory accesses arising from a logical processor in protected mode (including enclave access) are subject to segmentation checks with the applicable segment register.

To ensure that outside entities do not modify the enclave's logical-to-linear address translation in an unexpected fashion, ENCLU[EENTER] and ENCLU[ERESUME] check that CS, DS, ES, and SS, if usable (i.e., not null), have segment base value of zero. A non-zero segment base value for these registers results in a #GP(0).

On enclave entry either via EENTER or ERESUME, the processor saves the contents of the external FS and GS registers, and loads these registers with values stored in the TCS at build time to enable the enclave’s use of these registers for accessing the thread-local storage inside the enclave. On EEXIT and AEX, the contents at time of entry are restored. On AEX, the values of FS and GS are saved in the SSA frame. On ERESUME, FS and GS are restored from the SSA frame. The details of these operations can be found in the descriptions of EENTER, ERESUME, EEXIT, and AEX flows.

37.5 PAGE-BASED ACCESS CONTROL

37.5.1 Access-control for Accesses that Originate from non-SGX Instructions

Intel SGX builds on the processor's paging mechanism to provide page-granular access-control for enclave pages. Enclave pages are only accessible from inside the currently executing enclave if they belong to that enclave. In addition, enclave accesses must conform to the access control requirements described in Section 37.3. or through certain Intel SGX instructions. Attempts to execute, read, or write to linear addresses mapped to EPC pages when not inside an enclave will result in the processor altering the access to preserve the confidentiality and integrity of the enclave. The exact behavior may be different between implementations.

37.5.2 Memory Accesses that Split across ELRANGE

Memory data accesses are allowed to split across ELRANGE (i.e., a part of the access is inside ELRANGE and a part of the access is outside ELRANGE) while the processor is inside an enclave. If an access splits across ELRANGE, the processor splits the access into two sub-accesses (one inside ELRANGE and the other outside ELRANGE), and each access is evaluated. A code-fetch access that splits across ELRANGE results in a #GP due to the portion that lies outside of the ELRANGE.

37.5.3 Implicit vs. Explicit Accesses

Memory accesses originating from Intel SGX instruction leaf functions are categorized as either explicit accesses or implicit accesses. Table 37-1 lists the implicit and explicit memory accesses made by Intel SGX leaf functions.

1. EPCM may allow write, read or execute access only for pages with page type PT_REG.

37.5.3.1 Explicit Accesses

Accesses to memory locations provided as explicit operands to Intel SGX instruction leaf functions, or their linked data structures are called explicit accesses.

Explicit accesses are always made using logical addresses. These accesses are subject to segmentation, paging, extended paging, and APIC-virtualization checks, and trigger any faults/exit associated with these checks when the access is made.

The interaction of explicit memory accesses with data breakpoints is leaf-function-specific, and is documented in Section 42.3.4.

37.5.3.2 Implicit Accesses

Accesses to data structures whose physical addresses are cached by the processor are called implicit accesses. These addresses are not passed as operands of the instruction but are implied by use of the instruction.

These accesses do not trigger any access-control faults/exits or data breakpoints. Table 37-1 lists memory objects that Intel SGX instruction leaf functions access either by explicit access or implicit access. The addresses of explicit access objects are passed via register operands with the second through fourth column of Table 37-1 matching implicitly encoded registers RBX, RCX, RDX.

Physical addresses used in different implicit accesses are cached via different instructions and for different durations. The physical address of SECS associated with each EPC page is cached at the time the page is added to the enclave via ENCLS[EADD] or ENCLS[EAUG], or when the page is loaded to EPC via ENCLS[ELDB] or ENCLS[ELDU]. This binding is severed when the corresponding page is removed from the EPC via ENCLS[EREMOVE] or ENCLS[EWB]. Physical addresses of TCS and SSA pages are cached at the time of most-recent enclave entry. Exit from an enclave (ENCLU[EEXIT] or AEX) flushes this caching. Details of Asynchronous Enclave Exit is described in Chapter 39.

The physical addresses that are cached for use by implicit accesses are derived from logical (or linear) addresses after checks such as segmentation, paging, EPT, and APIC virtualization checks. These checks may trigger exceptions or VM exits. Note, however, that such exception or VM exits may not occur after a physical address is cached and used for an implicit access.

Table 37-1. List of Implicit and Explicit Memory Access by Intel® SGX Enclave Instructions

Instr. Leaf	Enum.	Explicit 1	Explicit 2	Explicit 3	Implicit
EACCEPT	SGX2	SECINFO	EPCPAGE		SECS
EACCEPTCOPY	SGX2	SECINFO	EPCPAGE (Src)	EPCPAGE (Dst)	
EADD	SGX1	PAGEINFO and linked structures	EPCPAGE		
EAUG	SGX2	PAGEINFO and linked structures	EPCPAGE		SECS
EBLOCK	SGX1	EPCPAGE			SECS
ECREATE	SGX1	PAGEINFO and linked structures	EPCPAGE		
EDBGRD	SGX1	EPCADDR	Destination		SECS
EDBGWR	SGX1	EPCADDR	Source		SECS
EENTER	SGX1	TCS and linked SSA			SECS
EEXIT	SGX1				SECS, TCS
EEXTEND	SGX1	SECS	EPCPAGE		
EGETKEY	SGX1	KEYREQUEST	KEY		SECS
EINIT	SGX1	SIGSTRUCT	SECS	EINITTOKEN	
ELDB/ELDU	SGX1	PAGEINFO and linked structures, PCMD	EPCPAGE	VAPAGE	
EMODPE	SGX2	SECINFO	EPCPAGE		
EMODPR	SGX2	SECINFO	EPCPAGE		SECS
EMODT	SGX2	SECINFO	EPCPAGE		SECS

Table 37-1. List of Implicit and Explicit Memory Access by Intel® SGX Enclave Instructions (Contd.)

Instr. Leaf	Enum.	Explicit 1	Explicit 2	Explicit 3	Implicit
EPA	SGX1	EPCADDR			
EREMOVE	SGX1	EPCPAGE			SECS
EREPORT	SGX1	TARGETINFO	REPORTDATA	OUTPUTDATA	SECS
ERESUME	SGX1	TCS and linked SSA			SECS
ETRACK	SGX1	EPCPAGE			
EWB	SGX1	PAGEINFO and linked structures, PCMD	EPCPAGE	VAPAGE	SECS
Asynchronous Enclave Exit*					SECS, TCS, SSA
*Details of Asynchronous Enclave Exit (AEX) is described in Section 39.4					

37.6 INTEL® SGX DATA STRUCTURES OVERVIEW

Enclave operation is managed via a collection of data structures. Many of the top-level data structures contain sub-structures. The top-level data structures relate to parameters that may be used in enclave setup/maintenance, by Intel SGX instructions, or AEX event. The top-level data structures are:

- SGX Enclave Control Structure (SECS)
- Thread Control Structure (TCS)
- State Save Area (SSA)
- Page Information (PAGEINFO)
- Security Information (SECINFO)
- Paging Crypto MetaData (PCMD)
- Enclave Signature Structure (SIGSTRUCT)
- EINIT Token Structure (EINITTOKEN)
- Report Structure (REPORT)
- Report Target Info (TARGETINFO)
- Key Request (KEYREQUEST)
- Version Array (VA)
- Enclave Page Cache Map (EPCM)

Details of the top-level data structures and associated sub-structures are listed in Section 37.7 through Section 37.19.

37.7 SGX ENCLAVE CONTROL STRUCTURE (SECS)

The SECS data structure requires 4K-Bytes alignment.

Table 37-2. Layout of SGX Enclave Control Structure (SECS)

Field	OFFSET (Bytes)	Size (Bytes)	Description
SIZE	0	8	Size of enclave in bytes; must be power of 2.
BASEADDR	8	8	Enclave Base Linear Address must be naturally aligned to size.
SSAFRAMESIZE	16	4	Size of one SSA frame in pages, including XSAVE, pad, GPR, and MISC (if CPUID.(EAX=12H, ECX=0):.EBX != 0).
MISCSELECT	20	4	Bit vector specifying which extended features are saved to the MISC region (see Section 37.7.2) of the SSA frame when an AEX occurs.

Table 37-2. Layout of SGX Enclave Control Structure (SECS) (Contd.)

Field	OFFSET (Bytes)	Size (Bytes)	Description
RESERVED	24	24	
ATTRIBUTES	48	16	Attributes of the Enclave, see Table 37-3.
MRENCLAVE	64	32	Measurement Register of enclave build process. See SIGSTRUCT for format.
RESERVED	96	32	
MRSIGNER	128	32	Measurement Register extended with the public key that verified the enclave. See SIGSTRUCT for format.
RESERVED	160	96	
ISVPRODID	256	2	Product ID of enclave.
ISVSVN	258	2	Security version number (SVN) of the enclave.
RESERVED	260	3836	The RESERVED field consists of the following: ▪ EID: An 8 byte Enclave Identifier. Its location is implementation specific. ▪ PAD: A 352 bytes padding pattern from the Signature (used for key derivation strings). It's location is implementation specific. ▪ The remaining 3476 bytes are reserved area. The entire 3836 byte field must be cleared prior to executing ECREATE or EREPORT.

37.7.1 ATTRIBUTES

The ATTRIBUTES data structure is comprised of bit-granular fields that are used in the SECS, the REPORT and the KEYREQUEST structures. CPUID.(EAX=12H, ECX=1) enumerates a bitmap of permitted 1-setting of bits in ATTRIBUTES.

Table 37-3. Layout of ATTRIBUTES Structure

Field	Bit Position	Description
INIT	0	This bit indicates if the enclave has been initialized by EINIT. It must be cleared when loaded as part of ECREATE. For EREPORT instruction, TARGET_INFO.ATTRIBUTES[EINIT] must always be 1 to match the state after EINIT has initialized the enclave.
DEBUG	1	If 1, the enclave permit debugger to read and write enclave data using EDBGRD and EDBGWR.
MODE64BIT	2	Enclave runs in 64-bit mode.
RESERVED	3	Must be Zero.
PROVISIONKEY	4	Provisioning Key is available from EGETKEY.
EINITTOKEN_KEY	5	EINIT token key is available from EGETKEY.
RESERVED	63:6	
XFRM	127:64	XSAVE Feature Request Mask. See Section 41.7.

37.7.2 SECS.MISCSELECT Field

CPUID.(EAX=12H, ECX=0):EBX[31:0] enumerates which extended information that the processor can save into the MISC region of SSA when an AEX occurs. An enclave writer can specify via SIGSTRUCT how to set the SECS.MISCSELECT field. The bit vector of MISCSELECT selects which extended information is to be saved in the MISC region of the SSA frame when an AEX is generated. The bit vector definition of extended information is listed in Table 37-4.

If CPUID.(EAX=12H, ECX=0):EBX[31:0] = 0, SECS.MISCSELECT field must be all zeros.

The SECS.MISCSELECT field determines the size of MISC region of the SSA frame, see Section 37.9.2.

Table 37-4. Bit Vector Layout of MISCSELECT Field of Extended Information

Field	Bit Position	Description
EXINFO	0	Report information about page fault and general protection exception that occurred inside an enclave.
Reserved	31:1	Reserved (0).

37.8 THREAD CONTROL STRUCTURE (TCS)

Each executing thread in the enclave is associated with a Thread Control Structure. It requires 4K-Bytes alignment.

Table 37-5. Layout of Thread Control Structure (TCS)

Field	OFFSET (Bytes)	Size (Bytes)	Description
STAGE	0	8	Enclave execution state of the thread controlled by this TCS. A value of 0 indicates that this TCS is available for enclave entry. A value of 1 indicates that a processer is currently executing an enclave in the context of this TCS.
FLAGS	8	8	The thread's execution flags (see Section 37.8.1).
OSSA	16	8	Offset of the base of the State Save Area stack, relative to the enclave base. Must be page aligned.
CSSA	24	4	Current slot index of an SSA frame, cleared by EADD and EACCEPT.
NSSA	28	4	Number of available slots for SSA frames.
OENTRY	32	8	Offset in enclave to which control is transferred on EENTER relative to the base of the enclave.
AEP	40	8	The value of the Asynchronous Exit Pointer that was saved at EENTER time.
OFSBASGX	48	8	Offset to add to the base address of the enclave for producing the base address of FS segment inside the enclave. Must be page aligned.
OGSBASGX	56	8	Offset to add to the base address of the enclave for producing the base address of GS segment inside the enclave. Must be page aligned.
FSLIMIT	64	4	Size to become the new FS limit in 32-bit mode.
GSLIMIT	68	4	Size to become the new GS limit in 32-bit mode.
RESERVED	72	4024	Must be zero.

37.8.1 TCS.FLAGS

Table 37-6. Layout of TCS.FLAGS Field

Field	Bit Position	Description
DBGOPTIN	0	If set, allows debugging features (single-stepping, breakpoints, etc.) to be enabled and active while executing in the enclave on this TCS. Hardware clears this bit on EADD. A debugger may later modify it if the enclave's ATTRIBUTES.DEBUG is set.
RESERVED	63:1	

37.8.2 State Save Area Offset (OSSA)

The OSSA points to a stack of State Save Area (SSA) frames (see Section 37.9) used to save the processor state when an interrupt or exception occurs while executing in the enclave.

37.8.3 Current State Save Area Frame (CSSA)

CSSA is the index of the current SSA frame that will be used by the processor to determine where to save the processor state on an interrupt or exception that occurs while executing in the enclave. It is an index into the array of frames addressed by OSSA. CSSA is incremented on an AEX and decremented on an ERESUME.

37.8.4 Number of State Save Area Frames (NSSA)

NSSA specifies the number of SSA frames available for this TCS. There must be at least one available SSA frame when EENTER-ing the enclave or the EENTER will fail.

37.9 STATE SAVE AREA (SSA) FRAME

When an AEX occurs while running in an enclave, the architectural state is saved in the thread's current SSA frame, which is pointed to by TCS.CSSA. An SSA frame must be page aligned, and contains the following regions:

- The XSAVE region starts at the base of the SSA frame, this region contains extended feature register state in an XSAVE/FXSAVE-compatible non-compacted format.
- A Pad region: software may choose to maintain a pad region separating the XSAVE region and the MISC region. Software choose the size of the pad region according to the sizes of the MISC and GPRSGX regions.
- The GPRSGX region. The GPRSGX region is the last region of an SSA frame (see Table 37-7). This is used to hold the processor general purpose registers (RAX … R15), the RIP, the outside RSP and RBP, RFLAGS and the AEX information.
- The MISC region (If CPUIDEAX=12H, ECX=0):EBX[31:0] != 0). The MISC region is adjacent to the GRPSGX region, and may contain zero or more components of extended information that would be saved when an AEX occurs. If the MISC region is absent, the region between the GPRSGX and XSAVE regions is the pad region that software can use. If the MISC region is present, the region between the MISC and XSAVE regions is the pad region that software can use. See additional details in Section 37.9.2.

Table 37-7. Top-to-Bottom Layout of an SSA Frame

Region	Offset (Byte)	Size (Bytes)	Description
XSAVE	0	Calculate using CPUID leaf 0DH information	The size of XSAVE region in SSA is derived from the enclave's support of the collection of processor extended states that would be managed by XSAVE. The enablement of those processor extended state components in conjunction with CPUID leaf 0DH information determines the XSAVE region size in SSA.
Pad	End of XSAVE region	Chosen by enclave writer	Ensure the end of GPRSGX region is aligned to the end of a 4KB page.
MISC	base of GPRSGX – sizeof(MISC)	Calculate from highest set bit of SECS.MISCSELECT	See Section 37.9.2.
GPRSGX	SSAFRAMESIZE – 176	176	See Table 37-8 for layout of the GPRSGX region.

37.9.1 GPRSGX Region

The layout of the GPRSGX region is shown in Table 37-8.

Table 37-8. Layout of GPRSGX Portion of the State Save Area

Field	OFFSET (Bytes)	Size (Bytes)	Description
RAX	0	8	
RCX	8	8	
RDX	16	8	
RBX	24	8	
RSP	32	8	
RBP	40	8	
RSI	48	8	
RDI	56	8	
R8	64	8	
R9	72	8	
R10	80	8	
R11	88	8	
R12	96	8	
R13	104	8	
R14	112	8	
R15	120	8	
RFLAGS	128	8	Flag register.
RIP	136	8	Instruction pointer.
URSP	144	8	Non-Enclave (outside) stack pointer. Saved by EENTER, restored on AEX.
URBP	152	8	Non-Enclave (outside) RBP pointer. Saved by EENTER, restored on AEX.
EXITINFO	160	4	Contains information about exceptions that cause AEXs, which might be needed by enclave software (see Section 37.9.1.1).
RESERVED	164	4	
FSBASE	168	8	FS BASE.
GSBASE	176	8	GS BASE.

37.9.1.1 EXITINFO

EXITINFO contains the information used to report exit reasons to software inside the enclave. It is a 4 byte field laid out as in Table 37-9. The VALID bit is set only for the exceptions conditions which are reported inside an enclave. See Table 37-10 for which exceptions are reported inside the enclave. If the exception condition is not one reported inside the enclave then VECTOR and EXIT_TYPE are cleared.

Table 37-9. Layout of EXITINFO Field

Field	Bit Position	Description
VECTOR	7:0	Exception number of exceptions reported inside enclave.
EXIT_TYPE	10:8	011b: Hardware exceptions. 110b: Software exceptions. Other values: Reserved.
RESERVED	30:11	Reserved as zero.
VALID	31	0: unsupported exceptions. 1: Supported exceptions. Includes two categories: • Unconditionally supported exceptions: #DE, #DB, #BP, #BR, #UD, #MF, #AC, #XM. • Conditionally supported exception: — #PF, #GP if SECS.MISCSELECT.EXINFO = 1.

37.9.1.2 VECTOR Field Definition

Table 37-10 contains the VECTOR field. This field contains information about some exceptions which occur inside the enclave. These vector values are the same as the values that would be used when vectoring into regular exception handlers. All values not shown are not reported inside an enclave.

Table 37-10. Exception Vectors

Name	Vector #	Description
#DE	0	Divider exception.
#DB	1	Debug exception.
#BP	3	Breakpoint exception.
#BR	5	Bound range exceeded exception.
#UD	6	Invalid opcode exception.
#GP	13	General protection exception. Only reported if SECS.MISCSELECT.EXINFO = 1.
#PF	14	Page fault exception. Only reported if SECS.MISCSELECT.EXINFO = 1.
#MF	16	x87 FPU floating-point error.
#AC	17	Alignment check exceptions.
#XM	19	SIMD floating-point exceptions.

37.9.2 MISC Region

The layout of the MISC region is shown in Table 37-11. The number of components that the processor supports in the MISC region corresponds to the set bits of CPUID.(EAX=12H, ECX=0):EBX[31:0] set to 1. Each set bit in CPUID.(EAX=12H, ECX=0):EBX[31:0] has a defined size for the corresponding component, as shown in Table 37-11. Enclave writers needs to do the following:

- Decide which MISC region components will be supported for the enclave.
- Allocate an SSA frame large enough to hold the components chosen above.
- Instruct each enclave builder software to set the appropriate bits in SECS.MISCSELECT.

The first component, EXINFO, starts next to the GPRSGX region. Additional components in the MISC region grow in ascending order within the MISC region towards the XSAVE region.

The size of the MISC region is calculated as follows:

- If CPUID.(EAX=12H, ECX=0):EBX[31:0] = 0, MISC region is not supported.
- If CPUID.(EAX=12H, ECX=0):EBX[31:0] != 0, the size of MISC region is derived from sum of the highest bit set in SECS.MISCSELECT and the size of the MISC component corresponding to that bit. Offset and size

information of currently defined MISC components are listed in Table 37-11. For example, if the highest bit set in SECS.MISCSELECT is bit 0, the MISC region offset is OFFSET(GPRSGX)-16 and size is 16 bytes.

- The processor saves a MISC component i in the MISC region if and only if SECS.MISCSELECT[i] is 1.

Table 37-11. Layout of MISC region of the State Save Area

MISC Components	OFFSET (Bytes)	Size (Bytes)	Description
EXINFO	Offset(GPRSGX) -16	16	if CPUID.(EAX=12H, ECX=0):EBX[0] = 1, exception information on #GP or #PF that occurred inside an enclave can be written to the EXINFO structure if specified by SECS.MISCSELECT[0] = 1.
Future Extension	Below EXINFO	TBD	Reserved. (Zero size if CPUID.(EAX=12H, ECX=0):EBX[31:1] =0).

37.9.2.1 EXINFO Structure

Table 37-12 contains the layout of the EXINFO structure that provides additional information.

Table 37-12. Layout of EXINFO Structure

Field	OFFSET (Bytes)	Size (Bytes)	Description
MADDR	0	8	If #PF: contains the page fault linear address that caused a page fault. If #GP: the field is cleared.
ERRCD	8	4	Exception error code for either #GP or #PF.
RESERVED	12	4	

37.9.2.2 Page Fault Error Codes

Table 37-13 contains page fault error code that may be reported in EXINFO.ERRCD.

Table 37-13. Page Fault Error Codes

Name	Bit Position	Description
P	0	Same as non-SGX page fault exception P flag.
W/R	1	Same as non-SGX page fault exception W/R flag.
U/S[1]	2	Always set to 1 (user mode reference).
RSVD	3	Same as non-SGX page fault exception RSVD flag.
I/D	4	Same as non-SGX page fault exception I/D flag.
PK	5	Protection Key induced fault.
RSVD	14:6	Reserved.
SGX	15	EPCM induced fault.
RSVD	31:5	Reserved.

NOTES:

1. Page faults incident to enclave mode that report U/S=0 are not reported in EXINFO

37.10 PAGE INFORMATION (PAGEINFO)

PAGEINFO is an architectural data structure that is used as a parameter to the EPC-management instructions. It requires 32-Byte alignment.

Table 37-14. Layout of PAGEINFO Data Structure

Field	OFFSET (Bytes)	Size (Bytes)	Description
LINADDR	0	8	Enclave linear address.
SRCPGE	8	8	Effective address of the page where contents are located.
SECINFO/PCMD	16	8	Effective address of the SECINFO or PCMD (for ELDU, ELDB, EWB) structure for the page.
SECS	24	8	Effective address of EPC slot that currently contains the SECS.

37.11 SECURITY INFORMATION (SECINFO)

The SECINFO data structure holds meta-data about an enclave page.

Table 37-15. Layout of SECINFO Data Structure

Field	OFFSET (Bytes)	Size (Bytes)	Description
FLAGS	0	8	Flags describing the state of the enclave page.
RESERVED	8	56	Must be zero.

37.11.1 SECINFO.FLAGS

The SECINFO.FLAGS are a set of fields describing the properties of an enclave page.

Table 37-16. Layout of SECINFO.FLAGS Field

Field	Bit Position	Description
R	0	If 1 indicates that the page can be read from inside the enclave; otherwise the page cannot be read from inside the enclave.
W	1	If 1 indicates that the page can be written from inside the enclave; otherwise the page cannot be written from inside the enclave.
X	2	If 1 indicates that the page can be executed from inside the enclave; otherwise the page cannot be executed from inside the enclave.
PENDING	3	If 1 indicates that the page is in the PENDING state; otherwise the page is not in the PENDING state.
MODIFIED	4	If 1 indicates that the page is in the MODIFIED state; otherwise the page is not in the MODIFIED state.
PR	5	If 1 indicates that a permission restriction operation on the page is in progress, otherwise a permission restriction operation is not in progress.
RESERVED	7:6	Must be zero.
PAGE_TYPE	15:8	The type of page that the SECINFO is associated with.
RESERVED	63:16	Must be zero.

37.11.2 PAGE_TYPE Field Definition

The SECINFO flags and EPC flags contain bits indicating the type of page.

Table 37-17. Supported PAGE_TYPE

TYPE	Value	Description
PT_SECS	0	Page is an SECS.
PT_TCS	1	Page is a TCS.
PT_REG	2	Page is a regular page.
PT_VA	3	Page is a Version Array.
PT_TRIM	4	Page is in trimmed state.
	All other	Reserved.

37.12 PAGING CRYPTO METADATA (PCMD)

The PCMD structure is used to keep track of crypto meta-data associated with a paged-out page. Combined with PAGEINFO, it provides enough information for the processor to verify, decrypt, and reload a paged-out EPC page. The size of the PCMD structure (128 bytes) is architectural.

EWB calculates the Message Authentication Code (MAC) value and writes out the PCMD. ELDB/U reads the fields and checks the MAC.

The format of PCMD is as follows:

Table 37-18. Layout of PCMD Data Structure

Field	OFFSET (Bytes)	Size (Bytes)	Description
SECINFO	0	64	Flags describing the state of the enclave page; R/W by software.
ENCLAVEID	64	8	Enclave Identifier used to establish a cryptographic binding between paged-out page and the enclave.
RESERVED	72	40	Must be zero.
MAC	112	16	Message Authentication Code for the page, page meta-data and reserved field.

37.13 ENCLAVE SIGNATURE STRUCTURE (SIGSTRUCT)

SIGSTRUCT is a structure created and signed by the enclave developer that contains information about the enclave. SIGSTRUCT is processed by the EINIT leaf function to verify that the enclave was properly built.

SIGSTRUCT includes ENCLAVEHASH as SHA256 digest, as defined in FIPS PUB 180-4. The digests are byte strings of length 32. Each of the 8 HASH dwords is stored in little-endian order.

SIGSTRUCT includes four 3072-bit integers (MODULUS, SIGNATURE, Q1, Q2). Each such integer is represented as a byte strings of length 384, with the most significant byte at the position "offset + 383", and the least significant byte at position "offset".

The (3072-bit integer) SIGNATURE should be an RSA signature, where: a) the RSA modulus (MODULUS) is a 3072-bit integer; b) the public exponent is set to 3; c) the signing procedure uses the EMSA-PKCS1-v1.5 format with DER encoding of the "DigestInfo" value as specified in of PKCS#1 v2.1/RFC 3447.

The 3072-bit integers Q1 and Q2 are defined by:

q1 = floor(Signature^2 / Modulus);

q2 = floor((Signature^3 - q1 * Signature * Modulus) / Modulus);

SIGSTRUCT must be page aligned

In column 5 of Table 37-19, 'Y' indicates that this field should be included in the signature generated by the developer.

Table 37-19. Layout of Enclave Signature Structure (SIGSTRUCT)

Field	OFFSET (Bytes)	Size (Bytes)	Description	Signed
HEADER	0	16	Must be byte stream 06000000E10000000000010000000000H	Y
VENDOR	16	4	Intel Enclave: 00008086H Non-Intel Enclave: 00000000H	Y
DATE	20	4	Build date is yyyymmdd in hex: yyyy=4 digit year, mm=1-12, dd=1-31	Y
HEADER2	24	16	Must be byte stream 01010000600000006000000001000000H	Y
SWDEFINED	40	4	Available for software use.	Y
RESERVED	44	84	Must be zero.	Y
MODULUS	128	384	Module Public Key (keylength=3072 bits).	N
EXPONENT	512	4	RSA Exponent = 3.	N
SIGNATURE	516	384	Signature over Header and Body.	N
MISCSELECT*	900	4	Bit vector specifying Extended SSA frame feature set to be used.	Y
MISCMASK*	904	4	Bit vector mask of MISCSELECT to enforce.	Y
RESERVED	908	20	Must be zero.	Y
ATTRIBUTES	928	16	Enclave Attributes that must be set.	Y
ATTRIBUTEMASK	944	16	Mask of Attributes to enforce.	Y
ENCLAVEHASH	960	32	MRENCLAVE of enclave this structure applies to.	Y
RESERVED	992	32	Must be zero.	Y
ISVPRODID	1024	2	ISV assigned Product ID.	Y
ISVSVN	1026	2	ISV assigned SVN (security version number).	Y
RESERVED	1028	12	Must be zero.	N
Q1	1040	384	Q1 value for RSA Signature Verification.	N
Q2	1424	384	Q2 value for RSA Signature Verification.	N
* If CPUID.(EAX=12H, ECX=0):EBX[31:0] = 0, MISCSELECT must be 0. If CPUID.(EAX=12H, ECX=0):EBX[31:0] !=0, enclave writers must specify MISCSELECT such that each cleared bit in MISCMASK must also specify the corresponding bit as 0 in MISCSELECT.				

37.14 EINIT TOKEN STRUCTURE (EINITTOKEN)

The EINIT token is used by EINIT to verify that the enclave is permitted to launch. EINIT token is generated by an enclave in possession of the EINITTOKEN key (the Launch Enclave).

EINIT token must be 512-Byte aligned.

Table 37-20. Layout of EINIT Token (EINITTOKEN)

Field	OFFSET (Bytes)	Size (Bytes)	MACed	Description
Valid	0	4	Y	Bit 0: 1: Valid; 0: Invalid. All other bits reserved.
RESERVED	4	44	Y	Must be zero.
ATTRIBUTES	48	16	Y	ATTRIBUTES of the Enclave.
MRENCLAVE	64	32	Y	MRENCLAVE of the Enclave.
RESERVED	96	32	Y	Reserved.
MRSIGNER	128	32	Y	MRSIGNER of the Enclave.
RESERVED	160	32	Y	Reserved.
CPUSVNLE	192	16	N	Launch Enclave's CPUSVN.
ISVPRODIDLE	208	02	N	Launch Enclave's ISVPRODID.
ISVSVNLE	210	02	N	Launch Enclave's ISVSVN.
RESERVED	212	24	N	Reserved.
MASKEDMISCSELECTLE	236	4		Launch Enclave's MASKEDMISCSELECT: set by the LE to the resolved MISCSELECT value, used by EGETKEY (after applying KEYREQUEST's masking).
MASKEDATTRIBUTESLE	240	16	N	Launch Enclave's MASKEDATTRIBUTES: This should be set to the LE's ATTRIBUTES masked with ATTRIBUTEMASK of the LE's KEYREQUEST.
KEYID	256	32	N	Value for key wear-out protection.
MAC	288	16	N	Message Authentication Code on EINITTOKEN using EINITTOKEN_KEY.

37.15 REPORT (REPORT)

The REPORT structure is the output of the EREPORT instruction, and must be 512-Byte aligned.

Table 37-21. Layout of REPORT

Field	OFFSET (Bytes)	Size (Bytes)	Description
CPUSVN	0	16	The security version number of the processor.
MISCSELECT	16	4	Bit vector specifying which extended features are saved to the MISC region of the SSA frame when an AEX occurs.
RESERVED	20	28	Must be zero.
ATTRIBUTES	48	16	ATTRIBUTES of the Enclave. See Section 37.7.1.
MRENCLAVE	64	32	The value of SECS.MRENCLAVE.
RESERVED	96	32	Reserved.
MRSIGNER	128	32	The value of SECS.MRSIGNER.
RESERVED	160	96	Zero.
ISVPRODID	256	02	Product ID of enclave.
ISVSVN	258	02	Security version number (SVN) of the enclave.
RESERVED	260	60	Zero.
REPORTDATA	320	64	Data provided by the user and protected by the REPORT's MAC, see Section 37.15.1.
KEYID	384	32	Value for key wear-out protection.
MAC	416	16	Message Authentication Code on the report using report key.

37.15.1 REPORTDATA

REPORTDATA is a 64-Byte data structure that is provided by the enclave and included in the REPORT. It can be used to securely pass information from the enclave to the target enclave.

37.16 REPORT TARGET INFO (TARGETINFO)

This structure is an input parameter to the EREPORT leaf function. The address of TARGETINFO is specified as an effective address in RBX. It is used to identify the target enclave which will be able to cryptographically verify the REPORT structure returned by EREPORT. TARGETINFO must be 512-Byte aligned.

Table 37-22. Layout of TARGETINFO Data Structure

Field	OFFSET (Bytes)	Size (Bytes)	Description
MEASUREMENT	0	32	The MRENCLAVE of the target enclave.
ATTRIBUTES	32	16	The ATTRIBUTES field of the target enclave.
RESERVED	48	4	
MISCSELECT	52	4	The MISCSELECT of the target enclave.
RESERVED	56	456	

37.17 KEY REQUEST (KEYREQUEST)

This structure is an input parameter to the EGETKEY leaf function. It is passed in as an effective address in RBX and must be 512-Byte aligned. It is used for selecting the appropriate key and any additional parameters required in the derivation of that key.

Table 37-23. Layout of KEYREQUEST Data Structure

Field	OFFSET (Bytes)	Size (Bytes)	Description
KEYNAME	0	02	Identifies the Key Required.
KEYPOLICY	02	02	Identifies which inputs are required to be used in the key derivation.
ISVSVN	04	02	The ISV security version number that will be used in the key derivation.
RESERVED	06	02	Must be zero.
CPUSVN	08	16	The security version number of the processor used in the key derivation.
ATTRIBUTEMASK	24	16	A mask defining which ATTRIBUTES bits will be included in key derivation.
KEYID	40	32	Value for key wear-out protection.
MISCMASK	72	4	A mask defining which MISCSELECT bits will be included in key derivation.
RESERVED	76	436	

37.17.1 KEY REQUEST KeyNames

Table 37-24. Supported KEYName Values

Key Name	Value	Description
EINITTOKEN_KEY	0	EINIT_TOKEN key
PROVISION_KEY	1	Provisioning Key
PROVISION_SEAL_KEY	2	Provisioning Seal Key
REPORT_KEY	3	Report Key
SEAL_KEY	4	Seal Key
	All other	Reserved

37.17.2 Key Request Policy Structure

Table 37-25. Layout of KEYPOLICY Field

Field	Bit Position	Description
MRENCLAVE	0	If 1, derive key using the enclave's MRENCLAVE measurement register.
MRSIGNER	1	If 1, derive key using the enclave's MRSIGNER measurement register.
RESERVED	15:2	Must be zero.

37.18 VERSION ARRAY (VA)

In order to securely store the versions of evicted EPC pages, Intel SGX defines a special EPC page type called a Version Array (VA). Each VA page contains 512 slots, each of which can contain an 8-byte version number for a page evicted from the EPC. When an EPC page is evicted, software chooses an empty slot in a VA page; this slot receives the unique version number of the page being evicted. When the EPC page is reloaded, there must be a VA slot that must hold the version of the page. If the page is successfully reloaded, the version in the VA slot is cleared.

VA pages can be evicted, just like any other EPC page. When evicting a VA page, a version slot in some other VA page must be used to hold the version for the VA being evicted. A Version Array Page must be 4K-Bytes aligned.

Table 37-26. Layout of Version Array Data Structure

Field	OFFSET (Bytes)	Size (Bytes)	Description
Slot 0	0	08	Version Slot 0
Slot 1	8	08	Version Slot 1
…			
Slot 511	4088	08	Version Slot 511

37.19 ENCLAVE PAGE CACHE MAP (EPCM)

EPCM is a secure structure used by the processor to track the contents of the EPC. The EPCM holds exactly one entry for each page that is currently loaded into the EPC. EPCM is not accessible by software, and the layout of EPCM fields is implementation specific.

Table 37-27. Content of an Enclave Page Cache Map Entry

Field	Description
VALID	Indicates whether the EPCM entry is valid.
R	Read access; indicates whether enclave accesses for reads are allowed from the EPC page referenced by this entry.
W	Write access; indicates whether enclave accesses for writes are allowed to the EPC page referenced by this entry.
X	Execute access; indicates whether enclave accesses for instruction fetches are allowed from the EPC page referenced by this entry.
PT	EPCM page type (PT_SECS, PT_TCS, PT_REG, PT_VA, PT_TRIM).
ENCLAVESECS	SECS identifier of the enclave to which the EPC page belongs.
ENCLAVEADDRESS	Linear enclave address of the EPC page.
BLOCKED	Indicates whether the EPC page is in the blocked state.
PENDING	Indicates whether the EPC page is in the pending state.
MODIFIED	Indicates whether the EPC page is in the modified state.
PR	Indicates whether the EPC page is in a permission restriction state.

23.Updates to Chapter 38, Volume 3D

Change bars show changes to Chapter 38 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3D:* System Programming guide, Part 4.

Changes to this chapter: Minor typo corrections to Section 38.1.2 “EADD and EEXTEND Interaction”, Section 38.1.4 “Intel® SGX Launch Control Configuration”, Section 38.4.3 “Keys”, Section 38.5.2 “OS Management of EPC Pages”, and Section 38.5.3 “Eviction of Enclave Pages”.

CHAPTER 38
ENCLAVE OPERATION

The following aspects of enclave operation are described in this chapter:

- Enclave creation: Includes loading code and data from outside of enclave into the EPC and establishing the enclave entity.
- Adding pages and measuring the enclave.
- Initialization of an enclave: Finalizes the cryptographic log and establishes the enclave identity and sealing identity.
- Enclave entry and exiting including:
  - — Controlled entry and exit.
  - — Asynchronous Enclave Exit (AEX) and resuming execution after an AEX.

38.1 CONSTRUCTING AN ENCLAVE

Figure 38-1 illustrates a typical Enclave memory layout.

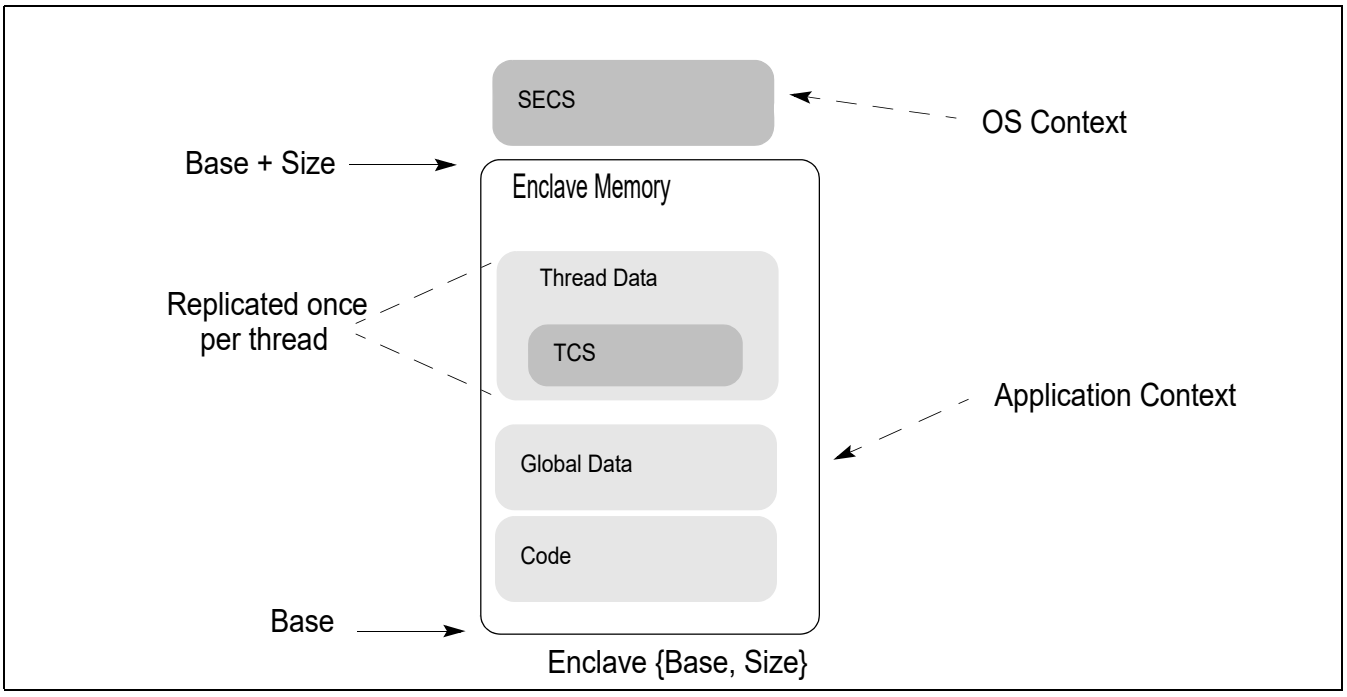


Figure 38-1. Enclave Memory Layout

The enclave creation, commitment of memory resources, and finalizing the enclave's identity with measurement comprises multiple phases. This process can be illustrated by the following exemplary steps:

1. The application hands over the enclave content along with additional information required by the enclave creation API to the enclave creation service running at privilege level 0.
2. The enclave creation service running at privilege level 0 uses the ECREATE leaf function to set up the initial environment, specifying base address and size of the enclave. This address range, the ELRANGE, is part of the application's address space. This reserves the memory range. The enclave will now reside in this address

region. ECREATE also allocates an Enclave Page Cache (EPC) page for the SGX Enclave Control Structure (SECS). Note that this page is not required to be a part of the enclave linear address space and is not required to be mapped into the process.

3. The enclave creation service uses the EADD leaf function to commit EPC pages to the enclave, and use EEXTEND to measure the committed memory content of the enclave. For each page to be added to the enclave:
   — Use EADD to add the new page to the enclave.
   — If the enclave developer requires measurement of the page as a proof for the content, use EEXTEND to add a measurement for 256 bytes of the page. Repeat this operation until the entire page is measured.
4. The enclave creation service uses the EINIT leaf function to complete the enclave creation process and finalize the enclave measurement to establish the enclave identity. Until an EINIT is executed, the enclave is not permitted to execute any enclave code (i.e. entering the enclave by executing EENTER would result in a fault).

38.1.1 ECREATE

The ECREATE leaf function sets up the initial environment for the enclave by reading an SGX Enclave Control Structure (SECS) that contains the enclave's address range (ELRANGE) as defined by BASEADDR and SIZE, the ATTRIBUTES and MISCSELECT bitmaps, and the SSAFRAMESIZE. It then securely stores this information in an Enclave Page Cache (EPC) page. ELRANGE is part of the application's address space. ECREATE also initializes a cryptographic log of the enclave's build process.

38.1.2 EADD and EEXTEND Interaction

Once the SECS has been created, enclave pages can be added to the enclave via EADD. This involves converting a free EPC page into either a PT_REG or a PT_TCS page.

When EADD is invoked, the processor will update the EPCM entry with the type of page (PT_REG or PT_TCS), the linear address used by the enclave to access the page, and the enclave access permissions for the page. It associates the page to the SECS provided as input. The EPCM entry information is used by hardware to manage access control to the page. EADD records EPCM information in the cryptographic log stored in the SECS and copies 4 KBytes of data from unprotected memory outside the EPC to the allocated EPC page.

System software is responsible for selecting a free EPC page. System software is also responsible for providing the type of page to be added, the attributes of the page, the contents of the page, and the SECS (enclave) to which the page is to be added as requested by the application. Incorrect data would lead to a failure of EADD or to an incorrect cryptographic log and a failure at EINIT time.

After a page has been added to an enclave, software can measure a 256 byte region as determined by the developer by invoking EEXTEND. Thus to measure an entire 4KB page, system software must execute EEXTEND 16 times. Each invocation of EEXTEND adds to the cryptographic log information about which region is being measured and the measurement of the section.

Entries in the cryptographic log define the measurement of the enclave and are critical in gaining assurance that the enclave was correctly constructed by the untrusted system software.

38.1.3 EINIT Interaction

Once system software has completed the process of adding and measuring pages, the enclave needs to be initialized by the EINIT leaf function. After an enclave is initialized, EADD and EEXTEND are disabled for that enclave (An attempt to execute EADD/EEXTEND to enclave after enclave initialization will result in a fault). The initialization process finalizes the cryptographic log and establishes the **enclave identity** and **sealing identity** used by EGETKEY and EREPORT.

A cryptographic hash of the log is stored as the **enclave identity**. Correct construction of the enclave results in the cryptographic hash matching the one built by the enclave owner and included as the ENCLAVEHASH field of SIGSTRUCT. The **enclave identity** provided by the EREPORT leaf function can be verified by a remote party.

The EINIT leaf function checks the EINIT token to validate that the enclave has been enabled on this platform. If the enclave is not correctly constructed, or the EINIT token is not valid for the platform, or SIGSTRUCT isn't properly signed, then EINIT will fail. See the EINIT leaf function for details on the error reporting.

The **enclave identity** is a cryptographic hash that reflects the enclave attributes and MISCSELECT value, content of the enclave, the order in which it was built, the addresses it occupies in memory, the security attributes, and access right permissions of each page. The **enclave identity** is established by the EINIT leaf function.

The **sealing identity** is managed by a sealing authority represented by the hash of the public key used to sign the SIGSTRUCT structure processed by EINIT. The sealing authority assigns a product ID (ISVPRODID) and security version number (ISVSVN) to a particular enclave identity.

EINIT establishes the sealing identity using the following steps:

1. Verifies that SIGSTRUCT is properly signed using the public key enclosed in the SIGSTRUCT.
2. Checks that the measurement of the enclave matches the measurement of the enclave specified in SIGSTRUCT.
3. Checks that the enclave's attributes and MISCSELECT values are compatible with those specified in SIGSTRUCT.
4. Finalizes the measurement of the enclave and records the **sealing identity** (the sealing authority, product id and security version number) and **enclave identity** in the SECS.
5. Sets the ATTRIBUTES.INIT bit for the enclave.

38.1.4 Intel® SGX Launch Control Configuration

Intel® SGX Launch Control is a set of controls that govern the creation of enclaves. Before the EINIT leaf function will successfully initialize an enclave, a designated Launch Enclave must create an EINITTOKEN for that enclave. Launch Enclaves have SECS.ATTRIBUTES.EINITTOKEN_KEY = 1, granting them access to the EINITTOKEN_KEY from the EGETKEY leaf function. EINITTOKEN_KEY must be used by the Launch Enclave when computing EINITTOKEN.MAC, the Message Authentication Code of the EINITTOKEN.

The hash of the public key used to sign the SIGSTRUCT of the Launch Enclave must equal the value in the IA32_SGXLEPUBKEYHASH MSRs. Only Launch Enclaves are allowed to launch without a valid token.

The IA32_SGXLEPUBKEYHASH MSRs are provided to designate the platform's Launch Enclave. IA32_SGXLEPUBKEYHASH defaults to digest of Intel's launch enclave signing key after reset.

IA32_FEATURE_CONTROL bit 17 controls the permissions on the IA32_SGXLEPUBKEYHASH MSRs when CPUID.(EAX=12H, ECX=00H):EAX[0] = 1. If IA32_FEATURE_CONTROL is locked with bit 17 set, IA32_SGXLEPUBKEYHASH MSRs are reconfigurable (writeable). If either IA32_FEATURE_CONTROL is not locked or bit 17 is clear, the MSRs are read only. By leaving these MSRs writable, system SW or a VMM can support a plurality of Launch Enclaves for hosting multiple execution environments. See Table 42.2.2 for more details.

38.2 ENCLAVE ENTRY AND EXITING

38.2.1 Controlled Entry and Exit

The EENTER leaf function is the method to enter the enclave under program control. To execute EENTER, software must supply an address of a TCS that is part of the enclave to be entered. The TCS holds the location inside the enclave to transfer control to and a pointer to the SSA frame inside the enclave that an AEX should store the register state to.

When a logical processor enters an enclave, the TCS is considered busy until the logical processors exits the enclave. An attempt to enter an enclave through a busy TCS results in a fault. Intel® SGX allows an enclave builder to define multiple TCSs, thereby providing support for multithreaded enclaves.

Software must also supply to EENTER the Asynchronous Exit Pointer (AEP) parameter. AEP is an address external to the enclave which an exception handler will return to using IRET. Typically the location would contain the ERESUME instruction. ERESUME transfers control back to the enclave, to the address retrieved from the enclave thread's saved state.

EENTER performs the following operations:

1. Check that TCS is not busy and flush all cached linear-to-physical mappings.
2. Change the mode of operation to be in enclave mode.
3. Save the old RSP, RBP for later restore on AEX (Software is responsible for setting up the new RSP, RBP to be used inside enclave).
4. Save XCR0 and replace it with the XFRM value for the enclave.
5. Check if software wishes to debug (applicable to a debuggable enclave):
   — If not debugging, then configure hardware so the enclave appears as a single instruction.
   — If debugging, then configure hardware to allow traps, breakpoints, and single steps inside the enclave.
6. Set the TCS as busy.
7. Transfer control from outside enclave to predetermined location inside the enclave specified by the TCS.

The EEXIT leaf function is the method of leaving the enclave under program control. EEXIT receives the target address outside of the enclave that the enclave wishes to transfer control to. It is the responsibility of enclave software to erase any secret from the registers prior to invoking EEXIT. To allow enclave software to easily perform an external function call and re-enter the enclave (using EEXIT and EENTER leaf functions), EEXIT returns the value of the AEP that was used when the enclave was entered.

EEXIT performs the following operations:

1. Clear enclave mode and flush all cached linear-to-physical mappings.
2. Mark TCS as not busy.
3. Transfer control from inside the enclave to a location on the outside specified as parameter to the EEXIT leaf function.

38.2.2 Asynchronous Enclave Exit (AEX)

Asynchronous and synchronous events, such as exceptions, interrupts, traps, SMIs, and VM exits may occur while executing inside an enclave. These events are referred to as Enclave Exiting Events (EEE). Upon an EEE, the processor state is securely saved inside the enclave (in the thread's current SSA frame) and then replaced by a synthetic state to prevent leakage of secrets. The process of securely saving state and establishing the synthetic state is called an Asynchronous Enclave Exit (AEX). Details of AEX is described in Chapter 39, "Enclave Exiting Events".

As part of most EEEs, the AEP is pushed onto the stack as the location of the eventing address. This is the location where control will return to after executing the IRET. The ERESUME leaf function can be executed from that point to reenter the enclave and resume execution from the interrupted point.

After AEX has completed, the logical processor is no longer in enclave mode and the exiting event is processed normally. Any new events that occur after the AEX has completed are treated as having occurred outside the enclave (e.g. a #PF in dispatching to an interrupt handler).

38.2.3 Resuming Execution after AEX

After system software has serviced the event that caused the logical processor to exit an enclave, the logical processor can continue enclave execution using ERESUME. ERESUME restores processor state and returns control to where execution was interrupted.

If the cause of the exit was an exception or a fault and was not resolved, the event will be triggered again if the enclave is re-entered using ERESUME. For example, if an enclave performs a divide by 0 operation, executing ERESUME will cause the enclave to attempt to re-execute the faulting instruction and result in another divide by 0 exception. Intel® SGX provides the means for an enclave developer to handle enclave exceptions from within the enclave. Software can enter the enclave at a different location and invoke the exception handler within the enclave by executing the EENTER leaf function. The exception handler within the enclave can read the fault information from the SSA frame and attempt to resolve the faulting condition or simply return and indicate to software that the enclave should be terminated (e.g. using EEXIT).

38.2.3.1 ERESUME Interaction

ERESUME restores registers depending on the mode of the enclave (32 or 64 bit).

- In 32-bit mode (IA32_EFER.LMA = 0 || CS.L = 0), the low 32-bits of the legacy registers (EAX, EBX, ECX, EDX, ESP, EBP, ESI, EDI, EIP and EFLAGS) are restored from the thread's GPR area of the current SSA frame. Neither the upper 32 bits of the legacy registers nor the 64-bit registers (R8 … R15) are loaded.
- In 64-bit mode (IA32_EFER.LMA = 1 && CS.L = 1), all 64 bits of the general processor registers (RAX, RBX, RCX, RDX, RSP, RBP, RSI, RDI, R8 … R15, RIP and RFLAGS) are loaded.

Extended features specified by SECS.ATTRIBUTES.XFRM are restored from the XSAVE area of the current SSA frame. The layout of the x87 area depends on the current values of IA32_EFER.LMA and CS.L:

- IA32_EFER.LMA = 0 || CS.L = 0
  - 32-bit load in the same format that XSAVE/FXSAVE uses with these values.
- IA32_EFER.LMA = 1 && CS.L = 1
  - 64-bit load in the same format that XSAVE/FXSAVE uses with these values as if REX.W = 1.

38.3 CALLING ENCLAVE PROCEDURES

38.3.1 Calling Convention

In standard call conventions subroutine parameters are generally pushed onto the stack. The called routine, being aware of its own stack layout, knows how to find parameters based on compile-time-computable offsets from the SP or BP register (depending on runtime conventions used by the compiler).

Because of the stack switch when calling an enclave, stack-located parameters cannot be found in this manner. Entering the enclave requires a modified parameter passing convention.

For example, the caller might push parameters onto the untrusted stack and then pass a pointer to those parameters in RAX to the enclave software. The exact choice of calling conventions is up to the writer of the edge routines; be those routines hand-coded or compiler generated.

38.3.2 Register Preservation

As with most systems, it is the responsibility of the callee to preserve all registers except that used for returning a value. This is consistent with conventional usage and tends to optimize the number of register save/restore operations that need be performed. It has the additional security result that it ensures that data is scrubbed from any registers that were used by enclave to temporarily contain secrets.

38.3.3 Returning to Caller

No registers are modified during EEXIT. It is the responsibility of software to remove secrets in registers before executing EEXIT.

38.4 INTEL® SGX KEY AND ATTESTATION

38.4.1 Enclave Measurement

During the enclave build process, two "measurements" are taken of each enclave and are stored in two 256-bit Measurement Registers (MR): MRENCLAVE and MRSIGNER. MRENCLAVE represents the enclave's contents and build process. MRSIGNER represents the entity that signed the enclave's SIGSTRUCT.

The values of the Measurement Registers are included in attestations to identify the enclave to remote parties. The MRs are also included in most keys, binding keys to enclaves with specific MRs.

38.4.1.1 MRENCLAVE

MRENCLAVE is a unique 256 bit value that identifies the code and data that was loaded into the enclave during the initial launch. It is computed as a SHA256 hash that is initialized by the ECREATE leaf function. EADD and EEXTEND leaf functions record information about each page and the content of those pages. The EINIT leaf function finalizes the hash, which is stored in SECS.MRENCLAVE. Any tampering with the build process, contents of a page, page permissions, etc will result in a different MRENCLAVE value.

Figure 38-2 illustrates a simplified flow of changes to the MRENCLAVE register when building an enclave:

- Enclave creation with ECREATE.
- Copying a non-enclave source page into the EPC of an un-initialized enclave with EADD.
- Updating twice of the MRENCLAVE after modifying the enclave's page content, i.e. EEXTEND twice.
- Finalizing the enclave build with EINIT.

Details on specific values inserted in the hash are available in the individual instruction definitions.

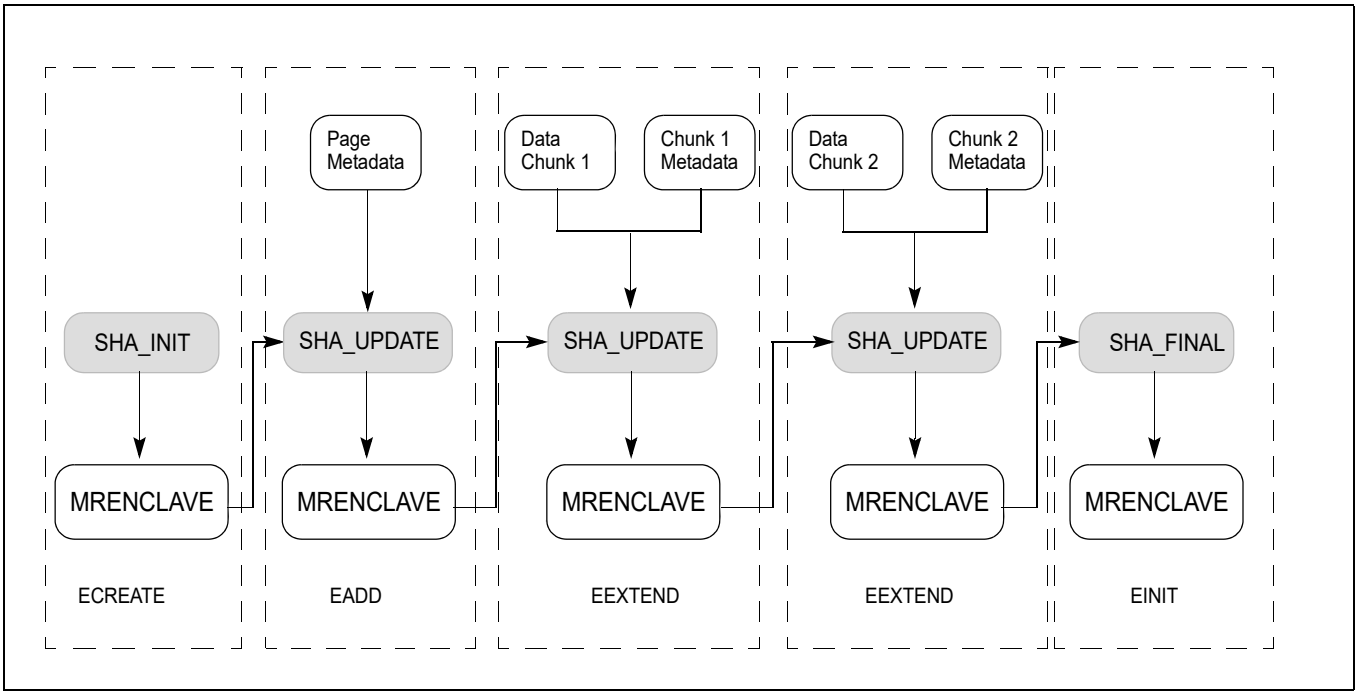


Figure 38-2. Measurement Flow of Enclave Build Process

38.4.1.2 MRSIGNER

Each enclave is signed using a 3072 bit RSA key. The signature is stored in the SIGSTRUCT. In the SIGSTRUCT, the enclave's signer also assigns a product ID (ISVPRODID) and a security version (ISVSVN) to the enclave. MRSIGNER is the SHA-256 hash of the signer's public key.

In attestation, MRSIGNER can be used to allow software to approve of an enclave based on the author rather than maintaining a list of MRENCLAVEs. It is used in key derivation to allow software to create a lineage of an application. By signing multiple enclaves with the same key, the enclaves will share the same keys and data. Combined with security version numbering, the author can release multiple versions of an application which can access keys for previous versions, but not future versions of that application.

38.4.2 Security Version Numbers (SVN)

Intel® SGX supports a versioning system that allows the signer to identify different versions of the same software released by an author. The security version is independent of the functional version an author uses and is intended to specify security equivalence. Multiple releases with functional enhancements may all share the same SVN if they all have the same security properties or posture. Each enclave has an SVN and the underlying hardware has an SVN.

The SVNs are attested to in EREPORT and are included in the derivation of most keys, thus providing separation between data for older/newer versions.

38.4.2.1 Enclave Security Version

In the SIGSTRUCT, the MRSIGNER is associated with a 16-bit Product ID (ISVPRODID) and a 16 bit integer SVN (ISVSVN). Together they define a specific group of versions of a specific product. Most keys, including the Seal Key, can be bound to this pair.

To support upgrading from one release to another, EGETKEY will return keys corresponding to any value less than or equal to the software's ISVSVN.

38.4.2.2 Hardware Security Version

CPUSVN is a 128 bit value that reflects the microcode update version and authenticated code modules supported by the processor. Unlike ISVSVN, CPUSVN is not an integer and cannot be compared mathematically. Not all values are valid CPUSVNs.

Software must ensure that the CPUSVN provided to EGETKEY is valid. EREPORT will return the CPUSVN of the current environment. Software can execute EREPORT with TARGETINFO set to zeros to retrieve a CPUSVN from REPORTDATA. Software can access keys for a CPUSVN recorded previously, provided that each of the elements reflected in CPUSVN are the same or have been upgraded.

38.4.3 Keys

Intel® SGX provides software with access to keys unique to each processor and rooted in HW keys inserted into the processor during manufacturing.

Each enclave requests keys using the EGETKEY leaf function. The key is based on enclave parameters such as measurement, the enclave signing key, security attributes of the enclave, and the Hardware Security version of the processor itself. A full list of parameter options is specified in the KEYREQUEST structure, see details in Section 37.17.

By deriving keys using enclave properties, SGX guarantees that if two enclaves call EGETKEY, they will receive a unique key only accessible by the respective enclave. It also guarantees that the enclave will receive the same key on every future execution of EGETKEY. Some parameters are optional or configurable by software. For example, a Seal key can be based on the signer of the enclave, resulting in a key available to multiple enclaves signed by the same party.

The EGETKEY leaf function provides several key types. Each key is specific to the processor, CPUSVN, and the enclave that executed EGETKEY. The EGETKEY instruction definition details how each of these keys is derived, see Table 40-56. Additionally,

- SEAL Key: The Seal key is a general purpose key for the enclave to use to protect secrets. Typical uses of the Seal key are encrypting and calculating MAC of secrets on disk. There are 2 types of Seal Key described in Section 38.4.3.1.
- REPORT Key: This key is used to compute the MAC on the REPORT structure. The EREPORT leaf function is used to compute this MAC, and destination enclave uses the Report key to verify the MAC. The software usage flow is detailed in Section 38.4.3.2.
- EINITTOKEN_KEY: This key is used by Launch Enclaves to compute the MAC on EINITTOKENs. These tokens are then verified in the EINIT leaf function. The key is only available to enclaves with ATTRIBUTE.EINITTOKEN_KEY set to 1.

- PROVISIONING Key and PROVISIONING SEAL Key: These keys are used by attestation key provisioning software to prove to remote parties that the processor is genuine and identify the currently executing TCB. These keys are only available to enclaves with ATTRIBUTE.PROVISIONKEY set to 1.

38.4.3.1 Sealing Enclave Data

Enclaves can protect persistent data using Seal keys to provide encryption and/or integrity protection. EGETKEY provides two types of Seal keys specified in KEYREQUEST.KEYPOLICY field: MRENCLAVE-based key and MRSIGNER-based key.

The MRENCLAVE-based keys are available only to enclave instances sharing the same MRENCLAVE. If a new version of the enclave is released, the Seal keys will be different. Retrieving previous data requires additional software support.

The MRSIGNER-based keys are bound to the 3 tuple (MRSIGNER, ISVPRODID, ISVSVN). These keys are available to any enclave with the same MRSIGNER and ISVPRODID and an ISVSVN equal to or greater than the key in questions. This is valuable for allowing new versions of the same software to retrieve keys created before an upgrade.

38.4.3.2 Using REPORTs for Local Attestation

SGX provides a means for enclaves to securely identify one another, this is referred to as "Local Attestation". SGX provides a hardware assertion, REPORT that contains calling enclaves Attributes, Measurements and User supplied data (described in detail in Section 37.15). Figure 38-3 shows the basic flow of information.

1. The source enclave determines the identity of the target enclave to populate TARGETINFO.
2. The source enclave calls EREPORT instruction to generate a REPORT structure. The EREPORT instruction conducts the following:
   — Populates the REPORT with identify information about the calling enclave.
   — Derives the Report Key that is returned when the target enclave executes the EGETKEY. TARGETINFO provides information about the target.
   — Computes a MAC over the REPORT using derived target enclave Report Key.
3. Non-enclave software copies the REPORT from source to destination.
4. The target enclave executes the EGETKEY instruction to request its REPORT key, which is the same key used by EREPORT at the source.
5. The target enclave verifies the MAC and can then inspect the REPORT to identify the source.

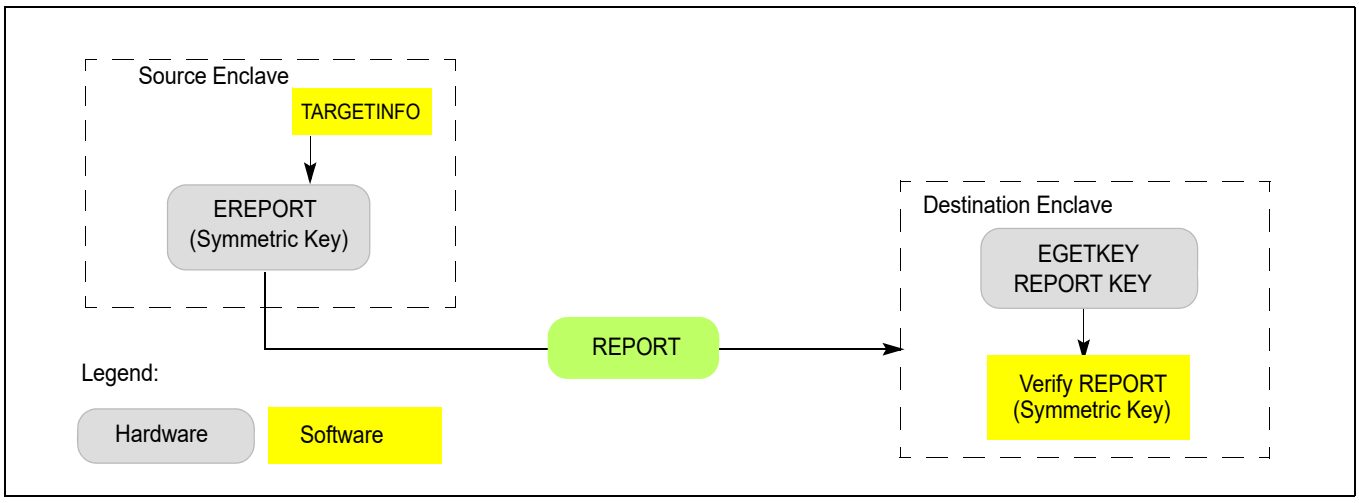


Figure 38-3. SGX Local Attestation

38.5 EPC AND MANAGEMENT OF EPC PAGES

EPC layout is implementation specific, and is enumerated through CPUID (see Table 36-6 for EPC layout). EPC is typically configured by BIOS at system boot time.

38.5.1 EPC Implementation

EPC must be properly protected against attacks. One example of EPC implementation could use a Memory Encryption Engine (MEE). An MEE provides a cost-effective mechanism of creating cryptographically protected volatile storage using platform DRAM. These units provide integrity, replay, and confidentiality protection. Details are implementation specific.

38.5.2 OS Management of EPC Pages

The EPC is a finite resource. SGX1 (i.e. CPUID.(EAX=12H, ECX=0):EAX.SGX1 = 1 but CPUID.(EAX=12H, ECX=0):EAX.SGX2 = 0) provides the EPC manager with leaf functions to manage this resource and properly swap pages out of and into the EPC. For that, the EPC manager would need to keep track of all EPC entries, type and state, context affiliation, and SECS affiliation.

Enclave pages that are candidates for eviction should be moved to BLOCKED state using EBLOCK instruction that ensures no new cached virtual to physical address mappings can be created by attempts to reference a BLOCKED page.

Before evicting blocked pages, EPC manager should execute ETRACK leaf function on that enclave and ensure that there are no stale cached virtual to physical address mappings for the blocked pages remain on any thread on the platform.

After removing all stale translations from blocked pages, system software should use the EWB leaf function for securely evicting pages out of the EPC. EWB encrypts a page in the EPC, writes it to unprotected memory, and invalidates the copy in EPC. In addition, EWB also creates a cryptographic MAC (PCMD.MAC) of the page and stores it in unprotected memory. A page can be reloaded back to the processor only if the data and MAC match. To ensure that only the latest version of the evicted page can be loaded back, the version of the evicted page is stored securely in a Version Array (VA) in EPC.

SGX1 includes two instructions for reloading pages that have been evicted by system software: ELDU and ELDB. The difference between the two instructions is the value of the paging state at the end of the instruction. ELDU results in a page being reloaded and set to an UNBLOCKED state, while ELDB results in a page loaded to a BLOCKED state.

ELDB is intended for use by a Virtual Machine Monitor (VMM). When a VMM reloads an evicted page, it needs to restore it to the correct state of the page (BLOCKED vs. UNBLOCKED) as it existed at the time the page was evicted. Based on the state of the page at eviction, the VMM chooses either ELDB or ELDU.

38.5.2.1 Enhancement to Managing EPC Pages

On processors supporting SGX2 (i.e. CPUID.(EAX=12H, ECX=0):EAX.SGX2 = 1), the EPC manager can manage EPC resources (while enclave is running) with more flexibility provided by the SGX2 leaf functions. The additional flexibility is described in Section 38.5.7 through Section 38.5.11.

38.5.3 Eviction of Enclave Pages

Intel SGX paging is optimized to allow the Operating System (OS) to evict multiple pages out of the EPC under a single synchronization.

The suggested flow for evicting a list of pages from the EPC is:

1. For each page to be evicted from the EPC:
   a. Select an empty slot in a Version Array (VA) page.
      - If no empty VA page slots exist, create a new VA page using the EPA leaf function.

b. Remove linear-address to physical-address mapping from the enclave context's mapping tables (page table and EPT tables).

c. Execute the EBLOCK leaf function for the target page. This sets the target page state to BLOCKED. At this point no new mappings of the page will be created. So any access which does not have the mapping cached in the TLB will generate a #PF.

2. For each enclave containing pages selected in step 1:

— Execute an ETRACK leaf function pointing to that enclave's SECS. This initiates the tracking process that ensures that all caching of linear-address to physical-address translations for the blocked pages is cleared.

3. For all logical processors executing in processes (OS) or guests (VMM) that contain the enclaves selected in step 1:

— Issue an IPI (inter-processor interrupt) to those threads. This causes those logical processors to asynchronously exit any enclaves they might be in, and as a result flush cached linear-address to physical-address translations that might hold stale translations to blocked pages. There is no need for additional measures such as performing a "TLB shootdown".

4. After enclaves exit, allow logical processors to resume normal operation, including enclave re-entry as the tracking logic keeps track of the activity.

5. For each page to be evicted:

— Evict the page using the EWB leaf function with parameters include the effective-address pointer to the EPC page, the VA slot, a 4K byte buffer to hold the encrypted page contents, and a 128 byte buffer to hold page metadata. The last three elements are tied together cryptographically and must be used to later reload the page.

At this point, system software has the only copy of each page data encrypted with its page metadata in main memory.

38.5.4 Loading an Enclave Page

To reload a previously evicted page, system software needs four elements: the VA slot used when the page was evicted, a buffer containing the encrypted page contents, a buffer containing the page metadata, and the parent SECS to associate this page with. If the VA page or the parent SECS are not already in the EPC, they must be reloaded first.

1. Execute ELDB/ELDU (depending on the desired BLOCKED state for the page), passing as parameters: the EPC page linear address, the VA slot, the encrypted page, and the page metadata.

2. Create a mapping in the enclave context's mapping tables (page tables and EPT tables) to allow the application to access that page (OS: system page table; VMM: EPT).

The ELDB/ELDU instruction marks the VA slot empty so that the page cannot be replayed at a later date.

38.5.5 Eviction of an SECS Page

The eviction of an SECS page is similar to the eviction of an enclave page. The only difference is that an SECS page cannot be evicted until all other pages belonging to the enclave have been evicted. Since all other pages have been evicted, there will be no threads executing inside the enclave and tracking with ETRACK isn't necessary. When reloading an enclave, the SECS page must be reloaded before all other constituent pages.

1. Ensure all pages are evicted from enclave.

2. Select an empty slot in a Version Array page.

— If no VA page exists with an empty slot, create a new one using the EPA function leaf.

3. Evict the page using the EWB leaf function with parameters include the effective-address pointer to the EPC page, the VA slot, a 4K byte buffer to hold the encrypted page contents and a 128 byte buffer to hold page metadata. The last three elements are tied together cryptographically and must be used to later reload the page.

38.5.6 Eviction of a Version Array Page

VA pages do not belong to any enclave and tracking with ETRACK isn't necessary. When evicting the VA page, a slot in a different VA page must be specified in order to provide versioning of the evicted VA page.

1. Select a slot in a Version Array page other than the page being evicted.
   — If no VA page exists with an empty slot, create a new one using the EPA leaf function.
2. Evict the page using the EWB leaf function with parameters include the effective-address pointer to the EPC page, the VA slot, a 4K byte buffer to hold the encrypted page contents, and a 128 byte buffer to hold page metadata. The last three elements are tied together cryptographically and must be used to later reload the page.

38.5.7 Allocating a Regular Page

On processors that support SGX2, allocating a new page to an already initialized enclave is accomplished by invoking the EAUG leaf function. Typically, the enclave requests that the OS allocate a new page at a particular location within the enclave's address space. Once allocated, the page remains in a pending state until the enclave executes the corresponding EACCEPT leaf function to accept the new page into the enclave. Page allocation operations may be batched to improve efficiency.

The typical process for allocating a regular page is as follows:

1. Enclave requests additional memory from OS when the current allocation becomes insufficient.
2. The OS invokes the EAUG leaf function to add a new memory page to the enclave.
   a. EAUG may only be called on a free EPC page.
   b. Successful completion of the EAUG instruction places the target page in the VALID and PENDING state.
   c. All dynamically created pages have the type PT_REG and content of all zeros.
3. The OS maps the page in the enclave context's mapping tables.
4. The enclave issues an EACCEPT instruction, which verifies the page's attributes and clears the PENDING state. At that point the page becomes accessible for normal enclave use.

38.5.8 Allocating a TCS Page

On processors that support SGX2, allocating a new TCS page to an already initialized enclave is a two-step process. First the OS allocates a regular page with a call to EAUG. This page must then be accepted and initialized by the enclave to which it belongs. Once the page has been initialized with appropriate values for a TCS page, the enclave requests the OS to change the page's type to PT_TCS. This change must also be accepted. As with allocating a regular page, TCS allocation operations may be batched.

A typical process for allocating a TCS page is as follows:

1. Enclave requests an additional page from the OS.
2. The OS invokes EAUG to add a new regular memory page to the enclave.
   a. EAUG may only be called on a free EPC page.
   b. Successful completion of the EAUG instruction places the target page in the VALID and PENDING state.
3. The OS maps the page in the enclave context's mapping tables.
4. The enclave issues an EACCEPT instruction, at which point the page becomes accessible for normal enclave use.
5. The enclave initializes the contents of the new page.
6. The enclave requests that the OS convert the page from type PT_REG to PT_TCS.
7. OS issues an EMODT instruction on the page.
   a. The parameters to EMODT indicate that the regular page should be converted into a TCS.

b. EMODT forces all access rights to a page to be removed because TCS pages may not be accessed by enclave code.

8. The enclave issues an EACCEPT instruction to confirm the requested modification.

38.5.9 Trimming a Page

On processors that support SGX2, Intel SGX supports the trimming of an enclave page as a special case of EMODT. Trimming allows an enclave to actively participate in the process of removing a page from the enclave (deallocation) by splitting the process into first removing it from the enclave's access and then removing it from the EPC using the EREMOVE leaf function. The page type PT_TRIM indicates that a page has been trimmed from the enclave's address space and that the page is no longer accessible to enclave software. Modifications to a page in the PT_TRIM state are not permitted; the page must be removed and then reallocated by the OS before the enclave may use the page again. Page deallocation operations may be batched to improve efficiency.

The typical process for trimming a page from an enclave is as follows:

1. Enclave signals OS that a particular page is no longer in use.
2. OS invokes the EMODT leaf function on the page, requesting that the page's type be changed to PT_TRIM.
   a. SECS and VA pages cannot be trimmed in this way, so the initial type of the page must be PT_REG or PT_TCS.
   b. EMODT may only be called on valid enclave pages.
3. OS invokes the ETRACK leaf function on the enclave containing the page to track removal the TLB addresses from all the processors.
4. Issue an IPI (inter-processor interrupt) to flush the stale linear-address to physical-address translations for all logical processors executing in processes that contain the enclave.
5. Enclave issues an EACCEPT leaf function.
6. The OS may now permanently remove the page from the EPC (by issuing EREMOVE).

38.5.10 Restricting the EPCM Permissions of a Page

On processors that support SGX2, restricting the EPCM permissions associated with an enclave page is accomplished using the EMODPR leaf function. This operation requires the cooperation of the OS to flush stale entries to the page and to update the page-table permissions of the page to match. Permissions restriction operations may be batched.

The typical process for restricting the permissions of an enclave page is as follows:

1. Enclave requests that the OS to restrict the permissions of an EPC page.
2. OS performs permission restriction, flushing cached linear-address to physical-address translations, and page-table modifications.
   a. Invokes the EMODPR leaf function to restrict permissions (EMODPR may only be called on VALID pages).
   b. Invokes the ETRACK leaf function on the enclave containing the page to track removal of the TLB addresses from all the processor.
   c. Issue an IPI (inter-processor interrupt) to flush the stale linear-address to physical-address translations for all logical processors executing in processes that contain the enclave.
   d. Sends IPIs to trigger enclave thread exit and TLB shootdown.
   e. OS informs the Enclave that all logical processors should now see the new restricted permissions.
3. Enclave invokes the EACCEPT leaf function.
   a. Enclave may access the page throughout the entire process.
   b. Successful call to EACCEPT guarantees that no stale cached linear-address to physical-address translations are present.

38.5.11 Extending the EPCM Permissions of a Page

On processors that support SGX2, extending the EPCM permissions associated with an enclave page is accomplished directly be the enclave using the EMODPE leaf function. After performing the EPCM permission extension, the enclave requests the OS to update the page table permissions to match the extended permission. Security wise, permission extension does not require enclave threads to leave the enclave as TLBs with stale references to the more restrictive permissions will be flushed on demand, but to allow forward progress, an OS needs to be aware that an application might signal a page fault.

The typical process for extending the permissions of an enclave page is as follows:

1. Enclave invokes EMODPE to extend the EPCM permissions associated with an EPC page (EMODPE may only be called on VALID pages).
2. Enclave requests that OS update the page tables to match the new EPCM permissions.
3. Enclave code resumes.
   a. If cached linear-address to physical-address translations are present to the more restrictive permissions, the enclave thread will page fault. The SGX2-aware OS will see that the page tables permit the access and resume the thread, which can now successfully access the page because exiting cleared the TLB.
   b. If cached linear-address to physical-address translations are not present, access to the page with the new permissions will succeed without an enclave exit.

38.6 CHANGES TO INSTRUCTION BEHAVIOR INSIDE AN ENCLAVE

This section covers instructions whose behavior changes when executed in enclave mode.

38.6.1 Illegal Instructions

The instructions listed in Table 38-1 are ring 3 instructions which become illegal when executed inside an enclave. Executing these instructions inside an enclave will generate an exception.

The first row of Table 38-1 enumerates instructions that may cause a VM exit for VMM emulation. Since a VMM cannot emulate enclave execution, execution of any these instructions inside an enclave results in an invalid-opcode exception (#UD) and no VM exit.

The second row of Table 38-1 enumerates I/O instructions that may cause a fault or a VM exit for emulation. Again, enclave execution cannot be emulated, so execution of any these instructions inside an enclave results in #UD.

The third row of Table 38-1 enumerates instructions that load descriptors from the GDT or the LDT or that change privilege level. The former class is disallowed because enclave software should not depend on the contents of the descriptor tables and the latter because enclave execution must be entirely with CPL = 3. Again, execution of any these instructions inside an enclave results in #UD.

The fourth row of Table 38-1 enumerates instructions that provide access to kernel information from user mode and can be used to aid kernel exploits from within enclave. Execution of any these instructions inside an enclave results in #UD

Table 38-1. Illegal Instructions Inside an Enclave

Instructions	Result	Comment
CPUID, GETSEC, RDPMC, SGDT, SIDT, SLDT, STR, VMCALL, VMFUNC	#UD	Might cause VM exit.
IN, INS/INSB/INSW/INSD, OUT, OUTS/OUTSB/OUTSW/OUTSD	#UD	I/O fault may not safely recover. May require emulation.
Far call, Far jump, Far Ret, INT n/INTO, IRET, LDS/LES/LFS/LGS/LSS, MOV to DS/ES/SS/FS/GS, POP DS/ES/SS/FS/GS, SYSCALL, SYSENTER	#UD	Access segment register could change privilege level.
SMSW	#UD	Might provide access to kernel information.
ENCLU[EENTER], ENCLU[ERESUME]	#GP	Cannot enter an enclave from within an enclave.

RDTSC and RDTSCP are legal inside an enclave for processors that support SGX2 (subject to the value of CR4.TSD). For processors which support SGX1 but not SGX2, RDTSC and RDTSCP will cause #UD.

RDTSC and RDTSCP instructions may cause a VM exit when inside an enclave.

Software developers must take into account that the RDTSC/RDTSCP results are not immune to influences by other software, e.g. the TSC can be manipulated by software outside the enclave.

38.6.2 RDRAND and RDSEED Instructions

These instructions may cause a VM exit if the "RDRAND exiting" VM-execution control is 1. Unlike other instructions that can cause VM exits, these instructions are legal inside an enclave. As noted in Section 6.5.5, any VM exit originating on an instruction boundary inside an enclave sets bit 27 of the exit-reason field of the VMCS. If a VMM receives a VM exit due to an attempt to execute either of these instructions determines (by that bit) that the execution was inside an enclave, it can do either of two things. It can clear the "RDRAND exiting" VM-execution control and execute VMRESUME; this will result in the enclave executing RDRAND or RDSEED again, and this time a VM exit will not occur. Alternatively, the VMM might choose to discontinue execution of this virtual machine.

NOTE

It is expected that VMMs that virtualize Intel SGX will not set "RDRAND exiting" to 1.

38.6.3 PAUSE Instruction

The PAUSE instruction may cause a VM exit if either of the "PAUSE exiting" and "PAUSE-loop exiting" VM-execution controls is 1. Unlike other instructions that can cause VM exits, the PAUSE instruction is legal inside an enclave.

If a VMM receives a VM exit due to the 1-setting of "PAUSE-loop exiting", it may take action to prevent recurrence of the PAUSE loop (e.g., by scheduling another virtual CPU of this virtual machine) and then execute VMRESUME; this will result in the enclave executing PAUSE again, but this time the PAUSE loop (and resulting VM exit) will not occur.

If a VMM receives a VM exit due to the 1-setting of "PAUSE exiting", it can do either of two things. It can clear the "PAUSE exiting" VM-execution control and execute VMRESUME; this will result in the enclave executing PAUSE again, but this time a VM exit will not occur. Alternatively, the VMM might choose to discontinue execution of this virtual machine.

NOTE

It is expected that VMMs that virtualize Intel SGX will not set "PAUSE exiting" to 1.

38.6.4 INT 3 Behavior Inside an Enclave

INT3 is legal inside an enclave, however, the behavior inside an enclave is different from its behavior outside an enclave. See Section 42.4.1 for details.

38.6.5 INVD Handling when Enclaves Are Enabled

Once processor reserved memory protections are activated (see Section 38.5), any execution of INVD will result in a #GP(0).

24.Updates to Chapter 39, Volume 3D

Change bars show changes to Chapter 39 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3D:* System Programming Guide, Part 4.

Changes to this chapter: Minor typo corrections to introduction paragraph, and Section 39.2 “State Saving by AEX”.

CHAPTER 39
ENCLAVE EXITING EVENTS

Certain events, such as exceptions and interrupts, incident to (but asynchronous with) enclave execution may cause control to transition outside of enclave mode. (Most of these also cause a change of privilege level.) To protect the integrity and security of the enclave, the processor will exit the enclave (and enclave mode) before invoking the handler for such an event. For that reason, such events are called **enclave-exiting events** (EEE); EEEs include external interrupts, non-maskable interrupts, system-management interrupts, exceptions, and VM exits.

The process of leaving an enclave in response to an EEE is called an **asynchronous enclave exit** (AEX). To protect the secrecy of the enclave, an AEX saves the state of certain registers within enclave memory and then loads those registers with fixed values called **synthetic state**.

39.1 COMPATIBLE SWITCH TO THE EXITING STACK OF AEX

AEXs load registers with a pre-determined synthetic state. These register may be later pushed onto the appropriate stack in a form as defined by the enclave-exiting event. To allow enclave execution to resume after the invoking handler has process the enclave exiting event, the asynchronous enclave exit loads the address of trampoline code outside of the enclave into RIP. This trampoline code eventually returns to the enclave by means of an ENCLU(ERESUME) leaf function. Prior to exiting the enclave the RSP and RBP registers are restored to their values prior to enclave entry.

The stack to be used is chosen using the same rules as for non-SGX mode:

- If there is a privilege level change, the stack will be the one associated with the new ring.
- If there is no privilege level change, the current application stack is used.
- If the IA-32e IST mechanism is used, the exit stack is chosen using that method.

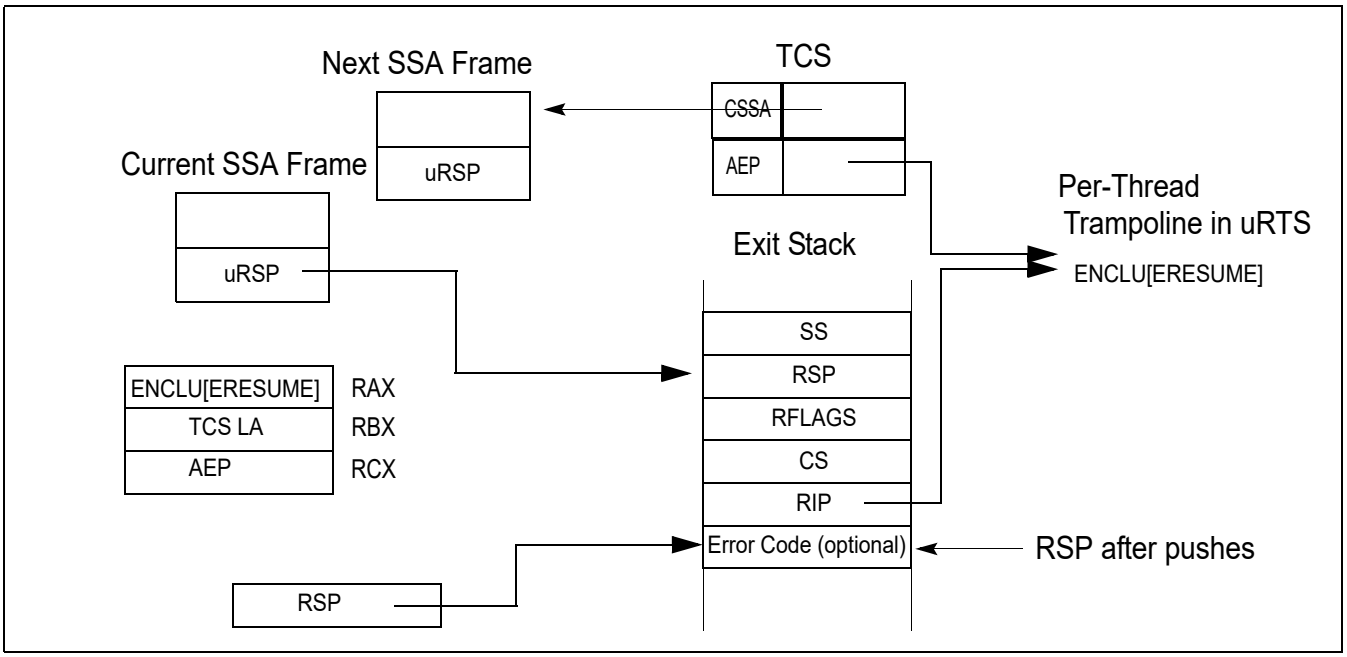


Figure 39-1. Exit Stack Just After Interrupt with Stack Switch

In all cases, the choice of exit stack and the information pushed onto it is consistent with non-SGX operation. Figure 39-1 shows the Application and Exiting Stacks after an exit with a stack switch. An exit without a stack switch uses the Application Stack. The ERESUME leaf index value is placed into RAX, the TCS pointer is placed in RBX and the AEP (see below) is placed into RCX to facilitate resuming the enclave after the exit.

Upon an AEX, the AEP (Asynchronous Exit Pointer) is loaded into the RIP. The AEP points to a trampoline code sequence which includes the ERESUME instruction that is later used to reenter the enclave.

The following bits of RFLAGS are cleared before RFLAGS is pushed onto the exit stack: CF, PF, AF, ZF, SF, OF, RF. The remaining bits are left unchanged.

39.2 STATE SAVING BY AEX

The State Save Area holds the processor state at the time of an AEX. To allow handling events within the enclave and re-entering it after an AEX, the SSA can be a stack of multiple SSA frames as illustrated in Figure 39-2.

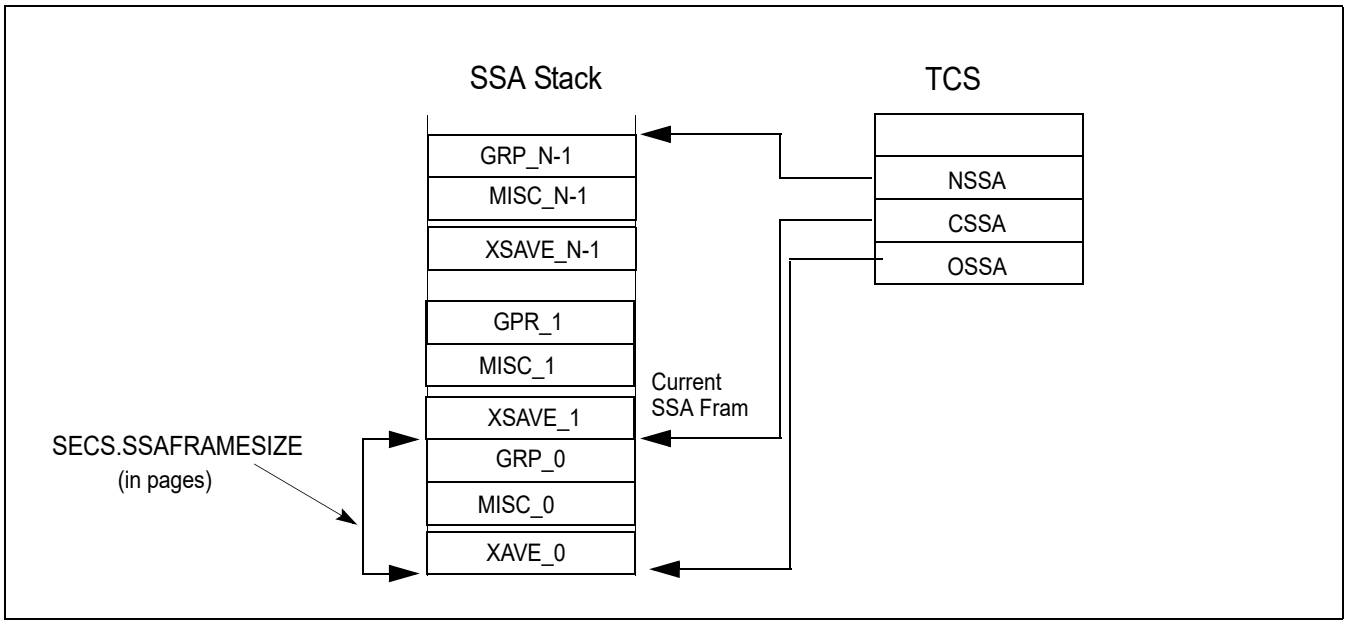


Figure 39-2. The SSA Stack

The location of the SSA frames to be used is controlled by the following variables in the TCS and the SECS:

- Size of a frame in the State Save Area (SECS.SSAFRAMESIZE): This defines the number of 4K Byte pages in a single frame in the State Save Area. The SSA frame size must be large enough to hold the GPR state, the XSAVE state, and the MISC state.
- Base address of the enclave (SECS.BASEADDR): This defines the enclave's base linear address from which the offset to the base of the SSA stack is calculated.
- Number of State Save Area Slots (TCS.NSSA): This defines the total number of slots (frames) in the State Save Area stack.
- Current State Save Area Slot (TCS.CSSA): This defines the slot to use on the next exit.
- State Save Area (TCS.OSSA): This defines the offset of the base address of a set of State Save Area slots from the enclave’s base address.

When an AEX occurs, hardware selects the SSA frame to use by examining TCS.CSSA. Processor state is saved into the SSA frame (see Section 39.4) and loaded with a synthetic state (as described in Section 39.3.1) to avoid leaking secrets, RSP and RBP are restored to their values prior to enclave entry, and TCS.CSSA is incremented. As will be described later, if an exception takes the last slot, it will not be possible to reenter the enclave to handle the

exception from within the enclave. A subsequent ERESUME restores the processor state from the current SSA frame and frees the SSA frame.

The format of the XSAVE section of SSA is identical to the format used by the XSAVE/XRSTOR instructions. On EENTER, CSSA must be less than NSSA, ensuring that there is at least one State Save Area slot available for exits. If there is no free SSA frame when executing EENTER, the entry will fail.

39.3 SYNTHETIC STATE ON ASYNCHRONOUS ENCLAVE EXIT

39.3.1 Processor Synthetic State on Asynchronous Enclave Exit

Table 39-1 shows the synthetic state loaded on AEX. The values shown are the lower 32 bits when the processor is in 32 bit mode and 64 bits when the processor is in 64 bit mode.

Table 39-1. GPR, x87 Synthetic States on Asynchronous Enclave Exit

Register	Value
RAX	3 (ENCLU[3] is ERESUME).
RBX	Pointer to TCS of interrupted enclave thread.
RCX	AEP of interrupted enclave thread.
RDX, RSI, RDI	0.
RSP	Restored from SSA.uRSP.
RBP	Restored from SSA.uRBP.
R8-R15	0 in 64-bit mode; unchanged in 32-bit mode.
RIP	AEP of interrupted enclave thread.
RFLAGS	CF, PF, AF, ZF, SF, OF, RF bits are cleared. All other bits are left unchanged.
x87/SSE State	Unless otherwise listed here, all x87 and SSE state are set to the INIT state. The INIT state is the state that would be loaded by the XRSTOR instruction with bits 1:0 both set in the requested feature bitmask (RFBM), and both clear in XSTATE_BV the XSAVE header.
FCW	On #MF exception: set to 037EH. On all other exits: set to 037FH.
FSW	On #MF exception: set to 8081H. On all other exits: set to 0H.
MXCSR	On #XM exception: set to 1F01H. On all other exits: set to 1FB0H.
CR2	If the event that caused the AEX is a #PF, and the #PF does not directly cause a VM exit, then the low 12 bits are cleared. If the #PF leads directly to a VM exit, CR2 is not updated (usual IA behavior). Note: The low 12 bits are not cleared if a #PF is encountered during the delivery of the EEE that caused the AEX. This is because the #PF was not the EEE.
FS, GS	Restored to values as of most recent EENTER/ERESUME.

39.3.2 Synthetic State for Extended Features

When CR4.OSXSAVE = 1, extended features (those controlled by XCR0[63:2]) are set to their respective INIT states when this corresponding bit of SECS.XFRM is set. The INIT state is the state that would be loaded by the XRSTOR instruction had the instruction mask and the XSTATE_BV field of the XSAVE header each contained the value XFRM. (When the AEX occurs in 32-bit mode, those features that do not exist in 32-bit mode are unchanged.)

39.3.3 Synthetic State for MISC Features

State represented by SECS.MISCSELECT might also be overridden by synthetic state after it has been saved into the SSA. State represented by MISCSELECT[0] is not overridden but if the exiting event is a page fault then lower 12 bits of CR2 are cleared.

39.4 AEX FLOW

On Enclave Exiting Events (interrupts, exceptions, VM exits or SMIs), the processor state is securely saved inside the enclave, a synthetic state is loaded and the enclave is exited. The EEE then proceeds in the usual exit-defined fashion. The following sections describes the details of an AEX:

1. The exact processor state saved into the current SSA frame depends on whether the enclave is a 32-bit or a 64-bit enclave. In 32-bit mode (IA32_EFER.LMA = 0 || CS.L = 0), the low 32 bits of the legacy registers (EAX, EBX, ECX, EDX, ESP, EBP, ESI, EDI, EIP and EFLAGS) are stored. The upper 32 bits of the legacy registers and the 64-bit registers (R8 … R15) are not stored.

   In 64-bit mode (IA32_EFER.LMA = 1 && CS.L = 1), all 64 bits of the general processor registers (RAX, RBX, RCX, RDX, RSP, RBP, RSI, RDI, R8 … R15, RIP and RFLAGS) are stored.

   The state of those extended features specified by SECS.ATTRIBUTES.XFRM are stored into the XSAVE area of the current SSA frame. The layout of the x87 and XMM portions (the 1st 512 bytes) depends on the current values of IA32_EFER.LMA and CS.L:

   If IA32_EFER.LMA = 0 || CS.L = 0, the same format (32-bit) that XSAVE/FXSAVE uses with these values.

   If IA32_EFER.LMA = 1 && CS.L = 1, the same format (64-bit) that XSAVE/FXSAVE uses with these values when REX.W = 1.

   The cause of the AEX is saved in the EXITINFO field. See Table 37-9 for details and values of the various fields.

   The state of those miscellaneous features (see Section 37.7.2) specified by SECS.MISCSELECT are stored into the MISC area of the current SSA frame.

2. Synthetic state is created for a number of processor registers to present an opaque view of the enclave state. Table 39-1 shows the values for GPRs, x87, SSE, FS, GS, Debug and performance monitoring on AEX. The synthetic state for other extended features (those controlled by XCR0[62:2]) is set to their respective INIT states when their corresponding bit of SECS.ATTRIBUTES.XFRM is set. The INIT state is that state as defined by the behavior of the XRSTOR instruction when HEADER.XSTATE_BV[n] is 0. Synthetic state of those miscellaneous features specified by SECS.MISCSELECT depends on the miscellaneous feature. There is no synthetic state required for the miscellaneous state controlled by SECS.MISCSELECT[0].

3. Any code and data breakpoints that were suppressed at the time of enclave entry are unsuppressed when exiting the enclave.

4. RFLAGS.TF is set to the value that it had at the time of the most recent enclave entry (except for the situation that the entry was opt-in for debug; see Section 42.2). In the SSA, RFLAGS.TF is set to 0.

5. RFLAGS.RF is set to 0 in the synthetic state. In the SSA, the value saved is the same as what would have been saved on stack in the non-SGX case (architectural value of RF). Thus, AEXs due to interrupts, traps, and code breakpoints save RF unmodified into SSA, while AEXs due to other faults save RF as 1 in the SSA.

   If the event causing AEX happened on intermediate iteration of a REP-prefixed instruction, then RF=1 is saved on SSA, irrespective of its priority.

6. Any performance monitoring activity (including PEBS) or profiling activity (LBR, Tracing using Intel PT) on the exiting thread that was suppressed due to the enclave entry on that thread is unsuppressed. Any counting that had been demoted from AnyThread counting to MyThread counting (on one logical processor) is promoted back to AnyThread counting.

39.4.1 AEX Operational Detail

Temp Variables in AEX Operational Flow

Name	Type	Size (bits)	Description
TMP_RIP	Effective Address	32/64	Address of instruction at which to resume execution on ERESUME.
TMP_MODE64	binary	1	((IA32_EFER.LMA = 1) && (CS.L = 1)).
TMP_BRANCH_RECORD	LBR Record	2x64	From/To address to be pushed onto LBR stack.

The pseudo code in this section describes the internal operations that are executed when an AEX occurs in enclave mode. These operations occur just before the normal interrupt or exception processing occurs.

```
(* Save RIP for later use *)
TMP_RIP = Linear Address of Resume RIP
(* Is the processor in 64-bit mode? *)
TMP_MODE64 ← ((IA32_EFER.LMA = 1) && (CS.L = 1));

(* Save all registers, When saving EFLAGS, the TF bit is set to 0 and
   the RF bit is set to what would have been saved on stack in the non-SGX case *)

 IF (TMP_MODE64 = 0)
   THEN
       Save EAX, EBX, ECX, EDX, ESP, EBP, ESI, EDI, EFLAGS, EIP into the current SSA frame using
CR_GPR_PA; (* see Table 40-4 for list of CREGs used to describe internal operation within Intel SGX *)
       SSA.RFLAGS.TF ← 0;
   ELSE    (* TMP_MODE64 = 1 *)
       Save RAX, RBX, RCX, RDX, RSP, RBP, RSI, RDI, R8-R15, RFLAGS, RIP into the current SSA frame using
CR_GPR_PA;
       SSA.RFLAGS.TF ← 0;
FI;
Save FS and GS BASE into SSA using CR_GPR_PA;

(* store XSAVE state into the current SSA frame's XSAVE area using the physical addresses
   that were determined and cached at enclave entry time with CR_XSAVE_PAGE_i. *)
For each XSAVE state i defined by (SECS.ATTRIBUTES.XFRM[i] = 1, destination address cached in
CR_XSAVE_PAGE_i)
  SSA.XSAVE.i ← XSAVE_STATE_i;

(* Clear bytes 8 to 23 of XSAVE_HEADER, i.e. the next 16 bytes after XHEADER_BV *)

CR_XSAVE_PAGE_0.XHEADER_BV[191:64] ← 0;

(* Clear bits in XHEADER_BV[63:0] that are not enabled in ATTRIBUTES.XFRM *)

CR_XSAVE_PAGE_0.XHEADER_BV[63:0] ←
   CR_XSAVE_PAGE_0.XHEADER_BV[63:0] & SECS(CR_ACTIVE_SECS).ATTRIBUTES.XFRM;
   Apply synthetic state to GPRs, RFLAGS, extended features, etc.

(* Restore the RSP and RBP from the current SSA frame's GPR area using the physical address
   that was determined and cached at enclave entry time with CR_GPR_PA. *)
RSP ← CR_GPR_PA.URSP;
RBP ← CR_GPR_PA.URBP;
```

```
(* Restore the FS and GS *)
FS.selector ← CR_SAVE_FS.selector;
FS.base ← CR_SAVE_FS.base;
FS.limit ← CR_SAVE_FS.limit;
FS.access_rights ← CR_SAVE_FS.access_rights;
GS.selector ← CR_SAVE_GS.selector;
GS.base ← CR_SAVE_GS.base;
GS.limit ← CR_SAVE_GS.limit;
GS.access_rights ← CR_SAVE_GS.access_rights;

(* Examine exception code and update enclave internal states*)
exception_code ← Exception or interrupt vector;

(* Indicate the exit reason in SSA *)
IF (exception_code = (#DE OR #DB OR #BP OR #BR OR #UD OR #MF OR #AC OR #XM ))
   THEN
       CR_GPR_PA.EXITINFO.VECTOR ← exception_code;
       IF (exception code = #BP)
           THEN CR_GPR_PA.EXITINFO.EXIT_TYPE ← 6;
           ELSE CR_GPR_PA.EXITINFO.EXIT_TYPE ← 3;
       FI;
       CR_GPR_PA.EXITINFO.VALID ← 1;
   ELSE IF (exception_code is #PF or #GP )
       THEN
       (* Check SECS.MISCSELECT using CR_ACTIVE_SECS *)
       IF (SECS.MISCSELECT[0] is set)
           THEN
           CR_GPR_PA.EXITINFO.VECTOR ← exception_code;
           CR_GPR_PA.EXITINFO.EXIT_TYPE ← 3;
           IF (exception_code is #PF)
               THEN
                   SSA.MISC.EXINFO. MADDR ← CR2;
                   SSA.MISC.EXINFO.ERRCD ← PFEC;
                   SSA.MISC.EXINFO.RESERVED ← 0;
           ELSE
               SSA.MISC.EXINFO. MADDR ← 0;
               SSA.MISC.EXINFO.ERRCD ← GPEC;
               SSA.MISC.EXINFO.RESERVED ← 0;
           FI;
           CR_GPR_PA.EXITINFO.VALID ← 1;
       FI;
   ELSE
       CR_GPR_PA.EXITINFO.VECTOR ← 0;
       CR_GPR_PA.EXITINFO.EXIT_TYPE ← 0
       CR_GPR_PA.REASON.VALID ← 0;
FI;

(* Execution will resume at the AEP *)
RIP ← CR_TCS_PA.AEP;

(* Set EAX to the ERESUME leaf index *)
EAX ← 3;
```

```
(* Put the TCS LA into RBX for later use by ERESUME *)
RBX ← CR_TCS_LA;

(* Put the AEP into RCX for later use by ERESUME *)
RCX ← CR_TCS_PA.AEP;

(* Increment the SSA frame # *)
CR_TCS_PA.CSSA ← CR_TCS_PA.CSSA + 1;

(* Restore XCR0 if needed *)
IF (CR4.OSXSAVE = 1)
   THEN XCR0 ← CR_SAVE_XCR0; FI;

Un-suppress all code breakpoints that are outside ELRANGE

(* Update the thread context to show not in enclave mode *)
CR_ENCLAVE_MODE ← 0;

(* Assure consistent translations. *)
Flush linear context including TLBs and paging-structure caches

IF (CR_DBGOPTIN = 0)
   THEN
      Un-suppress all breakpoints that overlap ELRANGE
      (* Clear suppressed breakpoint matches *)
      Restore suppressed breakpoint matches
      (* Restore TF *)
      RFLAGS.TF ← CR_SAVE_TF;
      Un-suppress monitor trap flag;
      Un-suppress branch recording facilities;
      Un-suppress all suppressed performance monitoring activity;
      Promote any sibling-thread counters that were demoted from AnyThread to MyThread during enclave
entry back to AnyThread;
FI;

IF the "monitor trap flag" VM-execution control is 1
   THEN Pend MTF VM Exit at the end of exit; FI;

(* Clear low 12 bits of CR2 on #PF *)
IF (Exception code is #PF)
   THEN CR2 ← CR2 & ~0xFFF; FI;

(* end_of_flow *)
(* Execution continues with normal event processing. *)
```

25.Updates to Chapter 40, Volume 3D

Change bars show changes to Chapter 40 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3D:* System Programming Guide, Part 4.

Change to this chapter: Minor typo corrections to introduction paragraph, Section 40.1 “Intel® SGX Instruction Syntax and Operation”, Section 40.1.3 “Information and Error Codes”, Section 40.1.4 “Internal CREGs”, Section 40.2 “Intel® SGX Instruction Reference”, instructions: ENCLU, EAUG, ECREATE, EDBGRD, EDBGWR, EEXTEND, EMODPR, ETRACK, EGETKEY. Correction to pseudocode in EBLOCK, ECREATE, EDBGRD, EDBGWR, EEXTEND, EINIT, ELDB/ELDU, EMODT, EREMOVE, ETRACK, EWB, EACCEPT, EACCEPTCOPY, EENTER, EGETKEY, EMODPE, EREPORT, ERESUME.

CHAPTER 40
SGX INSTRUCTION REFERENCES

This chapter describes the supervisor and user level instructions provided by Intel® Software Guard Extensions (Intel® SGX). In general, various functionality is encoded as leaf functions within the ENCLS (supervisor) and ENCLU (user) instruction mnemonics. Different leaf functions are encoded by specifying an input value in the EAX register of the respective instruction mnemonic.

40.1 INTEL® SGX INSTRUCTION SYNTAX AND OPERATION

ENCLS and ENCLU instruction mnemonics for all leaf functions are covered in this section.

For all instructions, the value of CS.D is ignored; addresses and operands are 64 bits in 64-bit mode and are otherwise 32 bits. Aside from EAX specifying the leaf number as input, each instruction leaf may require all or some subset of the RBX/RCX/RDX as input parameters. Some leaf functions may return data or status information in one or more of the general purpose registers.

40.1.1 ENCLS Register Usage Summary

Table 40-1 summarizes the implicit register usage of supervisor mode enclave instructions.

Table 40-1. Register Usage of Privileged Enclave Instruction Leaf Functions

Instr. Leaf	EAX	RBX	RCX	RDX
ECREATE	00H (In)	PAGEINFO (In, EA)	EPCPAGE (In, EA)	
EADD	01H (In)	PAGEINFO (In, EA)	EPCPAGE (In, EA)	
EINIT	02H (In)	SIGSTRUCT (In, EA)	SECS (In, EA)	EINITTOKEN (In, EA)
EREMOVE	03H (In)		EPCPAGE (In, EA)	
EDBGRD	04H (In)	Result Data (Out)	EPCPAGE (In, EA)	
EDBGWR	05H (In)	Source Data (In)	EPCPAGE (In, EA)	
EEXTEND	06H (In)	SECS (In, EA)	EPCPAGE (In, EA)	
ELDB	07H (In)	PAGEINFO (In, EA)	EPCPAGE (In, EA)	VERSION (In, EA)
ELDU	08H (In)	PAGEINFO (In, EA)	EPCPAGE (In, EA)	VERSION (In, EA)
EBLOCK	09H (In)		EPCPAGE (In, EA)	
EPA	0AH (In)	PT_VA (In)	EPCPAGE (In, EA)	
EWB	0BH (In)	PAGEINFO (In, EA)	EPCPAGE (In, EA)	VERSION (In, EA)
ETRACK	0CH (In)		EPCPAGE (In, EA)	
EAUG	0DH (In)	PAGEINFO (In, EA)	EPCPAGE (In, EA)	
EMODPR	0EH (In)	SECINFO (In, EA)	EPCPAGE (In, EA)	
EMODT	0FH (In)	SECINFO (In, EA)	EPCPAGE (In, EA)	
EA: Effective Address				

40.1.2 ENCLU Register Usage Summary

Table 40-2 Summarized the implicit register usage of user mode enclave instructions.

Table 40-2. Register Usage of Unprivileged Enclave Instruction Leaf Functions

Instr. Leaf	EAX	RBX	RCX	RDX
EREPORT	00H (In)	TARGETINFO (In, EA)	REPORTDATA (In, EA)	OUTPUTDATA (In, EA)
EGETKEY	01H (In)	KEYREQUEST (In, EA)	KEY (In, EA)	
EENTER	02H (In)	TCS (In, EA)	AEP (In, EA)	
	RBX.CSSA (Out)		Return (Out, EA)	
ERESUME	03H (In)	TCS (In, EA)	AEP (In, EA)	
EEXIT	04H (In)	Target (In, EA)	Current AEP (Out)	
EACCEPT	05H (In)	SECINFO (In, EA)	EPCPAGE (In, EA)	
EMODPE	06H (In)	SECINFO (In, EA)	EPCPAGE (In, EA)	
EACCEPTCOPY	07H (In)	SECINFO (In, EA)	EPCPAGE (In, EA)	EPCPAGE (In, EA)
EA: Effective Address				

40.1.3 Information and Error Codes

Information and error codes are reported by various instruction leaf functions to show an abnormal termination of the instruction or provide information which may be useful to the developer. Table 40-3 shows the various codes and the instruction which generated the code. Details of the meaning of the code is provided in the individual instruction.

Table 40-3. Error or Information Codes for Intel® SGX Instructions

Name	Value	Returned By
No Error	0	
SGX_INVALID_SIG_STRUCT	1	EINIT
SGX_INVALID_ATTRIBUTE	2	EINIT, EGETKEY
SGX_BLSTATE	3	EBLOCK
SGX_INVALID_MEASUREMENT	4	EINIT
SGX_NOTBLOCKABLE	5	EBLOCK
SGX_PG_INVLD	6	EBLOCK
SGX_LOCKFAIL	7	EBLOCK, EMODPR, EMODT
SGX_INVALID_SIGNATURE	8	EINIT
SGX_MAC_COMPARE_FAIL	9	ELDB, ELDU
SGX_PAGE_NOT_BLOCKED	10	EWB
SGX_NOT_TRACKED	11	EWB, EACCEPT
SGX_VA_SLOT_OCCUPIED	12	EWB
SGX_CHILD_PRESENT	13	EWB, EREMOVE
SGX_ENCLAVE_ACT	14	EREMOVE
SGX_ENTRYEPOCH_LOCKED	15	EBLOCK
SGX_INVALID_EINITTOKEN	16	EINIT
SGX_PREV_TRK_INCMPL	17	ETRACK
SGX_PG_IS_SECS	18	EBLOCK
SGX_PAGE_ATTRIBUTES_MISMATCH	19	EACCEPT, EACCEPTCOPY
SGX_PAGE_NOT_MODIFIABLE	20	EMODPR, EMODT
SGX_PAGE_NOT_DEBUGGABLE	21	EDBGRD, EDBGWR

Table 40-3. Error or Information Codes for Intel® SGX Instructions

Name	Value	Returned By
SGX_INVALID_CPUSVN	32	EINIT, EGETKEY
SGX_INVALID_ISVSVN	64	EGETKEY
SGX_UNMASKED_EVENT	128	EINIT
SGX_INVALID_KEYNAME	256	EGETKEY

40.1.4 Internal CREGs

The CREGs as shown in Table 5-4 are hardware specific registers used in this document to indicate values kept by the processor. These values are used while executing in enclave mode or while executing an Intel SGX instruction. These registers are not software visible and are implementation specific. The values in Table 40-4 appear at various places in the pseudo-code of this document. They are used to enhance understanding of the operations.

Table 40-4. List of Internal CREG

Name	Size (Bits)	Scope
CR_ENCLAVE_MODE	1	LP
CR_DBGOPTIN	1	LP
CR_TCS_LA	64	LP
CR_TCS_PA	64	LP
CR_ACTIVE_SECS	64	LP
CR_ELRANGE	128	LP
CR_SAVE_TF	1	LP
CR_SAVE_FS	64	LP
CR_GPR_PA	64	LP
CR_XSAVE_PAGE_n	64	LP
CR_SAVE_DR7	64	LP
CR_SAVE_PERF_GLOBAL_CTRL	64	LP
CR_SAVE_DEBUGCTL	64	LP
CR_SAVE_PEBS_ENABLE	64	LP
CR_CPUSVN	128	PACKAGE
CR_SGXOWNEREPOCH	128	PACKAGE
CR_SAVE_XCR0	64	LP
CR_SGX_ATTRIBUTES_MASK	128	LP
CR_PAGING_VERSION	64	PACKAGE
CR_VERSION_THRESHOLD	64	PACKAGE
CR_NEXT_EID	64	PACKAGE
CR_BASE_PK	128	PACKAGE
CR_SEAL_FUSES	128	PACKAGE

40.1.5 Concurrent Operation Restrictions

To protect the integrity of Intel SGX data structures, under certain conditions, Intel SGX disallows certain leaf functions from operating concurrently. Listed below are some examples of concurrency that are not allowed.

- For example, Intel SGX disallows the following leafs to concurrently operate on the same EPC page.

  - ECREATE, EADD, and EREMOVE are not allowed to operate on the same EPC page concurrently with themselves.
  - EADD, EEXTEND, and EINIT leafs are not allowed to operate on the same SECS concurrently.
- Intel SGX disallows the EREMOVE leaf from removing pages from an enclave that is in use.
- Intel SGX disallows entry (EENTER and ERESUME) to an enclave while a page from that enclave is being removed.

When disallowed operation is detected, a leaf function causes an exception. To prevent such exceptions, software must serialize leaf functions or prevent these leaf functions from accessing the same resource.

40.1.5.1 Concurrency Tables of Intel® SGX Instructions

Concurrent restriction of an individual leaf function (ENCLS or ENCLU) with another Intel SGX instruction leaf functions is listed under the **Concurrency Restriction** paragraph of the respective reference pages of the leaf function.

Each cell in the table for a given Intel SGX Instruction leaf details the concurrency restriction when that instruction references the same EPC page (as an explicit or an implicit parameter) as referenced by a concurrent instruction leaf executed on another logical processor. The concurrency behavior of the instruction leaf if focus shown in a given row is denoted by the following:

- 'N': The instructions listed in a given row heading may not execute concurrently with the instruction leaf shown in the respective column. Software should serialize them. For example, multiple ETRACK operations on the same enclave are not allowed to execute concurrently on the same SECS page.
- 'Y': The instruction leaf listed in a given row may execute concurrently with the instruction leaf shown in the respective column. For instance, multiple ELDB/ELDUs are allowed to execute concurrently as long as the selected EPC page is not the same page.
- 'C': The instruction leaf listed in a given row heading may return an error code when executed concurrently with the instruction leaf shown in the respective column.
- 'U': These two instruction leaves may complete, but the occurrence these two simultaneous flows are considered a user program error for which the processor does not enforce any restriction.
- A grey cell indicates the concurrency behavior of the instruction in focus (in the row header) may be different than that of the concurrent instruction (in the column header). The concurrent instruction's behavior is detailed in its respective concurrency table. For example, EBLOCK's SECS parameter is implicit, thus it is always shown as 'Y' in the table. However a concurrent instruction may return an error code when accessing the same page.

For instance, multiple ELDB/ELDUs are allowed to execute as long as the selected EPC page is not the same page. Multiple ETRACK operations are not allowed to execute concurrently.

40.2 INTEL® SGX INSTRUCTION REFERENCE

ENCLS—Execute an Enclave System Function of Specified Leaf Number

Opcode/ Instruction	Op/En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F 01 CF ENCLS	NP	V/V	SGX1	This instruction is used to execute privileged Intel SGX leaf functions that are used for managing and debugging the enclaves.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Implicit Register Operands
NP	NA	NA	NA	See Section 40.3

Description

The ENCLS instruction invokes the specified privileged Intel SGX leaf function for managing and debugging enclaves. Software specifies the leaf function by setting the appropriate value in the register EAX as input. The registers RBX, RCX, and RDX have leaf-specific purpose, and may act as input, as output, or may be unused. In 64-bit mode, the instruction ignores upper 32 bits of the RAX register.

The ENCLS instruction produces an invalid-opcode exception (#UD) if CR0.PE = 0 or RFLAGS.VM = 1, or if it is executed in system-management mode (SMM). Additionally, any attempt to execute the instruction when CPL > 0 results in #UD. The instruction produces a general-protection exception (#GP) if CR0.PG = 0 or if an attempt is made to invoke an undefined leaf function.

In VMX non-root operation, execution of ENCLS may cause a VM exit if the "enable ENCLS exiting" VM-execution control is 1. In this case, execution of individual leaf functions of ENCLS is governed by the ENCLS-exiting bitmap field in the VMCS. Each bit in that field corresponds to the index of an ENCLS leaf function (as provided in EAX).

Software in VMX root operation can thus intercept the invocation of various ENCLS leaf functions in VMX non-root operation by setting the "enable ENCLS exiting" VM-execution control and setting the corresponding bits in the ENCLS-exiting bitmap.

Addresses and operands are 32 bits outside 64-bit mode (IA32_EFER.LMA = 0 || CS.L = 0) and are 64 bits in 64-bit mode (IA32_EFER.LMA = 1 || CS.L = 1). CS.D value has no impact on address calculation. The DS segment is used to create linear addresses.

Segment override prefixes and address-size override prefixes are ignored, and is the REX prefix in 64-bit mode.

Operation

```
IF TSX_ACTIVE
    THEN GOTO TSX_ABORT_PROCESSING; FI;

IF CR0.PE = 0 or RFLAGS.VM = 1 or in SMM or CPUID.SGX_LEAF.0:EAX.SE1 = 0
    THEN #UD; FI;

IF (CPL > 0)
    THEN #UD; FI;

IF in VMX non-root operation and the "enable ENCLS exiting" VM-execution control is 1
    THEN
        IF EAX < 63 and ENCLS_exiting_bitmap[EAX] = 1 or EAX> 62 and ENCLS_exiting_bitmap[63] = 1
            THEN VM exit;
        FI;
FI;
IF IA32_FEATURE_CONTROL.LOCK = 0 or IA32_FEATURE_CONTROL.SGX_ENABLE = 0
    THEN #GP(0); FI;

IF EAX is invalid leaf number)
```

```
    THEN #GP(0); FI;

IF CR0.PG = 0
    THEN #GP(0); FI;

(* DS must not be an expanded down segment *)
IF not in 64-bit mode and DS.Type is expand-down data
    THEN #GP(0); FI;

Jump to leaf specific flow
```

Flags Affected

See individual leaf functions

Protected Mode Exceptions

#UD	If any of the LOCK/OSIZE/REP/VEX prefix is used.
	If current privilege level is not 0.
	If CPUID.(EAX=12H,ECX=0):EAX.SGX1 [bit 0] = 0.
	If logical processor is in SMM.
#GP(0)	If IA32_FEATURE_CONTROL.LOCK = 0.
	If IA32_FEATURE_CONTROL.SGX_ENABLE = 0.
	If input value in EAX encodes an unsupported leaf.
	If data segment expand down.
	If CR0.PG=0.

Real-Address Mode Exceptions

#UD	ENCLS is not recognized in real mode.

Virtual-8086 Mode Exceptions

#UD	ENCLS is not recognized in virtual-8086 mode.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#UD	If any of the LOCK/OSIZE/REP/VEX prefix is used.
	If current privilege level is not 0.
	If CPUID.(EAX=12H,ECX=0):EAX.SGX1 [bit 0] = 0.
	If logical processor is in SMM.
#GP(0)	If IA32_FEATURE_CONTROL.LOCK = 0.
	If IA32_FEATURE_CONTROL.SGX_ENABLE = 0.
	If input value in EAX encodes an unsupported leaf.

ENCLU—Execute an Enclave User Function of Specified Leaf Number

Opcode/ Instruction	Op/En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F 01 D7 ENCLU	NP	V/V	SGX1	This instruction is used to execute non-privileged Intel SGX leaf functions.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Implicit Register Operands
NP	NA	NA	NA	See Section 40.4

Description

The ENCLU instruction invokes the specified non-privileged Intel SGX leaf functions. Software specifies the leaf function by setting the appropriate value in the register EAX as input. The registers RBX, RCX, and RDX have leaf-specific purpose, and may act as input, as output, or may be unused. In 64-bit mode, the instruction ignores upper 32 bits of the RAX register.

The ENCLU instruction produces an invalid-opcode exception (#UD) if CR0.PE = 0 or RFLAGS.VM = 1, or if it is executed in system-management mode (SMM). Additionally, any attempt to execute this instruction when CPL < 3 results in #UD. The instruction produces a general-protection exception (#GP) if either CR0.PG or CR0.NE is 0, or if an attempt is made to invoke an undefined leaf function. The ENCLU instruction produces a device not available exception (#NM) if CR0.TS = 1.

Addresses and operands are 32 bits outside 64-bit mode (IA32_EFER.LMA = 0 or CS.L = 0) and are 64 bits in 64-bit mode (IA32_EFER.LMA = 1 and CS.L = 1). CS.D value has no impact on address calculation. The DS segment is used to create linear addresses.

Segment override prefixes and address-size override prefixes are ignored, as is the REX prefix in 64-bit mode.

Operation

```
IN_64BIT_MODE← 0;
IF TSX_ACTIVE
    THEN GOTO TSX_ABORT_PROCESSING; FI;

IF CR0.PE= 0 or RFLAGS.VM = 1 or in SMM or CPUID.SGX_LEAF.0:EAX.SE1 = 0
    THEN #UD; FI;

IF CR0.TS = 1
    THEN #NM; FI;

IF CPL < 3
    THEN #UD; FI;

IF IA32_FEATURE_CONTROL.LOCK = 0 or IA32_FEATURE_CONTROL.SGX_ENABLE = 0
    THEN #GP(0); FI;

IF EAX is invalid leaf number
    THEN #GP(0); FI;

IF CR0.PG = 0 or CR0.NE = 0
    THEN #GP(0); FI;

IN_64BIT_MODE ← IA32_EFER.LMA AND CS.L ? 1 : 0;
(* Check not in 16-bit mode and DS is not a 16-bit segment *)
IF not in 64-bit mode and (CS.D = 0 or DS.B = 0)
```

```
    THEN #GP(0); FI;

IF CR_ENCLAVE_MODE = 1 and (EAX = 2 or EAX = 3) (* EENTER or ERESUME *)
    THEN #GP(0); FI;

IF CR_ENCLAVE_MODE = 0 and (EAX = 0 or EAX = 1 or EAX = 4 or EAX = 5 or EAX = 6 or EAX = 7)
(* EREPORT, EGETKEY, EEXIT, EACCEPT, EMODPE, or EACCEPTCOPY *)
    THEN #GP(0); FI;

Jump to leaf specific flow
```

Flags Affected

See individual leaf functions

Protected Mode Exceptions

#UD	If any of the LOCK/OSIZE/REP/VEX prefix is used.
	If current privilege level is not 3.
	If CPUID.(EAX=12H,ECX=0):EAX.SGX1 [bit 0] = 0.
	If logical processor is in SMM.
#GP(0)	If IA32_FEATURE_CONTROL.LOCK = 0.
	If IA32_FEATURE_CONTROL.SGX_ENABLE = 0.
	If input value in EAX encodes an unsupported leaf.
	If input value in EAX encodes EENTER/ERESUME and ENCLAVE_MODE = 1.
	If input value in EAX encodes EGETKEY/EREPORT/EEXIT/EACCEPT/EACCEPTCOPY/EMODPE and ENCLAVE_MODE = 0.
	If operating in 16-bit mode.
	If data segment is in 16-bit mode.
	If CR0.PG = 0 or CR0.NE= 0.
#NM	If CR0.TS = 1.

Real-Address Mode Exceptions

#UD	ENCLS is not recognized in real mode.

Virtual-8086 Mode Exceptions

#UD	ENCLS is not recognized in virtual-8086 mode.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#UD	If any of the LOCK/OSIZE/REP/VEX prefix is used.
	If current privilege level is not 3.
	If CPUID.(EAX=12H,ECX=0):EAX.SGX1 [bit 0] = 0.
	If logical processor is in SMM.
#GP(0)	If IA32_FEATURE_CONTROL.LOCK = 0.
	If IA32_FEATURE_CONTROL.SGX_ENABLE = 0.
	If input value in EAX encodes an unsupported leaf.
	If input value in EAX encodes EENTER/ERESUME and ENCLAVE_MODE = 1.
	If input value in EAX encodes EGETKEY/EREPORT/EEXIT/EACCEPT/EACCEPTCOPY/EMODPE and ENCLAVE_MODE = 0.

	If CR0.NE= 0.
#NM	If CR0.TS = 1.

40.3 INTEL® SGX SYSTEM LEAF FUNCTION REFERENCE

Leaf functions available with the ENCLS instruction mnemonic are covered in this section. In general, each instruction leaf requires EAX to specify the leaf function index and/or additional implicit registers specifying leaf-specific input parameters. An instruction operand encoding table provides details of each implicit register usage and associated input/output semantics.

In many cases, an input parameter specifies an effective address associated with a memory object inside or outside the EPC, the memory addressing semantics of these memory objects are also summarized in a separate table.

EADD—Add a Page to an Uninitialized Enclave

Opcode/ Instruction	Op/En	64/32 bit Mode Support	CPUID Feature Flag	Description
EAX = 01H ENCLS[EADD]	IR	V/V	SGX1	This leaf function adds a page to an uninitialized enclave.

Instruction Operand Encoding

Op/En	EAX	RBX	RCX
IR	EADD (In)	Address of a PAGEINFO (In)	Address of the destination EPC page (In)

Description

This leaf function copies a source page from non-enclave memory into the EPC, associates the EPC page with an SECS page residing in the EPC, and stores the linear address and security attributes in EPCM. As part of the association, the enclave offset and the security attributes are measured and extended into the SECS.MRENCLAVE. This instruction can only be executed when current privilege level is 0.

RBX contains the effective address of a PAGEINFO structure while RCX contains the effective address of an EPC page. The table below provides additional information on the memory parameter of EADD leaf function.

EADD Memory Parameter Semantics

PAGEINFO	PAGEINFO.SECS	PAGEINFO.SRCPGE	PAGEINFO.SECINFO	EPCPAGE
Read access permitted by Non Enclave	Read/Write access permitted by Enclave	Read access permitted by Non Enclave	Read access permitted by Non Enclave	Write access permitted by Enclave

The instruction faults if any of the following:

EADD Faulting Conditions

The operands are not properly aligned.	Unsupported security attributes are set.
Refers to an invalid SECS.	Reference is made to an SECS that is locked by another thread.
The EPC page is locked by another thread.	RCX does not contain an effective address of an EPC page.
The EPC page is already valid.	If security attributes specifies a TCS and the source page specifies unsupported TCS values or fields.
The SECS has been initialized.	The specified enclave offset is outside of the enclave address space.

Concurrency Restrictions

Table 40-5. Concurrency Restrictions of EADD with Other Intel® SGX Operations 1 of 2

Operation		EEXIT			EADD		EBLOCK		ECREATE	EDBGRD/WR		EENTER/ERESUME			EEXTEND		EGETKEY		EINIT	ELDB/ELDU			EPA
	Param	TCS	SSA	SECS	Targ	SECS	Targ	SECS	SECS	Targ	SECS	TCS	SSA	SECS	Targ	SECS	Param	SECS	SECS	Targ	VA	SECS	VA
EADD	Targ				N		N		N	N		N			N				N	N	N	N	N
	SECS			N		N	Y	Y	N		Y			N		N		N	N			Y	N

Table 40-6. Concurrency Restrictions of EADD with Other Intel® SGX Operations 2 of 2

Operation		EREMOVE		EREPORT		ETRACK	EWB			EAUG		EMODPE		EMODPR		EMODT		EACCEPT			EACCEPTCOPY		
	Param	Targ	SECS	Param	SECS	SECS	SRC	VA	SECS	Targ	SECS	Targ	SECINFO	Targ	SECS	Targ	SECS	Targ	SECINFO	SECS	Targ	SRC	SECINFO
EADD	Targ	N				N	N	N		N	N			N		N							
	SECS	N	Y		N	Y	N		Y	N	N				N	N	N			N			

Operation

Temp Variables in EADD Operational Flow

Name	Type	Size (bits)	Description
TMP_SRCPGE	Effective Address	32/64	Effective address of the source page.
TMP_SECS	Effective Address	32/64	Effective address of the SECS destination page.
TMP_SECINFO	Effective Address	32/64	Effective address of an SECINFO structure which contains security attributes of the page to be added.
SCRATCH_SECINFO	SECINFO	512	Scratch storage for holding the contents of DS:TMP_SECINFO.
TMP_LINADDR	Unsigned Integer	64	Holds the linear address to be stored in the EPCM and used to calculate TMP_ENCLAVEOFFSET.
TMP_ENCLAVEOFFSET	Enclave Offset	64	The page displacement from the enclave base address.
TMPUPDATEFIELD	SHA256 Buffer	512	Buffer used to hold data being added to TMP_SECS.MRENCLAVE.

```
IF (DS:RBX is not 32Byte Aligned)
    THEN #GP(0); FI;

IF (DS:RCX is not 4KByte Aligned)
    THEN #GP(0); FI;

IF (DS:RCX does not resolve within an EPC)
    THEN #PF(DS:RCX); FI;

TMP_SRCPGE ← DS:RBX.SRCPGE;
TMP_SECS ← DS:RBX.SECS;
TMP_SECINFO ← DS:RBX.SECINFO;
TMP_LINADDR ← DS:RBX.LINADDR;

IF (DS:TMP_SRCPGE is not 4KByte aligned or DS:TMP_SECS is not 4KByte aligned or
    DS:TMP_SECINFO is not 64Byte aligned or TMP_LINADDR is not 4KByte aligned)
    THEN #GP(0); FI;

IF (DS:TMP_SECS does not resolve within an EPC)
    THEN #PF(DS:TMP_SECS); FI;

SCRATCH_SECINFO ← DS:TMP_SECINFO;

(* Check for mis-configured SECINFO flags*)
IF (SCRATCH_SECINFO reserved fields are not zero or
    ! (SCRATCH_SECINFO.FLAGS.PT is PT_REG or SCRATCH_SECINFO.FLAGS.PT is PT_TCS) )
    THEN #GP(0); FI;
```

```
(* Check the EPC page for concurrency *)
IF (EPC page in use)
    THEN #GP(0); FI;

IF (EPCM(DS:RCX).VALID ≠ 0)
    THEN #PF(DS:RCX); FI;

(* Check the SECS for concurrency *)
IF (SECS is not available for EADD)
    THEN #GP(0); FI;

IF (EPCM(DS:TMP_SECS).VALID = 0 or EPCM(DS:TMP_SECS).PT ≠ PT_SECS)
    THEN #PF(DS:TMP_SECS); FI;

(* Copy 4KBytes from source page to EPC page*)
DS:RCX[32767:0] ← DS:TMP_SRCPGE[32767:0];

CASE (SCRATCH_SECINFO.FLAGS.PT)
{
    PT_TCS:
        IF (DS:RCX.RESERVED ≠ 0) #GP(0); FI;
        IF ( (DS:TMP_SECS.ATTIBUTES.MODE64BIT = 0) and
            ((DS:TCS.FSLIMIT & 0FFFH ≠ 0FFFH) or (DS:TCS.GSLIMIT & 0FFFH ≠ 0FFFH) )) #GP(0); FI;
        BREAK;
    PT_REG:
        IF (SCRATCH_SECINFO.FLAGS.W = 1 and SCRATCH_SECINFO.FLAGS.R = 0) #GP(0); FI;
        BREAK;
ESAC;

(* Check the enclave offset is within the enclave linear address space *)
IF (TMP_LINADDR < DS:TMP_SECS.BASEADDR or TMP_LINADDR ≥ DS:TMP_SECS.BASEADDR + DS:TMP_SECS.SIZE)
    THEN #GP(0); FI;

(* Check concurrency of measurement resource*)
IF (Measurement being updated)
    THEN #GP(0); FI;

(* Check if the enclave to which the page will be added is already in Initialized state *)
IF (DS:TMP_SECS already initialized)
    THEN #GP(0); FI;

(* For TCS pages, force EPCM.rwx bits to 0 and no debug access *)
IF (SCRATCH_SECINFO.FLAGS.PT = PT_TCS)
    THEN
        SCRATCH_SECINFO.FLAGS.R ← 0;
        SCRATCH_SECINFO.FLAGS.W ← 0;
        SCRATCH_SECINFO.FLAGS.X ← 0;
        (DS:RCX).FLAGS.DBGOPTIN ← 0; // force TCS.FLAGS.DBGOPTIN off
        DS:RCX.CSSA ← 0;
        DS:RCX.AEP ← 0;
        DS:RCX.STATE ← 0;
FI;

(* Add enclave offset and security attributes to MRENCLAVE *)
```

```
TMP_ENCLAVEOFFSET ← TMP_LINADDR - DS:TMP_SECS.BASEADDR;
TMPUPDATEFIELD[63:0] ← 0000000044444145H; // "EADD"
TMPUPDATEFIELD[127:64] ← TMP_ENCLAVEOFFSET;
TMPUPDATEFIELD[511:128] ← SCRATCH_SECINFO[375:0]; // 48 bytes
DS:TMP_SECS.MRENCLAVE ← SHA256UPDATE(DS:TMP_SECS.MRENCLAVE, TMPUPDATEFIELD)
INC enclave's MRENCLAVE update counter;

(* Add enclave offset and security attributes to MRENCLAVE *)
EPCM(DS:RCX).R ← SCRATCH_SECINFO.FLAGS.R;
EPCM(DS:RCX).W ← SCRATCH_SECINFO.FLAGS.W;
EPCM(DS:RCX).X ← SCRATCH_SECINFO.FLAGS.X;
EPCM(DS:RCX).PT ← SCRATCH_SECINFO.FLAGS.PT;
EPCM(DS:RCX).ENCLAVEADDRESS ← TMP_LINADDR;

(* associate the EPCPAGE with the SECS by storing the SECS identifier of DS:TMP_SECS *)
Update EPCM(DS:RCX) SECS identifier to reference DS:TMP_SECS identifier;

(* Set EPCM entry fields *)
EPCM(DS:RCX).BLOCKED ← 0;
EPCM(DS:RCX).PENDING ← 0;
EPCM(DS:RCX).MODIFIED ← 0;
EPCM(DS:RCX).VALID ← 1;
```

Flags Affected

None

Protected Mode Exceptions

#GP(0)	If a memory operand effective address is outside the DS segment limit.
	If a memory operand is not properly aligned.
	If an enclave memory operand is outside of the EPC.
	If an enclave memory operand is the wrong type.
	If a memory operand is locked.
	If the enclave is initialized.
	If the enclave's MRENCLAVE is locked.
	If the TCS page reserved bits are set.
#PF(error code)	If a page fault occurs in accessing memory operands.
	If the EPC page is valid.

64-Bit Mode Exceptions

#GP(0)	If a memory operand is non-canonical form.
	If a memory operand is not properly aligned.
	If an enclave memory operand is outside of the EPC.
	If an enclave memory operand is the wrong type.
	If a memory operand is locked.
	If the enclave is initialized.
	If the enclave's MRENCLAVE is locked.
	If the TCS page reserved bits are set.
#PF(error code)	If a page fault occurs in accessing memory operands.
	If the EPC page is valid.

EAUG—Add a Page to an Initialized Enclave

Opcode/ Instruction	Op/En	64/32 bit Mode Support	CPUID Feature Flag	Description
EAX = 0DH ENCLS[EAUG]	IR	V/V	SGX2	This leaf function adds a page to an initialized enclave.

Instruction Operand Encoding

Op/En	EAX	RBX	RCX
IR	EAUG (In)	Address of a SECINFO (In)	Address of the destination EPC page (In)

Description

This leaf function zeroes a page of EPC memory, associates the EPC page with an SECS page residing in the EPC, and stores the linear address and security attributes in the EPCM. As part of the association, the security attributes are configured to prevent access to the EPC page until a corresponding invocation of the EACCEPT leaf or EACCEPTCOPY leaf confirms the addition of the new page into the enclave. This instruction can only be executed when current privilege level is 0.

RBX contains the effective address of a PAGEINFO structure while RCX contains the effective address of an EPC page. The table below provides additional information on the memory parameter of the EAUG leaf function.

EAUG Memory Parameter Semantics

PAGEINFO	PAGEINFO.SECS	PAGEINFO.SRCPGE	PAGEINFO.SECINFO	EPCPAGE
Read access permitted by Non Enclave	Read/Write access permitted by Enclave	Must be zero	Read access permitted by Non Enclave	Write access permitted by Enclave

The instruction faults if any of the following:

EAUG Faulting Conditions

The operands are not properly aligned.	Unsupported security attributes are set.
Refers to an invalid SECS.	Reference is made to an SECS that is locked by another thread.
The EPC page is locked by another thread.	RCX does not contain an effective address of an EPC page.
The EPC page is already valid.	The specified enclave offset is outside of the enclave address space.
The SECS has been initialized.	

Concurrency Restrictions

Table 40-7. Concurrency Restrictions of EAUG with Other Intel® SGX Operations 1 of 2

Operation		EEXIT			EADD		EBLOCK		ECREATE	EDBGRD/WR		EENTER/ERESUME			EEXTEND		EGETKEY		EINIT	ELDB/ELDU			EPA
	Param	TCS	SSA	SECS	Targ	SECS	Targ	SECS	SECS	Targ	SECS	TCS	SSA	SECS	Targ	SECS	Param	SECS	SECS	Targ	VA	SECS	VA
EAUG	Targ				N	N	N		N	N		N			N				N	N	N	N	N
	SECS			Y	N	N		Y	N		Y			Y		N		Y	N	N		Y	N

Table 40-8. Concurrency Restrictions of EAUG with Other Intel® SGX Operations 2 of 2

Operation		EREMOVE		EREPORT		ETRACK	EWB			EAUG		EMODPE		EMODPR		EMODT		EACCEPT			EACCEPTCOPY		
	Param	Targ	SECS	Param	SECS	SECS	SRC	VA	SECS	Targ	SECS	Targ	SECINFO	Targ	SECS	Targ	SECS	Targ	SECINFO	SECS	Targ	SRC	SECINFO
EAUG	Targ	N				N	N	N		N	N			N		N							
	SECS	N	Y		Y	Y	N		Y	N	Y				Y	N	Y			Y			

Operation

Temp Variables in EAUG Operational Flow

Name	Type	Size (bits)	Description
TMP_SECS	Effective Address	32/64	Effective address of the SECS destination page.
TMP_SECINFO	Effective Address	32/64	Effective address of an SECINFO structure which contains security attributes of the page to be added.
SCRATCH_SECINFO	SECINFO	512	Scratch storage for holding the contents of DS:TMP_SECINFO.
TMP_LINADDR	Unsigned Integer	64	Holds the linear address to be stored in the EPCM and used to calculate TMP_ENCLAVEOFFSET.

```
IF (DS:RBX is not 32Byte Aligned)
    THEN #GP(0); FI;

IF (DS:RCX is not 4KByte Aligned)
    THEN #GP(0); FI;

IF (DS:RCX does not resolve within an EPC)
    THEN #PF(DS:RCX); FI;

TMP_SECS ← DS:RBX.SECS;
TMP_LINADDR ← DS:RBX.LINADDR;

IF ( DS:TMP_SECS is not 4KByte aligned or TMP_LINADDR is not 4KByte aligned )
    THEN #GP(0); FI;

IF ( (DS:RBX.SRCPAGE is not 0) or (DS:RBX.SECINFO is not 0) )
    THEN #GP(0); FI;

IF (DS:TMP_SECS does not resolve within an EPC)
    THEN #PF(DS:TMP_SECS); FI;

(* Check the EPC page for concurrency *)
IF (EPC page in use)
    THEN #GP(0); FI;

IF (EPCM(DS:RCX).VALID ≠ 0)
    THEN #PF(DS:RCX); FI;

(* Check the SECS for concurrency *)
IF (SECS is not available for EAUG)
    THEN #GP(0); FI;
```

```
IF (EPCM(DS:TMP_SECS).VALID = 0 or EPCM(DS:TMP_SECS).PT ≠ PT_SECS)
    THEN #PF(DS:TMP_SECS); FI;

(* Check if the enclave to which the page will be added is in the Initialized state *)
IF (DS:TMP_SECS is not initialized)
    THEN #GP(0); FI;

(* Check the enclave offset is within the enclave linear address space *)
IF ( (TMP_LINADDR < DS:TMP_SECS.BASEADDR) or (TMP_LINADDR ≥ DS:TMP_SECS.BASEADDR + DS:TMP_SECS.SIZE) )
    THEN #GP(0); FI;

(* Clear the content of EPC page*)
DS:RCX[32767:0] ← 0;

(* Set EPCM security attributes *)
EPCM(DS:RCX).R ← 1;
EPCM(DS:RCX).W ← 1;
EPCM(DS:RCX).X ← 0;
EPCM(DS:RCX).PT ← PT_REG;
EPCM(DS:RCX).ENCLAVEADDRESS ← TMP_LINADDR;
EPCM(DS:RCX).BLOCKED ← 0;
EPCM(DS:RCX).PENDING ← 1;
EPCM(DS:RCX).MODIFIED ← 0;
EPCM(DS:RCX).PR ← 0;

(* associate the EPCPAGE with the SECS by storing the SECS identifier of DS:TMP_SECS *)
Update EPCM(DS:RCX) SECS identifier to reference DS:TMP_SECS identifier;

(* Set EPCM valid fields *)
EPCM(DS:RCX).VALID ← 1;
```

Flags Affected

None

Protected Mode Exceptions

#GP(0)	If a memory operand effective address is outside the DS segment limit.
	If a memory operand is not properly aligned.
	If a memory operand is locked.
	If the enclave is not initialized.
#PF(error code)	If a page fault occurs in accessing memory operands.

64-Bit Mode Exceptions

#GP(0)	If a memory operand is non-canonical form.
	If a memory operand is not properly aligned.
	If a memory operand is locked.
	If the enclave is not initialized.
#PF(error code)	If a page fault occurs in accessing memory operands.

EBLOCK—Mark a page in EPC as Blocked

Opcode/ Instruction	Op/En	64/32 bit Mode Support	CPUID Feature Flag	Description
EAX = 09H ENCLS[EBLOCK]	IR	V/V	SGX1	This leaf function marks a page in the EPC as blocked.

Instruction Operand Encoding

Op/En	EAX		RCX
IR	EBLOCK (In)	Return error code (Out)	Effective address of the EPC page (In)

Description

This leaf function causes an EPC page to be marked as BLOCKED. This instruction can only be executed when current privilege level is 0.

The content of RCX is an effective address of an EPC page. The DS segment is used to create linear address. Segment override is not supported.

An error code is returned in RAX.

The table below provides additional information on the memory parameter of EBLOCK leaf function.

EBLOCK Memory Parameter Semantics

EPCPAGE
Read/Write access permitted by Enclave

The error codes are:

Table 40-9. EBLOCK Return Value in RAX

Error Code (see Table 40-3)	Description
No Error	EBLOCK successful
SGX_BLKSTATE	Page already blocked. This value is used to indicate to a VMM that the page was already in BLOCKED state as a result of EBLOCK and thus will need to be restored to this state when it is eventually reloaded (using ELDB).
SGX_ENTRYEPOCH_LOCKED	SECS locked for Entry Epoch update. This value indicates that an ETRACK is currently executing on the SECS. The EBLOCK should be reattempted.
SGX_NOTBLOCKABLE	Page type is not one which can be blocked
SGX_PG_INVLD	Page is not valid and cannot be blocked
SGX_LOCKFAIL	Page is being written by EADD, EAUG, ECREATE, ELDU/B, EMODT, or EWB

Concurrency Restrictions

Table 40-10. Concurrency Restrictions of EBLOCK with Other Intel® SGX Operations 1 of 2

Operation		EEXIT			EADD		EBLOCK		ECREATE	EDBGRD/WR		EENTER/ERESUME			EEXTEND		EGETKEY		EINIT	ELDB/ELDU			EPA
	Param	TCS	SSA	SECS	Targ	SECS	Targ	SECS	SECS	Targ	SECS	TCS	SSA	SECS	Targ	SECS	Param	SECS	SECS	Targ	VA	SECS	VA
EBLOCK	Targ	Y	Y	Y	N	C	C	C	N	Y	C	Y	Y	C	Y	C	Y	C	Y	N	C	C	N
	SECS			Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y

Table 40-11. Concurrency Restrictions of EBLOCK with Other Intel® SGX Operations 2 of 2

Operation		EREMOVE		EREPORT		ETRACK	EWB			EAUG		EMODPE		EMODPR		EMODT		EACCEPT			EACCEPTCOPY		
	Param	Targ	SECS	Param	SECS	SECS	SRC	VA	SECS	Targ	SECS	Targ	SECINFO	Targ	SECS	Targ	SECS	Targ	SECINFO	SECS	Targ	SRC	SECINFO
EBLOCK	Targ	N	C	Y	C	C	N	C	C	N	C	Y	Y	Y	C	N	C	Y	Y	C	Y	Y	Y
	SECS	Y	Y	Y	Y	U	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y

Operation

Temp Variables in EBLOCK Operational Flow

Name	Type	Size (Bits)	Description
TMP_BLKSTATE	Integer	64	Page is already blocked.

```
IF (DS:RCX is not 4KByte Aligned)
    THEN #GP(0); FI;

IF (DS:RCX does not resolve within an EPC)
    THEN #PF(DS:RCX); FI;

RFLAGS.ZF,CF,PF,AF,OF,SF ← 0;
RAX← 0;

(* Check the EPC page for concurrency*)
IF (EPC page in use)
    THEN
        RFLAGS.ZF ← 1;
        RAX← SGX_LOCKFAIL;
        GOTO DONE;
FI;

IF (EPCM(DS:RCX). VALID = 0)
    THEN
        RFLAGS.ZF ← 1;
        RAX← SGX_PG_INVLD;
        GOTO DONE;
FI;

IF ( (EPCM(DS:RCX).PT ≠ PT_REG) and (EPCM(DS:RCX).PT ≠ PT_TCS) and (EPCM(DS:RCX).PT ≠ PT_TRIM) )
    THEN
        RFLAGS.CF ← 1;
        IF (EPCM(DS:RCX).PT = PT_SECS)
            THEN RAX← SGX_PG_IS_SECS;
            ELSE RAX← SGX_NOTBLOCKABLE;
        FI;
        GOTO DONE;
FI;

(* Check if the page is already blocked and report blocked state *)
TMP_BLKSTATE ← EPCM(DS:RCX).BLOCKED;

(* at this point, the page must be valid and PT_TCS or PT_REG or PT_TRIM*)
```

```
IF (TMP_BLKSTATE = 1) )
    THEN
        RFLAGS.CF ← 1;
        RAX← SGX_BLKSTATE;
    ELSE
        EPCM(DS:RCX).BLOCKED ← 1
FI;
DONE:
```

Flags Affected

Sets ZF if SECS is in use or invalid, otherwise cleared. Sets CF if page is BLOCKED or not blockable, otherwise cleared. Clears PF, AF, OF, SF.

Protected Mode Exceptions

#GP(0)	If a memory operand effective address is outside the DS segment limit.
	If a memory operand is not properly aligned.
	If the specified EPC resource is in use.
#PF(error code)	If a page fault occurs in accessing memory operands.
	If a memory operand is not an EPC page.

64-Bit Mode Exceptions

#GP(0)	If a memory operand is non-canonical form.
	If a memory operand is not properly aligned.
	If the specified EPC resource is in use.
#PF(error code)	If a page fault occurs in accessing memory operands.
	If a memory operand is not an EPC page.

ECREATE—Create an SECS page in the Enclave Page Cache

Opcode/ Instruction	Op/En	64/32 bit Mode Support	CPUID Feature Flag	Description
EAX = 00H ENCLS[ECREATE]	IR	V/V	SGX1	This leaf function begins an enclave build by creating an SECS page in EPC.

Instruction Operand Encoding

Op/En	EAX	RBX	RCX
IR	ECREATE (In)	Address of a PAGEINFO (In)	Address of the destination SECS page (In)

Description

ENCLS[ECREATE] is the first instruction executed in the enclave build process. ECREATE copies an SECS structure outside the EPC into an SECS page inside the EPC. The internal structure of SECS is not accessible to software.

ECREATE will set up fields in the protected SECS and mark the page as valid inside the EPC. ECREATE initializes or checks unused fields.

Software sets the following fields in the source structure: SECS:BASEADDR, SECS:SIZE in bytes, and ATTRIBUTES. SECS:BASEADDR must be naturally aligned on an SECS.SIZE boundary. SECS.SIZE must be at least 2 pages (8192).

The source operand RBX contains an effective address of a PAGEINFO structure. PAGEINFO contains an effective address of a source SECS and an effective address of an SECINFO. The SECS field in PAGEINFO is not used.

The RCX register is the effective address of the destination SECS. It is an address of an empty slot in the EPC. The SECS structure must be page aligned. SECINFO flags must specify the page as an SECS page.

ECREATE Memory Parameter Semantics

PAGEINFO	PAGEINFO.SRCPGE	PAGEINFO.SECINFO	EPCPAGE
Read access permitted by Non Enclave	Read access permitted by Non Enclave	Read access permitted by Non Enclave	Write access permitted by Enclave

ECREATE will fault if the SECS target page is in use; already valid; outside the EPC. It will also fault if addresses are not aligned; unused PAGEINFO fields are not zero.

If the amount of space needed to store the SSA frame is greater than the amount specified in SECS.SSAFRAME-SIZE, a #GP(0) results. The amount of space needed for an SSA frame is computed based on DS:TMP_SECS.ATTRIBUTES.XFRM size. Details of computing the size can be found Section 41.7.

Concurrency Restrictions

Table 40-12. Concurrency Restrictions of ECREATE with Other Intel® SGX Operations 1 of 2

Operation		EEXIT			EADD		EBLOCK		ECREATE	EDBGRD/WR		EENTER/ERESUME			EEXTEND		EGETKEY		EINIT	ELDB/ELDU			EPA
	Param	TCS	SSA	SECS	Targ	SECS	Targ	SECS	SECS	Targ	SECS	TCS	SSA	SECS	Targ	SECS	Param	SECS	SECS	Targ	VA	SECS	VA
ECREATE	SECS				N	N	N		N	N		N			N				N	N	N	N	N

Table 40-13. Concurrency Restrictions of ECREATE with Other Intel® SGX Operations 2 of 2

Operation		EREMOVE		EREPORT		ETRACK	EWB			EAUG		EMODPE		EMODPR		EMODT		EACCEPT			EACCEPTCOPY		
	Param	Targ	SECS	Param	SECS	SECS	SRC	VA	SECS	Targ	SECS	Targ	SECINFO	Targ	SECS	Targ	SECS	Targ	SECINFO	SECS	Targ	SRC	SECINFO
ECREATE	SECS	N				N	N	N		N	N			N		N							

Operation

Temp Variables in ECREATE Operational Flow

Name	Type	Size (Bits)	Description
TMP_SRCPGE	Effective Address	32/64	Effective address of the SECS source page.
TMP_SECS	Effective Address	32/64	Effective address of the SECS destination page.
TMP_SECINFO	Effective Address	32/64	Effective address of an SECINFO structure which contains security attributes of the SECS page to be added.
TMP_XSIZE	SSA Size	64	The size calculation of SSA frame.
TMP_MISC_SIZE	MISC Field Size	64	Size of the selected MISC field components.
TMPUPDATEFIELD	SHA256 Buffer	512	Buffer used to hold data being added to TMP_SECS.MRENCLAVE.

```
IF (DS:RBX is not 32Byte Aligned)
    THEN #GP(0); FI;

IF (DS:RCX is not 4KByte Aligned)
    THEN #GP(0); FI;

IF (DS:RCX does not resolve within an EPC)
    THEN #PF(DS:RCX); FI;

TMP_SRCPGE ← DS:RBX.SRCPGE;
TMP_SECINFO ← DS:RBX.SECINFO;

IF (DS:TMP_SRCPGE is not 4KByte aligned or DS:TMP_SECINFO is not 64Byte aligned)
    THEN #GP(0); FI;

IF (DS:RBX.LINADDR ! = 0 or DS:RBX.SECS ≠ 0)
    THEN #GP(0); FI;

(* Check for misconfigured SECINFO flags*)
IF (DS:TMP_SECINFO reserved fields are not zero or DS:TMP_SECINFO.FLAGS.PT ≠ PT_SECS) )
    THEN #GP(0); FI;

TMP_SECS ← RCX;

IF (EPC entry in use)
    THEN #GP(0); FI;

IF (EPCM(DS:RCX).VALID = 1)
    THEN #PF(DS:RCX); FI;

(* Copy 4KBytes from source page to EPC page*)
DS:RCX[32767:0] ← DS:TMP_SRCPGE[32767:0];

(* Check lower 2 bits of XFRM are set *)
IF ( ( DS:TMP_SECS.ATTRIBUTES.XFRM BitwiseAND 03H) ≠ 03H)
    THEN #GP(0); FI;

IF (XFRM is illegal)
```

```
    THEN #GP(0); FI;

(* Make sure that the SECS does not have any unsupported MISCSELECT options*)
IF ( !(CPUID.(EAX=12H, ECX=0):EBX[31:0] & DS:TMP_SECS.MISCSELECT[31:0]) )
    THEN
        EPCM(DS:TMP_SECS).EntryLock.Release();
        #GP(0);
FI;

( * Compute size of MISC area *)
TMP_MISC_SIZE ← compute_misc_region_size();

(* Compute the size required to save state of the enclave on async exit, see Section 41.7.2.2*)
TMP_XSIZE ← compute_xsave_size(DS:TMP_SECS.ATTRIBUTES.XFRM) + GPR_SIZE + TMP_MISC_SIZE;

(* Ensure that the declared area is large enough to hold XSAVE and GPR stat *)
IF ( ( DS:TMP_SECS.SSAFRAMESIZE*4096 < TMP_XSIZE)
    THEN #GP(0); FI;

IF ( (DS:TMP_SECS.ATTRIBUTES.MODE64BIT = 1) and (DS:TMP_SECS.BASEADDR is not canonical) )
    THEN #GP(0); FI;

IF ( (DS:TMP_SECS.ATTRIBUTES.MODE64BIT = 0) and (DS:TMP_SECS.BASEADDR and 0FFFFFFFF00000000H) )
    THEN #GP(0); FI;

IF ( (DS:TMP_SECS.ATTRIBUTES.MODE64BIT = 0) and (DS:TMP_SECS.SIZE ≥ 2 ^ (CPUID.(EAX=12H, ECX=0):.EDX[7:0]) ) )
    THEN #GP(0); FI;

IF ( (DS:TMP_SECS.ATTRIBUTES.MODE64BIT = 1) and (DS:TMP_SECS.SIZE ≥ 2 ^ (CPUID.(EAX=12H, ECX=0):.EDX[15:8]) ) )
    THEN #GP(0); FI;

(* Enclave size must be at least 8192 bytes and must be power of 2 in bytes*)
IF (DS:TMP_SECS.SIZE < 8192 or popcnt(DS:TMP_SECS.SIZE) > 1)
    THEN #GP(0); FI;

(* Ensure base address of an enclave is aligned on size*)
IF ( ( DS:TMP_SECS.BASEADDR and (DS:TMP_SECS.SIZE-1) )
    THEN #GP(0); FI;

* Ensure the SECS does not have any unsupported attributes*)
IF ( ( DS:TMP_SECS.ATTRIBUTES and (~CR_SGX_ATTRIBUTES_MASK) )
    THEN #GP(0); FI;

IF ( ( DS:TMP_SECS reserved fields are not zero)
    THEN #GP(0); FI;

Clear DS:TMP_SECS to Uninitialized;
DS:TMP_SECS.MRENCLAVE ← SHA256INITIALIZE(DS:TMP_SECS.MRENCLAVE);
DS:TMP_SECS.ISVSVN ← 0;
DS:TMP_SECS.ISVPRODID ← 0;

(* Initialize hash updates etc*)
Initialize enclave's MRENCLAVE update counter;
```

```
(* Add "ECREATE" string and SECS fields to MRENCLAVE *)
TMPUPDATEFIELD[63:0] ← 0045544145524345H; // "ECREATE"
TMPUPDATEFIELD[95:64] ← DS:TMP_SECS.SSAFRAMESIZE;
TMPUPDATEFIELD[159:96] ← DS:TMP_SECS.SIZE;
TMPUPDATEFIELD[511:160] ← 0;
DS:TMP_SECS.MRENCLAVE ← SHA256UPDATE(DS:TMP_SECS.MRENCLAVE, TMPUPDATEFIELD)
INC enclave's MRENCLAVE update counter;

(* Set EID *)
DS:TMP_SECS.EID ← LockedXAdd(CR_NEXT_EID, 1);

(* Set the EPCM entry, first create SECS identifier and store the identifier in EPCM *)
EPCM(DS:TMP_SECS).PT ← PT_SECS;
EPCM(DS:TMP_SECS).ENCLAVEADDRESS ← 0;
EPCM(DS:TMP_SECS).R ← 0;
EPCM(DS:TMP_SECS).W ← 0;
EPCM(DS:TMP_SECS).X ← 0;

(* Set EPCM entry fields *)
EPCM(DS:RCX).BLOCKED ← 0;
EPCM(DS:RCX).PENDING ← 0;
EPCM(DS:RCX).MODIFIED ← 0;
EPCM(DS:RCX).PR ← 0;
EPCM(DS:RCX).VALID ← 1;
```

Flags Affected

None

Protected Mode Exceptions

#GP(0)	If a memory operand effective address is outside the DS segment limit.
	If a memory operand is not properly aligned.
	If the reserved fields are not zero.
	If PAGEINFO.SECS is not zero.
	If PAGEINFO.LINADDR is not zero.
	If the SECS destination is locked.
	If SECS.SSAFRAMESIZE is insufficient.
#PF(error code)	If a page fault occurs in accessing memory operands.
	If the SECS destination is outside the EPC.

64-Bit Mode Exceptions

#GP(0)	If a memory address is non-canonical form.
	If a memory operand is not properly aligned.
	If the reserved fields are not zero.
	If PAGEINFO.SECS is not zero.
	If PAGEINFO.LINADDR is not zero.
	If the SECS destination is locked.
	If SECS.SSAFRAMESIZE is insufficient.
#PF(error code)	If a page fault occurs in accessing memory operands.
	If the SECS destination is outside the EPC.

EDBGRD—Read From a Debug Enclave

Opcode/ Instruction	Op/En	64/32 bit Mode Support	CPUID Feature Flag	Description
EAX = 04H ENCLS[EDBGRD]	IR	V/V	SGX1	This leaf function reads a dword/quadword from a debug enclave.

Instruction Operand Encoding

Op/En	EAX	RBX	RCX
IR	EDBGRD (In)	Data read from a debug enclave (Out)	Address of source memory in the EPC (In)

Description

This leaf function copies a quadword/doubleword from an EPC page belonging to a debug enclave into the RBX register. Eight bytes are read in 64-bit mode, four bytes are read in non-64-bit modes. The size of data read cannot be overridden.

The effective address of the source location inside the EPC is provided in the register RCX.

EDBGRD Memory Parameter Semantics

EPCQW
Read access permitted by Enclave

The error codes are:

Table 40-14. EDBGRD Return Value in RAX

Error Code (see Table 40-3)	Description
No Error	EDBGRD successful
SGX_PAGE_NOT_DEBUGGABLE	The EPC page cannot be accessed because it is in the PENDING or MODIFIED state

The instruction faults if any of the following:

EDBGRD Faulting Conditions

RCX points into a page that is an SECS.	RCX does not resolve to a naturally aligned linear address.
RCX points to a page that does not belong to an enclave that is in debug mode.	RCX points to a location inside a TCS that is beyond the architectural size of the TCS (SGX_TCS_LIMIT).
An operand causing any segment violation.	May page fault.
CPL > 0.	

This instruction ignores the EPCM RWX attributes on the enclave page. Consequently, violation of EPCM RWX attributes via EDBGRD does not result in a #GP.

Concurrency Restrictions

Table 40-15. Concurrency Restrictions of EDBGRD with Other Intel® SGX Operations 1 of 2

Operation		EEXIT			EADD		EBLOCK		ECREATE	EDBGRD/WR		EENTER/ERESUME			EEXTEND		EGETKEY		EINIT	ELDB/ELDU			EPA
	Param	TCS	SSA	SECS	Targ	SECS	Targ	SECS	SECS	Targ	SECS	TCS	SSA	SECS	Targ	SECS	Param	SECS	SECS	Targ	VA	SECS	VA
EDBGRD	Targ	Y	Y		N		Y		N	Y		Y	Y		Y		Y		N	N	Y		N
	SECS			Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y

Table 40-16. Concurrency Restrictions of EDBGRD with Other Intel® SGX Operations 2 of 2

Operation		EREMOVE		EREPORT		ETRACK	EWB			EAUG		EMODPE		EMODPR		EMODT		EACCEPT			EACCEPTCOPY		
	Param	Targ	SECS	Param	SECS	SECS	SRC	VA	SECS	Targ	SECS	Targ	SECINFO	Targ	SECS	Targ	SECS	Targ	SECINFO	SECS	Targ	SRC	SECINFO
EDBGRD	Targ	N		Y		N	N	Y		N		Y	Y	Y		N			Y			Y	Y
	SECS	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y

Operation

Temp Variables in EDBGRD Operational Flow

Name	Type	Size (Bits)	Description
TMP_MODE64	Binary	1	((IA32_EFER.LMA = 1) && (CS.L = 1))
TMP_SECS		64	Physical address of SECS of the enclave to which source operand belongs

```
TMP_MODE64 ← ((IA32_EFER.LMA = 1) && (CS.L = 1));

IF ( (TMP_MODE64 = 1) and (DS:RCX is not 8Byte Aligned) )
    THEN #GP(0); FI;

IF ( (TMP_MODE64 = 0) and (DS:RCX is not 4Byte Aligned) )
    THEN #GP(0); FI;

IF (DS:RCX does not resolve within an EPC)
    THEN #PF(DS:RCX); FI;

(* make sure no other Intel SGX instruction is accessing EPCM *)
IF (Other EPCM modifying instructions executing)
    THEN #GP(0); FI;

IF (EPCM(DS:RCX). VALID = 0)
    THEN #PF(DS:RCX); FI;

(* make sure that DS:RCX (SOURCE) is pointing to a PT_REG or PT_TCS or PT_VA *)
IF ( (EPCM(DS:RCX).PT ≠ PT_REG) and (EPCM(DS:RCX).PT ≠ PT_TCS) and (EPCM(DS:RCX).PT ≠ PT_VA))
    THEN #PF(DS:RCX); FI;

(* make sure that DS:RCX points to an accessible EPC page *)
IF ( (EPCM(DS:RCX).PENDING is not 0 or (EPCM(DS:RCX).MODIFIED is not 0) )
    THEN
        RFLAGS.ZF ← 1;
```

```
        RAX ← SGX_PAGE_NOT_DEBUGGABLE;
        GOTO DONE;
FI;

(* If source is a TCS, then make sure that the offset into the page is not beyond the TCS size*)
IF ( ( EPCM(DS:RCX). PT = PT_TCS) and ((DS:RCX) & FFFH ≥ SGX_TCS_LIMIT) )
    THEN #GP(0); FI;

(* make sure the enclave owning the PT_REG or PT_TCS page allow debug *)
IF ( (EPCM(DS:RCX).PT = PT_REG) or (EPCM(DS:RCX).PT = PT_TCS) )
    THEN
        TMP_SECS ← GET_SECS_ADDRESS;
        IF (TMP_SECS.ATTRIBUTES.DEBUG = 0)
            THEN #GP(0); FI;
        IF ( (TMP_MODE64 = 1) )
            THEN RBX[63:0] ← (DS:RCX)[63:0];
            ELSE EBX[31:0] ← (DS:RCX)[31:0];
        FI;
    ELSE
        TMP_64BIT_VAL[63:0] ← (DS:RCX)[63:0] & (~07H); // Read contents from VA slot
        IF (TMP_MODE64 = 1)
            THEN
                IF (TMP_64BIT_VAL ≠ 0H)
                    THEN RBX[63:0] ← 0FFFFFFFFFFFFFFFFH;
                    ELSE RBX[63:0] ← 0H;
                FI;
            ELSE
                IF (TMP_64BIT_VAL ≠ 0H)
                    THEN EBX[31:0] ← 0FFFFFFFFH;
                    ELSE EBX[31:0] ← 0H;
                FI;
FI;

(* clear EAX and ZF to indicate successful completion *)
RAX ← 0;
RFLAGS.ZF ← 0;

DONE:
(* clear flags *)
RFLAGS.CF,PF,AF,OF,SF ← 0;
```

Flags Affected

None

Protected Mode Exceptions

#GP(0)	If the address in RCS violates DS limit or access rights.
	If DS segment is unusable.
	If RCX points to a memory location not 4Byte-aligned.
	If the address in RCX points to a page belonging to a non-debug enclave.
	If the address in RCX points to a page which is not PT_TCS, PT_REG or PT_VA.
	If the address in RCX points to a location inside TCS that is beyond SGX_TCS_LIMIT.
#PF(error code)	If a page fault occurs in accessing memory operands.

	If the address in RCX points to a non-EPC page.
	If the address in RCX points to an invalid EPC page.

64-Bit Mode Exceptions

#GP(0)	If RCX is non-canonical form.
	If RCX points to a memory location not 8Byte-aligned.
	If the address in RCX points to a page belonging to a non-debug enclave.
	If the address in RCX points to a page which is not PT_TCS, PT_REG or PT_VA.
	If the address in RCX points to a location inside TCS that is beyond SGX_TCS_LIMIT.
#PF(error code)	If a page fault occurs in accessing memory operands.
	If the address in RCX points to a non-EPC page.
	If the address in RCX points to an invalid EPC page.

EDBGWR—Write to a Debug Enclave

Opcode/ Instruction	Op/En	64/32 bit Mode Support	CPUID Feature Flag	Description
EAX = 05H ENCLS[EDBGWR]	IR	V/V	SGX1	This leaf function writes a dword/quadword to a debug enclave.

Instruction Operand Encoding

Op/En	EAX	RBX	RCX
IR	EDBGWR (In)	Data to be written to a debug enclave (In)	Address of Target memory in the EPC (In)

Description

This leaf function copies the content in EBX/RBX to an EPC page belonging to a debug enclave. Eight bytes are written in 64-bit mode, four bytes are written in non-64-bit modes. The size of data cannot be overridden.

The effective address of the source location inside the EPC is provided in the register RCX.

EDBGWR Memory Parameter Semantics

EPCQW
Write access permitted by Enclave

The instruction faults if any of the following:

EDBGWR Faulting Conditions

RCX points into a page that is an SECS.	RCX does not resolve to a naturally aligned linear address.
RCX points to a page that does not belong to an enclave that is in debug mode.	RCX points to a location inside a TCS that is not the FLAGS word.
An operand causing any segment violation.	May page fault.
CPL > 0.	

The error codes are:

Table 40-17. EDBGWR Return Value in RAX

Error Code (see Table 40-3)	Description
No Error	EDBGWR successful
SGX_PAGE_NOT_DEBUGGABLE	The EPC page cannot be accessed because it is in the PENDING or MODIFIED state

This instruction ignores the EPCM RWX attributes on the enclave page. Consequently, violation of EPCM RWX attributes via EDBGRD does not result in a #GP.

Concurrency Restrictions

Table 40-18. Concurrency Restrictions of EDBGWR with Other Intel® SGX Operations 1 of 2

Operation		EEXIT			EADD		EBLOCK		ECREATE	EDBGRD/WR		EENTER/ERESUME			EEXTEND		EGETKEY		EINIT	ELDB/ELDU			EPA
	Param	TCS	SSA	SECS	Targ	SECS	Targ	SECS	SECS	Targ	SECS	TCS	SSA	SECS	Targ	SECS	Param	SECS	SECS	Targ	VA	SECS	VA
EDBGWR	Targ	Y	Y		N		Y		N	Y		Y	Y		Y		Y		N	N	Y		N
	SECS			Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y

Table 40-19. Concurrency Restrictions of EDBGWR with Other Intel® SGX Operations 2 of 2

Operation		EREMOVE		EREPORT		ETRACK	EWB			EAUG		EMODPE		EMODPR		EMODT		EACCEPT			EACCEPTCOPY		
	Param	Targ	SECS	Param	SECS	SECS	SRC	VA	SECS	Targ	SECS	Targ	SECINFO	Targ	SECS	Targ	SECS	Targ	SECINFO	SECS	Targ	SRC	SECINFO
EDBGWR	Targ	N		Y		N	N	Y		N		Y	Y	Y		N			Y			Y	Y
	SECS	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y

Operation

Temp Variables in EDBGWR Operational Flow

Name	Type	Size (Bits)	Description
TMP_MODE64	Binary	1	((IA32_EFER.LMA = 1) && (CS.L = 1)).
TMP_SECS		64	Physical address of SECS of the enclave to which source operand belongs.

```
TMP_MODE64 ← ((IA32_EFER.LMA = 1) && (CS.L = 1));

IF ( (TMP_MODE64 = 1) and (DS:RCX is not 8Byte Aligned) )
    THEN #GP(0); FI;

IF ( (TMP_MODE64 = 0) and (DS:RCX is not 4Byte Aligned) )
    THEN #GP(0); FI;

IF (DS:RCX does not resolve within an EPC)
    THEN #PF(DS:RCX); FI;

(* make sure no other Intel SGX instruction is accessing EPCM *)
IF (Other EPCM modifying instructions executing)
    THEN #GP(0); FI;

IF (EPCM(DS:RCX). VALID = 0)
    THEN #PF(DS:RCX); FI;

(* make sure that DS:RCX (DST) is pointing to a PT_REG or PT_TCS *)
IF ( (EPCM(DS:RCX).PT ≠ PT_REG) and (EPCM(DS:RCX).PT ≠ PT_TCS) )
    THEN #PF(DS:RCX); FI;

(* make sure that DS:RCX points to an accessible EPC page *)
IF ( (EPCM(DS:RCX).PENDING is not 0) or (EPCM(DS:RCS).MODIFIED is not 0) )
    THEN
        RFLAGS.ZF ← 1;
        RAX ← SGX_PAGE_NOT_DEBUGGABLE;
        GOTO DONE;
```

```
FI;

(* If destination is a TCS, then make sure that the offset into the page can only point to the FLAGS field*)
IF ( ( EPCM(DS:RCX). PT = PT_TCS) and ((DS:RCX) & FF8H ≠ offset_of_FLAGS & 0FF8H) )
    THEN #GP(0); FI;

(* Locate the SECS for the enclave to which the DS:RCX page belongs *)
TMP_SECS ← GET_SECS_PHYS_ADDRESS(EPCM(DS:RCX).ENCLAVESECS);

(* make sure the enclave owning the PT_REG or PT_TCS page allow debug *)
IF (TMP_SECS.ATTRIBUTES.DEBUG = 0)
    THEN #GP(0); FI;

IF ( (TMP_MODE64 = 1) )
    THEN (DS:RCX)[63:0] ← RBX[63:0];
    ELSE (DS:RCX)[31:0] ← EBX[31:0];
FI;

(* clear EAX and ZF to indicate successful completion *)
RAX ← 0;
RFLAGS.ZF ← 0;

DONE:
(* clear flags *)
RFLAGS.CF,PF,AF,OF,SF ← 0
```

Flags Affected

None

Protected Mode Exceptions

#GP(0)	If the address in RCS violates DS limit or access rights.
	If DS segment is unusable.
	If RCX points to a memory location not 4Byte-aligned.
	If the address in RCX points to a page belonging to a non-debug enclave.
	If the address in RCX points to a page which is not PT_TCS or PT_REG.
	If the address in RCX points to a location inside TCS that is not the FLAGS word.
#PF(error code)	If a page fault occurs in accessing memory operands.
	If the address in RCX points to a non-EPC page.
	If the address in RCX points to an invalid EPC page.

64-Bit Mode Exceptions

#GP(0)	If RCX is non-canonical form.
	If RCX points to a memory location not 8Byte-aligned.
	If the address in RCX points to a page belonging to a non-debug enclave.
	If the address in RCX points to a page which is not PT_TCS or PT_REG.
	If the address in RCX points to a location inside TCS that is not the FLAGS word.
#PF(error code)	If a page fault occurs in accessing memory operands.
	If the address in RCX points to a non-EPC page.
	If the address in RCX points to an invalid EPC page.

EEXTEND—Extend Uninitialized Enclave Measurement by 256 Bytes

Opcode/ Instruction	Op/En	64/32 bit Mode Support	CPUID Feature Flag	Description
EAX = 06H ENCLS[EEXTEND]	IR	V/V	SGX1	This leaf function measures 256 bytes of an uninitialized enclave page.

Instruction Operand Encoding

Op/En	EAX	EBX	RCX
IR	EEXTEND (In)	Effective address of the SECS of the data chunk (In)	Effective address of a 256-byte chunk in the EPC (In)

Description

This leaf function updates the MRENCLAVE measurement register of an SECS with the measurement of an EXTEND string compromising of "EEXTEND" || ENCLAVEOFFSET || PADDING || 256 bytes of the enclave page. This instruction can only be executed when current privilege level is 0 and the enclave is uninitialized.

RBX contains the effective address of the SECS of the region to be measured. The address must be the same as the one used to add the page into the enclave.

RCX contains the effective address of the 256 byte region of an EPC page to be measured. The DS segment is used to create linear addresses. Segment override is not supported.

EEXTEND Memory Parameter Semantics

EPC[RCX]
Read access by Enclave

The instruction faults if any of the following:

EEXTEND Faulting Conditions

RBX points to an address not 4KBytes aligned.	RBX does not resolve to an SECS.
RBX does not point to an SECS page.	RBX does not point to the SECS page of the data chunk.
RCX points to an address not 256B aligned.	RCX points to an unused page or a SECS.
RCX does not resolve in an EPC page.	If SECS is locked.
If the SECS is already initialized.	May page fault.
CPL > 0.	

Concurrency Restrictions

Table 40-20. Concurrency Restrictions of EEXTEND with Other Intel® SGX Operations 1 of 2

Operation		EEXIT			EADD		EBLOCK		ECREATE	EDBGRD/WR		EENTER/ ERESUME			EEXTEND		EGETKEY		EINIT	ELDB/ELDU			EPA
	Param	TCS	SSA	SECS	Targ	SECS	Targ	SECS	SECS	Targ	SECS	TCS	SSA	SECS	Targ	SECS	Param	SECS	SECS	Targ	VA	SECS	VA
EEXTEND	Targ	N	N		N		Y		N	Y					Y				N	N			N
	SECS	Y	Y	Y	Y	N	Y	Y	Y	Y	Y	Y	Y	N	Y	N	Y	N	N	Y	Y	Y	Y

Table 40-21. Concurrency Restrictions of EEXTEND with Other Intel® SGX Operations 2 of 2

Operation		EREMOVE		EREPORT		ETRACK	EWB			EAUG		EMODPE		EMODPR		EMODT		EACCEPT			EACCEPTCOPY		
	Param	Targ	SECS	Param	SECS	SECS	SRC	VA	SECS	Targ	SECS	Targ	SECINFO	Targ	SECS	Targ	SECS	Targ	SECINFO	SECS	Targ	SRC	SECINFO
EEXTEND	Targ	N					N			N				N		N							
	SECS	Y	Y	Y	N	Y	Y	Y	Y	Y	N	Y	Y	Y	N	Y	N	Y	Y	N	Y	Y	Y

Operation

Temp Variables in EEXTEND Operational Flow

Name	Type	Size (Bits)	Description
TMP_SECS		64	Physical address of SECS of the enclave to which source operand belongs.
TMP_ENCLAVEOFFSET	Enclave Offset	64	The page displacement from the enclave base address.
TMPUPDATEFIELD	SHA256 Buffer	512	Buffer used to hold data being added to TMP_SECS.MRENCLAVE.

```
TMP_MODE64 ← ((IA32_EFER.LMA = 1) && (CS.L = 1));

IF (DS:RBX is not 4096 Byte Aligned)
    THEN #GP(0); FI;

IF (DS:RBX does resolve to an EPC page)
    THEN #PF(DS:RBX); FI;

IF (DS:RCX is not 256Byte Aligned)
    THEN #GP(0); FI;

IF (DS:RCX does not resolve within an EPC)
    THEN #PF(DS:RCX); FI;

(* make sure no other Intel SGX instruction is accessing EPCM *)
IF (Other instructions accessing EPCM)
    THEN #GP(0); FI;

IF (EPCM(DS:RCX). VALID = 0)
    THEN #PF(DS:RCX); FI;

(* make sure that DS:RCX (DST) is pointing to a PT_REG or PT_TCS *)
IF ( (EPCM(DS:RCX).PT ≠ PT_REG) and (EPCM(DS:RCX).PT ≠ PT_TCS) )
    THEN #PF(DS:RCX); FI;

TMP_SECS ← Get_SECS_ADDRESS();

IF (DS:RBX does not resolve to TMP_SECS)
    THEN #GP(0); FI;

(* make sure no other instruction is accessing MRENCLAVE or ATTRIBUETS.INIT *)
IF ( (Other instruction accessing MRENCLAVE) or (Other instructions checking or updating the initialized state of the SECS))
    THEN #GP(0); FI;

(* Calculate enclave offset *)
```

```
TMP_ENCLAVEOFFSET ← EPCM(DS:RCX).ENCLAVEADDRESS - TMP_SECS.BASEADDR;
TMP_ENCLAVEOFFSET ← TMP_ENCLAVEOFFSET + (DS:RCX & 0FFFH)

(* Add EEXTEND message and offset to MRENCLAVE *)
TMPUPDATEFIELD[63:0] ← 00444E4554584545H; // "EEXTEND"
TMPUPDATEFIELD[127:64] ← TMP_ENCLAVEOFFSET;
TMPUPDATEFIELD[511:128] ← 0; // 48 bytes
TMP_SECS.MRENCLAVE ← SHA256UPDATE(TMP_SECS.MRENCLAVE, TMPUPDATEFIELD)
INC enclave's MRENCLAVE update counter;

(*Add 256 bytes to MRENCLAVE, 64 byte at a time *)
TMP_SECS.MRENCLAVE ← SHA256UPDATE(TMP_SECS.MRENCLAVE, DS:RCX[511:0] );
TMP_SECS.MRENCLAVE ← SHA256UPDATE(TMP_SECS.MRENCLAVE, DS:RCX[1023: 512] );
TMP_SECS.MRENCLAVE ← SHA256UPDATE(TMP_SECS.MRENCLAVE, DS:RCX[1535: 1024] );
TMP_SECS.MRENCLAVE ← SHA256UPDATE(TMP_SECS.MRENCLAVE, DS:RCX[2047: 1536] );
INC enclave's MRENCLAVE update counter by 4;
```

Flags Affected

None

Protected Mode Exceptions

#GP(0)	If the address in RBX is outside the DS segment limit.
	If RBX points to an SECS page which is not the SECS of the data chunk.
	If the address in RCX is outside the DS segment limit.
	If RCX points to a memory location not 256Byte-aligned.
	If another instruction is accessing MRENCLAVE.
	If another instruction is checking or updating the SECS.
	If the enclave is already initialized.
#PF(error code)	If a page fault occurs in accessing memory operands.
	If the address in RBX points to a non-EPC page.
	If the address in RCX points to a page which is not PT_TCS or PT_REG.
	If the address in RCX points to a non-EPC page.
	If the address in RCX points to an invalid EPC page.

64-Bit Mode Exceptions

#GP(0)	If RBX is non-canonical form.
	If RBX points to an SECS page which is not the SECS of the data chunk.
	If RCX is non-canonical form.
	If RCX points to a memory location not 256 Byte-aligned.
	If another instruction is accessing MRENCLAVE.
	If another instruction is checking or updating the SECS.
	If the enclave is already initialized.
#PF(error code)	If a page fault occurs in accessing memory operands.
	If the address in RBX points to a non-EPC page.
	If the address in RCX points to a page which is not PT_TCS or PT_REG.
	If the address in RCX points to a non-EPC page.
	If the address in RCX points to an invalid EPC page.

EINIT—Initialize an Enclave for Execution

Opcode/ Instruction	Op/En	64/32 bit Mode Support	CPUID Feature Flag	Description
EAX = 02H ENCLS[EINIT]	IR	V/V	SGX1	This leaf function initializes the enclave and makes it ready to execute enclave code.

Instruction Operand Encoding

Op/En	EAX		RBX	RCX	RDX
IR	EINIT (In)	Error code (Out)	Address of SIGSTRUCT (In)	Address of SECS (In)	Address of EINITTOKEN (In)

Description

This leaf function is the final instruction executed in the enclave build process. After EINIT, the MRENCLAVE measurement is complete, and the enclave is ready to start user code execution using the EENTER instruction.

EINIT takes the effective address of a SIGSTRUCT and EINITTOKEN. The SIGSTRUCT describes the enclave including MRENCLAVE, ATTRIBUTES, ISVSVN, a 3072 bit RSA key, and a signature using the included key. SIGSTRUCT must be populated with two values, q1 and q2. These are calculated using the formulas shown below:

q1 = floor(Signature^2 / Modulus);

q2 = floor((Signature^3 - q1 * Signature * Modulus) / Modulus);

The EINITTOKEN contains the MRENCLAVE, MRSIGNER, and ATTRIBUTES. These values must match the corresponding values in the SECS. If the EINITTOKEN was created with a debug launch key, the enclave must be in debug mode as well.

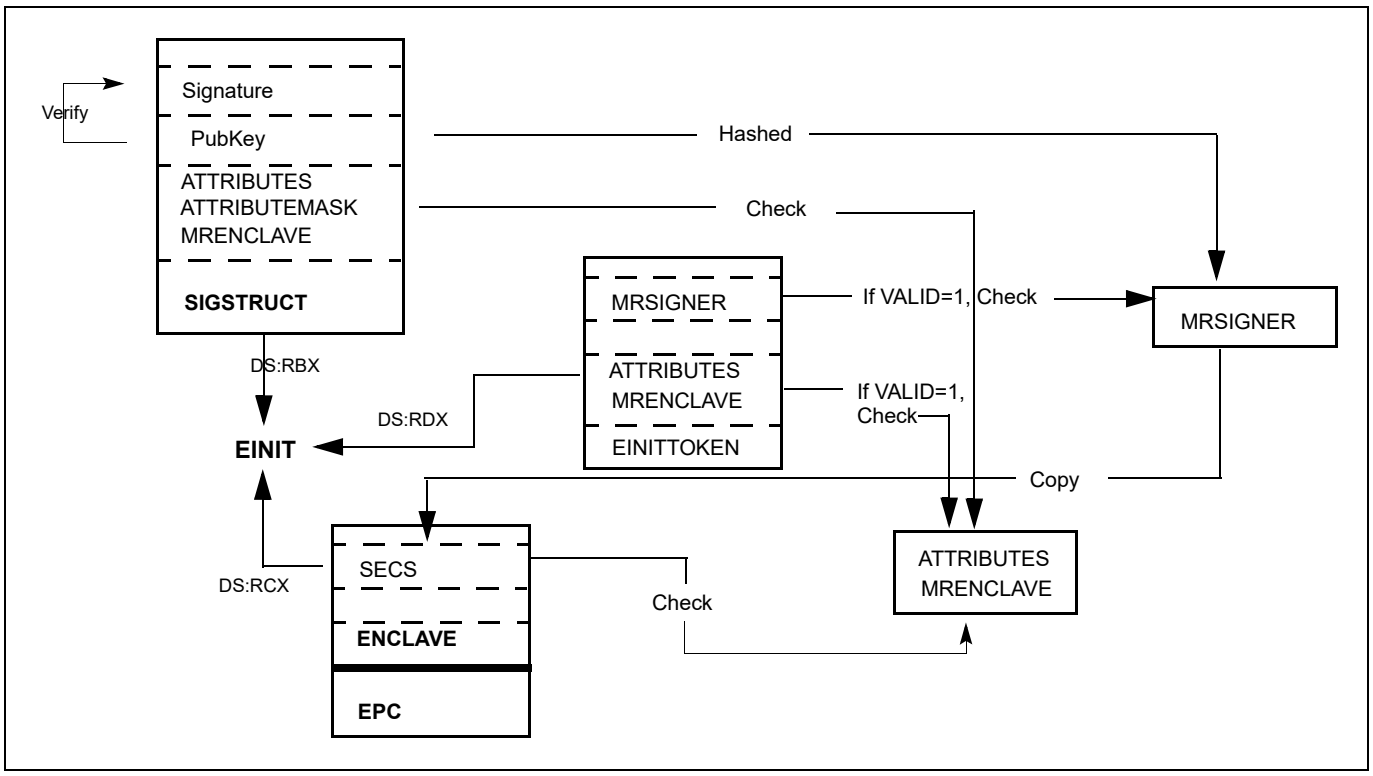


Figure 40-1. Relationships Between SECS, SIGSTRUCT and EINITTOKEN

EINIT Memory Parameter Semantics

SIGSTRUCT	SECS	EINITTOKEN
Access by non-Enclave	Read/Write access by Enclave	Access by non-Enclave

EINIT performs the following steps, which can be seen in Figure 40-1:

Validates that SIGSTRUCT is signed using the enclosed public key.

Checks that the completed computation of SECS.MRENCLAVE equals SIGSTRUCT.HASHENCLAVE.

Checks that no reserved bits are set to 1 in SIGSTRUCT.ATTRIBUTES and no reserved bits in SIGSTRUCT.ATTRIBUTESMASK are set to 0.

Checks that no controlled ATTRIBUTES bits are set in SIGSTRUCT.ATTRIBUTES unless the SHA256 digest of SIGSTRUCT.MODULUS equals IA32_SGX_LEPUBKEYHASH.

Checks that SIGSTRUCT.ATTRIBUTES equals the result of logically and-ing SIGSTRUCT.ATTRIBUTEMASK with SECS.ATTRIBUTES.

If EINITTOKEN.VALID is 0, checks that the SHA256 digest of SIGSTRUCT.MODULUS equals IA32_SGX_LEPUBKEYHASH.

If EINITTOKEN.VALID is 1, checks the validity of EINITTOKEN.

If EINITTOKEN.VALID is 1, checks that EINITTOKEN.MRENCLAVE equals SECS.MRENCLAVE.

If EINITTOKEN.VALID is 1 and EINITTOKEN.ATTRIBUTES.DEBUG is 1, SECS.ATTRIBUTES.DEBUG must be 1.

Commits SECS.MRENCLAVE, and sets SECS.MRSIGNER, SECS.ISVSVN, and SECS.ISVPRODID based on SIGSTRUCT.

Update the SECS as Initialized.

Periodically, EINIT polls for certain asynchronous events. If such an event is detected, it completes with failure code (ZF=1 and RAX = SGX_UNMASKED_EVENT), and RIP is incremented to point to the next instruction. These events includes external interrupts, non-maskable interrupts, system-management interrupts, machine checks, INIT signals, and the VMX-preemption timer. EINIT does not fail if the pending event is inhibited (e.g., external interrupts could be inhibited due to blocking by MOV SS blocking or by STI).

The following bits in RFLAGS are cleared: CF, PF, AF, OF, and SF. When the instruction completes with an error, RFLAGS.ZF is set to 1, and the corresponding error bit is set in RAX. If no error occurs, RFLAGS.ZF is cleared and RAX is set to 0.

The error codes are:

Table 40-22. EINIT Return Value in RAX

Error Code (see Table 40-3)	Description
No Error	EINIT successful
SGX_INVALID_SIG_STRUCT	If SIGSTRUCT contained an invalid value
SGX_INVALID_ATTRIBUTE	If SIGSTRUCT contains an unauthorized attributes mask
SGX_INVALID_MEASUREMENT	If SIGSTRUCT contains an incorrect measurement If EINITTOKEN contains an incorrect measurement
SGX_INVALID_SIGNATURE	If signature does not validate with enclosed public key
SGX_INVALID_LICENSE	If license is invalid
SGX_INVALID_CPUSVN	If license SVN is unsupported
SGX_UNMASKED_EVENT	If an unmasked event is received before the instruction completes its operation

Concurrency Restrictions

Table 40-23. Concurrency Restrictions of EINIT with Other Intel® SGX Operations 1 of 2

Operation		EEXIT			EADD		EBLOCK		ECREATE	EDBGRD/WR		EENTER/ERESUME			EEXTEND		EGETKEY		EINIT	ELDB/ELDU			EPA
	Param	TCS	SSA	SECS	Targ	SECS	Targ	SECS	SECS	Targ	SECS	TCS	SSA	SECS	Targ	SECS	Param	SECS	SECS	Targ	VA	SECS	VA
EINIT	SECS			N	N	N	Y	Y	N	N	Y			N	N	N		N	N	N		Y	N

Table 40-24. Concurrency Restrictions of EINIT with Other Intel® SGX Operations 2 of 2

Operation		EREMOVE		EREPORT		ETRACK	EWB			EAUG		EMODPE		EMODPR		EMODT		EACCEPT			EACCEPTCOPY		
	Param	Targ	SECS	Param	SECS	SECS	SRC	VA	SECS	Targ	SECS	Targ	SECINFO	Targ	SECS	Targ	SECS	Targ	SECINFO	SECS	Targ	SRC	SECINFO
EINIT	SECS	N	Y		N	Y	N		Y	N	N				N	N	N			N			

Operation

Temp Variables in EINIT Operational Flow

Name	Type	Size	Description
TMP_SIG	SIGSTRUCT	1808Bytes	Temp space for SIGSTRUCT.
TMP_TOKEN	EINITTOKEN	304Bytes	Temp space for EINITTOKEN.
TMP_MRENCLAVE		32Bytes	Temp space for calculating MRENCLAVE.
TMP_MRSIGNER		32Bytes	Temp space for calculating MRSIGNER.
CONTROLLED_ATTRIBUTES	ATTRIBUTES	16Bytes	Constant mask of all ATTRIBUTE bits that can only be set for authorized enclaves.
TMP_KEYDEPENDENCIES	Buffer	224Bytes	Temp space for key derivation.
TMP_EINITTOKENKEY		16Bytes	Temp space for the derived EINITTOKEN Key.
TMP_SIG_PADDING	PKCS Padding Buffer	352Bytes	The value of the top 352 bytes from the computation of Signature[3] modulo MRSIGNER.

```
(* make sure SIGSTRUCT and SECS are aligned *)
IF ( (DS:RBX is not 4KByte Aligned) or (DS:RCX is not 4KByte Aligned) )
    THEN #GP(0); FI;

(* make sure the EINITTOKEN is aligned *)
IF (DS:RDX is not 512Byte Aligned)
    THEN #GP(0); FI;

(* make sure the SECS is inside the EPC *)
IF (DS:RCX does not resolve within an EPC)
    THEN #PF(DS:RCX); FI;

TMP_SIG[14463:0] ← DS:RBX[14463:0]; // 1808 bytes
TMP_TOKEN[2423:0] ← DS:RDX[2423:0]; // 304 bytes

(* Verify SIGSTRUCT Header. *)
IF ( (TMP_SIG.HEADER ≠ 06000000E10000000000010000000000h) or
    ((TMP_SIG.VENDOR ≠ 0) and (TMP_SIG.VENDOR ≠ 00008086h) ) or
    (TMP_SIG HEADER2 ≠ 01010000600000006000000001000000h) or
```

```
    (TMP_SIG.EXPONENT  ≠ 00000003h) or (Reserved space is not 0's) )
    THEN
        RFLAGS.ZF ← 1;
        RAX ← SGX_INVALID_SIG_STRUCT;
        GOTO EXIT;
FI;

(* Open "Event Window" Check for Interrupts. Verify signature using embedded public key, q1, and q2. Save upper 352 bytes of the
PKCS1.5 encoded message into the TMP_SIG_PADDING*)
IF (interrupt was pending) THEN
    RFLAGS.ZF ← 1;
    RAX ← SGX_UNMASKED_EVENT;
    GOTO EXIT;
FI
IF (signature failed to verify) THEN
    RFLAGS.ZF ← 1;
    RAX ← SGX_INVALID_SIGNATURE;
    GOTO EXIT;
FI;
(*Close "Event Window" *)

(* make sure no other Intel SGX instruction is modifying SECS*)
IF (Other instructions modifying SECS)
    THEN #GP(0); FI;

IF ( (EPCM(DS:RCX). VALID = 0) or (EPCM(DS:RCX).PT ≠ PT_SECS) )
    THEN #PF(DS:RCX); FI;

(* make sure no other instruction is accessing MRENCLAVE or ATTRIBUETS.INIT *)
IF ( (Other instruction modifying MRENCLAVE) or (Other instructions modifying the SECS's Initialized state))
    THEN #GP(0); FI;

(* Calculate finalized version of MRENCLAVE *)
(* SHA256 algorithm requires one last update that compresses the length of the hashed message into the output SHA256 digest *)
TMP_ENCLAVE ←   SHA256FINAL( (DS:RCX).MRENCLAVE, enclave's MRENCLAVE update count *512);

(* Verify MRENCLAVE from SIGSTRUCT *)
IF (TMP_SIG.ENCLAVEHASH ≠ TMP_MRENCLAVE)
    RFLAGS.ZF ← 1;
    RAX ← SGX_INVALID_MEASUREMENT;
    GOTO EXIT;
FI;

TMP_MRSIGNER ← SHA256(TMP_SIG.MODULUS)

(* if controlled ATTRIBUTES are set, SIGSTRUCT must be signed using an authorized key *)
CONTROLLED_ATTRIBUTES ← 0000000000000020H;
IF ( ( (DS:RCX.ATTRIBUTES & CONTROLLED_ATTRIBUTES) ≠ 0) and (TMP_MRSIGNER ≠ IA32_SGXLEPUBKEYHASH) )
    RFLAGS.ZF ← 1;
    RAX ← SGX_INVALID_ATTRIBUTE;
    GOTO EXIT;
FI;

(* Verify SIGSTRUCT.ATTRIBUTE requirements are met *)
```

```
IF ( (DS:RCX.ATTRIBUTES & TMP_SIG.ATTRIBUTEMASK) ≠ (TMP_SIG.ATTRIBUTE & TMP_SIG.ATTRIBUTEMASK) )
    RFLAGS.ZF ← 1;
    RAX ← SGX_INVALID_ATTRIBUTE;
    GOTO EXIT;
FI;

( *Verify SIGSTRUCT.MISCSELECT requirements are met *)
IF ( (DS:RCX.MISCSELECT & TMP_SIG.MISCMASK) ≠ (TMP_SIG.MISCSELECT & TMP_SIG.MISCMASK) )
    THEN
        RFLAGS.ZF ← 1;
        RAX ← SGX_INVALID_ATTRIBUTE;
    GOTO EXIT
FI;

(* if EINITTOKEN.VALID[0] is 0, verify the enclave is signed by an authorized key *)
IF (TMP_TOKEN.VALID[0] = 0)
    IF (TMP_MRSIGNER ≠ IA32_SGXLEPUBKEYHASH)
        RFLAGS.ZF ← 1;
        RAX ← SGX_INVALID_EINITTOKEN;
        GOTO EXIT;
    FI;
    GOTO COMMIT;
FI;

(* Debug Launch Enclave cannot launch Production Enclaves *)
IF ( (DS:RDX.MASKEDATTRIBUTESLE.DEBUG = 1) and (DS:RCX.ATTRIBUTES.DEBUG = 0) )
    RFLAGS.ZF ← 1;
    RAX ← SGX_INVALID_EINITTOKEN;
    GOTO EXIT;
FI;

(* Check reserve space in EINIT token includes reserved regions and upper bits in valid field *)
IF (TMP_TOKEN reserved space is not clear)
    RFLAGS.ZF ← 1;
    RAX ← SGX_INVALID_EINITTOKEN;
    GOTO EXIT;
FI;

(* EINIT token must be ≤ CR_CPUSVN *)
IF (TMP_TOKEN.CPUSVN > CR_CPUSVN)
    RFLAGS.ZF ← 1;
    RAX ← SGX_INVALID_CPUSVN;
    GOTO EXIT;
FI;

(* Derive Launch key used to calculate EINITTOKEN.MAC *)
HARDCODED_PKCS1_5_PADDING[15:0] ← 0100H;
HARDCODED_PKCS1_5_PADDING[2655:16] ← SignExtend330Byte(-1); // 330 bytes of 0FFH
HARDCODED_PKCS1_5_PADDING[2815:2656] ← 2004000501020403650148866009060D30313000H;

TMP_KEYDEPENDENCIES.KEYNAME ← EINITTOKEN_KEY;
TMP_KEYDEPENDENCIES.ISVPRODID ← TMP_TOKEN.ISVPRODIDLE;
TMP_KEYDEPENDENCIES.ISVSVN ← TMP_TOKEN.ISVSVN;
TMP_KEYDEPENDENCIES.SGXOWNEREPOCH ← CR_SGXOWNEREPOCH;
```

```
TMP_KEYDEPENDENCIES.ATTRIBUTES ← TMP_TOKEN.MASKEDATTRIBUTESLE;
TMP_KEYDEPENDENCIES.ATTRIBUTESMASK ← 0;
TMP_KEYDEPENDENCIES.MRENCLAVE ← 0;
TMP_KEYDEPENDENCIES.MRSIGNER ← IA32_SGXLEPUBKEYHASH;
TMP_KEYDEPENDENCIES.KEYID ← TMP_TOKEN.KEYID;
TMP_KEYDEPENDENCIES.SEAL_KEY_FUSES ← CR_SEAL_FUSES;
TMP_KEYDEPENDENCIES.CPUSVN ← TMP_TOKEN.CPUSVN;
TMP_KEYDEPENDENCIES.MISCSELECT ← TMP_TOKEN.MASKEDMISCSELECTLE;
TMP_KEYDEPENDENCIES.MISCMASK ← 0;
TMP_KEYDEPENDENCIES.PADDING ← HARDCODED_PKCS1_5_PADDING;

(* Calculate the derived key*)
TMP_EINITTOKENKEY ← derivekey(TMP_KEYDEPENDENCIES);

(* Verify EINITTOKEN was generated using this CPU's Launch key and that it has not been modified since issuing by the Launch
Enclave. Only 192 bytes of EINITTOKEN are CMACed *)
IF (TMP_TOKEN.MAC ≠ CMAC(TMP_EINITTOKENKEY, TMP_TOKEN[1535:0] ) )
    RFLAGS.ZF ← 1;
    RAX ← SGX_INVALID_EINITTOKEN;
    GOTO EXIT;
FI;

(* Verify EINITTOKEN (RDX) is for this enclave *)
IF (TMP_TOKEN.MRENCLAVE ≠ TMP_MRENCLAVE) or (TMP_TOKEN.MRSIGNER ≠ TMP_MRSIGNER) )
    RFLAGS.ZF ← 1;
    RAX ← SGX_INVALID_MEASUREMENT;
    GOTO EXIT;
FI;

(* Verify ATTRIBUTES in EINITTOKEN are the same as the enclave's *)
IF (TMP_TOKEN.ATTRIBUTES ≠ DS:RCX.ATTRIBUTES)
    RFLAGS.ZF ← 1;
    RAX ← SGX_INVALID_EINIT_ATTRIBUTE;
    GOTO EXIT;
FI;

COMMIT:
(* Commit changes to the SECS; Set ISVPRODID, ISVSVN, MRSIGNER, INIT ATTRIBUTE fields in SECS (RCX) *)
DS:RCX.MRENCLAVE ← TMP_MRENCLAVE;
(* MRSIGNER stores a SHA256 in little endian implemented natively on x86 *)
DS:RCX.MRSIGNER ← TMP_MRSIGNER;
DS:RCX.ISVPRODID ← TMP_SIG.ISVPRODID;
DS:RCX.ISVSVN ← TMP_SIG.ISVSVN;
DS:RCX.PADDING ← TMP_SIG_PADDING;

(* Mark the SECS as initialized *)
Update DS:RCX to initialized;

(* Set RAX and ZF for success*)
    RFLAGS.ZF ← 0;
    RAX ← 0;
EXIT:
RFLAGS.CF,PF,AF,OF,SF ← 0;
```

Flags Affected

ZF is cleared if successful, otherwise ZF is set and RAX contains the error code. CF, PF, AF, OF, SF are cleared.

Protected Mode Exceptions

#GP(0)	If a memory operand is not properly aligned. If another instruction is modifying the SECS. If the enclave is already initialized. If the SECS.MRENCLAVE is in use.
#PF(error code)	If a page fault occurs in accessing memory operands. If RCX does not resolve in an EPC page. If the memory address is not a valid, uninitialized SECS.

64-Bit Mode Exceptions

#GP(0)	If a memory operand is not properly aligned. If another instruction is modifying the SECS. If the enclave is already initialized. If the SECS.MRENCLAVE is in use.
#PF(error code)	If a page fault occurs in accessing memory operands. If RCX does not resolve in an EPC page. If the memory address is not a valid, uninitialized SECS.

ELDB/ELDU—Load an EPC page and Marked its State

Opcode/ Instruction	Op/En	64/32 bit Mode Support	CPUID Feature Flag	Description
EAX = 07H ENCLS[ELDB]	IR	V/V	SGX1	This leaf function loads, verifies an EPC page and marks the page as blocked.
EAX = 08H ENCLS[ELDU]	IR	V/V	SGX1	This leaf function loads, verifies an EPC page and marks the page as unblocked.

Instruction Operand Encoding

Op/En	EAX		RBX	RCX	RDX
IR	ELDB/ELDU (In)	Return error code (Out)	Address of the PAGEINFO (In)	Address of the EPC page (In)	Address of the version-array slot (In)

Description

This leaf function copies a page from regular main memory to the EPC. As part of the copying process, the page is cryptographically authenticated and decrypted. This instruction can only be executed when current privilege level is 0.

The ELDB leaf function sets the BLOCK bit in the EPCM entry for the destination page in the EPC after copying. The ELDU leaf function clears the BLOCK bit in the EPCM entry for the destination page in the EPC after copying.

RBX contains the effective address of a PAGEINFO structure; RCX contains the effective address of the destination EPC page; RDX holds the effective address of the version array slot that holds the version of the page.

The table below provides additional information on the memory parameter of ELDB/ELDU leaf functions.

EBLDB/ELDBU Memory Parameter Semantics

PAGEINFO	PAGEINFO.SRCPGE	PAGEINFO.PCMD	PAGEINFO.SECS	EPCPAGE	Version-Array Slot
Non-enclave read access	Non-enclave read access	Non-enclave read access	Enclave read/write access	Read/Write access permitted by Enclave	Read/Write access permitted by Enclave

The error codes are:

Table 40-25. ELDB/ELDU Return Value in RAX

Error Code (see Table 40-3)	Description
No Error	ELDB/ELDU successful
SGX_MAC_COMPARE_FAIL	If the MAC check fails

Concurrency Restrictions

Table 40-26. Concurrency Restrictions of ELDB/ELDU with Intel® SGX Instructions - 1of 2

Operation		EEXIT			EADD		EBLOCK		ECREATE	EDBGRD/WR		EENTER/ERESUME			EEXTEND		EGETKEY		EINIT	ELDB/ELDU			EPA
	Param	Targ	VA	SECS	Targ	SECS	Targ	SECS	SECS	Targ	SECS	TCS	SSA	SECS	Targ	SECS	Param	SECS	SECS	Targ	VA	SECS	VA
ELDB/ELDU	Targ				N		N		N	N		N			N				N	N	N	N	N
	VA				N				N	Y										N	Y		N
	SECS			Y	N	Y		Y	N		Y			Y		Y		Y	Y	N		Y	

Table 40-27. Concurrency Restrictions of ELDB/ELDU with Intel® SGX Instructions - 2 of 2

Operation		EREMOVE		EREPORT		ETRACK	EWB			EAUG		EMODPE		EMODPR		EMODT		EACCEPT			EACCEPTCOPY		
	Param	Targ	SECS	Param	SECS	SECS	SRC	VA	SECS	Targ	SECS	Targ	SECINFO	Targ	SECS	Targ	SECS	Targ	SECINFO	SECS	Targ	SRC	SECINFO
ELDB/ELDU	Targ	N				N	N	N		N	N			N		N							
	VA	N					N	Y		N						N							
	SECS	N	Y		Y	Y			Y	N	Y				Y		Y			Y			

Operation

Temp Variables in ELDB/ELDU Operational Flow

Name	Type	Size (Bits)	Description
TMP_SRCPGE	Memory page	4KBytes	
TMP_SECS	Memory page	4KBytes	
TMP_PCMD	PCMD	128 Bytes	
TMP_HEADER	MACHEADER	128 Bytes	
TMP_VER	UINT64	64	
TMP_MAC	UINT128	128	
TMP_PK	UINT128	128	Page encryption/MAC key.
SCRATCH_PCMD	PCMD	128 Bytes	

```
(* Check PAGEINFO and EPCPAGE alignment *)
IF ( (DS:RBX is not 32Byte Aligned) or (DS:RCX is not 4KByte Aligned) )
    THEN #GP(0); FI;

IF (DS:RCX does not resolve within an EPC)
    THEN #PF(DS:RCX); FI;

(* Check VASLOT alignment *)
IF (DS:RDX is not 8Byte aligned)
    THEN #GP(0); FI;

IF (DS:RDX does not resolve within an EPC)
    THEN #PF(DS:RDX); FI;

TMP_SRCPGE ← DS:RBX.SRCPGE;
TMP_SECS ← DS:RBX.SECS;
TMP_PCMD ← DS:RBX.PCMD;

(* Check alignment of PAGEINFO (RBX)linked parameters. Note: PCMD pointer is overlaid on top of PAGEINFO.SECINFO field *)
IF ( (DS:TMP_PCMD is not 128Byte aligned) or (DS:TMP_SRCPGE is not 4KByte aligned) )
    THEN #GP(0); FI;

(* Check concurrency of EPC and VASLOT by other Intel SGX instructions *)
IF ( (other instructions accessing EPC) or (Other instructions modifying VA slot) )
    THEN #GP(0); FI;

(* Verify EPCM attributes of EPC page, VA, and SECS *)
```

```
IF (EPCM(DS:RCX).VALID = 1)
    THEN #PF(DS:RCX); FI;

IF ( (EPCM(DS:RDX & ~0FFFH).VALID = 0) or (EPCM(DS:RDX & ~0FFFH).PT ≠ PT_VA) )
    THEN #PF(DS:RDX); FI;

(* Copy PCMD into scratch buffer *)
SCRATCH_PCMD[1023: 0]← DS:TMP_PCMD[1023:0];

(* Zero out TMP_HEADER*)
TMP_HEADER[sizeof(TMP_HEADER)-1: 0]← 0;

TMP_HEADER.SECINFO ← SCRATCH_PCMD.SECINFO;
TMP_HEADER.RSVD ← SCRATCH_PCMD.RSVD;
TMP_HEADER.LINADDR ← DS:RBX.LINADDR;

(* Verify various attributes of SECS parameter *)
IF ( (TMP_HEADER.SECINFO.FLAGS.PT = PT_REG) or (TMP_HEADER.SECINFO.FLAGS.PT = PT_TCS) or
     (TMP_HEADER.SECINFO.FLAGS.PT = PT_TRIM) )
    THEN
        IF ( DS:TMP_SECS is not 4KByte aligned)
            THEN #GP(0) FI;
        IF (DS:TMP_SECS does not resolve within an EPC)
            THEN #PF(DS:TMP_SECS) FI;
        IF ( Other instructions modifying SECS)
            THEN #GP(0) FI;
        IF ( (EPCM(DS:TMP_SECS).VALID = 0) or (EPCM(DS:TMP_SECS).PT ≠ PT_SECS) )
            THEN #PF(DS:TMP_SECS) FI;
    ELSIF ( (TMP_HEADER.SECINFO.FLAGS.PT = PT_SECS) or (TMP_HEADER.SECINFO.FLAGS.PT = PT_VA) )
        IF ( ( TMP_SECS ≠ 0) )
            THEN #GP(0) FI;
    ELSE
            #GP(0)
FI;

IF ( (TMP_HEADER.SECINFO.FLAGS.PT = PT_REG) or (TMP_HEADER.SECINFO.FLAGS.PT = PT_TCS) or
     (TMP_HEADER.SECINFO.FLAGS.PT = PT_TRIM) )
    THEN
        TMP_HEADER.EID ← DS:TMP_SECS.EID;
    ELSE
        (* These pages do not have any parent, and hence no EID binding *)
        TMP_HEADER.EID ← 0;
FI;

(* Copy 4KBytes SRCPGE to secure location *)
DS:RCX[32767: 0]← DS:TMP_SRCPGE[32767: 0];
TMP_VER ← DS:RDX[63:0];

(* Decrypt and MAC page. AES_GCM_DEC has 2 outputs, {plain text, MAC} *)
(* Parameters for AES_GCM_DEC {Key, Counter, ..} *)
{DS:RCX, TMP_MAC} ← AES_GCM_DEC(CR_BASE_PK, TMP_VER << 32, TMP_HEADER, 128, DS:RCX, 4096);

IF ( (TMP_MAC ≠ DS:TMP_PCMD.MAC) )
    THEN
```

```
        RFLAGS.ZF ← 1;
        RAX← SGX_MAC_COMPARE_FAIL;
        GOTO ERROR_EXIT;
FI;

(* Check version before committing *)
IF (DS:RDX ≠ 0)
    THEN #GP(0);
    ELSE
        DS:RDX← TMP_VER;
FI;

(* Commit EPCM changes *)
EPCM(DS:RCX).PT ← TMP_HEADER.SECINFO.FLAGS.PT;
EPCM(DS:RCX).RWX ← TMP_HEADER.SECINFO.FLAGS.RWX;
EPCM(DS:RCX).PENDING ← TMP_HEADER.SECINFO.FLAGS.PENDING;
EPCM(DS:RCX).MODIFIED ← TMP_HEADER.SECINFO.FLAGS.MODIFIED;
EPCM(DS:RCX).PR ← TMP_HEADER.SECINFO.FLAGS.PR;
EPCM(DS:RCX).ENCLAVEADDRESS ← TMP_HEADER.LINADDR;

IF ( (EAX = 07H) and (TMP_HEADER.SECINFO.FLAGS.PT is NOT PT_SECS or PT_VA))
    THEN
        EPCM(DS:RCX).BLOCKED ← 1;
    ELSE
        EPCM(DS:RCX).BLOCKED ← 0;
FI;

EPCM(DS:RCX). VALID ← 1;

RAX← 0;
RFLAGS.ZF ← 0;

ERROR_EXIT:
RFLAGS.CF,PF,AF,OF,SF ← 0;
```

Flags Affected

Sets ZF if unsuccessful, otherwise cleared and RAX returns error code. Clears CF, PF, AF, OF, SF.

Protected Mode Exceptions

#GP(0)	If a memory operand effective address is outside the DS segment limit.
	If a memory operand is not properly aligned.
	If the instruction's EPC resource is in use by others.
	If the instruction fails to verify MAC.
	If the version-array slot is in use.
	If the parameters fail consistency checks.
#PF(error code)	If a page fault occurs in accessing memory operands.
	If a memory operand expected to be in EPC does not resolve to an EPC page.
	If one of the EPC memory operands has incorrect page type.
	If the destination EPC page is already valid.

64-Bit Mode Exceptions

#GP(0)	If a memory operand is non-canonical form. If a memory operand is not properly aligned. If the instruction's EPC resource is in use by others. If the instruction fails to verify MAC. If the version-array slot is in use. If the parameters fail consistency checks.
#PF(error code)	If a page fault occurs in accessing memory operands. If a memory operand expected to be in EPC does not resolve to an EPC page. If one of the EPC memory operands has incorrect page type. If the destination EPC page is already valid.

EMODPR—Restrict the Permissions of an EPC Page

Opcode/ Instruction	Op/En	64/32 bit Mode Support	CPUID Feature Flag	Description
EAX = 0EH ENCLS[EMODPR]	IR	V/V	SGX2	This leaf function restricts the access rights associated with a EPC page in an initialized enclave.

Instruction Operand Encoding

Op/En	EAX		RBX	RCX
IR	EMODPR (In)	Return Error Code (Out)	Address of a SECINFO (In)	Address of the destination EPC page (In)

Description

This leaf function restricts the access rights associated with an EPC page in an initialized enclave. THE RWX bits of the SECINFO parameter are treated as a permissions mask; supplying a value that does not restrict the page permissions will have no effect. This instruction can only be executed when current privilege level is 0.

RBX contains the effective address of a SECINFO structure while RCX contains the effective address of an EPC page. The table below provides additional information on the memory parameter of the EMODPR leaf function.

EMODPR Memory Parameter Semantics

SECINFO	EPCPAGE
Read access permitted by Non Enclave	Read/Write access permitted by Enclave

The instruction faults if any of the following:

EMODPR Faulting Conditions

The operands are not properly aligned.	If unsupported security attributes are set.
The Enclave is not initialized.	SECS is locked by another thread.
The EPC page is locked by another thread.	RCX does not contain an effective address of an EPC page in the running enclave.
The EPC page is not valid.	

The error codes are:

Table 40-28. EMODPR Return Value in RAX

Error Code (see Table 40-3)	Description
No Error	EMODPR successful
SGX_PAGE_NOT_MODIFIABLE	The EPC page cannot be modified because it is in the PENDING or MODIFIED state
SGX_LOCKFAIL	Page is being written by EADD, EAUG, ECREATE, ELDU/B, EMODT, or EWB

Concurrency Restrictions

Table 40-29. Concurrency Restrictions of EMODPR with Other Intel® SGX Operations 1 of 2

Operation		EEXIT			EADD		EBLOCK		ECREATE	EDBGRD/WR		EENTER/ERESUME			EEXTEND		EGETKEY		EINIT	ELDB/ELDU			EPA
	Param	TCS	SSA	SECS	Targ	SECS	Targ	SECS	SECS	Targ	SECS	TCS	SSA	SECS	Targ	SECS	Param	SECS	SECS	Targ	VA	SECS	VA
EMODPR	Targ		Y		N		Y		N	Y			Y		N		Y			N			N
	SECS			Y	Y	N	Y	Y	Y	Y	Y	Y	Y	Y	Y	N	Y	Y	N	Y	Y	Y	Y

Table 40-30. Concurrency Restrictions of EMODPR with Other Intel® SGX Operations 2 of 2

Operation		EREMOVE		EREPORT		ETRACK	EWB			EAUG		EMODPE		EMODPR		EMODT		EACCEPT			EACCEPTCOPY		
	Param	Targ	SECS	Param	SECS	SECS	SRC	VA	SECS	Targ	SECS	Targ	SECINFO	Targ	SECS	Targ	SECS	Targ	SECINFO	SECS	Targ	SRC	SECINFO
EMODPR	Targ	N		Y	Y		N			N		C	Y	C		C		C	Y		C	Y	Y
	SECS	Y	Y		Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y

Operation

Temp Variables in EMODPR Operational Flow

Name	Type	Size (bits)	Description
TMP_SECS	Effective Address	32/64	Physical address of SECS to which EPC operand belongs.
SCRATCH_SECINFO	SECINFO	512	Scratch storage for holding the contents of DS:RBX.

```
IF (DS:RBX is not 64Byte Aligned)
    THEN #GP(0); FI;

IF (DS:RCX is not 4KByte Aligned)
    THEN #GP(0); FI;

IF (DS:RCX does not resolve within an EPC)
    THEN #PF(DS:RCX); FI;

SCRATCH_SECINFO ← DS:RBX;

(* Check for mis-configured SECINFO flags*)
IF ( (SCRATCH_SECINFO reserved fields are not zero ) or
    ( (SCRATCH_SECINFO.FLAGS.R is 0 and SCRATCH_SECINFO.FLAGS.W is not 0) )
    THEN #GP(0); FI;

(* Check concurrency with SGX1 or SGX2 instructions on the EPC page *)
IF (SGX1 or other SGX2 instructions accessing EPC page)
    THEN #GP(0); FI;

IF (EPCM(DS:RCX).VALID is 0 )
    THEN #PF(DS:RCX); FI;

(* Check the EPC page for concurrency *)
IF (EPC page in use by another SGX2 instruction)
    THEN
        RFLAGS.ZF ← 1;
        RAX ← SGX_LOCKFAIL;
        GOTO DONE;
FI;

IF ( (EPCM(DS:RCX).PENDING is not 0 or (EPCM(DS:RCX).MODIFIED is not 0) )
    THEN
        RFLAGS.ZF ← 1;
        RAX ← SGX_PAGE_NOT_MODIFIABLE;
        GOTO DONE;
```

```
FI;

IF (EPCM(DS:RCX).PT is not PT_REG)
    THEN #PF(DS:RCX); FI;

TMP_SECS ← GET_SECS_ADDRESS

IF (TMP_SECS.ATTRIBUTES.INIT = 0)
  THEN #GP(0); FI;

(* Set the PR bit to indicate that permission restriction is in progress *)
EPCM(DS:RCX).PR ← 1;

(* Update EPCM permissions *)
EPCM(DS:RCX).R ← EPCM(DS:RCX).R & SCRATCH_SECINFO.FLAGS.R;
EPCM(DS:RCX).W ← EPCM(DS:RCX).W & SCRATCH_SECINFO.FLAGS.W;
EPCM(DS:RCX).X ← EPCM(DS:RCX).X & SCRATCH_SECINFO.FLAGS.X;

RFLAGS.ZF ← 0;
RAX ← 0;

DONE:
RFLAGS.CF,PF,AF,OF,SF ← 0;
```

Flags Affected

Sets ZF if page is not modifiable or if other SGX2 instructions are executing concurrently, otherwise cleared. Clears CF, PF, AF, OF, SF.

Protected Mode Exceptions

#GP(0)	If a memory operand effective address is outside the DS segment limit.
	If a memory operand is not properly aligned.
	If a memory operand is locked.
#PF(error code)	If a page fault occurs in accessing memory operands.
	If a memory operand is not an EPC page.

64-Bit Mode Exceptions

#GP(0)	If a memory operand is non-canonical form.
	If a memory operand is not properly aligned.
	If a memory operand is locked.
#PF(error code)	If a page fault occurs in accessing memory operands.
	If a memory operand is not an EPC page.

EMODT—Change the Type of an EPC Page

Opcode/ Instruction	Op/En	64/32 bit Mode Support	CPUID Feature Flag	Description
EAX = 0FH ENCLS[EMODT]	IR	V/V	SGX2	This leaf function changes the type of an existing EPC page.

Instruction Operand Encoding

Op/En	EAX		RBX	RCX
IR	EMODT (In)	Return Error Code (Out)	Address of a SECINFO (In)	Address of the destination EPC page (In)

Description

This leaf function modifies the type of an EPC page. The security attributes are configured to prevent access to the EPC page at its new type until a corresponding invocation of the EACCEPT leaf confirms the modification. This instruction can only be executed when current privilege level is 0.

RBX contains the effective address of a SECINFO structure while RCX contains the effective address of an EPC page. The table below provides additional information on the memory parameter of the EMODT leaf function.

EMODT Memory Parameter Semantics

SECINFO	EPCPAGE
Read access permitted by Non Enclave	Read/Write access permitted by Enclave

The instruction faults if any of the following:

EMODT Faulting Conditions

The operands are not properly aligned.	If unsupported security attributes are set.
The Enclave is not initialized.	SECS is locked by another thread.
The EPC page is locked by another thread.	RCX does not contain an effective address of an EPC page in the running enclave.
The EPC page is not valid.	

The error codes are:

Table 40-31. EMODT Return Value in RAX

Error Code (see Table 40-3)	Description
No Error	EMODT successful
SGX_PAGE_NOT_MODIFIABLE	The EPC page cannot be modified because it is in the PENDING or MODIFIED state
SGX_LOCKFAIL	Page is being written by EADD, EAUG, ECREATE, ELDU/B, EMODPR, or EWB

Concurrency Restrictions

Table 40-32. Concurrency Restrictions of EMODT with Other Intel® SGX Operations 1 of 2

Operation		EEXIT			EADD		EBLOCK		ECREATE	EDBGRD/WR		EENTER/ERESUME			EEXTEND		EGETKEY		EINIT	ELDB/ELDU			EPA
	Param	TCS	SSA	SECS	Targ	SECS	Targ	SECS	SECS	Targ	SECS	TCS	SSA	SECS	Targ	SECS	Param	SECS	SECS	Targ	VA	SECS	VA
EMODT	Targ	Y	Y		C	C	C		C	C		C	Y		C		Y		N	C	C		C
	SECS			Y	Y	N	Y	Y	Y	Y	Y	Y	Y	Y	Y	N	Y	Y	N	Y	Y	Y	Y

Table 40-33. Concurrency Restrictions of EMODT with Other Intel® SGX Operations 2 of 2

Operation		EREMOVE		EREPORT		ETRACK	EWB			EAUG		EMODPE		EMODPR		EMODT		EACCEPT			EACCEPTCOPY		
	Param	Targ	SECS	Param	SECS	SECS	SRC	VA	SECS	Targ	SECS	Targ	SECINFO	Targ	SECS	Targ	SECS	Targ	SECINFO	SECS	Targ	SRC	SECINFO
EMODT	Targ	C		Y			C	C		C	C	C	Y	C		C	Y	C	Y		C	Y	Y
	SECS	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y		Y	Y	Y	Y	Y	Y	Y

Operation

Temp Variables in EMODT Operational Flow

Name	Type	Size (bits)	Description
TMP_SECS	Effective Address	32/64	Physical address of SECS to which EPC operand belongs.
SCRATCH_SECINFO	SECINFO	512	Scratch storage for holding the contents of DS:RBX.

```
IF (DS:RBX is not 64Byte Aligned)
    THEN #GP(0); FI;

IF (DS:RCX is not 4KByte Aligned)
    THEN #GP(0); FI;

IF (DS:RCX does not resolve within an EPC)
    THEN #PF(DS:RCX); FI;

SCRATCH_SECINFO ← DS:RBX;

(* Check for mis-configured SECINFO flags*)
IF ( (SCRATCH_SECINFO reserved fields are not zero ) or
    !(SCRATCH_SECINFO.FLAGS.PT is PT_TCS or SCRATCH_SECINFO.FLAGS.PT is PT_TRIM) )
    THEN #GP(0); FI;

(* Check concurrency with SGX1 instructions on the EPC page *)
IF (other SGX1 instructions accessing EPC page)
    THEN THEN
        RFLAGS ← 1;
        RAX ← SGX_LOCKFAIL;
        GOTO DONE;
FI;

IF (EPCM(DS:RCX).VALID is 0)
    THEN #PF(DS:RCX); FI;

(* Check the EPC page for concurrency *)
IF (EPC page in use by another SGX2 instruction)
    THEN
        RFLAGS ← 1;
        RAX ← SGX_LOCKFAIL;
        GOTO DONE;
FI;

IF (!(EPCM(DS:RCX).PT is PT_REG or
```

```
    (EPCM(DS:RCX).PT is PT_TCS and SCRATCH_SECINFO.FLAGS.PT is PT_TRIM)))
        THEN #PF(DS:RCX); FI;

IF ( (EPCM(DS:RCX).PENDING is not 0 or (EPCM(DS:RCX).MODIFIED is not 0) )
    THEN
        RFLAGS ← 1;
        RAX ← SGX_PAGE_NOT_MODIFIABLE;
        GOTO DONE;
FI;

TMP_SECS ← GET_SECS_ADDRESS

IF (TMP_SECS.ATTRIBUTES.INIT = 0)
    THEN #GP(0); FI;

(* Update EPCM fields *)
EPCM(DS:RCX).PR ← 0;
EPCM(DS:RCX).MODIFIED ← 1;
EPCM(DS:RCX).R ← 0;
EPCM(DS:RCX).W ← 0;
EPCM(DS:RCX).X ← 0;
EPCM(DS:RCX).PT ← SCRATCH_SECINFO.FLAGS.PT;

RFLAGS.ZF ← 0;
RAX ← 0;

DONE:
RFLAGS.CF,PF,AF,OF,SF ← 0;
```

Flags Affected

Sets ZF if page is not modifiable or if other SGX2 instructions are executing concurrently, otherwise cleared. Clears CF, PF, AF, OF, SF.

Protected Mode Exceptions

#GP(0)	If a memory operand effective address is outside the DS segment limit.
	If a memory operand is not properly aligned.
	If a memory operand is locked.
#PF(error code)	If a page fault occurs in accessing memory operands.
	If a memory operand is not an EPC page.

64-Bit Mode Exceptions

#GP(0)	If a memory operand is non-canonical form.
	If a memory operand is not properly aligned.
	If a memory operand is locked.
#PF(error code)	If a page fault occurs in accessing memory operands.
	If a memory operand is not an EPC page.

EPA—Add Version Array

Opcode/ Instruction	Op/En	64/32 bit Mode Support	CPUID Feature Flag	Description
EAX = 0AH ENCLS[EPA]	IR	V/V	SGX1	This leaf function adds a Version Array to the EPC.

Instruction Operand Encoding

Op/En	EAX	RBX	RCX
IR	EPA (In)	PT_VA (In, Constant)	Effective address of the EPC page (In)

Description

This leaf function creates an empty version array in the EPC page whose logical address is given by DS:RCX, and sets up EPCM attributes for that page. At the time of execution of this instruction, the register RBX must be set to PT_VA.

The table below provides additional information on the memory parameter of EPA leaf function.

EPA Memory Parameter Semantics

EPCPAGE
Write access permitted by Enclave

Concurrency Restrictions

Table 40-34. Concurrency Restrictions of EPA with Other Intel® SGX Operations 1 of 2

Operation		EEXIT			EADD		EBLOCK		ECREATE	EDBGRD/WR		EENTER/ERESUME			EEXTEND		EGETKEY		EINIT	ELDB/ELDU			EPA
	Param	TCS	SSA	SECS	Targ	SECS	Targ	SECS	SECS	Targ	SECS	TCS	SSA	SECS	Targ	SECS	Param	SECS	SECS	Targ	VA	SECS	VA
EPA	VA				N	N	N		N	N		N			N				N	N	N		N

Table 40-35. Concurrency Restrictions of EPA with Other Intel® SGX Operations 2 of 2

Operation		EREMOVE		EREPORT		ETRACK	EWB			EAUG		EMODPE		EMODPR		EMODT		EACCEPT			EACCEPTCOPY		
	Param	Targ	SECS	Param	SECS	SECS	SRC	VA	SECS	Targ	SECS	Targ	SECINFO	Targ	SECS	Targ	SECS	Targ	SECINFO	SECS	Targ	SRC	SECINFO
EPA	VA	N				N	N	N		N	N			N		N							

Operation

```
IF (RBX ≠ PT_VA or DS:RCX is not 4KByte Aligned)
    THEN #GP(0); FI;

IF (DS:RCX does not resolve within an EPC)
    THEN #PF(DS:RCX); FI;

(* Check concurrency with other Intel SGX instructions *)
IF (Other Intel SGX instructions accessing the page)
    THEN #GP(0); FI;
```

```
(* Check EPC page must be empty *)
IF (EPCM(DS:RCX). VALID ≠ 0)
    THEN #PF(DS:RCX); FI;

(* Clears EPC page *)
DS:RCX[32767:0] ← 0;

EPCM(DS:RCX).PT ← PT_VA;
EPCM(DS:RCX).ENCLAVEADDRESS ← 0;
EPCM(DS:RCX).BLOCKED ← 0;
EPCM(DS:RCX).PENDING ← 0;
EPCM(DS:RCX).MODIFIED ← 0;
EPCM(DS:RCX).PR ← 0;
EPCM(DS:RCX).RWX ← 0;
EPCM(DS:RCX).VALID ← 1;
```

Flags Affected

None

Protected Mode Exceptions

#GP(0)	If a memory operand effective address is outside the DS segment limit.
	If a memory operand is not properly aligned.
	If another Intel SGX instruction is accessing the EPC page.
	If RBX is not set to PT_VA.
#PF(error code)	If a page fault occurs in accessing memory operands.
	If a memory operand is not an EPC page.
	If the EPC page is valid.

64-Bit Mode Exceptions

#GP(0)	If a memory operand is non-canonical form.
	If a memory operand is not properly aligned.
	If another Intel SGX instruction is accessing the EPC page.
	If RBX is not set to PT_VA.
#PF(error code)	If a page fault occurs in accessing memory operands.
	If a memory operand is not an EPC page.
	If the EPC page is valid.

EREMOVE—Remove a page from the EPC

Opcode/ Instruction	Op/En	64/32 bit Mode Support	CPUID Feature Flag	Description
EAX = 03H ENCLS[EREMOVE]	IR	V/V	SGX1	This leaf function removes a page from the EPC.

Instruction Operand Encoding

Op/En	EAX	RCX
IR	EREMOVE (In)	Effective address of the EPC page (In)

Description

This leaf function causes an EPC page to be un-associated with its SECS and be marked as unused. This instruction leaf can only be executed when the current privilege level is 0.

The content of RCX is an effective address of an EPC page. The DS segment is used to create linear address. Segment override is not supported.

The instruction fails if the operand is not properly aligned or does not refer to an EPC page or the page is in use by another thread, or other threads are running in the enclave to which the page belongs. In addition the instruction fails if the operand refers to an SECS with associations.

EREMOVE Memory Parameter Semantics

EPCPAGE
Write access permitted by Enclave

The instruction faults if any of the following:

EREMOVE Faulting Conditions

The memory operand is not properly aligned.	The memory operand does not resolve in an EPC page.
Refers to an invalid SECS.	Refers to an EPC page that is locked by another thread.
Another Intel SGX instruction is accessing the EPC page.	RCX does not contain an effective address of an EPC page.
the EPC page refers to an SECS with associations.	

The error codes are:

Table 40-36. EREMOVE Return Value in RAX

Error Code (see Table 40-3)	Description
No Error	EREMOVE successful
SGX_CHILD_PRESENT	If the SECS still have enclave pages loaded into EPC
SGX_ENCLAVE_ACT	If there are still logical processors executing inside the enclave

Concurrency Restrictions

Table 40-37. Concurrency Restrictions of EREMOVE with Other Intel® SGX Operations 1 of 2

Operation		EEXIT			EADD		EBLOCK		ECREATE	EDBGRD/WR		EENTER/ERESUME			EEXTEND		EGETKEY		EINIT	ELDB/ELDU			EPA
	Param	TCS	SSA	SECS	Targ	SECS	Targ	SECS	SECS	Targ	SECS	TCS	SSA	SECS	Targ	SECS	Param	SECS	SECS	Targ	VA	SECS	VA
EREMOVE	Targ	C	C	C	N	N	N	C	N	N	C	N	C	C	N	C	C	C	N	N	N	N	N
	SECS			C	Y	Y	Y	Y	Y	Y	Y	Y	Y	C	Y	Y	Y	C	Y	Y	Y	Y	Y

Table 40-38. Concurrency Restrictions of EREMOVE with Other Intel® SGX Operations 2 of 2

Operation		EREMOVE		EREPORT		ETRACK	EWB			EAUG		EMODPE		EMODPR		EMODT		EACCEPT			EACCEPTCOPY		
	Param	Targ	SECS	Param	SECS	SECS	SRC	VA	SECS	Targ	SECS	Targ	SECINFO	Targ	SECS	Targ	SECS	Targ	SECINFO	SECS	Targ	SRC	SECINFO
EREMOVE	Targ	N	C	C	C	N	N	N	C	N	N	C	C	N	C	N	C	C	C	C	C	C	C
	SECS	Y	Y	Y	C	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	C	C	C	Y	Y	Y

Operation

Temp Variables in EREMOVE Operational Flow

Name	Type	Size (Bits)	Description
TMP_SECS	Effective Address	32/64	Effective address of the SECS destination page.

```
IF (DS:RCX is not 4KByte Aligned)
    THEN #GP(0); FI;

IF (DS:RCX does not resolve to an EPC page)
    THEN #PF(DS:RCX); FI;

TMP_SECS ← Get_SECS_ADDRESS();

(* Check the EPC page for concurrency *)
IF (EPC page being referenced by another Intel SGX instruction)
    THEN #GP(0); FI;

(* if DS:RCX is already unused, nothing to do*)
IF ( (EPCM(DS:RCX).VALID = 0) or (EPCM(DS:RCX).PT = PT_TRIM AND EPCM(DS:RCX).MODIFIED = 0))
    THEN GOTO DONE;
FI;

IF ( (EPCM(DS:RCX).PT = PT_VA) OR
    ((EPCM(DS:RCX).PT = PT_TRIM) AND (EPCM(DS:RCX).MODIFIED = 0)) )
    THEN
        EPCM(DS:RCX).VALID ← 0;
        GOTO DONE;
FI;

IF (EPCM(DS:RCX).PT = PT_SECS)
    THEN
        IF (DS:RCX has an EPC page associated with it)
            THEN
```

```
                RFLAGS.ZF ← 1;
                RAX← SGX_CHILD_PRESENT;
                GOTO ERROR_EXIT;
        FI;
        EPCM(DS:RCX).VALID ← 0;
        GOTO DONE;
FI;

IF (Other threads active using SECS)
    THEN
        RFLAGS.ZF ← 1;
        RAX← SGX_ENCLAVE_ACT;
        GOTO ERROR_EXIT;
FI;

IF ( (EPCM(DS:RCX).PT is PT_REG) or (EPCM(DS:RCX).PT is PT_TCS) or (EPCM(DS:RCX).PT is PT_TRIM) )
    THEN
        EPCM(DS:RCX).VALID ← 0;
        GOTO DONE;
FI;

DONE:
RAX← 0;
RFLAGS.ZF ← 0;

ERROR_EXIT:
RFLAGS.CF,PF,AF,OF,SF ← 0;
```

Flags Affected

Sets ZF if unsuccessful, otherwise cleared and RAX returns error code. Clears CF, PF, AF, OF, SF

Protected Mode Exceptions

#GP(0)	If a memory operand effective address is outside the DS segment limit.
	If a memory operand is not properly aligned.
	If another Intel SGX instruction is accessing the page.
#PF(error code)	If a page fault occurs in accessing memory operands.
	If the memory operand is not an EPC page.

64-Bit Mode Exceptions

#GP(0)	If the memory operand is non-canonical form.
	If a memory operand is not properly aligned.
	If another Intel SGX instruction is accessing the page.
#PF(error code)	If a page fault occurs in accessing memory operands.
	If the memory operand is not an EPC page.

ETRACK—Activates EBLOCK Checks

Opcode/ Instruction	Op/En	64/32 bit Mode Support	CPUID Feature Flag	Description
EAX = 0CH ENCLS[ETRACK]	IR	V/V	SGX1	This leaf function activates EBLOCK checks.

Instruction Operand Encoding

Op/En	EAX		RCX
IR	ETRACK (In)	Return error code (Out)	Pointer to the SECS of the EPC page (In)

Description

This leaf function provides the mechanism for hardware to track that software has completed the required TLB address clears successfully. The instruction can only be executed when the current privilege level is 0.

The content of RCX is an effective address of an EPC page.

The table below provides additional information on the memory parameter of ETRACK leaf function.

ETRACK Memory Parameter Semantics

EPCPAGE
Read/Write access permitted by Enclave

The error codes are:

Table 40-39. ETRACK Return Value in RAX

Error Code (see Table 40-3)	Description
No Error	ETRACK successful
SGX_PREV_TRK_INCMPL	All processors did not complete the previous shoot-down sequence

Concurrency Restrictions

Table 40-40. Concurrency Restrictions of ETRACK with Other Intel® SGX Operations 1 of 2

Operation		EEXIT			EADD		EBLOCK		ECREATE	EDBGRD/WR		EENTER/ ERESUME			EEXTEND		EGETKEY		EINIT	ELDB/ELDU			EPA
	Param	TCS	SSA	SECS	Targ	SECS	Targ	SECS	SECS	Targ	SECS	TCS	SSA	SECS	Targ	SECS	Param	SECS	SECS	Targ	VA	SECS	VA
ETRACK	SECS			Y	N	Y		Y	N	N	Y			Y		Y		Y	Y	N		Y	N

Table 40-41. Concurrency Restrictions of ETRACK with Other Intel® SGX Operations 2 of 2

Operation		EREMOVE		EREPORT		ETRACK	EWB			EAUG		EMODPE		EMODPR		EMODT		EACCEPT			EACCEPTCOPY		
	Param	Targ	SECS	Param	SECS	SECS	SRC	VA	SECS	Targ	SECS	Targ	SECINFO	Targ	SECS	Targ	SECS	Targ	SECINFO	SECS	Targ	SRC	SECINFO
ETRACK	SECS	N	Y		Y	N	N		Y	N	Y				Y		Y			Y			

Operation

```
IF (DS:RCX is not 4KByte Aligned)
    THEN #GP(0); FI;

IF (DS:RCX does not resolve within an EPC)
    THEN #PF(DS:RCX); FI;
```

```
(* Check concurrency with other Intel SGX instructions *)
IF (Other Intel SGX instructions using tracking facility on this SECS)
    THEN #GP(0); FI;

IF (EPCM(DS:RCX). VALID = 0)
    THEN #PF(DS:RCX); FI;

IF (EPCM(DS:RCX).PT ≠ PT_SECS)
    THEN #PF(DS:RCX); FI;

(* All processors must have completed the previous tracking cycle*)
IF ( (DS:RCX).TRACKING ≠ 0) )
    THEN
        RFLAGS.ZF ← 1;
        RAX← SGX_PREV_TRK_INCMPL;
        GOTO DONE;
    ELSE
        RAX← 0;
        RFLAGS.ZF ← 0;
FI;

DONE:
RFLAGS.CF,PF,AF,OF,SF ← 0;
```

Flags Affected

Sets ZF if SECS is in use or invalid, otherwise cleared. Clears CF, PF, AF, OF, SF.

Protected Mode Exceptions

#GP(0)	If a memory operand effective address is outside the DS segment limit.
	If a memory operand is not properly aligned.
	If another thread is concurrently using the tracking facility on this SECS.
#PF(error code)	If a page fault occurs in accessing memory operands.
	If a memory operand is not an EPC page.

64-Bit Mode Exceptions

#GP(0)	If a memory operand is non-canonical form.
	If a memory operand is not properly aligned.
	If the specified EPC resource is in use.
#PF(error code)	If a page fault occurs in accessing memory operands.
	If a memory operand is not an EPC page.

Table 40-42. ETRACK Return Value in RAX

Error Code (see Table 40-3)	Value	Description
No Error	0	ETRACK successful
SGX_PREV_TRK_INCMPL		All processors did not complete the previous shoot-down sequence

EWB—Invalidate an EPC Page and Write out to Main Memory

Opcode/ Instruction	Op/En	64/32 bit Mode Support	CPUID Feature Flag	Description
EAX = 0BH ENCLS[EWB]	IR	V/V	SGX1	This leaf function invalidates an EPC page and writes it out to main memory.

Instruction Operand Encoding

Op/En	EAX		RBX	RCX	RDX
IR	EWB (In)	Error code (Out)	Address of an PAGEINFO (In)	Address of the EPC page (In)	Address of a VA slot (In)

Description

This leaf function copies a page from the EPC to regular main memory. As part of the copying process, the page is cryptographically protected. This instruction can only be executed when current privilege level is 0.

The table below provides additional information on the memory parameter of EPA leaf function.

EWB Memory Parameter Semantics

PAGEINFO	PAGEINFO.SRCPGE	PAGEINFO.PCMD	EPCPAGE	VASLOT
Non-EPC R/W access	Non-EPC R/W access	Non-EPC R/W access	EPC R/W access	EPC R/W access

The error codes are:

Table 40-43. EWB Return Value in RAX

Error Code (see Table 40-3)	Description
No Error	EWB successful
SGX_PAGE_NOT_BLOCKED	If page is not marked as blocked
SGX_NOT_TRACKED	If EWB is racing with ETRACK instruction
SGX_VA_SLOT_OCCUPIED	Version array slot contained valid entry
SGX_CHILD_PRESENT	Child page present while attempting to page out enclave

Concurrency Restrictions

Table 40-44. Concurrency Restrictions of EWB with Intel® SGX Instructions - 1of 2

Operation		EEXIT			EADD		EBLOCK		ECREATE	EDBGRD/WR		EENTER/ERESUME			EEXTEND		EGETKEY		EINIT	ELDB/ELDU			EPA
	Param	Targ	VA	SECS	Targ	SECS	Targ	SECS	SECS	Targ	SECS	TCS	SSA	SECS	Targ	SECS	Param	SECS	SECS	Targ	VA	SECS	VA
EWB	Src	C	C	C	N	N	N	C	N	N	C	N	C	C	N	C	C	C	N	N	N		N
	VA				N				N	Y										N	Y		N
	SECS			Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y

Table 40-45. Concurrency Restrictions of EWB with Intel® SGX Instructions - 2 of 2

Operation		EREMOVE		EREPORT		ETRACK	EWB			EAUG		EMODPE		EMODPR		EMODT		EACCEPT			EACCEPTCOPY		
	Param	Targ	SECS	Param	SECS	SECS	SRC	VA	SECS	Targ	SECS	Targ	SECINFO	Targ	SECS	Targ	SECS	Targ	SECINFO	SECS	Targ	SRC	SECINFO

Table 40-45. Concurrency Restrictions of EWB with Intel® SGX Instructions - 2 of 2

Operation		EREMOVE		EREPORT		ETRACK	EWB			EAUG		EMODPE		EMODPR		EMODT		EACCEPT			EACCEPTCOPY		
EWB	Src	N	C	C	C	N	N	N	C	N	N	C	C	N	C	N	C	C	C	C	C	C	C
	VA	N					N	Y		N						N							
	SECS	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y

Operation

Temp Variables in EWB Operational Flow

Name	Type	Size (Bytes)	Description
TMP_SRCPGE	Memory page	4096	
TMP_PCMD	PCMD	128	
TMP_SECS	SECS	4096	
TMP_BPEPOCH	UINT64	8	
TMP_BPREFCOUNT	UINT64	8	
TMP_HEADER	MAC Header	128	
TMP_PCMD_ENCLAVEID	UINT64	8	
TMP_VER	UINT64	8	
TMP_PK	UINT128	16	

```
IF ( (DS:RBX is not 32Byte Aligned) or (DS:RCX is not 4KByte Aligned) )
    THEN #GP(0); FI;

IF (DS:RCX does not resolve within an EPC)
    THEN #PF(DS:RCX); FI;

IF (DS:RDX is not 8Byte Aligned)
    THEN #GP(0); FI;

IF (DS:RDX does not resolve within an EPC)
    THEN #PF(DS:RDX); FI;

(* EPCPAGE and VASLOT should not resolve to the same EPC page*)
IF (DS:RCX and DS:RDX resolve to the same EPC page)
    THEN #GP(0); FI;

TMP_SRCPGE ← DS:RBX.SRCPGE;
(* Note PAGEINFO.PCMD is overlaid on top of PAGEINFO.SECINFO *)
TMP_PCMD ← DS:RBX.PCMD;

If (DS:RBX.LINADDR ≠ 0) OR (DS:RBX.SECS ≠ 0)
    THEN #GP(0); FI;

IF ( (DS:TMP_PCMD is not 128Byte Aligned) or (DSTMP_SRCPGE is not 4KByte Aligned) )
    THEN #GP(0); FI;

(* Check for concurrent Intel SGX instruction access to the page *)
IF (Other Intel SGX instruction is accessing page)
```

```
    THEN #GP(0); FI;

(*Check if the VA Page is being removed or changed*)
IF (VA Page is being modified)
    THEN #GP(0); FI;

(* Verify that EPCPAGE and VASLOT page are valid EPC pages and DS:RDX is VA *)
IF (EPCM(DS:RCX).VALID = 0)
    THEN #PF(DS:RCX); FI;

IF ( (EPCM(DS:RDX & ~0FFFH).VALID = 0) or (EPCM(DS:RDX & ~FFFH).PT is not PT_VA) )
    THEN #PF(DS:RDX); FI;

(* Perform page-type-specific exception checks *)
IF ( (EPCM(DS:RCX).PT is PT_REG) or (EPCM(DS:RCX).PT is PT_TCS) or (EPCM(DS:RCX).PT is PT_TRIM ) )
    THEN
        TMP_SECS = Obtain SECS through EPCM(DS:RCX)
    (* Check that EBLOCK has occurred correctly *)
    IF (EBLOCK is not correct)
        THEN #GP(0); FI;
FI;

RFLAGS.ZF,CF,PF,AF,OF,SF ← 0;
RAX ← 0;

(* Perform page-type-specific checks *)
IF ( (EPCM(DS:RCX).PT is PT_REG) or (EPCM(DS:RCX).PT is PT_TCS) or (EPCM(DS:RCX).PT is PT_TRIM ))
    THEN
        (* check to see if the page is evictable *)
        IF (EPCM(DS:RCX).BLOCKED = 0)
            THEN
                RAX ← SGX_PAGE NOT_BLOCKED;
                RFLAGS.ZF ← 1;
                GOTO ERROR_EXIT;
        FI;
        (* Check if tracking done correctly *)
        IF (Tracking not correct)
            THEN
                RAX ← SGX_NOT_TRACKED;
                RFLAGS.ZF ← 1;
                GOTO ERROR_EXIT;
        FI;

        (* Obtain EID to establish cryptographic binding between the paged-out page and the enclave *)
        TMP_HEADER.EID ← TMP_SECS.EID;

        (* Obtain EID as an enclave handle for software *)
        TMP_PCMD_ENCLAVEID ← TMP_SECS.EID;
    ELSE IF (EPCM(DS:RCX).PT is PT_SECS)
        (*check that there are no child pages inside the enclave *)
        IF (DS:RCX has an EPC page associated with it)
            THEN
                RAX ← SGX_CHILD_PRESENT;
                RFLAGS.ZF ← 1;
```

```
                GOTO ERROR_EXIT;
        FI:
        TMP_HEADER.EID ← 0;
        (* Obtain EID as an enclave handle for software *)
        TMP_PCMD_ENCLAVEID ← (DS:RCX).EID;
    ELSE IF (EPCM(DS:RCX).PT is PT_VA)
        TMP_HEADER.EID ← 0; // Zero is not a special value
        (* No enclave handle for VA pages*)
        TMP_PCMD_ENCLAVEID ← 0;
FI;

(* Zero out TMP_HEADER*)
TMP_HEADER[ sizeof(TMP_HEADER)-1 : 0] ← 0;

TMP_HEADER.LINADDR ← EPCM(DS:RCX).ENCLAVEADDRESS;
TMP_HEADER.SECINFO.FLAGS.PT ← EPCM(DS:RCX).PT;
TMP_HEADER.SECINFO.FLAGS.RWX ← EPCM(DS:RCX).RWX;
TMP_HEADER.SECINFO.FLAGS.PENDING ← EPCM(DS:RCX).PENDING;
TMP_HEADER.SECINFO.FLAGS.MODIFIED ← EPCM(DS:RCX).MODIFIED;
TMP_HEADER.SECINFO.FLAGS.PR ← EPCM(DS:RCX).PR;

(* Encrypt the page, DS:RCX could be encrypted in place. AES-GCM produces 2 values, {ciphertext, MAC}. *)
(* AES-GCM input parameters: key, GCM Counter, MAC_HDR, MAC_HDR_SIZE, SRC, SRC_SIZE)*)
{DS:TMP_SRCPGE, DS:TMP_PCMD.MAC} ← AES_GCM_ENC(CR_BASE_PK), (TMP_VER << 32),
    TMP_HEADER, 128, DS:RCX, 4096);

(* Write the output *)
Zero out DS:TMP_PCMD.SECINFO
DS:TMP_PCMD.SECINFO.FLAGS.PT ← EPCM(DS:RCX).PT;
DS:TMP_PCMD.SECINFO.FLAGS.RWX ← EPCM(DS:RCX).RWX;
DS:TMP_PCMD.SECINFO.FLAGS.PENDING ← EPCM(DS:RCX).PENDING;
DS:TMP_PCMD.SECINFO.FLAGS.MODIFIED ← EPCM(DS:RCX).MODIFIED;
DS:TMP_PCMD.SECINFO.FLAGS.PR ← EPCM(DS:RCX).PR;
DS:TMP_PCMD.RESERVED ← 0;
DS:TMP_PCMD.ENCLAVEID ← TMP_PCMD_ENCLAVEID;
DS:RBX.LINADDR ← EPCM(DS:RCX).ENCLAVEADDRESS;

(*Check if version array slot was empty *)
IF ([DS.RDX])
    THEN
        RAX ← SGX_VA_SLOT_OCCUPIED
        RFLAGS.CF ← 1;
FI;

(* Write version to Version Array slot *)
[DS.RDX] ← TMP_VER;

(* Free up EPCM Entry *)
EPCM.(DS:RCX).VALID ← 0;
ERROR_EXIT:
```

Flags Affected

ZF is set if page is not blocked, not tracked, or a child is present. Otherwise cleared.

CF is set if VA slot is previously occupied, Otherwise cleared.

Protected Mode Exceptions

#GP(0)	If a memory operand effective address is outside the DS segment limit.
	If a memory operand is not properly aligned.
	If the EPC page and VASLOT resolve to the same EPC page.
	If another Intel SGX instruction is concurrently accessing either the target EPC, VA, or SECS pages.
	If the tracking resource is in use.
	If the EPC page or the version array page is invalid.
	If the parameters fail consistency checks.
#PF(error code)	If a page fault occurs in accessing memory operands.
	If a memory operand is not an EPC page.
	If one of the EPC memory operands has incorrect page type.

64-Bit Mode Exceptions

#GP(0)	If a memory operand is non-canonical form.
	If a memory operand is not properly aligned.
	If the EPC page and VASLOT resolve to the same EPC page.
	If another Intel SGX instruction is concurrently accessing either the target EPC, VA, or SECS pages.
	If the tracking resource is in use.
	If the EPC page or the version array page in invalid.
	If the parameters fail consistency checks.
#PF(error code)	If a page fault occurs in accessing memory operands.
	If a memory operand is not an EPC page.
	If one of the EPC memory operands has incorrect page type.

40.4 INTEL® SGX USER LEAF FUNCTION REFERENCE

40.4.1 Instruction Column in the Instruction Summary Table

Leaf functions available with the ENCLU instruction mnemonic are covered in this section. In general, each instruction leaf requires EAX to specify the leaf function index and/or additional registers specifying leaf-specific input parameters. An instruction operand encoding table provides details of the implicitly-encoded register usage and associated input/output semantics.

In many cases, an input parameter specifies an effective address associated with a memory object inside or outside the EPC, the memory addressing semantics of these memory objects are also summarized in a separate table.

EACCEPT—Accept Changes to an EPC Page

Opcode/ Instruction	Op/En	64/32 bit Mode Support	CPUID Feature Flag	Description
EAX = 05H ENCLU[EACCEPT]	IR	V/V	SGX2	This leaf function accepts changes made by system software to an EPC page in the running enclave.

Instruction Operand Encoding

Op/En	EAX		RBX	RCX
IR	EACCEPT (In)	Return Error Code (Out)	Address of a SECINFO (In)	Address of the destination EPC page (In)

Description

This leaf function accepts changes to a page in the running enclave by verifying that the security attributes specified in the SECINFO match the security attributes of the page in the EPCM. This instruction leaf can only be executed when inside the enclave.

RBX contains the effective address of a SECINFO structure while RCX contains the effective address of an EPC page. The table below provides additional information on the memory parameter of the EACCEPT leaf function.

EACCEPT Memory Parameter Semantics

SECINFO	EPCPAGE (Destination)
Read access permitted by Non Enclave	Read access permitted by Enclave

The instruction faults if any of the following:

EACCEPT Faulting Conditions

The operands are not properly aligned.	If security attributes of the SECINFO page make the page inaccessible.
The EPC page is locked by another thread.	RBX does not contain an effective address in an EPC page in the running enclave.
The EPC page is not valid.	RCX does not contain an effective address of an EPC page in the running enclave.
SECINFO contains an invalid request.	Page type is PT_REG and MODIFIED bit is 0.
	Page type is PT_TCS or PT_TRIM and PENDING bit is 0 and MODIFIED bit is 1.

The error codes are:

Table 40-46. EACCEPT Return Value in RAX

Error Code (see Table 40-3)	Description
No Error	EACCEPT successful
SGX_PAGE_ATTRIBUTES_MISMATCH	The attributes of the target EPC page do not match the expected values
SGX_NOT_TRACKED	The OS did not complete an ETRACK on the target page

Concurrency Restrictions

Table 40-47. Concurrency Restrictions of EACCEPT with Intel® SGX Instructions - 1of 2

Operation		EEXIT			EADD		EBLOCK		ECREATE	EDBGRD/WR		EENTER/ERESUME			EEXTEND		EGETKEY		EINIT	ELDB/ELDU			EPA
	Param	Targ	VA	SECS	Targ	SECS	Targ	SECS	SECS	Targ	SECS	TCS	SSA	SECS	Targ	SECS	Param	SECS	SECS	Targ	VA	SECS	VA

Table 40-47. Concurrency Restrictions of EACCEPT with Intel® SGX Instructions - 1of 2

Operation		EEXIT			EADD		EBLOCK		ECREATE	EDBGRD/WR		EENTER/ERESUME			EEXTEND		EGETKEY		EINIT	ELDB/ELDU			EPA
EACCEPT	Targ	C	Y							Y		C	Y				Y						
	SECINFO		U							Y			U				U						
	SECS			Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y

Table 40-48. Concurrency Restrictions of EACCEPT with Intel® SGX Instructions - 2 of 2

Operation		EREMOVE		EREPORT		ETRACK	EWB			EAUG		EMODPE		EMODPR		EMODT		EACCEPT			EACCEPTCOPY		
	Param	Targ	SECS	Param	SECS	SECS	SRC	VA	SECS	Targ	SECS	Targ	SECINFO	Targ	SECS	Targ	SECS	Targ	SECINFO	SECS	Targ	SRC	SECINFO
EACCEPT	Targ			Y								N	Y	N		N		N	Y		N	Y	C
	SECINFO			U								Y	Y					Y	Y			U	Y
	SECS	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y

Operation

Temp Variables in EACCEPT Operational Flow

Name	Type	Size (bits)	Description
TMP_SECS	Effective Address	32/64	Physical address of SECS to which EPC operands belongs.
SCRATCH_SECINFO	SECINFO	512	Scratch storage for holding the contents of DS:RBX.

```
IF (DS:RBX is not 64Byte Aligned)
    THEN #GP(0); FI;

IF (DS:RBX is not within CR_ELRANGE)
    THEN #GP(0); FI;

IF (DS:RBX does not resolve within an EPC)
    THEN #PF(DS:RBX); FI;

IF ( (EPCM(DS:RBX &~FFFH).VALID = 0) or (EPCM(DS:RBX &~FFFH).R = 0) or (EPCM(DS:RBX &~FFFH).PENDING ≠ 0) or
    (EPCM(DS:RBX &~FFFH).MODIFIED ≠ 0) or (EPCM(DS:RBX &~FFFH).BLOCKED ≠ 0) or
    (EPCM(DS:RBX &~FFFH).PT ≠ PT_REG) or (EPCM(DS:RBX &~FFFH).ENCLAVESECS ≠ CR_ACTIVE_SECS) or
    (EPCM(DS:RBX &~FFFH).ENCLAVEADDRESS ≠ (DS:RBX & FFFH)) )
    THEN #PF(DS:RBX); FI;

(* Copy 64 bytes of contents *)
SCRATCH_SECINFO ← DS:RBX;

(* Check for mis-configured SECINFO flags*)
IF (SCRATCH_SECINFO reserved fields are not zero ) )
    THEN #GP(0); FI;

IF (DS:RCX is not 4KByte Aligned)
    THEN #GP(0); FI;

IF (DS:RCX is not within CR_ELRANGE)
    THEN #GP(0); FI;
```

```
IF (DS:RCX does not resolve within an EPC)
    THEN #PF(DS:RCX); FI;

(* Check that the combination of requested PT, PENDING and MODIFIED is legal *)
IF (NOT (((SCRATCH_SECINFO.FLAGS.PT is PT_REG) and
    ((SCRATCH_SECINFO.FLAGS.PR is 1) or
    (SCRATCH_SECINFO.FLAGS.PENDING is 1)) and
    (SCRATCH_SECINFO.FLAGS.MODIFIED is 0)) or
    ((SCRATCH_SECINFO.FLAGS.PT is PT_TCS or PT_TRIM) and
    (SCRATCH_SECINFO.FLAGS.PR is 0) and
    (SCRATCH_SECINFO.FLAGS.PENDING is 0) and
    (SCRATCH_SECINFO.FLAGS.MODIFIED is 1) )))
        THEN #GP(0); FI

(* Check security attributes of the destination EPC page *)
If ( (EPCM(DS:RCX).VALID is 0) or (EPCM(DS:RCX).BLOCKED is not 0) or
    ((EPCM(DS:RCX).PT is not PT_REG) and (EPCM(DS:RCX).PT is not PT_TCS) and (EPCM(DS:RCX).PT is not PT_TRIM)) or
    (EPCM(DS:RCX).ENCLAVESECS ≠ CR_ACTIVE_SECS))
    THEN #PF((DS:RCX); FI;

(* Check the destination EPC page for concurrency *)
IF ( EPC page in use )
    THEN #GP(0); FI;

(* Re-Check security attributes of the destination EPC page *)
IF ( (EPCM(DS:RCX).VALID is 0) or (EPCM(DS:RCX).ENCLAVESECS ≠ CR_ACTIVE_SECS) )
    THEN #PF(DS:RCX); FI;

(* Verify that accept request matches current EPC page settings *)
IF ( (EPCM(DS:RCX).ENCLAVEADDRESS ≠ DS:RCX) or (EPCM(DS:RCX).PENDING ≠ SCRATCH_SECINFO.FLAGS.PENDING) or
    (EPCM(DS:RCX).MODIFIED ≠ SCRATCH_SECINFO.FLAGS.MODIFIED) or (EPCM(DS:RCX).R ≠ SCRATCH_SECINFO.FLAGS.R) or
    (EPCM(DS:RCX).W ≠ SCRATCH_SECINFO.FLAGS.W) or (EPCM(DS:RCX).X ≠ SCRATCH_SECINFO.FLAGS.X) or
    (EPCM(DS:RCX).PT ≠ SCRATCH_SECINFO.FLAGS.PT) )
    THEN
        RFLAGS.ZF ← 1;
        RAX ← SGX_PAGE_ATTRIBUTES_MISMATCH;
        GOTO DONE;
FI;
(* Check that all required threads have left enclave *)
IF (Tracking not correct)
    THEN
        RFLAGS.ZF ← 1;
        RAX ← SGX_NOT_TRACKED;
        GOTO DONE;
FI;

(* Get pointer to the SECS to which the EPC page belongs *)
TMP_SECS = << Obtain physical address of SECS through EPCM(DS:RCX)>>
(* For TCS pages, perform additional checks *)
IF (SCRATCH_SECINFO.FLAGS.PT = PT_TCS)
    THEN
        IF (DS:RCX.RESERVED ≠ 0) #GP(0); FI;
FI;
```

```
(* Check that TCS.FLAGS.DBGOPTIN, TCS stack, and TCS status are correctly initialized *)
IF ( ((DS:RCX).FLAGS.DBGOPTIN is not 0) or ((DS:RCX).CSSA ≥ (DS:RCX).NSSA) or ((DS:RCX).AEP is not 0) or ((DS:RCX).STATE is not 0)
    THEN #GP(0); FI;

(* Check consistency of FS & GS Limit *)
IF ( (TMP_SECS.ATTRIBUTES.MODE64BIT is 0) and ((DS:RCX.FSLIMIT & FFFH ≠ FFFH) or (DS:RCX.GSLIMIT & FFFH ≠ FFFH)) )
    THEN #GP(0); FI;

(* Clear PENDING/MODIFIED flags to mark accept operation complete *)
EPCM(DS:RCX).PENDING ← 0;
EPCM(DS:RCX).MODIFIED ← 0;
EPCM(DS:RCX).PR ← 0;

(* Clear EAX and ZF to indicate successful completion *)
RFLAGS.ZF ← 0;
RAX ← 0;

DONE:
RFLAGS.CF,PF,AF,OF,SF ← 0;
```

Flags Affected

Sets ZF if page cannot be accepted, otherwise cleared. Clears CF, PF, AF, OF, SF

Protected Mode Exceptions

#GP(0)	If a memory operand effective address is outside the DS segment limit.
	If a memory operand is not properly aligned.
	If a memory operand is locked.
#PF(error code)	If a page fault occurs in accessing memory operands.
	If a memory operand is not an EPC page.
	If EPC page has incorrect page type or security attributes.

64-Bit Mode Exceptions

#GP(0)	If a memory operand is non-canonical form.
	If a memory operand is not properly aligned.
	If a memory operand is locked.
#PF(error code)	If a page fault occurs in accessing memory operands.
	If a memory operand is not an EPC page.
	If EPC page has incorrect page type or security attributes.

EACCEPTCOPY—Initialize a Pending Page

Opcode/ Instruction	Op/En	64/32 bit Mode Support	CPUID Feature Flag	Description
EAX = 07H ENCLU[EACCEPTCOPY]	IR	V/V	SGX2	This leaf function initializes a dynamically allocated EPC page from another page in the EPC.

Instruction Operand Encoding

Op/En	EAX		RBX	RCX	RDX
IR	EACCEPTCOPY (In)	Return Error Code (Out)	Address of a SECINFO (In)	Address of the destination EPC page (In)	Address of the source EPC page (In)

Description

This leaf function copies the contents of an existing EPC page into an uninitialized EPC page (created by EAUG). After initialization, the instruction may also modify the access rights associated with the destination EPC page. This instruction leaf can only be executed when inside the enclave.

RBX contains the effective address of a SECINFO structure while RCX and RDX each contain the effective address of an EPC page. The table below provides additional information on the memory parameter of the EACCEPTCOPY leaf function.

EACCEPTCOPY Memory Parameter Semantics

SECINFO	EPCPAGE (Destination)	EPCPAGE (Source)
Read access permitted by Non Enclave	Read/Write access permitted by Enclave	Read access permitted by Enclave

The instruction faults if any of the following:

EACCEPTCOPY Faulting Conditions

The operands are not properly aligned.	If security attributes of the SECINFO page make the page inaccessible.
The EPC page is locked by another thread.	If security attributes of the source EPC page make the page inaccessible.
The EPC page is not valid.	RBX does not contain an effective address in an EPC page in the running enclave.
SECINFO contains an invalid request.	RCX/RDX does not contain an effective address of an EPC page in the running enclave.

The error codes are:

Table 40-49. EACCEPTCOPY Return Value in RAX

Error Code (see Table 40-3)	Description
No Error	EACCEPTCOPY successful
SGX_PAGE_ATTRIBUTES_MISMATCH	The attributes of the target EPC page do not match the expected values

Concurrency Restrictions

Table 40-50. Concurrency Restrictions of EACCEPTCOPY with Intel® SGX Instructions - 1of 2

Operation		EEXIT			EADD		EBLOCK		ECREATE	EDBGRD/WR		EENTER/ERESUME			EEXTEND		EGETKEY		EINIT	ELDB/ELDU			EPA
	Param	Targ	VA	SECS	Targ	SECS	Targ	SECS	SECS	Targ	SECS	TCS	SSA	SECS	Targ	SECS	Param	SECS	SECS	Targ	VA	SECS	VA
EACCEPTCOPY	Targ																						
	Src		U							Y			U				Y						
	SECINFO		U							Y			U				U						

Table 40-51. Concurrency Restrictions of EACCEPTCOPY with Intel® SGX Instructions - 2 of 2

Operation		EREMOVE		EREPORT		ETRACK	EWB			EAUG		EMODPE		EMODPR		EMODT		EACCEPT			EACCEPTCOPY		
	Param	Targ	SECS	Param	SECS	SECS	SRC	VA	SECS	Targ	SECS	Targ	SECINFO	Targ	SECS	Targ	SECS	Targ	SECINFO	SECS	Targ	SRC	SECINFO
EACCEPTCOPY	Targ													N		N		N			N		
	Src			Y								Y	Y					Y	U			Y	Y
	SECINFO			U								Y	Y					Y	Y			Y	Y

Operation

Temp Variables in EACCEPTCOPY Operational Flow

Name	Type	Size (bits)	Description
SCRATCH_SECINFO	SECINFO	512	Scratch storage for holding the contents of DS:RBX.

```
IF (DS:RBX is not 64Byte Aligned)
    THEN #GP(0); FI;

IF ( (DS:RCX is not 4KByte Aligned) or (DS:RDX is not 4KByte Aligned) )
    THEN #GP(0); FI;

IF ((DS:RBX is not within CR_ELRANGE) or (DS:RCX is not within CR_ELRANGE) or (DS:RDX is not within CR_ELRANGE))
    THEN #GP(0); FI;

IF (DS:RBX does not resolve within an EPC)
    THEN #PF(DS:RBX); FI;

IF (DS:RCX does not resolve within an EPC)
    THEN #PF(DS:RCX); FI;

IF (DS:RDX does not resolve within an EPC)
    THEN #PF(DS:RDX); FI;

IF ( (EPCM(DS:RBX &~FFFH).VALID = 0) or (EPCM(DS:RBX &~FFFH).R = 0) or (EPCM(DS:RBX &~FFFH).PENDING ≠ 0) or
    (EPCM(DS:RBX &~FFFH).MODIFIED ≠ 0) or (EPCM(DS:RBX &~FFFH).BLOCKED ≠ 0) or (EPCM(DS:RBX &~FFFH).PT ≠ PT_REG) or
    (EPCM(DS:RBX &~FFFH).ENCLAVESECS ≠ CR_ACTIVE_SECS) or
    (EPCM(DS:RBX &~FFFH).ENCLAVEADDRESS ≠ DS:RBX) )
    THEN #PF(DS:RBX); FI;
```

```
(* Copy 64 bytes of contents *)
SCRATCH_SECINFO ← DS:RBX;

(* Check for mis-configured SECINFO flags*)
IF ( (SCRATCH_SECINFO reserved fields are not zero ) or ((SCRATCH_SECINFO.FLAGS.R=0) AND(SCRATCH_SECINFO.FLAGS.W≠0 ) or
    (SCRATCH_SECINFO.FLAGS.PT is not PT_REG) )
    THEN #GP(0); FI;

(* Check security attributes of the source EPC page *)
IF ( (EPCM(DS:RDX).VALID = 0) or (EPCM(DS:RCX).R = 0) or (EPCM(DS:RDX).PENDING ≠ 0) or (EPCM(DS:RDX).MODIFIED ≠ 0) or
    (EPCM(DS:RDX).BLOCKED ≠ 0) or (EPCM(DS:RDX).PT ≠ PT_REG) or (EPCM(DS:RDX).ENCLAVESECS ≠ CR_ACTIVE_SECS) or
    (EPCM(DS:RDX).ENCLAVEADDRESS ≠ DS:RDX))
    THEN #PF(DS:RDX); FI;

(* Check security attributes of the destination EPC page *)
IF ( (EPCM(DS:RCX).VALID = 0) or (EPCM(DS:RCX).PENDING ≠ 1) or (EPCM(DS:RCX).MODIFIED ≠ 0) or
    (EPCM(DS:RDX).BLOCKED ≠ 0) or (EPCM(DS:RCX).PT ≠ PT_REG) or (EPCM(DS:RCX).ENCLAVESECS ≠ CR_ACTIVE_SECS) )
    THEN
        RFLAGS.ZF ← 1;
        RAX ← SGX_PAGE_ATTRIBUTES_MISMATCH;
        GOTO DONE;
FI;

(* Check the destination EPC page for concurrency *)
IF (destination EPC page in use )
    THEN #GP(0); FI;

(* Re-Check security attributes of the destination EPC page *)
IF ( (EPCM(DS:RCX).VALID = 0) or (EPCM(DS:RCX).PENDING ≠ 1) or (EPCM(DS:RCX).MODIFIED ≠ 0) or
    (EPCM(DS:RCX).R ≠ 1) or (EPCM(DS:RCX).W ≠ 1) or (EPCM(DS:RCX).X ≠ 0) or
    (EPCM(DS:RCX).PT ≠ SCRATCH_SECINFO.FLAGS.PT) or (EPCM(DS:RCX).ENCLAVESECS ≠ CR_ACTIVE_SECS) or
    (EPCM(DS:RCX).ENCLAVEADDRESS ≠ DS:RCX))
    THEN
        RFLAGS.ZF ← 1;
        RAX ← SGX_PAGE_ATTRIBUTES_MISMATCH;
        GOTO DONE;
FI;

(* Copy 4KBbytes form the source to destination EPC page*)
DS:RCX[32767:0] ← DS:RDX[32767:0];

(* Update EPCM permissions *)
EPCM(DS:RCX).R ← SCRATCH_SECINFO.FLAGS.R;
EPCM(DS:RCX).W ← SCRATCH_SECINFO.FLAGS.W;
EPCM(DS:RCX).X ← SCRATCH_SECINFO.FLAGS.X;
EPCM(DS:RCX).PENDING ← 0;

RFLAGS.ZF ← 0;
RAX ← 0;

DONE:
RFLAGS.CF,PF,AF,OF,SF ← 0;
```

Flags Affected

Sets ZF if page is not modifiable, otherwise cleared. Clears CF, PF, AF, OF, SF

Protected Mode Exceptions

#GP(0)	If a memory operand effective address is outside the DS segment limit.
	If a memory operand is not properly aligned.
	If a memory operand is locked.
#PF(error code)	If a page fault occurs in accessing memory operands.
	If a memory operand is not an EPC page.
	If EPC page has incorrect page type or security attributes.

64-Bit Mode Exceptions

#GP(0)	If a memory operand is non-canonical form.
	If a memory operand is not properly aligned.
	If a memory operand is locked.
#PF(error code)	If a page fault occurs in accessing memory operands.
	If a memory operand is not an EPC page.
	If EPC page has incorrect page type or security attributes.

EENTER—Enters an Enclave

Opcode/ Instruction	Op/En	64/32 bit Mode Support	CPUID Feature Flag	Description
EAX = 02H ENCLU[EENTER]	IR	V/V	SGX1	This leaf function is used to enter an enclave.

Instruction Operand Encoding

Op/En	EAX		RBX	RCX	
IR	EENTER (In)	Content of RBX.CSSA (Out)	Address of a TCS (In)	Address of AEP (In)	Address of IP following EENTER (Out)

Description

The ENCLU[EENTER] instruction transfers execution to an enclave. At the end of the instruction, the logical processor is executing in enclave mode at the RIP computed as EnclaveBase + TCS.OENTRY. If the target address is not within the CS segment (32-bit) or is not canonical (64-bit), a #GP(0) results.

EENTER Memory Parameter Semantics

TCS
Enclave access

EENTER is a serializing instruction. The instruction faults if any of the following occurs:

Address in RBX is not properly aligned.	Any TCS.FLAGS's must-be-zero bit is not zero.
TCS pointed to by RBX is not valid or available or locked.	Current 32/64 mode does not match the enclave mode in SECS.ATTRIBUTES.MODE64.
The SECS is in use.	Either of TCS-specified FS and GS segment is not a subsets of the current DS segment.
Any one of DS, ES, CS, SS is not zero.	If XSAVE available, CR4.OSXSAVE = 0, but SECS.ATTRIBUTES.XFRM ≠ 3.
CR4.OSFXSR ≠ 1.	If CR4.OSXSAVE = 1, SECS.ATTRIBUTES.XFRM is not a subset of XCR0.

The following operations are performed by EENTER:

- RSP and RBP are saved in the current SSA frame on EENTER and are automatically restored on EEXIT or interrupt.
- The AEP contained in RCX is stored into the TCS for use by AEXs.FS and GS (including hidden portions) are saved and new values are constructed using TCS.OFSBASE/GSBASE (32 and 64-bit mode) and TCS.OFSLIMIT/GSLIMIT (32-bit mode only). The resulting segments must be a subset of the DS segment.
- If CR4.OSXSAVE == 1, XCR0 is saved and replaced by SECS.ATTRIBUTES.XFRM. The effect of RFLAGS.TF depends on whether the enclave entry is opt-in or opt-out (see Section 42.1.2):
  - — On opt-out entry, TF is saved and cleared (it is restored on EEXIT or AEX). Any attempt to set TF via a POPF instruction while inside the enclave clears TF (see Section 42.2.5).
  - — On opt-in entry, a single-step debug exception is pended on the instruction boundary immediately after EENTER (see Section 42.2.2).
- All code breakpoints that do not overlap with ELRANGE are also suppressed. If the entry is an opt-out entry, all code and data breakpoints that overlap with the ELRANGE are suppressed.
- On opt-out entry, a number of performance monitoring counters and behaviors are modified or suppressed (see Section 42.2.3):

— All performance monitoring activity on the current thread is suppressed except for incrementing and firing of FIXED_CTR1 and FIXED_CTR2.

— PEBS is suppressed.

— AnyThread counting on other threads is demoted to MyThread mode and IA32_PERF_GLOBAL_STATUS[60] on that thread is set

— If the opt-out entry on a hardware thread results in suppression of any performance monitoring, then the processor sets IA32_PERF_GLOBAL_STATUS[60] and IA32_PERF_GLOBAL_STATUS[63].

Concurrency Restrictions

Table 40-52. Concurrency Restrictions of EENTER with Intel® SGX Instructions - 1of 2

Operation		EEXIT			EADD		EBLOCK		ECREATE	EDBGRD/WR		EENTER/ERESUME			EEXTEND		EGETKEY		EINIT	ELDB/ELDU			EPA
	Param	Targ	VA	SECS	Targ	SECS	Targ	SECS	SECS	Targ	SECS	TCS	SSA	SECS	Targ	SECS	Param	SECS	SECS	Targ	VA	SECS	VA
EENTER	TCS	N			N				N	Y		N								N			N
	SSA		U							Y		Y	U				U						
	SECS			Y	Y	N	Y	Y	Y	Y	Y	Y	Y	Y	Y	N	Y	Y	N	Y	Y	Y	Y

Table 40-53. Concurrency Restrictions of EENTER with Intel® SGX Instructions - 2 of 2

Operation		EREMOVE		EREPORT		ETRACK	EWB			EAUG		EMODPE		EMODPR		EMODT		EACCEPT			EACCEPTCOPY		
	Param	Targ	SECS	Param	SECS	SECS	SRC	VA	SECS	Targ	SECS	Targ	SECINFO	Targ	SECS	Targ	SECS	Targ	SECINFO	SECS	Targ	SRC	SECINFO
EENTER	TCS	N					N			N						N							
	SSA			U								Y	U					Y	U			U	U
	SECS	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y		Y	Y	Y	Y

Operation

Temp Variables in EENTER Operational Flow

Name	Type	Size (Bits)	Description
TMP_FSBASE	Effective Address	32/64	Proposed base address for FS segment.
TMP_GSBASE	Effective Address	32/64	Proposed base address for FS segment.
TMP_FSLIMIT	Effective Address	32/64	Highest legal address in proposed FS segment.
TMP_GSLIMIT	Effective Address	32/64	Highest legal address in proposed GS segment.
TMP_XSIZE	integer	64	Size of XSAVE area based on SECS.ATTRIBUTES.XFRM.
TMP_SSA_PAGE	Effective Address	32/64	Pointer used to iterate over the SSA pages in the current frame.
TMP_GPR	Effective Address	32/64	Address of the GPR area within the current SSA frame.

```
TMP_MODE64 ← ((IA32_EFER.LMA = 1) && (CS.L = 1));

(* Make sure DS is usable, expand up *)
IF (TMP_MODE64 = 0 and (DS not usable or ( ( DS[S] = 1) and (DS[bit 11] = 0) and DS[bit 10] = 1) ) ) )
    THEN #GP(0); FI;

(* Check that CS, SS, DS, ES.base is 0 *)
IF (TMP_MODE64 = 0)
    THEN
        IF(CS.base ≠ 0 or DS.base ≠ 0) #GP(0); FI;
```

```
            IF(ES usable and ES.base ≠ 0) #GP(0); FI;
            IF(SS usable and SS.base ≠ 0) #GP(0); FI;
            IF(SS usable and SS.B = 0) #GP(0); FI;
FI;

IF (DS:RBX is not 4KByte Aligned)
    THEN #GP(0); FI;

IF (DS:RBX does not resolve within an EPC)
    THEN #PF(DS:RBX); FI;

(* Check AEP is canonical*)
IF (TMP_MODE64 = 1 and (CS:RCX is not canonical) )
    THEN #GP(0); FI;

(* Check concurrency of TCS operation*)
IF (Other Intel SGX instructions is operating on TCS)
    THEN #GP(0); FI;

(* TCS verification *)
IF (EPCM(DS:RBX).VALID = 0)
    THEN #PF(DS:RBX); FI;

IF (EPCM(DS:RBX).BLOCKED = 1)
    THEN #PF(DS:RBX); FI;

IF ( (EPCM(DS:RBX).ENCLAVEADDRESS ≠ DS:RBX) or (EPCM(DS:RBX).PT ≠ PT_TCS) )
    THEN #PF(DS:RBX); FI;

IF ((EPCM(DS:RBX).PENDING = 1) or (EPCM(DS:RBX).MODIFIED = 1))
    THEN #PF(DS:RBX); FI;

IF ( (DS:RBX).OSSA is not 4KByte Aligned)
    THEN #GP(0); FI;

(* Check proposed FS and GS *)
IF ( ( (DS:RBX).OFSBASE is not 4KByte Aligned) or ( (DS:RBX).OGSBASE is not 4KByte Aligned) )
    THEN #GP(0); FI;

(* Get the SECS for the enclave in which the TCS resides *)
TMP_SECS ← Address of SECS for TCS;

(* Check proposed FS/GS segments fall within DS *)
IF (TMP_MODE64 = 0)
    THEN
        TMP_FSBASE ← (DS:RBX).OFSBASE + TMP_SECS.BASEADDR;
        TMP_FSLIMIT ← (DS:RBX).OFSBASE + TMP_SECS.BASEADDR + (DS:RBX).FSLIMIT;
        TMP_GSBASE ← (DS:RBX).OGSBASE + TMP_SECS.BASEADDR;
        TMP_GSLIMIT ← (DS:RBX).OGSBASE + TMP_SECS.BASEADDR + (DS:RBX).GSLIMIT;
        (* if FS wrap-around, make sure DS has no holes*)
        IF (TMP_FSLIMIT < TMP_FSBASE)
            THEN
                IF (DS.limit < 4GB) THEN #GP(0); FI;
            ELSE
```

```
                IF (TMP_FSLIMIT > DS.limit) THEN #GP(0); FI;
        FI;
        (* if GS wrap-around, make sure DS has no holes*)
        IF (TMP_GSLIMIT < TMP_GSBASE)
            THEN
                IF (DS.limit < 4GB) THEN #GP(0); FI;
            ELSE
                IF (TMP_GSLIMIT > DS.limit) THEN #GP(0); FI;
        FI;
    ELSE
        TMP_FSBASE ← (DS:RBX).OFSBASE + TMP_SECS.BASEADDR;
        TMP_GSBASE ← (DS:RBX).OGSBASE + TMP_SECS.BASEADDR;
        IF ( (TMP_FSBASE is not canonical) or (TMP_GSBASE is not canonical))
            THEN #GP(0); FI;
FI;

(* Ensure that the FLAGS field in the TCS does not have any reserved bits set *)
IF ( ( (DS:RBX).FLAGS & FFFFFFFFFFFFFFFEH) ≠ 0)
    THEN #GP(0); FI;

(* SECS must exist and enclave must have previously been EINITted *)
IF (the enclave is not already initialized)
    THEN #GP(0); FI;

(* make sure the logical processor's operating mode matches the enclave *)
IF ( (TMP_MODE64 ≠ TMP_SECS.ATTRIBUTES.MODE64BIT) )
    THEN #GP(0); FI;

IF (CR4.OSFXSR = 0)
    THEN #GP(0); FI;

(* Check for legal values of SECS.ATTRIBUTES.XFRM *)
IF (CR4.OSXSAVE = 0)
    THEN
        IF (TMP_SECS.ATTRIBUTES.XFRM ≠ 03H) THEN #GP(0); FI;
    ELSE
        IF ( (TMP_SECS.ATTRIBUTES.XFRM & XCR0) ≠ TMP_SECS.ATTRIBUES.XFRM) THEN #GP(0); FI;
FI;

(* Make sure the SSA contains at least one more frame *)
IF ( (DS:RBX).CSSA ≥ (DS:RBX).NSSA)
    THEN #GP(0); FI;

(* Compute linear address of SSA frame *)
TMP_SSA ← (DS:RBX).OSSA + TMP_SECS.BASEADDR + 4096 * TMP_SECS.SSAFRAMESIZE * (DS:RBX).CSSA;
TMP_XSIZE ← compute_XSAVE_frame_size(TMP_SECS.ATTRIBUTES.XFRM);

FOR EACH TMP_SSA_PAGE = TMP_SSA to TMP_SSA + TMP_XSIZE
    (* Check page is read/write accessible *)
    Check that DS:TMP_SSA_PAGE is read/write accessible;
    If a fault occurs, release locks, abort and deliver that fault;

    IF (DS:TMP_SSA_PAGE does not resolve to EPC page)
        THEN #PF(DS:TMP_SSA_PAGE); FI;
```

```
    IF (EPCM(DS:TMP_SSA_PAGE).VALID = 0)
        THEN #PF(DS:TMP_SSA_PAGE); FI;
    IF (EPCM(DS:TMP_SSA_PAGE).BLOCKED = 1)
        THEN #PF(DS:TMP_SSA_PAGE); FI;
    IF ((EPCM(DS:TMP_SSA_PAGE).PENDING = 1) or (EPCM(DS:TMP_SSA_PAGE).MODIFIED = 1))
        THEN #PF(DS:TMP_SSA_PAGE); FI;
    IF ( ( EPCM(DS:TMP_SSA_PAGE).ENCLAVEADDRESS ≠ DS:TMP_SSA_PAGE) or (EPCM(DS:TMP_SSA_PAGE).PT ≠ PT_REG) or
        (EPCM(DS:TMP_SSA_PAGE).ENCLAVESECS ≠ EPCM(DS:RBX).ENCLAVESECS) or
        (EPCM(DS:TMP_SSA_PAGE).R = 0) or (EPCM(DS:TMP_SSA_PAGE).W = 0) )
        THEN #PF(DS:TMP_SSA_PAGE); FI;
    CR_XSAVE_PAGE_n ← Physical_Address(DS:TMP_SSA_PAGE);
ENDFOR

(* Compute address of GPR area*)
TMP_GPR ← TMP_SSA + 4096 * DS:TMP_SECS.SSAFRAMESIZE -- sizeof(GPRSGX_AREA);
If a fault occurs; release locks, abort and deliver that fault;

IF (DS:TMP_GPR does not resolve to EPC page)
    THEN #PF(DS:TMP_GPR); FI;
IF (EPCM(DS:TMP_GPR).VALID = 0)
    THEN #PF(DS:TMP_GPR); FI;
IF (EPCM(DS:TMP_GPR).BLOCKED = 1)
    THEN #PF(DS:TMP_GPR); FI;
IF ((EPCM(DS:TMP_GPR).PENDING = 1) or (EPCM(DS:TMP_GPR).MODIFIED = 1))
    THEN #PF(DS:TMP_GPR); FI;
IF ( ( EPCM(DS:TMP_GPR).ENCLAVEADDRESS ≠ DS:TMP_GPR) or (EPCM(DS:TMP_GPR).PT ≠ PT_REG) or
    (EPCM(DS:TMP_GPR).ENCLAVESECS EPCM(DS:RBX).ENCLAVESECS) or
    (EPCM(DS:TMP_GPR).R = 0) or (EPCM(DS:TMP_GPR).W = 0) )
    THEN #PF(DS:TMP_GPR); FI;

IF (TMP_MODE64 = 0)
    THEN
        IF (TMP_GPR + (GPR_SIZE -1) is not in DS segment) THEN #GP(0); FI;
FI;

CR_GPR_PA ← Physical_Address (DS: TMP_GPR);

(* Validate TCS.OENTRY *)
TMP_TARGET ← (DS:RBX).OENTRY + TMP_SECS.BASEADDR;
IF (TMP_MODE64 = 1)
    THEN
        IF (TMP_TARGET is not canonical) THEN #GP(0); FI;
    ELSE
        IF (TMP_TARGET > CS limit) THEN #GP(0); FI;
FI;

(* Ensure the enclave is not already active and this thread is the only one using the TCS*)
IF (DS:RBX.STATE = ACTIVE))
    THEN #GP(0); FI;

CR_ENCLAVE_MODE ← 1;
CR_ACTIVE_SECS ← TMP_SECS;
CR_ELRANGE ← (TMPSECS.BASEADDR, TMP_SECS.SIZE);
```

```
(* Save state for possible AEXs *)
CR_TCS_PA ← Physical_Address (DS:RBX);
CR_TCS_LA ← RBX;
CR_TCS_LA.AEP ← RCX;

(* Save the hidden portions of FS and GS *)
CR_SAVE_FS_selector ← FS.selector;
CR_SAVE_FS_base ← FS.base;
CR_SAVE_FS_limit ← FS.limit;
CR_SAVE_FS_access_rights ← FS.access_rights;
CR_SAVE_GS_selector ← GS.selector;
CR_SAVE_GS_base ← GS.base;
CR_SAVE_GS_limit ← GS.limit;
CR_SAVE_GS_access_rights ← GS.access_rights;

(* If XSAVE is enabled, save XCR0 and replace it with SECS.ATTRIBUTES.XFRM*)
IF (CR4.OSXSAVE = 1)
    CR_SAVE_XCR0 ← XCR0;
    XCR0 ← TMP_SECS.ATTRIBUTES.XFRM;
FI;

RCX ← RIP;
RIP ← TMP_TARGET;
RAX ← (DS:RBX).CSSA;
(* Save the outside RSP and RBP so they can be restored on interrupt or EEXIT *)
DS:TMP_SSA.U_RSP ← RSP;
DS:TMP_SSA.U_RBP ← RBP;

(* Do the FS/GS swap *)
FS.base ← TMP_FSBASE;
FS.limit ← DS:RBX.FSLIMIT;
FS.type ← 0001b;
FS.W ← DS.W;
FS.S ← 1;
FS.DPL ← DS.DPL;
FS.G ← 1;
FS.B ← 1;
FS.P ← 1;
FS.AVL ← DS.AVL;
FS.L ← DS.L;
FS.unusable ← 0;
FS.selector ← 0BH;

GS.base ← TMP_GSBASE;
GS.limit ← DS:RBX.GSLIMIT;
GS.type ← 0001b;
GS.W ← DS.W;
GS.S ← 1;
GS.DPL ← DS.DPL;
GS.G ← 1;
GS.B ← 1;
GS.P ← 1;
GS.AVL ← DS.AVL;
GS.L ← DS.L;
```

```
GS.unusable ← 0;
GS.selector ← 0BH;

CR_DBGOPTIN ← TCS.FLAGS.DBGOPTIN;
Suppress_all_code_breakpoints_that_are_outside_ELRANGE;

IF (CR_DBGOPTIN = 0)
    THEN
        Suppress_all_code_breakpoints_that_overlap_with_ELRANGE;
        CR_SAVE_TF ← RFLAGS.TF;
        RFLAGS.TF ← 0;
        Suppress_monitor_trap_flag for the source of the execution of the enclave;
        Suppress any pending debug exceptions;
        Suppress any pending MTF VM exit;
    ELSE
        IF RFLAGS.TF = 1
            THEN pend a single-step #DB at the end of EENTER; FI;
        IF the "monitor trap flag" VM-execution control is set
            THEN pend an MTF VM exit at the end of EENTER; FI;
FI;

Flush_linear_context;
Allow_front_end_to_begin_fetch_at_new_RIP;
```

Flags Affected

RFLAGS.TF is cleared on opt-out entry

Protected Mode Exceptions

#GP(0)	If DS:RBX is not page aligned.
	If the enclave is not initialized.
	If part or all of the FS or GS segment specified by TCS is outside the DS segment or not properly aligned.
	If the thread is not in the INACTIVE state.
	If CS, DS, ES or SS bases are not all zero.
	If executed in enclave mode.
	If any reserved field in the TCS FLAG is set.
	If the target address is not within the CS segment.
	If CR4.OSFXSR = 0.
	If CR4.OSXSAVE = 0 and SECS.ATTRIBUTES.XFRM ≠ 3.
	If CR4.OSXSAVE = 1and SECS.ATTRIBUTES.XFRM is not a subset of XCR0.
#PF(error code)	If a page fault occurs in accessing memory.
	If DS:RBX does not point to a valid TCS.
	If one or more pages of the current SSA frame are not readable/writable, or do not resolve to a valid PT_REG EPC page.

64-Bit Mode Exceptions

#GP(0)	If DS:RBX is not page aligned.
	If the enclave is not initialized.
	If the thread is not in the INACTIVE state.
	If CS, DS, ES or SS bases are not all zero.
	If executed in enclave mode.

	If part or all of the FS or GS segment specified by TCS is outside the DS segment or not properly aligned.
	If the target address is not canonical.
	If CR4.OSFXSR = 0.
	If CR4.OSXSAVE = 0 and SECS.ATTRIBUTES.XFRM ≠ 3.
	If CR4.OSXSAVE = 1and SECS.ATTRIBUTES.XFRM is not a subset of XCR0.
#PF(error code)	If a page fault occurs in accessing memory operands.
	If DS:RBX does not point to a valid TCS.
	If one or more pages of the current SSA frame are not readable/writable, or do not resolve to a valid PT_REG EPC page.

EEXIT—Exits an Enclave

Opcode/ Instruction	Op/En	64/32 bit Mode Support	CPUID Feature Flag	Description
EAX = 04H ENCLU[EEXIT]	IR	V/V	SGX1	This leaf function is used to exit an enclave.

Instruction Operand Encoding

Op/En	EAX	RBX	RCX
IR	EEXIT (In)	Target address outside the enclave (In)	Address of the current AEP (In)

Description

The ENCLU[EEXIT] instruction exits the currently executing enclave and branches to the location specified in RBX. RCX receives the current AEP. If RBX is not within the CS (32-bit mode) or is not canonical (64-bit mode) a #GP(0) results.

EEXIT Memory Parameter Semantics

Target Address
Non-Enclave read and execute access

If RBX specifies an address that is inside the enclave, the instruction will complete normally. The fetch of the next instruction will occur in non-enclave mode, but will attempt to fetch from inside the enclave. This has the effect of abort page semantics on the next destination.

If secrets are contained in any registers, it is responsibility of enclave software to clear those registers.

If XCR0 was modified on enclave entry, it is restored to the value it had at the time of the most recent EENTER or ERESUME.

If the enclave is opt-out, RFLAGS.TF is loaded from the value previously saved on EENTER.

Code and data breakpoints are unsuppressed.

Performance monitoring counters are unsuppressed.

Concurrency Restrictions

Table 40-54. Concurrency Restrictions of EEXIT with Intel® SGX Instructions - 1of 2

Operation		EEXIT			EADD		EBLOCK		ECREATE	EDBGRD/WR		EENTER/ERESUME			EEXTEND		EGETKEY		EINIT	ELDB/ELDU			EPA
	Param	Targ	VA	SECS	Targ	SECS	Targ	SECS	SECS	Targ	SECS	TCS	SSA	SECS	Targ	SECS	Param	SECS	SECS	Targ	VA	SECS	VA
EEXIT	TCS				Y	Y	Y		Y	Y		N	Y		Y	Y	Y		Y	Y	Y	Y	Y
	SSA		U		Y	Y	Y		Y	Y		Y	U		Y	Y	U		Y	Y	Y	Y	Y
	SECS			Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y

Table 40-55. Concurrency Restrictions of EEXIT with Intel® SGX Instructions - 2 of 2

Operation		EREMOVE		EREPORT		ETRACK	EWB			EAUG		EMODPE		EMODPR		EMODT		EACCEPT			EACCEPTCOPY		
	Param	Targ	SECS	Param	SECS	SECS	SRC	VA	SECS	Targ	SECS	Targ	SECINFO	Targ	SECS	Targ	SECS	Targ	SECINFO	SECS	Targ	SRC	SECINFO
EEXIT	TCS	Y		Y		Y	Y	Y		Y	Y	Y	Y	Y		Y		Y	Y		Y	Y	Y
	SSA	Y				Y	Y	Y		Y	Y	Y	U	Y		Y		Y	U		Y	U	U
	SECS	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y

Operation

Temp Variables in EEXIT Operational Flow

Name	Type	Size (Bits)	Description
TMP_RIP	Effective Address	32/64	Saved copy of CRIP for use when creating LBR.

```
TMP_MODE64 ← ((IA32_EFER.LMA = 1) && (CS.L = 1));

IF (TMP_MODE64 = 1)
    THEN
        IF (RBX is not canonical) THEN #GP(0); FI;
    ELSE
        IF (RBX > CS limit) THEN #GP(0); FI;
FI;

TMP_RIP ← CRIP;
RIP ← RBX;

(* Return current AEP in RCX *)
RCX ← CR_TCS_PA.AEP;

(* Do the FS/GS swap *)
FS.selector ← CR_SAVE_FS.selector;
FS.base ← CR_SAVE_FS.base;
FS.limit ← CR_SAVE_FS.limit;
FS.access_rights ← CR_SAVE_FS.access_rights;
GS.selector ← CR_SAVE_GS.selector;
GS.base ← CR_SAVE_GS.base;
GS.limit ← CR_SAVE_GS.limit;
GS.access_rights ← CR_SAVE_GS.access_rights;

(* Restore XCR0 if needed *)
IF (CR4.OSXSAVE = 1)
    XCR0 ← CR_SAVE__XCR0;
FI;

Unsuppress_all_code_breakpoints_that_are_outside_ELRANGE;

IF (CR_DBGOPTIN = 0)
    THEN
        UnSuppress_all_code_breakpoints_that_overlap_with_ELRANGE;
        Restore suppressed breakpoint matches;
        RFLAGS.TF ← CR_SAVE_TF;
        UnSuppress_montior_trap_flag;
        UnSuppress_LBR_Generation;
        UnSuppress_performance monitoring_activity;
        Restore performance monitoring counter AnyThread demotion to MyThread in enclave back to AnyThread
FI;

IF RFLAGS.TF = 1
    THEN Pend Single-Step #DB at the end of EEXIT;
FI;
```

```
IF the "monitor trap flag" VM-execution control is set
    THEN pend a MTF VM exit at the end of EEXIT;
FI;

CR_ENCLAVE_MODE ← 0;
CR_TCS_PA.STATE ← INACTIVE;

(* Assure consistent translations *)
Flush_linear_context;
```

Flags Affected

RFLAGS.TF is restored from the value previously saved in EENTER or ERESUME.

Protected Mode Exceptions

#GP(0)	If executed outside an enclave.
	If RBX is outside the CS segment.
#PF(error code)	If a page fault occurs in accessing memory.

64-Bit Mode Exceptions

#GP(0)	If executed outside an enclave.
	If RBX is not canonical.
#PF(error code)	If a page fault occurs in accessing memory operands.

EGETKEY—Retrieves a Cryptographic Key

Opcode/ Instruction	Op/En	64/32 bit Mode Support	CPUID Feature Flag	Description
EAX = 04H ENCLU[EGETKEY]	IR	V/V	SGX1	This leaf function retrieves a cryptographic key.

Instruction Operand Encoding

Op/En	EAX	RBX	RCX
IR	EGETKEY (In)	Address to a KEYREQUEST (In)	Address of the OUTPUTDATA (In)

Description

The ENCLU[EGETKEY] instruction returns a 128-bit secret key from the processor specific key hierarchy. The register RBX contains the effective address of a KEYREQUEST structure, which the instruction interprets to determine the key being requested. The Requesting Keys section below provides a description of the keys that can be requested. The RCX register contains the effective address where the key will be returned. Both the addresses in RBX & RCX should be locations inside the enclave.

EGETKEY derives keys using a processor unique value to create a specific key based on a number of possible inputs. This instruction leaf can only be executed inside an enclave.

EEGETKEY Memory Parameter Semantics

KEYREQUEST	OUTPUTDATA
Enclave read access	Enclave write access

After validating the operands, the instruction determines which key is to be produced and performs the following actions:

- The instruction assembles the derivation data for the key based on the Table 40-56
- Computes derived key using the derivation data and package specific value
- Outputs the calculated key to the address in RCX

The instruction fails with #GP(0) if the operands are not properly aligned. Successful completion of the instruction will clear RFLAGS.{ZF, CF, AF, OF, SF, PF}. The instruction returns an error code if the user tries to request a key based on an invalid CPUSVN or ISVSVN (when the user request is accepted, see the table below), requests a key for which it has not been granted the attribute to request, or requests a key that is not supported by the hardware. These checks may be performed in any order. Thus, an indication by error number of one cause (for example, invalid attribute) does not imply that there are not also other errors. Different processors may thus give different error numbers for the same Enclave. The correctness of software should not rely on the order resulting from the checks documented in this section. In such cases the ZF flag is set and the corresponding error bit (SGX_INVALID_SVN, SGX_INVALID_ATTRIBUTE, SGX_INVALID_KEYNAME) is set in RAX and the data at the address specified by RCX is unmodified.

Requesting Keys

The KEYREQUEST structure (see Section 37.17.1) identifies the key to be provided. The Keyrequest.KeyName field identifies which type of key is requested.

Deriving Keys

Key derivation is based on a combination of the enclave specific values (see Table 40-56) and a processor key. Depending on the key being requested a field may either be included by definition or the value may be included from the KeyRequest. A “yes” in Table 40-56 indicates the value for the field is included from its default location, identified in the source row, and a “request” indicates the values for the field is included from its corresponding KeyRequest field.

Table 40-56. Key Derivation

	Key Name	Attributes	Owner Epoch	CPU SVN	ISV SVN	ISV PRODID	MRENCLAVE	MRSIGNER	RAND
Source	Key Dependent Constant	Y← SECS.ATTRIBUTES and SECS.MISCSELECT; R←AttribMask & SECS.ATTRIBUTES and SECS.MISCSELECT;	CR_SGX OWNER EPOCH	Y← CPUSVN Register; R← Req.CPUSVN;	R← Req.ISVSVN;	SECS. ISVID	SECS. MRENCLAVE	SECS. MRSIGNER	Req. KEYID
EINITTOKEN	Yes	Request	Yes	Request	Request	Yes	No	Yes	Request
Report	Yes	Yes	Yes	Yes	No	No	Yes	No	Request
Seal	Yes	Request	Yes	Request	Request	Yes	Request	Request	Request
Provisioning	Yes	Request	No	Request	Request	Yes	No	Yes	Yes
Provisioning Seal	Yes	Request	No	Request	Request	Yes	No	Yes	Yes

Keys that permit the specification of a CPU or ISV's code's SVNs have additional requirements. The caller may not request a key for an SVN beyond the current CPU or ISV SVN, respectively.

Several keys are access controlled. Access to the Provisioning Key and Provisioning Seal key requires the enclave's ATTRIBUTES.PROVISIONKEY be set. The EINITTOKEN Key requires ATTRIBUTES.EINITTOKEN_KEY be set and SECS.MRSIGNER equal IA32_SGXLEPUBKEYHASH.

Some keys are derived based on a hardcode PKCS padding constant (352 byte string):

HARDCODED_PKCS1_5_PADDING[15:0] ← 0100H;

HARDCODED_PKCS1_5_PADDING[2655:16] ← SignExtend330Byte(-1); // 330 bytes of 0FFH

HARDCODED_PKCS1_5_PADDING[2815:2656] ← 2004000501020403650148866009060D30313000H;

The error codes are:

Table 40-57. EGETKEY Return Value in RAX

Error Code (see Table 40-3)	Value	Description
No Error	0	EGETKEY successful
SGX_INVALID_ATTRIBUTE		The KEYREQUEST contains a KEYNAME for which the enclave is not authorized
SGX_INVALID_CPUSVN		If KEYREQUEST.CPUSVN is an unsupported platforms CPUSVN value
SGX_INVALID_ISVSVN		If KEYREQUEST.ISVSVN is greater than the enclave's ISV_SVN
SGX_INVALID_KEYNAME		If KEYREQUEST.KEYNAME is an unsupported value

Concurrency Restrictions

Table 40-58. Concurrency Restrictions of EGETKEY with Other Intel® SGX Operations 1 of 2

Operation		EEXIT			EADD		EBLOCK		ECREATE	EDBGRD/WR		EENTER/ ERESUME			EEXTEND		EGETKEY		EINIT	ELDB/ELDU			EPA
	Param	TCS	SSA	SECS	Targ	SECS	Targ	SECS	SECS	Targ	SECS	TCS	SSA	SECS	Targ	SECS	Param	SECS	SECS	Targ	VA	SECS	VA
EGETKEY	Param		U							Y			U				U						
	SECS			Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y

Table 40-59. Concurrency Restrictions of EGETKEY with Other Intel® SGX Operations 2 of 2

Operation		EREMOVE		EREPORT		ETRACK	EWB			EAUG		EMODPE		EMODPR		EMODT		EACCEPT			EACCEPTCOPY		
	Param	Targ	SECS	Param	SECS	SECS	SRC	VA	SECS	Targ	SECS	Targ	SECINFO	Targ	SECS	Targ	SECS	Targ	SECINFO	SECS	Targ	SRC	SECINFO
EGETKEY	Param			U								Y	U					Y	U			Y	U
	SECS	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y

Operation

Temp Variables in EGETKEY Operational Flow

Name	Type	Size (Bits)	Description
TMP_CURRENTSECS			Address of the SECS for the currently executing enclave.
TMP_KEYDEPENDENCIES			Temp space for key derivation.
TMP_ATTRIBUTES		128	Temp Space for the calculation of the sealable Attributes.
TMP_OUTPUTKEY		128	Temp Space for the calculation of the key.

```
(* Make sure KEYREQUEST is properly aligned and inside the current enclave *)
IF ( (DS:RBX is not 512Byte aligned) or (DS:RBX is within CR_ELRANGE) )
    THEN #GP(0); FI;

(* Make sure DS:RBX is an EPC address and the EPC page is valid *)
IF ( (DS:RBX does not resolve to an EPC address) or (EPCM(DS:RBX).VALID = 0) )
    THEN #PF(DS:RBX); FI;

IF (EPCM(DS:RBX).BLOCKED = 1)
    THEN #PF(DS:RBX); FI;

(* Check page parameters for correctness *)
IF ( (EPCM(DS:RBX).PT ≠ PT_REG) or (EPCM(DS:RBX).ENCLAVESECS ≠ CR_ACTIVE_SECS) or (EPCM(DS:RBX).PENDING = 1) or
    (EPCM(DS:RBX).MODIFIED = 1) or (EPCM(DS:RBX).ENCLAVEADDRESS ≠ (DS:RBX & ~0FFFH) ) or (EPCM(DS:RBX).R = 0) )
    THEN #PF(DS:RBX);
FI;

(* Make sure OUTPUTDATA is properly aligned and inside the current enclave *)
IF ( (DS:RCX is not 16Byte aligned) or (DS:RCX is not within CR_ELRANGE) )
    THEN #GP(0); FI;

(* Make sure DS:RCX is an EPC address and the EPC page is valid *)
IF ( (DS:RCX does not resolve to an EPC address) or (EPCM(DS:RCX).VALID = 0) )
    THEN #PF(DS:RCX); FI;

IF (EPCM(DS:RCX).BLOCKED = 1)
    THEN #PF(DS:RCX); FI;

(* Check page parameters for correctness *)
IF ( (EPCM(DS:RCX).PT ≠ PT_REG) or (EPCM(DS:RCX).ENCLAVESECS ≠ CR_ACTIVE_SECS) or (EPCM(DS:RCX).PENDING = 1) or
    (EPCM(DS:RCX).MODIFIED = 1) or (EPCM(DS:RCX).ENCLAVEADDRESS ≠ (DS:RCX & ~0FFFH) ) or (EPCM(DS:RCX).W = 0) )
    THEN #PF(DS:RCX);
FI;
```

```
(* Verify RESERVED spaces in KEYREQUEST are valid *)
IF ( (DS:RBX).RESERVED ≠ 0) or (DS:RBX.KEYPOLICY.RESERVED ≠ 0) )
    THEN #GP(0); FI;

TMP_CURRENTSECS ← CR_ACTIVE_SECS;

(* Determine which enclave attributes that must be included in the key. Attributes that must always be include INIT & DEBUG *)
REQUIRED_SEALING_MASK[127:0] ← 00000000 00000000 00000000 00000003H;
TMP_ATTRIBUTES ← (DS:RBX.ATTRIBUTEMASK | REQUIRED_SEALING_MASK) & TMP_CURRENTSECS.ATTRIBUTES;

(* Compute MISCSELECT fields to be included *)
TMP_MISCSELECT ← DS:RBX.MISCMASK & TMP_CURRENTSECS.MISCSELECT

CASE (DS:RBX.KEYNAME)
    SEAL_KEY:
        IF (DS:RBX.CPUSVN is beyond current CPU configuration)
            THEN
                RFLAGS.ZF ← 1;
                RAX ← SGX_INVALID_CPUSVN;
                GOTO EXIT;
        FI;
        IF (DS:RBX.ISVSVN > TMP_CURRENTSECS.ISVSVN)
            THEN
                RFLAGS.ZF ← 1;
                RAX ← SGX_INVALID_ISVSVN;
                GOTO EXIT;
        FI;
        // Include enclave identity?
        TMP_MRENCLAVE ← 0;
        IF (DS:RBX.KEYPOLICY.MRENCLAVE = 1)
            THEN TMP_MRENCLAVE ← TMP_CURRENTSECS.MRENCLAVE;
        FI;
        // Include enclave author?
        TMP_MRSIGNER ← 0;
        IF (DS:RBX.KEYPOLICY.MRSIGNER = 1)
            THEN TMP_MRSIGNER ← TMP_CURRENTSECS.MRSIGNER;
        FI;
        //Determine values key is based on
        TMP_KEYDEPENDENCIES.KEYNAME ← SEAL_KEY;
        TMP_KEYDEPENDENCIES.ISVPRODID ← TMP_CURRENTSECS.ISVPRODID;
        TMP_KEYDEPENDENCIES.ISVSVN ← DS:RBX.ISVSVN;
        TMP_KEYDEPENDENCIES.SGXOWNEREPOCH ← CR_SGXOWNEREPOCH;
        TMP_KEYDEPENDENCIES.ATTRIBUTES ← TMP_ATTRIBUTES;
        TMP_KEYDEPENDENCIES.ATTRIBUTESMASK ← DS:RBX.ATTRIBUTEMASK;
        TMP_KEYDEPENDENCIES.MRENCLAVE ← TMP_MRENCLAVE;
        TMP_KEYDEPENDENCIES.MRSIGNER ← TMP_MRSIGNER;
        TMP_KEYDEPENDENCIES.KEYID ← DS:RBX.KEYID;
        TMP_KEYDEPENDENCIES.SEAL_KEY_FUSES ← CR_SEAL_FUSES;
        TMP_KEYDEPENDENCIES.CPUSVN ← DS:RBX.CPUSVN;
        TMP_KEYDEPENDENCIES.PADDING ← TMP_CURRENTSECS.PADDING;
        TMP_KEYDEPENDENCIES.MISCSELECT ← TMP_MISCSELECT;
        TMP_KEYDEPENDENCIES.MISCMASK ← ~DS:RBX.MISCMASK;
        BREAK;
    REPORT_KEY:
```

```
        //Determine values key is based on
        TMP_KEYDEPENDENCIES.KEYNAME ← REPORT_KEY;
        TMP_KEYDEPENDENCIES.ISVPRODID ← 0;
        TMP_KEYDEPENDENCIES.ISVSVN ← 0;
        TMP_KEYDEPENDENCIES.SGXOWNEREPOCH ← CR_SGXOWNEREPOCH;
        TMP_KEYDEPENDENCIES.ATTRIBUTES ← TMP_CURRENTSECS.ATTRIBUTES;
        TMP_KEYDEPENDENCIES.ATTRIBUTESMASK ← 0;
        TMP_KEYDEPENDENCIES.MRENCLAVE ← TMP_CURRENTSECS.MRENCLAVE;
        TMP_KEYDEPENDENCIES.MRSIGNER ← 0;
        TMP_KEYDEPENDENCIES.KEYID ← DS:RBX.KEYID;
        TMP_KEYDEPENDENCIES.SEAL_KEY_FUSES ← CR_SEAL_FUSES;
        TMP_KEYDEPENDENCIES.CPUSVN ← CR_CPUSVN;
        TMP_KEYDEPENDENCIES.PADDING ← HARDCODED_PKCS1_5_PADDING;
        TMP_KEYDEPENDENCIES.MISCSELECT ← TMP_CURRENTSECS.MISCSELECT;
        TMP_KEYDEPENDENCIES.MISCMASK ← 0;
        BREAK;
EINITTOKEN_KEY:
        (* Check ENCLAVE has LAUNCH capability *)
        IF (TMP_CURRENTSECS.ATTRIBUTES.LAUNCHKEY = 0)
                THEN
                        RFLAGS.ZF ← 1;
                        RAX ← SGX_INVALID_ATTRIBUTE;
                        GOTO EXIT;
        FI;
        IF (DS:RBX.CPUSVN is beyond current CPU configuration)
                THEN
                        RFLAGS.ZF ← 1;
                        RAX ← SGX_INVALID_CPUSVN;
                        GOTO EXIT;
        FI;
        IF (DS:RBX.ISVSVN > TMP_CURRENTSECS.ISVSVN)
                THEN
                        RFLAGS.ZF ← 1;
                        RAX ← SGX_INVALID_ISVSVN;
                        GOTO EXIT;
        FI;
        (* Determine values key is based on *)
        TMP_KEYDEPENDENCIES.KEYNAME ← EINITTOKEN_KEY;
        TMP_KEYDEPENDENCIES.ISVPRODID ← TMP_CURRENTSECS.ISVPRODID
        TMP_KEYDEPENDENCIES.ISVSVN ← DS:RBX.ISVSVN;
        TMP_KEYDEPENDENCIES.SGXOWNEREPOCH ← CR_SGXOWNEREPOCH;
        TMP_KEYDEPENDENCIES.ATTRIBUTES ← TMP_ATTRIBUTES;
        TMP_KEYDEPENDENCIES.ATTRIBUTESMASK ← 0;
        TMP_KEYDEPENDENCIES.MRENCLAVE ← 0;
        TMP_KEYDEPENDENCIES.MRSIGNER ← TMP_CURRENTSECS.MRSIGNER;
        TMP_KEYDEPENDENCIES.KEYID ← DS:RBX.KEYID;
        TMP_KEYDEPENDENCIES.SEAL_KEY_FUSES ← CR_SEAL_FUSES;
        TMP_KEYDEPENDENCIES.CPUSVN ← DS:RBX.CPUSVN;
        TMP_KEYDEPENDENCIES.PADDING ← TMP_CURRENTSECS.PADDING;
        TMP_KEYDEPENDENCIES.MISCSELECT ← TMP_MISCSELECT;
        TMP_KEYDEPENDENCIES.MISCMASK ← 0;
        BREAK;
PROVISION_KEY:
(* Check ENCLAVE has PROVISIONING capability *)
```

```
        IF (TMP_CURRENTSECS.ATTRIBUTES.PROVISIONKEY = 0)
            THEN
                RFLAGS.ZF ← 1;
                RAX ← SGX_INVALID_ATTRIBUTE;
                GOTO EXIT;
        FI;
        IF (DS:RBX.CPUSVN is beyond current CPU configuration)
            THEN
                RFLAGS.ZF ← 1;
                RAX ← SGX_INVALID_CPUSVN;
                GOTO EXIT;
        FI;
        IF (DS:RBX.ISVSVN > TMP_CURRENTSECS.ISVSVN)
            THEN
                RFLAGS.ZF ← 1;
                RAX ← SGX_INVALID_ISVSVN;
                GOTO EXIT;
        FI;
        (* Determine values key is based on *)
        TMP_KEYDEPENDENCIES.KEYNAME ← PROVISION_KEY;
        TMP_KEYDEPENDENCIES.ISVPRODID ← TMP_CURRENTSECS.ISVPRODID;
        TMP_KEYDEPENDENCIES.ISVSVN ← DS:RBX.ISVSVN;
        TMP_KEYDEPENDENCIES.SGXOWNEREPOCH ← 0;
        TMP_KEYDEPENDENCIES.ATTRIBUTES ← TMP_ATTRIBUTES;
        TMP_KEYDEPENDENCIES.ATTRIBUTESMASK ← DS:RBX.ATTRIBUTEMASK;
        TMP_KEYDEPENDENCIES.MRENCLAVE ← 0;
        TMP_KEYDEPENDENCIES.MRSIGNER ← TMP_CURRENTSECS.MRSIGNER;
        TMP_KEYDEPENDENCIES.KEYID ← 0;
        TMP_KEYDEPENDENCIES.SEAL_KEY_FUSES ← 0;
        TMP_KEYDEPENDENCIES.CPUSVN ← DS:RBX.CPUSVN;
        TMP_KEYDEPENDENCIES.PADDING ← TMP_CURRENTSECS.PADDING;
        TMP_KEYDEPENDENCIES.MISCSELECT ← TMP_MISCSELECT;
        TMP_KEYDEPENDENCIES.MISCMASK ← ~DS:RBX.MISCMASK;
        BREAK;
    PROVISION_SEAL_KEY:
        (* Check ENCLAVE has PROVISIONING capability *)
        IF (TMP_CURRENTSECS.ATTRIBUTES.PROVISIONKEY = 0)
            THEN
                RFLAGS.ZF ← 1;
                RAX ← SGX_INVALID_ATTRIBUTE;
                GOTO EXIT;
        FI;
        IF (DS:RBX.CPUSVN is beyond current CPU configuration)
            THEN
                RFLAGS.ZF ← 1;
                RAX ← SGX_INVALID_CPUSVN;
                GOTO EXIT;
        FI;
        IF (DS:RBX.ISVSVN > TMP_CURRENTSECS.ISVSVN)
            THEN
                RFLAGS.ZF ← 1;
                RAX ← SGX_INVALID_ISVSVN;
                GOTO EXIT;
        FI;
```

```
        (* Determine values key is based on *)
        TMP_KEYDEPENDENCIES.KEYNAME ← PROVISION_SEAL_KEY;
        TMP_KEYDEPENDENCIES.ISVPRODID ← TMP_CURRENTSECS.ISVPRODID;
        TMP_KEYDEPENDENCIES.ISVSVN ← DS:RBX.ISVSVN;
        TMP_KEYDEPENDENCIES.SGXOWNEREPOCH ← 0;
        TMP_KEYDEPENDENCIES.ATTRIBUTES ← TMP_ATTRIBUTES;
        TMP_KEYDEPENDENCIES.ATTRIBUTESMASK ← DS:RBX.ATTRIBUTEMASK;
        TMP_KEYDEPENDENCIES.MRENCLAVE ← 0;
        TMP_KEYDEPENDENCIES.MRSIGNER ← TMP_CURRENTSECS.MRSIGNER;
        TMP_KEYDEPENDENCIES.KEYID ← 0;
        TMP_KEYDEPENDENCIES.SEAL_KEY_FUSES ← CR_SEAL_FUSES;
        TMP_KEYDEPENDENCIES.CPUSVN ← DS:RBX.CPUSVN;
        TMP_KEYDEPENDENCIES.PADDING ← TMP_CURRENTSECS.PADDING;
        TMP_KEYDEPENDENCIES.MISCSELECT ← TMP_MISCSELECT;
        TMP_KEYDEPENDENCIES.MISCMASK ← ~DS:RBX.MISCMASK;
        BREAK;
    DEFAULT:
        (* The value of KEYNAME is invalid *)
        RFLAGS.ZF ← 1;
        RAX ← SGX_INVALID_KEYNAME;
        GOTO EXIT:
ESAC;

(* Calculate the final derived key and output to the address in RCX *)
TMP_OUTPUTKEY ← derivekey(TMP_KEYDEPENDENCIES);
DS:RCX[15:0] ← TMP_OUTPUTKEY;
RAX ← 0;
RFLAGS.ZF ← 0;

EXIT:
RFLAGS.CF ← 0;
RFLAGS.PF ← 0;
RFLAGS.AF ← 0;
RFLAGS.OF ← 0;
RFLAGS.SF ← 0;
```

Flags Affected

ZF is cleared if successful, otherwise ZF is set. CF, PF, AF, OF, SF are cleared.

Protected Mode Exceptions

#GP(0)	If a memory operand effective address is outside the current enclave.
	If an effective address is not properly aligned.
	If an effective address is outside the DS segment limit.
	If KEYREQUEST format is invalid.
#PF(error code)	If a page fault occurs in accessing memory.

64-Bit Mode Exceptions

#GP(0)	If a memory operand effective address is outside the current enclave.
	If an effective address is not properly aligned.
	If an effective address is not canonical.
	If KEYREQUEST format is invalid.
#PF(error code)	If a page fault occurs in accessing memory operands.

EMODPE—Extend an EPC Page Permissions

Opcode/ Instruction	Op/En	64/32 bit Mode Support	CPUID Feature Flag	Description
EAX = 06H ENCLU[EMODPE]	IR	V/V	SGX2	This leaf function extends the access rights of an existing EPC page.

Instruction Operand Encoding

Op/En	EAX	RBX	RCX
IR	EMODPE (In)	Address of a SECINFO (In)	Address of the destination EPC page (In)

Description

This leaf function extends the access rights associated with an existing EPC page in the running enclave. THE RWX bits of the SECINFO parameter are treated as a permissions mask; supplying a value that does not extend the page permissions will have no effect. This instruction leaf can only be executed when inside the enclave.

RBX contains the effective address of a SECINFO structure while RCX contains the effective address of an EPC page. The table below provides additional information on the memory parameter of the EMODPE leaf function.

EMODPE Memory Parameter Semantics

SECINFO	EPCPAGE
Read access permitted by Non Enclave	Read access permitted by Enclave

The instruction faults if any of the following:

EMODPE Faulting Conditions

The operands are not properly aligned.	If security attributes of the SECINFO page make the page inaccessible.
The EPC page is locked by another thread.	RBX does not contain an effective address in an EPC page in the running enclave.
The EPC page is not valid.	RCX does not contain an effective address of an EPC page in the running enclave.
SECINFO contains an invalid request.	

Concurrency Restrictions

Table 40-60. Concurrency Restrictions of EMODPE with Other Intel® SGX Operations 1 of 2

Operation		EEXIT			EADD		EBLOCK		ECREATE	EDBGRD/WR		EENTER/ ERESUME			EEXTEND		EGETKEY		EINIT	ELDB/ELDU			EPA
	Param	TCS	SSA	SECS	Targ	SECS	Targ	SECS	SECS	Targ	SECS	TCS	SSA	SECS	Targ	SECS	Param	SECS	SECS	Targ	VA	SECS	VA
EMODPE	Targ		Y							Y			Y				Y						
	SECINFO		U							Y			U				U						

Table 40-61. Concurrency Restrictions of EMODPE with Other Intel® SGX Operations 2 of 2

Operation		EREMOVE		EREPORT		ETRACK	EWB			EAUG		EMODPE		EMODPR		EMODT		EACCEPT			EACCEPTCOPY		
	Param	Targ	SECS	Param	SECS	SECS	SRC	VA	SECS	Targ	SECS	Targ	SECINFO	Targ	SECS	Targ	SECS	Targ	SECINFO	SECS	Targ	SRC	SECINFO
EMODPE	Targ			Y								N	Y	N		N		N	Y			Y	Y
	SECINFO			U								Y	Y					Y	Y			Y	Y

Operation

Temp Variables in EMODPE Operational Flow

Name	Type	Size (bits)	Description
SCRATCH_SECINFO	SECINFO	512	Scratch storage for holding the contents of DS:RBX.

```
IF (DS:RBX is not 64Byte Aligned)
    THEN #GP(0); FI;

IF (DS:RCX is not 4KByte Aligned)
    THEN #GP(0); FI;

IF ((DS:RBX is not within CR_ELRANGE) or (DS:RCX is not within CR_ELRANGE) )
    THEN #GP(0); FI;

IF (DS:RBX does not resolve within an EPC)
    THEN #PF(DS:RBX); FI;

IF (DS:RCX does not resolve within an EPC)
    THEN #PF(DS:RCX); FI;

IF ( (EPCM(DS:RBX).VALID = 0) or (EPCM(DS:RBX).R = 0) or (EPCM(DS:RBX).PENDING ≠ 0) or (EPCM(DS:RBX).MODIFIED ≠ 0) or
    (EPCM(DS:RBX).BLOCKED ≠ 0) or (EPCM(DS:RBX).PT ≠ PT_REG) or (EPCM(DS:RBX).ENCLAVESECS ≠ CR_ACTIVE_SECS) or
    (EPCM(DS:RBX).ENCLAVEADDRESS ≠ (DS:RBX & ~0xFFF)) )
    THEN #PF(DS:RBX); FI;

SCRATCH_SECINFO ← DS:RBX;

(* Check for mis-configured SECINFO flags*)
IF (SCRATCH_SECINFO reserved fields are not zero )
    THEN #GP(0); FI;

(* Check security attributes of the EPC page *)
IF ( (EPCM(DS:RCX).VALID = 0) or (EPCM(DS:RCX).PENDING ≠ 0) or (EPCM(DS:RCX).MODIFIED ≠ 0) or
    (EPCM(DS:RCX).BLOCKED ≠ 0) or (EPCM(DS:RCX).PT ≠ PT_REG) or (EPCM(DS:RCX).ENCLAVESECS ≠ CR_ACTIVE_SECS) )
    THEN #PF(DS:RCX); FI;

(* Check the EPC page for concurrency *)
IF (EPC page in use by another SGX2 instruction)
    THEN #GP(0); FI;

(* Re-Check security attributes of the EPC page *)
IF ( (EPCM(DS:RCX).VALID = 0) or (EPCM(DS:RCX).PENDING ≠ 0) or (EPCM(DS:RCX).MODIFIED ≠ 0) or
    (EPCM(DS:RCX).PT ≠ PT_REG) or (EPCM(DS:RCX).ENCLAVESECS ≠ CR_ACTIVE_SECS) or
    (EPCM(DS:RCX).ENCLAVEADDRESS ≠ DS:RCX))
    THEN #PF(DS:RCX); FI;

(* Check for mis-configured SECINFO flags*)
IF ( (EPCM(DS:RCX).R = 0) and (SCRATCH_SECINFO.FLAGS.R = 0) and (SCRATCH_SECINFO.FLAGS.W ≠ 0) ))
    THEN #GP(0); FI;
```

```
(* Update EPCM permissions *)
EPCM(DS:RCX).R ← EPCM(DS:RCX).R | SCRATCH_SECINFO.FLAGS.R;
EPCM(DS:RCX).W ← EPCM(DS:RCX).W | SCRATCH_SECINFO.FLAGS.W;
EPCM(DS:RCX).X ← EPCM(DS:RCX).X | SCRATCH_SECINFO.FLAGS.X;
```

Flags Affected

None

Protected Mode Exceptions

#GP(0)	If a memory operand effective address is outside the DS segment limit.
	If a memory operand is not properly aligned.
	If a memory operand is locked.
#PF(error code)	If a page fault occurs in accessing memory operands.

64-Bit Mode Exceptions

#GP(0)	If a memory operand is non-canonical form.
	If a memory operand is not properly aligned.
	If a memory operand is locked.
#PF(error code)	If a page fault occurs in accessing memory operands.

EREPORT—Create a Cryptographic Report of the Enclave

Opcode/ Instruction	Op/En	64/32 bit Mode Support	CPUID Feature Flag	Description
EAX = 00H ENCLU[EREPORT]	IR	V/V	SGX1	This leaf function creates a cryptographic report of the enclave.

Instruction Operand Encoding

Op/En	EAX	RBX	RCX	RDX
IR	EREPORT (In)	Address of TARGETINFO (In)	Address of REPORTDATA (In)	Address where the REPORT is written to in an OUTPUTDATA (In)

Description

This leaf function creates a cryptographic REPORT that describes the contents of the enclave. This instruction leaf can only be executed when inside the enclave. The cryptographic report can be used by other enclaves to determine that the enclave is running on the same platform.

RBX contains the effective address of the MRENCLAVE value of the enclave that will authenticate the REPORT output, using the REPORT key delivered by EGETKEY command for that enclave. RCX contains the effective address of a 64-byte REPORTDATA structure, which allows the caller of the instruction to associate data with the enclave from which the instruction is called. RDX contains the address where the REPORT will be output by the instruction.

EREPORT Memory Parameter Semantics

TARGETINFO	REPORTDATA	OUTPUTDATA
Read access by Enclave	Read access by Enclave	Read/Write access by Enclave

This instruction leaf perform the following:

1. Validate the 3 operands (RBX, RCX, RDX) are inside the enclave.
2. Compute a report key for the target enclave, as indicated by the value located in RBX(TARGETINFO).
3. Assemble the enclave SECS data to complete the REPORT structure (including the data provided using the RCX (REPORTDATA) operand).
4. Computes a crytpographic hash over REPORT structure.
5. Add the computed hash to the REPORT structure.
6. Output the completed REPORT structure to the address in RDX (OUTPUTDATA).

The instruction fails if the operands are not properly aligned.

CR_REPORT_KEYID, used to provide key wearout protection, is populated with a statistically unique value on boot of the platform by a trusted entity within the SGX TCB.

The instruction faults if any of the following:

EREPORT Faulting Conditions

An effective address not properly aligned.	An memory address does not resolve in an EPC page.
If accessing an invalid EPC page.	If the EPC page is blocked.
May page fault.	

Concurrency Restrictions

Table 40-62. Concurrency Restrictions of EREPORT with Other Intel® SGX Operations 1 of 2

Operation		EEXIT			EADD		EBLOCK		ECREATE	EDBGRD/WR		EENTER/ERESUME			EEXTEND		EGETKEY		EINIT	ELDB/ELDU			EPA
	Param	TCS	SSA	SECS	Targ	SECS	Targ	SECS	SECS	Targ	SECS	TCS	SSA	SECS	Targ	SECS	Param	SECS	SECS	Targ	VA	SECS	VA
EREPORT	Param		U							Y			U				U						
	SECS			Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y

Table 40-63. Concurrency Restrictions of EREPORT with Other Intel® SGX Operations 2 of 2

Operation		EREMOVE		EREPORT		ETRACK	EWB			EAUG		EMODPE		EMODPR		EMODT		EACCEPT			EACCEPTCOPY		
	Param	Targ	SECS	Param	SECS	SECS	SRC	VA	SECS	Targ	SECS	Targ	SECINFO	Targ	SECS	Targ	SECS	Targ	SECINFO	SECS	Targ	SRC	SECINFO
EREPORT	Param			U								Y	U					Y	U			Y	U
	SECS	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y

Operation

Temp Variables in EREPORT Operational Flow

Name	Type	Size (bits)	Description
TMP_ATTRIBUTES		32	Physical address of SECS of the enclave to which source operand belongs.
TMP_CURRENTSECS			Address of the SECS for the currently executing enclave.
TMP_KEYDEPENDENCIES			Temp space for key derivation.
TMP_REPORTKEY		128	REPORTKEY generated by the instruction.
TMP_REPORT		3712	

```
TMP_MODE64 ← ((IA32_EFER.LMA = 1) && (CS.L = 1));

(* Address verification for TARGETINFO (RBX) *)
IF ( (DS:RBX is not 512Byte Aligned) or (DS:RBX is not within CR_ELRANGE) )
    THEN #GP(0); FI;

IF (DS:RBX does not resolve within an EPC)
    THEN #PF(DS:RBX); FI;

IF (EPCM(DS:RBX). VALID = 0)
    THEN #PF(DS:RBX); FI;

IF (EPCM(DS:RBX).BLOCKED = 1) )
    THEN #PF(DS:RBX); FI;

(* Check page parameters for correctness *)
IF ( (EPCM(DS:RBX).PT ≠ PT_REG) or (EPCM(DS:RBX).ENCLAVESECS ≠ CR_ACTIVE_SECS) or (EPCM(DS:RBX).PENDING = 1) or
    (EPCM(DS:RBX).MODIFIED = 1) or (EPCM(DS:RBX).ENCLAVEADDRESS ≠ (DS:RBX & ~0FFFH) ) or (EPCM(DS:RBX).R = 0) )
```

```
    THEN #PF(DS:RBX);
FI;

(* Address verification for REPORTDATA (RCX) *)
IF ( (DS:RCX is not 128Byte Aligned) or (DS:RCX is not within CR_ELRANGE) )
    THEN #GP(0); FI;

IF (DS:RCX does not resolve within an EPC)
    THEN #P(DS:RCX); FI;

IF (EPCM(DS:RCX). VALID = 0)
    THEN #PF(DS:RCX); FI;

IF (EPCM(DS:RCX).BLOCKED = 1) )
    THEN #PF(DS:RCX); FI;

(* Check page parameters for correctness *)
IF ( (EPCM(DS:RCX).PT ≠ PT_REG) or (EPCM(DS:RCX).ENCLAVESECS ≠ CR_ACTIVE_SECS) or (EPCM(DS:RCX).PENDING = 1) or
    (EPCM(DS:RCX).MODIFIED = 1) or (EPCM(DS:RCX).ENCLAVEADDRESS ≠ (DS:RCX & ~0FFFH) ) or (EPCM(DS:RCX).R = 0) )
    THEN #PF(DS:RCX);
FI;

(* Address verification for OUTPUTDATA (RDX) *)
IF ( (DS:RDX is not 512Byte Aligned) or (DS:RDX is not within CR_ELRANGE) )
    THEN #GP(0); FI;

IF (DS:RDX does not resolve within an EPC)
    THEN #PF(DS:RDX); FI;

IF (EPCM(DS:RDX). VALID = 0)
    THEN #PF(DS:RDX); FI;

IF (EPCM(DS:RDX).BLOCKED = 1) )
    THEN #PF(DS:RDX); FI;

(* Check page parameters for correctness *)
IF ( (EPCM(DS:RDX).PT ≠ PT_REG) or (EPCM(DS:RDX).ENCLAVESECS ≠ CR_ACTIVE_SECS) or (EPCM(DS:RCX).PENDING = 1) or
    (EPCM(DS:RCX).MODIFIED = 1) or (EPCM(DS:RDX).ENCLAVEADDRESS ≠ (DS:RDX & ~0FFFH) ) or (EPCM(DS:RDX).W = 0) )
    THEN #PF(DS:RDX);
FI;

(* REPORT MAC needs to be computed over data which cannot be modified *)
TMP_REPORT.CPUSVN ← CR_CPUSVN;
TMP_REPORT.ISVPRODID ← TMP_CURRENTSECS.ISVPRODID;
TMP_REPORT.ISVSVN ← TMP_CURRENTSECS..ISVSVN;
TMP_REPORT.ATTRIBUTES ← TMP_CURRENTSECS.ATTRIBUTES;
TMP_REPORT.REPORTDATA ← DS:RCX[511:0];
TMP_REPORT.MRENCLAVE ← TMP_CURRENTSECS.MRENCLAVE;
TMP_REPORT.MRSIGNER ← TMP_CURRENTSECS.MRSIGNER;
TMP_REPORT.MRRESERVED ← 0;
TMP_REPORT.KEYID[255:0] ← CR_REPORT_KEYID;
TMP_REPORT.MISCSELECT ← TMP_CURRENTSECS.MISCSELECT;
```

```
(* Derive the report key *)
TMP_KEYDEPENDENCIES.KEYNAME ← REPORT_KEY;
TMP_KEYDEPENDENCIES.ISVPRODID ← 0;
TMP_KEYDEPENDENCIES.ISVSVN ← 0;
TMP_KEYDEPENDENCIES.SGXOWNEREPOCH ← CR_SGXOWNEREPOCH;
TMP_KEYDEPENDENCIES.ATTRIBUTES ← DS:RBX.ATTRIBUTES;
TMP_KEYDEPENDENCIES.ATTRIBUTESMASK ← 0;
TMP_KEYDEPENDENCIES.MRENCLAVE ← DS:RBX.MEASUREMENT;
TMP_KEYDEPENDENCIES.MRSIGNER ← 0;
TMP_KEYDEPENDENCIES.KEYID ← TMP_REPORT.KEYID;
TMP_KEYDEPENDENCIES.SEAL_KEY_FUSES ← CR_SEAL_FUSES;
TMP_KEYDEPENDENCIES.CPUSVN ← CR_CPUSVN;
TMP_KEYDEPENDENCIES.PADDING ← TMP_CURRENTSECS.PADDING;
TMP_KEYDEPENDENCIES.MISCSELECT ← DS:RBX.MISCSELECT;
TMP_KEYDEPENDENCIES.MISCMASK ← 0;

(* Calculate the derived key*)
TMP_REPORTKEY ← derive_key(TMP_KEYDEPENDENCIES);

(* call cryptographic CMAC function, CMAC data are not including MAC&KEYID *)
TMP_REPORT.MAC ← cmac(TMP_REPORTKEY, TMP_REPORT[3071:0] );
DS:RDX[3455: 0] ← TMP_REPORT;
```

Flags Affected

None

Protected Mode Exceptions

#GP(0)	If the address in RCS is outside the DS segment limit.
	If a memory operand is not properly aligned.
	If a memory operand is not in the current enclave.
#PF(error code)	If a page fault occurs in accessing memory operands.

64-Bit Mode Exceptions

#GP(0)	If RCX is non-canonical form.
	If a memory operand is not properly aligned.
	If a memory operand is not in the current enclave.
#PF(error code)	If a page fault occurs in accessing memory operands.

ERESUME—Re-Enters an Enclave

Opcode/ Instruction	Op/En	64/32 bit Mode Support	CPUID Feature Flag	Description
EAX = 03H ENCLU[ERESUME]	IR	V/V	SGX1	This leaf function is used to re-enter an enclave after an interrupt.

Instruction Operand Encoding

Op/En	RAX	RBX	RCX
IR	ERESUME (In)	Address of a TCS (In)	Address of AEP (In)

Description

The ENCLU[ERESUME] instruction resumes execution of an enclave that was interrupted due to an exception or interrupt, using the machine state previously stored in the SSA.

ERESUME Memory Parameter Semantics

TCS
Enclave read/write access

The instruction faults if any of the following:

Address in RBX is not properly aligned.	Any TCS.FLAGS's must-be-zero bit is not zero.
TCS pointed to by RBX is not valid or available or locked.	Current 32/64 mode does not match the enclave mode in SECS.ATTRIBUTES.MODE64.
The SECS is in use by another enclave.	Either of TCS-specified FS and GS segment is not a subset of the current DS segment.
Any one of DS, ES, CS, SS is not zero.	If XSAVE available, CR4.OSXSAVE = 0, but SECS.ATTRIBUTES.XFRM ≠ 3.
CR4.OSFXSR ≠ 1.	If CR4.OSXSAVE = 1, SECS.ATTRIBUTES.XFRM is not a subset of XCR0.
Offsets 520-535 of the XSAVE area not 0.	The bit vector stored at offset 512 of the XSAVE area must be a subset of SECS.ATTRIBUTES.XFRM.
The SSA frame is not valid or in use.	

The following operations are performed by ERESUME:

- RSP and RBP are saved in the current SSA frame on EENTER and are automatically restored on EEXIT or an asynchronous exit due to any Interrupt event.
- The AEP contained in RCX is stored into the TCS for use by AEXs.FS and GS (including hidden portions) are saved and new values are constructed using TCS.OFSBASE/GSBASE (32 and 64-bit mode) and TCS.OFSLIMIT/GSLIMIT (32-bit mode only). The resulting segments must be a subset of the DS segment.
- If CR4.OSXSAVE == 1, XCR0 is saved and replaced by SECS.ATTRIBUTES.XFRM. The effect of RFLAGS.TF depends on whether the enclave entry is opt-in or opt-out (see Section 42.1.2):
  - — On opt-out entry, TF is saved and cleared (it is restored on EEXIT or AEX). Any attempt to set TF via a POPF instruction while inside the enclave clears TF (see Section 42.2.5).
  - — On opt-in entry, a single-step debug exception is pended on the instruction boundary immediately after EENTER (see Section 42.2.3).
- All code breakpoints that do not overlap with ELRANGE are also suppressed. If the entry is an opt-out entry, all code and data breakpoints that overlap with the ELRANGE are suppressed.

- On opt-out entry, a number of performance monitoring counters and behaviors are modified or suppressed (see Section 42.2.3):
  - All performance monitoring activity on the current thread is suppressed except for incrementing and firing of FIXED_CTR1 and FIXED_CTR2.
  - PEBS is suppressed.
  - AnyThread counting on other threads is demoted to MyThread mode and IA32_PERF_GLOBAL_STATUS[60] on that thread is set.
  - If the opt-out entry on a hardware thread results in suppression of any performance monitoring, then the processor sets IA32_PERF_GLOBAL_STATUS[60] and IA32_PERF_GLOBAL_STATUS[63].

Concurrency Restrictions

Table 40-64. Concurrency Restrictions of ERESUME with Intel® SGX Instructions - 1of 2

Operation		EEXIT			EADD		EBLOCK		ECREATE	EDBGRD/WR		EENTER/ERESUME			EEXTEND		EGETKEY		EINIT	ELDB/ELDU			EPA
	Param	Targ	VA	SECS	Targ	SECS	Targ	SECS	SECS	Targ	SECS	TCS	SSA	SECS	Targ	SECS	Param	SECS	SECS	Targ	VA	SECS	VA
ERESUME	TCS	N			N				N	Y		N								N			N
	SSA		U							Y		Y	U				U						
	SECS			Y	Y	N	Y	Y	Y	Y	Y	Y	Y	Y	Y	N	Y	Y	N	Y	Y	Y	Y

Table 40-65. Concurrency Restrictions of ERESUME with Intel® SGX Instructions - 2 of 2

Operation		EREMOVE		EREPORT		ETRACK	EWB			EAUG		EMODPE		EMODPR		EMODT		EACCEPT			EACCEPTCOPY		
	Param	Targ	SECS	Param	SECS	SECS	SRC	VA	SECS	Targ	SECS	Targ	SECINFO	Targ	SECS	Targ	SECS	Targ	SECINFO	SECS	Targ	SRC	SECINFO
ERESUME	TCS	N					N			N						N							
	SSA			U								Y	U					Y	U			U	U
	SECS	Y	Y	Y	Y	Y	Y		Y		Y				Y		Y			Y			

Operation

Temp Variables in ERESUME Operational Flow

Name	Type	Size	Description
TMP_FSBASE	Effective Address	32/64	Proposed base address for FS segment.
TMP_GSBASE	Effective Address	32/64	Proposed base address for FS segment.
TMP_FSLIMIT	Effective Address	32/64	Highest legal address in proposed FS segment.
TMP_GSLIMIT	Effective Address	32/64	Highest legal address in proposed GS segment.
TMP_TARGET	Effective Address	32/64	Address of first instruction inside enclave at which execution is to resume.
TMP_SECS	Effective Address	32/64	Physical address of SECS for this enclave.
TMP_SSA	Effective Address	32/64	Address of current SSA frame.
TMP_XSIZE	integer	64	Size of XSAVE area based on SECS.ATTRIBUTES.XFRM.
TMP_SSA_PAGE	Effective Address	32/64	Pointer used to iterate over the SSA pages in the current frame.
TMP_GPR	Effective Address	32/64	Address of the GPR area within the current SSA frame.
TMP_BRANCH_RECORD	LBR Record		From/to addresses to be pushed onto the LBR stack.

TMP_MODE64 ← ((IA32_EFER.LMA = 1) && (CS.L = 1));

```
(* Make sure DS is usable, expand up *)
IF (TMP_MODE64 = 0 and (DS not usable or ( ( DS[S] = 1) and (DS[bit 11] = 0) and DS[bit 10] = 1) ) ) )
    THEN #GP(0); FI;

(* Check that CS, SS, DS, ES.base is 0 *)
IF (TMP_MODE64 = 0)
    THEN
        IF(CS.base ≠ 0 or DS.base ≠ 0) #GP(0); FI;
        IF(ES usable and ES.base ≠ 0) #GP(0); FI;
        IF(SS usable and SS.base ≠ 0) #GP(0); FI;
        IF(SS usable and SS.B = 0) #GP(0); FI;
FI;

IF (DS:RBX is not 4KByte Aligned)
    THEN #GP(0); FI;

IF (DS:RBX does not resolve within an EPC)
    THEN #PF(DS:RBX); FI;

(* Check AEP is canonical*)
IF (TMP_MODE64 = 1 and (CS:RCX is not canonical) )
    THEN #GP(0); FI;

(* Check concurrency of TCS operation*)
IF (Other Intel SGX instructions is operating on TCS)
    THEN #GP(0); FI;

(* TCS verification *)
IF (EPCM(DS:RBX).VALID = 0)
    THEN #PF(DS:RBX); FI;

IF (EPCM(DS:RBX).BLOCKED = 1)
    THEN #PF(DS:RBX); FI;

IF ((EPCM(DS:RBX).PENDING = 1) or (EPCM(DS:RBX).MODIFIED = 1))
    THEN #PF(DS:RBX); FI;

IF ( (EPCM(DS:RBX).ENCLAVEADDRESS ≠ DS:RBX) or (EPCM(DS:RBX).PT ≠ PT_TCS) )
    THEN #PF(DS:RBX); FI;

IF ( (DS:RBX).OSSA is not 4KByte Aligned)
    THEN #GP(0); FI;

(* Check proposed FS and GS *)
IF ( ( (DS:RBX).OFSBASE is not 4KByte Aligned) or ( (DS:RBX).OGSBASE is not 4KByte Aligned) )
    THEN #GP(0); FI;

(* Get the SECS for the enclave in which the TCS resides *)
TMP_SECS ← Address of SECS for TCS;

(* Make sure that the FLAGS field in the TCS does not have any reserved bits set *)
IF ( ( (DS:RBX).FLAGS & & FFFFFFFFFFFFFFFEH) ≠ 0)
    THEN #GP(0); FI;
```

```
(* SECS must exist and enclave must have previously been EINITted *)
IF (the enclave is not already initialized)
    THEN #GP(0); FI;

(* make sure the logical processor's operating mode matches the enclave *)
IF ( (TMP_MODE64 ≠ TMP_SECS.ATTRIBUTES.MODE64BIT) )
    THEN #GP(0); FI;

IF (CR4.OSFXSR = 0)
    THEN #GP(0); FI;

(* Check for legal values of SECS.ATTRIBUTES.XFRM *)
IF (CR4.OSXSAVE = 0)
    THEN
        IF (TMP_SECS.ATTRIBUTES.XFRM ≠ 03H) THEN #GP(0); FI;
    ELSE
        IF ( (TMP_SECS.ATTRIBUTES.XFRM & XCR0) ≠ TMP_SECS.ATTRIBUTES.XFRM) THEN #GP(0); FI;
FI;

(* Make sure the SSA contains at least one active frame *)
IF ( (DS:RBX).CSSA = 0)
    THEN #GP(0); FI;

(* Compute linear address of SSA frame *)
TMP_SSA ← (DS:RBX).OSSA + TMP_SECS.BASEADDR + 4096 * TMP_SECS.SSAFRAMESIZE * ( (DS:RBX).CSSA - 1);
TMP_XSIZE ← compute_XSAVE_frame_size(TMP_SECS.ATTRIBUTES.XFRM);

FOR EACH TMP_SSA_PAGE = TMP_SSA to TMP_SSA + TMP_XSIZE
    (* Check page is read/write accessible *)
    Check that DS:TMP_SSA_PAGE is read/write accessible;
    If a fault occurs, release locks, abort and deliver that fault;
    IF (DS:TMP_SSA_PAGE does not resolve to EPC page)
        THEN #PF(DS:TMP_SSA_PAGE); FI;
    IF (EPCM(DS:TMP_SSA_PAGE).VALID = 0)
        THEN #PF(DS:TMP_SSA_PAGE); FI;
    IF (EPCM(DS:TMP_SSA_PAGE).BLOCKED = 1)
        THEN #PF(DS:TMP_SSA_PAGE); FI;
    IF ((EPCM(DS:TMP_SSA_PAGE).PENDING = 1) or (EPCM(DS:TMP_SSA_PAGE_.MODIFIED = 1))
        THEN #PF(DS:TMP_SSA_PAGE); FI;
    IF ( ( EPCM(DS:TMP_SSA_PAGE).ENCLAVEADDRESS ≠ DS:TMPSSA_PAGE) or (EPCM(DS:TMP_SSA_PAGE).PT ≠ PT_REG) or
        (EPCM(DS:TMP_SSA_PAGE).ENCLAVESECS ≠ EPCM(DS:RBX).ENCLAVESECS) or
        (EPCM(DS:TMP_SSA_PAGE).R = 0) or (EPCM(DS:TMP_SSA_PAGE).W = 0) )
        THEN #PF(DS:TMP_SSA_PAGE); FI;
    CR_XSAVE_PAGE_n ← Physical_Address(DS:TMP_SSA_PAGE);
ENDFOR

(* Compute address of GPR area*)
TMP_GPR ← TMP_SSA + 4096 * DS:TMP_SECS.SSAFRAMESIZE -- sizeof(GPRSGX_AREA);
Check that DS:TMP_SSA_PAGE is read/write accessible;
If a fault occurs, release locks, abort and deliver that fault;
IF (DS:TMP_GPR does not resolve to EPC page)
    THEN #PF(DS:TMP_GPR); FI;
IF (EPCM(DS:TMP_GPR).VALID = 0)
    THEN #PF(DS:TMP_GPR); FI;
```

```
IF (EPCM(DS:TMP_GPR).BLOCKED = 1)
    THEN #PF(DS:TMP_GPR); FI;
IF ((EPCM(DS:TMP_GPR).PENDING = 1) or (EPCM(DS:TMP_GPR).MODIFIED = 1))
    THEN #PF(DS:TMP_GPR); FI;
IF ( ( EPCM(DS:TMP_GPR).ENCLAVEADDRESS ≠ DS:TMP_GPR) or (EPCM(DS:TMP_GPR).PT ≠ PT_REG) or
    (EPCM(DS:TMP_GPR).ENCLAVESECS ≠ EPCM(DS:RBX).ENCLAVESECS) or
    (EPCM(DS:TMP_GPR).R = 0) or (EPCM(DS:TMP_GPR).W = 0) )
    THEN #PF(DS:TMP_GPR); FI;

IF (TMP_MODE64 = 0)
    THEN
        IF (TMP_GPR + (GPR_SIZE -1) is not in DS segment) THEN #GP(0); FI;
FI;

CR_GPR_PA ← Physical_Address (DS: TMP_GPR);

TMP_TARGET ← (DS:TMP_GPR).RIP;
IF (TMP_MODE64 = 1)
    THEN
        IF (TMP_TARGET is not canonical) THEN #GP(0); FI;
    ELSE
        IF (TMP_TARGET > CS limit) THEN #GP(0); FI;
FI;

(* Check proposed FS/GS segments fall within DS *)
IF (TMP_MODE64 = 0)
    THEN
        TMP_FSBASE ← (DS:RBX).OFSBASE + TMP_SECS.BASEADDR;
        TMP_FSLIMIT ← (DS:RBX).OFSBASE + TMP_SECS.BASEADDR + (DS:RBX).FSLIMIT;
        TMP_GSBASE ← (DS:RBX).OGSBASE + TMP_SECS.BASEADDR;
        TMP_GSLIMIT ← (DS:RBX).OGSBASE + TMP_SECS.BASEADDR + (DS:RBX).GSLIMIT;
        (* if FS wrap-around, make sure DS has no holes*)
        IF (TMP_FSLIMIT < TMP_FSBASE)
            THEN
                IF (DS.limit < 4GB) THEN #GP(0); FI;
            ELSE
                IF (TMP_FSLIMIT > DS.limit) THEN #GP(0); FI;
        FI;
        (* if GS wrap-around, make sure DS has no holes*)
        IF (TMP_GSLIMIT < TMP_GSBASE)
            THEN
                IF (DS.limit < 4GB) THEN #GP(0); FI;
            ELSE
                IF (TMP_GSLIMIT > DS.limit) THEN #GP(0); FI;
        FI;
    ELSE
        TMP_FSBASE ← (DS:RBX).OFSBASE + TMP_SECS.BASEADDR;
        TMP_GSBASE ← (DS:RBX).OGSBASE + TMP_SECS.BASEADDR;
        IF ( (TMP_FSBASE is not canonical) or (TMP_GSBASE is not canonical))
            THEN #GP(0); FI;
FI;

(* Ensure the enclave is not already active and this thread is the only one using the TCS*)
IF (DS:RBX.STATE = ACTIVE))
```

```
    THEN #GP(0); FI;

(* SECS.ATTRIBUTES.XFRM selects the features to be saved. *)
(* CR_XSAVE_PAGE_n: A list of 1 or more physical address of pages that contain the XSAVE area. *)
XRSTOR(TMP_MODE64, SECS.ATTRIBUTES.XFRM, CR_XSAVE_PAGE_n);

IF (XRSTOR failed with #GP)
    THEN
        DS:RBX.STATE ← INACTIVE;
        #GP(0);
FI;

CR_ENCLAVE_MODE ← 1;
CR_ACTIVE_SECS ← TMP_SECS;
CR_ELRANGE ← (TMP_SECS.BASEADDR, TMP_SECS.SIZE);

(* Save sate for possible AEXs *)
CR_TCS_PA ← Physical_Address (DS:RBX);
CR_TCS_LA ← RBX;
CR_TCS_LA.AEP ← RCX;

(* Save the hidden portions of FS and GS *)
CR_SAVE_FS_selector ← FS.selector;
CR_SAVE_FS_base ← FS.base;
CR_SAVE_FS_limit ← FS.limit;
CR_SAVE_FS_access_rights ← FS.access_rights;
CR_SAVE_GS_selector ← GS.selector;
CR_SAVE_GS_base ← GS.base;
CR_SAVE_GS_limit ← GS.limit;
CR_SAVE_GS_access_rights ← GS.access_rights;

RIP ← TMP_TARGET;

Restore_GPRs from DS:TMP_GPR;

(*Restore the RFLAGS values from SSA*)
RFLAGS.CF ← DS:TMP_GPR.RFLAGS.CF;
RFLAGS.PF ← DS:TMP_GPR.RFLAGS.PF;
RFLAGS.AF ← DS:TMP_GPR.RFLAGS.AF;
RFLAGS.ZF ← DS:TMP_GPR.RFLAGS.ZF;
RFLAGS.SF ← DS:TMP_GPR.RFLAGS.SF;
RFLAGS.DF ← DS:TMP_GPR.RFLAGS.DF;
RFLAGS.OF ← DS:TMP_GPR.RFLAGS.OF;
RFLAGS.NT ← DS:TMP_GPR.RFLAGS.NT;
RFLAGS.AC ← DS:TMP_GPR.RFLAGS.AC;
RFLAGS.ID ← DS:TMP_GPR.RFLAGS.ID;
RFLAGS.RF ← DS:TMP_GPR.RFLAGS.RF;
RFLAGS.VM ← 0;
IF (RFLAGS.IOPL = 3)
    THEN RFLAGS.IF = DS:TMP_GPR.IF; FI;

IF (TCS.FLAGS.OPTIN = 0)
    THEN RFLAGS.TF = 0; FI;
```

```
(* If XSAVE is enabled, save XCR0 and replace it with SECS.ATTRIBUTES.XFRM*)
IF (CR4.OSXSAVE = 1)
    CR_SAVE_XCR0 ← XCR0;
    XCR0 ← TMP_SECS.ATTRIBUTES.XFRM;
FI;

(* Pop the SSA stack*)
(DS:RBX).CSSA ← (DS:RBX).CSSA -1;

(* Do the FS/GS swap *)
FS.base ← TMP_FSBASE;
FS.limit ← DS:RBX.FSLIMIT;
FS.type ← 0001b;
FS.W ← DS.W;
FS.S ← 1;
FS.DPL ← DS.DPL;
FS.G ← 1;
FS.B ← 1;
FS.P ← 1;
FS.AVL ← DS.AVL;
FS.L ← DS.L;
FS.unusable ← 0;
FS.selector ← 0BH;

GS.base ← TMP_GSBASE;
GS.limit ← DS:RBX.GSLIMIT;
GS.type ← 0001b;
GS.W ← DS.W;
GS.S ← 1;
GS.DPL ← DS.DPL;
GS.G ← 1;
GS.B ← 1;
GS.P ← 1;
GS.AVL ← DS.AVL;
GS.L ← DS.L;
GS.unusable ← 0;
GS.selector ← 0BH;

CR_DBGOPTIN ← TCS.FLAGS.DBGOPTIN;
Suppress all code breakpoints that are outside ELRANGE;

IF (CR_DBGOPTIN = 0)
    THEN
        Suppress all code breakpoints that overlap with ELRANGE;
        CR_SAVE_TF ← RFLAGS.TF;
        RFLAGS.TF ← 0;
        Suppress any MTF VM exits during execution of the enclave;
        Clear all pending debug exceptions;
        Clear any pending MTF VM exit;
    ELSE
        Clear all pending debug exceptions;
        Clear pending MTF VM exits;
FI;
```

```
(* Assure consistent translations *)
Flush_linear_context;
Clear_Monitor_FSM;
Allow_front_end_to_begin_fetch_at_new_RIP;
```

Flags Affected

RFLAGS.TF is cleared on opt-out entry

Protected Mode Exceptions

#GP(0)	If DS:RBX is not page aligned.
	If the enclave is not initialized.
	If the thread is not in the INACTIVE state.
	If CS, DS, ES or SS bases are not all zero.
	If executed in enclave mode.
	If part or all of the FS or GS segment specified by TCS is outside the DS segment.
	If any reserved field in the TCS FLAG is set.
	If the target address is not within the CS segment.
	If CR4.OSFXSR = 0.
	If CR4.OSXSAVE = 0 and SECS.ATTRIBUTES.XFRM ≠ 3.
	If CR4.OSXSAVE = 1and SECS.ATTRIBUTES.XFRM is not a subset of XCR0.
#PF(error code)	If a page fault occurs in accessing memory.
	If DS:RBX does not point to a valid TCS.
	If one or more pages of the current SSA frame are not readable/writable, or do not resolve to a valid PT_REG EPC page.

64-Bit Mode Exceptions

#GP(0)	If DS:RBX is not page aligned.
	If the enclave is not initialized.
	If the thread is not in the INACTIVE state.
	If CS, DS, ES or SS bases are not all zero.
	If executed in enclave mode.
	If part or all of the FS or GS segment specified by TCS is outside the DS segment.
	If any reserved field in the TCS FLAG is set.
	If the target address is not canonical.
	If CR4.OSFXSR = 0.
	If CR4.OSXSAVE = 0 and SECS.ATTRIBUTES.XFRM ≠ 3.
	If CR4.OSXSAVE = 1and SECS.ATTRIBUTES.XFRM is not a subset of XCR0.
#PF(error code)	If a page fault occurs in accessing memory operands.
	If DS:RBX does not point to a valid TCS.
	If one or more pages of the current SSA frame are not readable/writable, or do not resolve to a valid PT_REG EPC page.

This page was intentionally left blank.

26. Updates to Appendix A, Volume 3D

Change bars show changes to Appendix A of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3D:* System Programming Guide, Part 4.

Changes to chapter: Correction to Section A.6 "Miscellaneous Data".

APPENDIX A
VMX CAPABILITY REPORTING FACILITY

The ability of a processor to support VMX operation and related instructions is indicated by CPUID.1:ECX.VMX[bit 5] = 1. A value 1 in this bit indicates support for VMX features.

Support for specific features detailed in Chapter 26 and other VMX chapters is determined by reading values from a set of capability MSRs. These MSRs are indexed starting at MSR address 480H. VMX capability MSRs are read-only; an attempt to write them (with WRMSR) produces a general-protection exception (#GP(0)). They do not exist on processors that do not support VMX operation; an attempt to read them (with RDMSR) on such processors produces a general-protection exception (#GP(0)).

A.1 BASIC VMX INFORMATION

The IA32_VMX_BASIC MSR (index 480H) consists of the following fields:

- Bits 30:0 contain the 31-bit VMCS revision identifier used by the processor. Processors that use the same VMCS revision identifier use the same size for VMCS regions (see subsequent item on bits 44:32).[1]
- Bit 31 is always 0.
- Bits 44:32 report the number of bytes that software should allocate for the VMXON region and any VMCS region. It is a value greater than 0 and at most 4096 (bit 44 is set if and only if bits 43:32 are clear).
- Bit 48 indicates the width of the physical addresses that may be used for the VMXON region, each VMCS, and data structures referenced by pointers in a VMCS (I/O bitmaps, virtual-APIC page, MSR areas for VMX transitions). If the bit is 0, these addresses are limited to the processor's physical-address width.[2] If the bit is 1, these addresses are limited to 32 bits. This bit is always 0 for processors that support Intel 64 architecture.
- If bit 49 is read as 1, the logical processor supports the dual-monitor treatment of system-management interrupts and system-management mode. See Section 34.15 for details of this treatment.
- Bits 53:50 report the memory type that should be used for the VMCS, for data structures referenced by pointers in the VMCS (I/O bitmaps, virtual-APIC page, MSR areas for VMX transitions), and for the MSEG header. If software needs to access these data structures (e.g., to modify the contents of the MSR bitmaps), it can configure the paging structures to map them into the linear-address space. If it does so, it should establish mappings that use the memory type reported bits 53:50 in this MSR.[3]

  As of this writing, all processors that support VMX operation indicate the write-back type. The values used are given in Table A-1.

Table A-1. Memory Types Recommended for VMCS and Related Data Structures

Value(s)	Field
0	Uncacheable (UC)
1–5	Not used
6	Write Back (WB)
7–15	Not used

1. Earlier versions of this manual specified that the VMCS revision identifier was a 32-bit field in bits 31:0 of this MSR. For all processors produced prior to this change, bit 31 of this MSR was read as 0.
2. On processors that support Intel 64 architecture, the pointer must not set bits beyond the processor's physical address width.
3. Alternatively, software may map any of these regions or structures with the UC memory type. (This may be necessary for the MSEG header.) Doing so is discouraged unless necessary as it will cause the performance of software accesses to those structures to suffer.

If software needs to access these data structures (e.g., to modify the contents of the MSR bitmaps), it can configure the paging structures to map them into the linear-address space. If it does so, it should establish mappings that use the memory type reported in this MSR.[1]

- If bit 54 is read as 1, the processor reports information in the VM-exit instruction-information field on VM exits due to execution of the INS and OUTS instructions (see Section 27.2.4). This reporting is done only if this bit is read as 1.
- Bit 55 is read as 1 if any VMX controls that default to 1 may be cleared to 0. See Appendix A.2 for details. It also reports support for the VMX capability MSRs IA32_VMX_TRUE_PINBASED_CTLS, IA32_VMX_TRUE_PROCBASED_CTLS, IA32_VMX_TRUE_EXIT_CTLS, and IA32_VMX_TRUE_ENTRY_CTLS. See Appendix A.3.1, Appendix A.3.2, Appendix A.4, and Appendix A.5 for details.
- The values of bits 47:45 and bits 63:56 are reserved and are read as 0.

A.2 RESERVED CONTROLS AND DEFAULT SETTINGS

As noted in Chapter 26, "VM Entries", certain VMX controls are reserved and must be set to a specific value (0 or 1) determined by the processor. The specific value to which a reserved control must be set is its **default setting**. Software can discover the default setting of a reserved control by consulting the appropriate VMX capability MSR (see Appendix A.3 through Appendix A.5).

Future processors may define new functionality for one or more reserved controls. Such processors would allow each newly defined control to be set either to 0 or to 1. Software that does not desire a control's new functionality should set the control to its default setting. For that reason, it is useful for software to know the default settings of the reserved controls.

Default settings partition the various controls into the following classes:

- **Always-flexible**. These have never been reserved.
- **Default0**. These are (or have been) reserved with a default setting of 0.
- **Default1**. They are (or have been) reserved with a default setting of 1.

As noted in Appendix A.1, a logical processor uses bit 55 of the IA32_VMX_BASIC MSR to indicate whether any of the default1 controls may be 0:

- If bit 55 of the IA32_VMX_BASIC MSR is read as 0, all the default1 controls are reserved and must be 1. VM entry will fail if any of these controls are 0 (see Section 26.2.1).
- If bit 55 of the IA32_VMX_BASIC MSR is read as 1, not all the default1 controls are reserved, and some (but not necessarily all) may be 0. The CPU supports four (4) new VMX capability MSRs: IA32_VMX_TRUE_PINBASED_CTLS, IA32_VMX_TRUE_PROCBASED_CTLS, IA32_VMX_TRUE_EXIT_CTLS, and IA32_VMX_TRUE_ENTRY_CTLS. See Appendix A.3 through Appendix A.5 for details. (These MSRs are not supported if bit 55 of the IA32_VMX_BASIC MSR is read as 0.)

See Section 31.5.1 for recommended software algorithms for proper capability detection of the default1 controls.

A.3 VM-EXECUTION CONTROLS

There are separate capability MSRs for the pin-based VM-execution controls, the primary processor-based VM-execution controls, and the secondary processor-based VM-execution controls. These are described in Appendix A.3.1, Appendix A.3.2, and Appendix A.3.3, respectively.

1. Alternatively, software may map any of these regions or structures with the UC memory type. (This may be necessary for the MSEG header.) Doing so is discouraged unless necessary as it will cause the performance of software accesses to those structures to suffer. The processor will continue to use the memory type reported in the VMX capability MSR IA32_VMX_BASIC with the exceptions noted.

A.3.1 Pin-Based VM-Execution Controls

The IA32_VMX_PINBASED_CTLS MSR (index 481H) reports on the allowed settings of **most** of the pin-based VM-execution controls (see Section 24.6.1):

- Bits 31:0 indicate the **allowed 0-settings** of these controls. VM entry allows control X (bit X of the pin-based VM-execution controls) to be 0 if bit X in the MSR is cleared to 0; if bit X in the MSR is set to 1, VM entry fails if control X is 0.

  Exceptions are made for the pin-based VM-execution controls in the default1 class (see Appendix A.2). These are bits 1, 2, and 4; the corresponding bits of the IA32_VMX_PINBASED_CTLS MSR are always read as 1. The treatment of these controls by VM entry is determined by bit 55 in the IA32_VMX_BASIC MSR:

  — If bit 55 in the IA32_VMX_BASIC MSR is read as 0, VM entry fails if any pin-based VM-execution control in the default1 class is 0.

  — If bit 55 in the IA32_VMX_BASIC MSR is read as 1, the IA32_VMX_TRUE_PINBASED_CTLS MSR (see below) reports which of the pin-based VM-execution controls in the default1 class can be 0 on VM entry.

- Bits 63:32 indicate the **allowed 1-settings** of these controls. VM entry allows control X to be 1 if bit 32+X in the MSR is set to 1; if bit 32+X in the MSR is cleared to 0, VM entry fails if control X is 1.

If bit 55 in the IA32_VMX_BASIC MSR is read as 1, the IA32_VMX_TRUE_PINBASED_CTLS MSR (index 48DH) reports on the allowed settings of **all** of the pin-based VM-execution controls:

- Bits 31:0 indicate the allowed 0-settings of these controls. VM entry allows control X to be 0 if bit X in the MSR is cleared to 0; if bit X in the MSR is set to 1, VM entry fails if control X is 0. There are no exceptions.
- Bits 63:32 indicate the allowed 1-settings of these controls. VM entry allows control X to be 1 if bit 32+X in the MSR is set to 1; if bit 32+X in the MSR is cleared to 0, VM entry fails if control X is 1.

It is necessary for software to consult only one of the capability MSRs to determine the allowed settings of the pin-based VM-execution controls:

- If bit 55 in the IA32_VMX_BASIC MSR is read as 0, all information about the allowed settings of the pin-based VM-execution controls is contained in the IA32_VMX_PINBASED_CTLS MSR. (The IA32_VMX_TRUE_PINBASED_CTLS MSR is not supported.)
- If bit 55 in the IA32_VMX_BASIC MSR is read as 1, all information about the allowed settings of the pin-based VM-execution controls is contained in the IA32_VMX_TRUE_PINBASED_CTLS MSR. Assuming that software knows that the default1 class of pin-based VM-execution controls contains bits 1, 2, and 4, there is no need for software to consult the IA32_VMX_PINBASED_CTLS MSR.

A.3.2 Primary Processor-Based VM-Execution Controls

The IA32_VMX_PROCBASED_CTLS MSR (index 482H) reports on the allowed settings of **most** of the primary processor-based VM-execution controls (see Section 24.6.2):

- Bits 31:0 indicate the allowed 0-settings of these controls. VM entry allows control X (bit X of the primary processor-based VM-execution controls) to be 0 if bit X in the MSR is cleared to 0; if bit X in the MSR is set to 1, VM entry fails if control X is 0.

  Exceptions are made for the primary processor-based VM-execution controls in the default1 class (see Appendix A.2). These are bits 1, 4–6, 8, 13–16, and 26; the corresponding bits of the IA32_VMX_PROCBASED_CTLS MSR are always read as 1. The treatment of these controls by VM entry is determined by bit 55 in the IA32_VMX_BASIC MSR:

  — If bit 55 in the IA32_VMX_BASIC MSR is read as 0, VM entry fails if any of the primary processor-based VM-execution controls in the default1 class is 0.

  — If bit 55 in the IA32_VMX_BASIC MSR is read as 1, the IA32_VMX_TRUE_PROCBASED_CTLS MSR (see below) reports which of the primary processor-based VM-execution controls in the default1 class can be 0 on VM entry.

- Bits 63:32 indicate the allowed 1-settings of these controls. VM entry allows control X to be 1 if bit 32+X in the MSR is set to 1; if bit 32+X in the MSR is cleared to 0, VM entry fails if control X is 1.

If bit 55 in the IA32_VMX_BASIC MSR is read as 1, the IA32_VMX_TRUE_PROCBASED_CTLS MSR (index 48EH) reports on the allowed settings of **all** of the primary processor-based VM-execution controls:

- Bits 31:0 indicate the allowed 0-settings of these controls. VM entry allows control X to be 0 if bit X in the MSR is cleared to 0; if bit X in the MSR is set to 1, VM entry fails if control X is 0. There are no exceptions.
- Bits 63:32 indicate the allowed 1-settings of these controls. VM entry allows control X to be 1 if bit 32+X in the MSR is set to 1; if bit 32+X in the MSR is cleared to 0, VM entry fails if control X is 1.

It is necessary for software to consult only one of the capability MSRs to determine the allowed settings of the primary processor-based VM-execution controls:

- If bit 55 in the IA32_VMX_BASIC MSR is read as 0, all information about the allowed settings of the primary processor-based VM-execution controls is contained in the IA32_VMX_PROCBASED_CTLS MSR. (The IA32_VMX_TRUE_PROCBASED_CTLS MSR is not supported.)
- If bit 55 in the IA32_VMX_BASIC MSR is read as 1, all information about the allowed settings of the processor-based VM-execution controls is contained in the IA32_VMX_TRUE_PROCBASED_CTLS MSR. Assuming that software knows that the default1 class of processor-based VM-execution controls contains bits 1, 4–6, 8, 13–16, and 26, there is no need for software to consult the IA32_VMX_PROCBASED_CTLS MSR.

A.3.3 Secondary Processor-Based VM-Execution Controls

The IA32_VMX_PROCBASED_CTLS2 MSR (index 48BH) reports on the allowed settings of the secondary processor-based VM-execution controls (see Section 24.6.2). VM entries perform the following checks:

- Bits 31:0 indicate the allowed 0-settings of these controls. These bits are always 0. This fact indicates that VM entry allows each bit of the secondary processor-based VM-execution controls to be 0 (reserved bits must be 0)
- Bits 63:32 indicate the allowed 1-settings of these controls; the 1-setting is not allowed for any reserved bit. VM entry allows control X (bit X of the secondary processor-based VM-execution controls) to be 1 if bit 32+X in the MSR is set to 1; if bit 32+X in the MSR is cleared to 0, VM entry fails if control X and the "activate secondary controls" primary processor-based VM-execution control are both 1.

The IA32_VMX_PROCBASED_CTLS2 MSR exists only on processors that support the 1-setting of the "activate secondary controls" VM-execution control (only if bit 63 of the IA32_VMX_PROCBASED_CTLS MSR is 1).

A.4 VM-EXIT CONTROLS

The IA32_VMX_EXIT_CTLS MSR (index 483H) reports on the allowed settings of **most** of the VM-exit controls (see Section 24.7.1):

- Bits 31:0 indicate the allowed 0-settings of these controls. VM entry allows control X (bit X of the VM-exit controls) to be 0 if bit X in the MSR is cleared to 0; if bit X in the MSR is set to 1, VM entry fails if control X is 0.

  Exceptions are made for the VM-exit controls in the default1 class (see Appendix A.2). These are bits 0–8, 10, 11, 13, 14, 16, and 17; the corresponding bits of the IA32_VMX_EXIT_CTLS MSR are always read as 1. The treatment of these controls by VM entry is determined by bit 55 in the IA32_VMX_BASIC MSR:

  — If bit 55 in the IA32_VMX_BASIC MSR is read as 0, VM entry fails if any VM-exit control in the default1 class is 0.

  — If bit 55 in the IA32_VMX_BASIC MSR is read as 1, the IA32_VMX_TRUE_EXIT_CTLS MSR (see below) reports which of the VM-exit controls in the default1 class can be 0 on VM entry.
- Bits 63:32 indicate the allowed 1-settings of these controls. VM entry allows control 32+X to be 1 if bit X in the MSR is set to 1; if bit 32+X in the MSR is cleared to 0, VM entry fails if control X is 1.

If bit 55 in the IA32_VMX_BASIC MSR is read as 1, the IA32_VMX_TRUE_EXIT_CTLS MSR (index 48FH) reports on the allowed settings of **all** of the VM-exit controls:

- Bits 31:0 indicate the allowed 0-settings of these controls. VM entry allows control X to be 0 if bit X in the MSR is cleared to 0; if bit X in the MSR is set to 1, VM entry fails if control X is 0. There are no exceptions.

- Bits 63:32 indicate the allowed 1-settings of these controls. VM entry allows control X to be 1 if bit 32+X in the MSR is set to 1; if bit 32+X in the MSR is cleared to 0, VM entry fails if control X is 1.

It is necessary for software to consult only one of the capability MSRs to determine the allowed settings of the VM-exit controls:

- If bit 55 in the IA32_VMX_BASIC MSR is read as 0, all information about the allowed settings of the VM-exit controls is contained in the IA32_VMX_EXIT_CTLS MSR. (The IA32_VMX_TRUE_EXIT_CTLS MSR is not supported.)
- If bit 55 in the IA32_VMX_BASIC MSR is read as 1, all information about the allowed settings of the VM-exit controls is contained in the IA32_VMX_TRUE_EXIT_CTLS MSR. Assuming that software knows that the default1 class of VM-exit controls contains bits 0–8, 10, 11, 13, 14, 16, and 17, there is no need for software to consult the IA32_VMX_EXIT_CTLS MSR.

A.5 VM-ENTRY CONTROLS

The IA32_VMX_ENTRY_CTLS MSR (index 484H) reports on the allowed settings of **most** of the VM-entry controls (see Section 24.8.1):

- Bits 31:0 indicate the allowed 0-settings of these controls. VM entry allows control X (bit X of the VM-entry controls) to be 0 if bit X in the MSR is cleared to 0; if bit X in the MSR is set to 1, VM entry fails if control X is 0.

  Exceptions are made for the VM-entry controls in the default1 class (see Appendix A.2). These are bits 0–8 and 12; the corresponding bits of the IA32_VMX_ENTRY_CTLS MSR are always read as 1. The treatment of these controls by VM entry is determined by bit 55 in the IA32_VMX_BASIC MSR:

  — If bit 55 in the IA32_VMX_BASIC MSR is read as 0, VM entry fails if any VM-entry control in the default1 class is 0.

  — If bit 55 in the IA32_VMX_BASIC MSR is read as 1, the IA32_VMX_TRUE_ENTRY_CTLS MSR (see below) reports which of the VM-entry controls in the default1 class can be 0 on VM entry.
- Bits 63:32 indicate the allowed 1-settings of these controls. VM entry fails if bit X is 1 in the VM-entry controls and bit 32+X is 0 in this MSR.

If bit 55 in the IA32_VMX_BASIC MSR is read as 1, the IA32_VMX_TRUE_ENTRY_CTLS MSR (index 490H) reports on the allowed settings of **all** of the VM-entry controls:

- Bits 31:0 indicate the allowed 0-settings of these controls. VM entry allows control X to be 0 if bit X in the MSR is cleared to 0; if bit X in the MSR is set to 1, VM entry fails if control X is 0. There are no exceptions.
- Bits 63:32 indicate the allowed 1-settings of these controls. VM entry allows control 32+X to be 1 if bit X in the MSR is set to 1; if bit 32+X in the MSR is cleared to 0, VM entry fails if control X is 1.

It is necessary for software to consult only one of the capability MSRs to determine the allowed settings of the VM-entry controls:

- If bit 55 in the IA32_VMX_BASIC MSR is read as 0, all information about the allowed settings of the VM-entry controls is contained in the IA32_VMX_ENTRY_CTLS MSR. (The IA32_VMX_TRUE_ENTRY_CTLS MSR is not supported.)
- If bit 55 in the IA32_VMX_BASIC MSR is read as 1, all information about the allowed settings of the VM-entry controls is contained in the IA32_VMX_TRUE_ENTRY_CTLS MSR. Assuming that software knows that the default1 class of VM-entry controls contains bits 0–8 and 12, there is no need for software to consult the IA32_VMX_ENTRY_CTLS MSR.

A.6 MISCELLANEOUS DATA

The IA32_VMX_MISC MSR (index 485H) consists of the following fields:

- Bits 4:0 report a value X that specifies the relationship between the rate of the VMX-preemption timer and that of the timestamp counter (TSC). Specifically, the VMX-preemption timer (if it is active) counts down by 1 every time bit X in the TSC changes due to a TSC increment.

- If bit 5 is read as 1, VM exits store the value of IA32_EFER.LMA into the "IA-32e mode guest" VM-entry control; see Section 27.2 for more details. This bit is read as 1 on any logical processor that supports the 1-setting of the "unrestricted guest" VM-execution control.
- Bits 8:6 report, as a bitmap, the activity states supported by the implementation:
  - — Bit 6 reports (if set) the support for activity state 1 (HLT).
  - — Bit 7 reports (if set) the support for activity state 2 (shutdown).
  - — Bit 8 reports (if set) the support for activity state 3 (wait-for-SIPI).

  If an activity state is not supported, the implementation causes a VM entry to fail if it attempts to establish that activity state. All implementations support VM entry to activity state 0 (active).
- If bit 14 is read as 1, Intel® Processor Trace (Intel PT) can be used in VMX operation. If the processor supports Intel PT but does not allow it to be used in VMX operation, execution of VMXON clears IA32_RTIT_CTL.TraceEn (see "VMXON—Enter VMX Operation" in Chapter 30); any attempt to write IA32_RTIT_CTL while in VMX operation (including VMX root operation) causes a general-protection exception.
- If bit 15 is read as 1, the RDMSR instruction can be used in system-management mode (SMM) to read the IA32_SMBASE MSR (MSR address 9EH). See Section 34.15.6.3.
- Bits 24:16 indicate the number of CR3-target values supported by the processor. This number is a value between 0 and 256, inclusive (bit 24 is set if and only if bits 23:16 are clear).
- Bits 27:25 is used to compute the recommended maximum number of MSRs that should appear in the VM-exit MSR-store list, the VM-exit MSR-load list, or the VM-entry MSR-load list. Specifically, if the value bits 27:25 of IA32_VMX_MISC is N, then 512 * (N + 1) is the recommended maximum number of MSRs to be included in each list. If the limit is exceeded, undefined processor behavior may result (including a machine check during the VMX transition).
- If bit 28 is read as 1, bit 2 of the IA32_SMM_MONITOR_CTL can be set to 1. VMXOFF unblocks SMIs unless IA32_SMM_MONITOR_CTL[bit 2] is 1 (see Section 34.14.4).
- If bit 29 is read as 1, software can use VMWRITE to write to any supported field in the VMCS; otherwise, VMWRITE cannot be used to modify VM-exit information fields.
- If bit 30 is read as 1, VM entry allows injection of a software interrupt, software exception, or privileged software exception with an instruction length of 0.
- Bits 63:32 report the 32-bit MSEG revision identifier used by the processor.
- Bits 13:9 and bit 31 are reserved and are read as 0.

A.7 VMX-FIXED BITS IN CR0

The IA32_VMX_CR0_FIXED0 MSR (index 486H) and IA32_VMX_CR0_FIXED1 MSR (index 487H) indicate how bits in CR0 may be set in VMX operation. They report on bits in CR0 that are allowed to be 0 and to be 1, respectively, in VMX operation. If bit X is 1 in IA32_VMX_CR0_FIXED0, then that bit of CR0 is fixed to 1 in VMX operation. Similarly, if bit X is 0 in IA32_VMX_CR0_FIXED1, then that bit of CR0 is fixed to 0 in VMX operation. It is always the case that, if bit X is 1 in IA32_VMX_CR0_FIXED0, then that bit is also 1 in IA32_VMX_CR0_FIXED1; if bit X is 0 in IA32_VMX_CR0_FIXED1, then that bit is also 0 in IA32_VMX_CR0_FIXED0. Thus, each bit in CR0 is either fixed to 0 (with value 0 in both MSRs), fixed to 1 (1 in both MSRs), or flexible (0 in IA32_VMX_CR0_FIXED0 and 1 in IA32_VMX_CR0_FIXED1).

A.8 VMX-FIXED BITS IN CR4

The IA32_VMX_CR4_FIXED0 MSR (index 488H) and IA32_VMX_CR4_FIXED1 MSR (index 489H) indicate how bits in CR4 may be set in VMX operation. They report on bits in CR4 that are allowed to be 0 and 1, respectively, in VMX operation. If bit X is 1 in IA32_VMX_CR4_FIXED0, then that bit of CR4 is fixed to 1 in VMX operation. Similarly, if bit X is 0 in IA32_VMX_CR4_FIXED1, then that bit of CR4 is fixed to 0 in VMX operation. It is always the case that, if bit X is 1 in IA32_VMX_CR4_FIXED0, then that bit is also 1 in IA32_VMX_CR4_FIXED1; if bit X is 0 in IA32_VMX_CR4_FIXED1, then that bit is also 0 in IA32_VMX_CR4_FIXED0. Thus, each bit in CR4 is either fixed to

0 (with value 0 in both MSRs), fixed to 1 (1 in both MSRs), or flexible (0 in IA32_VMX_CR4_FIXED0 and 1 in IA32_VMX_CR4_FIXED1).

A.9 VMCS ENUMERATION

The IA32_VMX_VMCS_ENUM MSR (index 48AH) provides information to assist software in enumerating fields in the VMCS.

As noted in Section 24.11.2, each field in the VMCS is associated with a 32-bit encoding which is structured as follows:

- Bits 31:15 are reserved (must be 0).
- Bits 14:13 indicate the field's width.
- Bit 12 is reserved (must be 0).
- Bits 11:10 indicate the field's type.
- Bits 9:1 is an index field that distinguishes different fields with the same width and type.
- Bit 0 indicates access type.

IA32_VMX_VMCS_ENUM indicates to software the highest index value used in the encoding of any field supported by the processor:

- Bits 9:1 contain the highest index value used for any VMCS encoding.
- Bit 0 and bits 63:10 are reserved and are read as 0.

A.10 VPID AND EPT CAPABILITIES

The IA32_VMX_EPT_VPID_CAP MSR (index 48CH) reports information about the capabilities of the logical processor with regard to virtual-processor identifiers (VPIDs, Section 28.1) and extended page tables (EPT, Section 28.2):

- If bit 0 is read as 1, the processor supports execute-only translations by EPT. This support allows software to configure EPT paging-structure entries in which bits 1:0 are clear (indicating that data accesses are not allowed) and bit 2 is set (indicating that instruction fetches are allowed).[1]
- Bit 6 indicates support for a page-walk length of 4.
- If bit 8 is read as 1, the logical processor allows software to configure the EPT paging-structure memory type to be uncacheable (UC); see Section 24.6.11.
- If bit 14 is read as 1, the logical processor allows software to configure the EPT paging-structure memory type to be write-back (WB).
- If bit 16 is read as 1, the logical processor allows software to configure a EPT PDE to map a 2-Mbyte page (by setting bit 7 in the EPT PDE).
- If bit 17 is read as 1, the logical processor allows software to configure a EPT PDPTE to map a 1-Gbyte page (by setting bit 7 in the EPT PDPTE).
- Support for the INVEPT instruction (see Chapter 30 and Section 28.3.3.1).
  - — If bit 20 is read as 1, the INVEPT instruction is supported.
  - — If bit 25 is read as 1, the single-context INVEPT type is supported.
  - — If bit 26 is read as 1, the all-context INVEPT type is supported.
- If bit 21 is read as 1, accessed and dirty flags for EPT are supported (see Section 28.2.4).

1. If the "mode-based execute control for EPT" VM-execution control is 1, setting bit 0 indicates also that software may also configure EPT paging-structure entries in which bits 1:0 are both clear and in which bit 10 is set (indicating a translation that can be used to fetch instructions from a supervisor-mode linear address or a user-mode linear address).

- If bit 22 is read as 1, the processor reports advanced VM-exit information for EPT violations (see Section 27.2.1). This reporting is done only if this bit is read as 1.
- Support for the INVVPID instruction (see Chapter 30 and Section 28.3.3.1).
  - — If bit 32 is read as 1, the INVVPID instruction is supported.
  - — If bit 40 is read as 1, the individual-address INVVPID type is supported.
  - — If bit 41 is read as 1, the single-context INVVPID type is supported.
  - — If bit 42 is read as 1, the all-context INVVPID type is supported.
  - — If bit 43 is read as 1, the single-context-retaining-globals INVVPID type is supported.
- Bits 5:1, bit 7, bits 13:9, bit 15, bits 19:18, bits 24:23, bits 31:27, bits 39:33, and bits 63:44 are reserved and are read as 0.

The IA32_VMX_EPT_VPID_CAP MSR exists only on processors that support the 1-setting of the "activate secondary controls" VM-execution control (only if bit 63 of the IA32_VMX_PROCBASED_CTLS MSR is 1) and that support either the 1-setting of the "enable EPT" VM-execution control (only if bit 33 of the IA32_VMX_PROCBASED_CTLS2 MSR is 1) or the 1-setting of the "enable VPID" VM-execution control (only if bit 37 of the IA32_VMX_PROCBASED_CTLS2 MSR is 1).

A.11 VM FUNCTIONS

The IA32_VMX_VMFUNC MSR (index 491H) reports on the allowed settings of the VM-function controls (see Section 24.6.14). VM entry allows bit X of the VM-function controls to be 1 if bit X in the MSR is set to 1; if bit X in the MSR is cleared to 0, VM entry fails if bit X of the VM-function controls, the "activate secondary controls" primary processor-based VM-execution control, and the "enable VM functions" secondary processor-based VM-execution control are all 1.

The IA32_VMX_VMFUNC MSR exists only on processors that support the 1-setting of the "activate secondary controls" VM-execution control (only if bit 63 of the IA32_VMX_PROCBASED_CTLS MSR is 1) and the 1-setting of the "enable VM functions" secondary processor-based VM-execution control (only if bit 45 of the IA32_VMX_PROCBASED_CTLS2 MSR is 1).

27. Updates to Chapter 1, Volume 4

Change bars show changes to Chapter 1 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 4:* Model-Specific Registers (MSRs).

Changes include: Updates to list of processors supported and minor typo correction in Figure 1-2 "Syntax for CPUID, CR, and MSR Data Presentation".

CHAPTER 1
ABOUT THIS MANUAL

The *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 4: Model-Specific Registers* (order number 335592) is part of a set that describes the architecture and programming environment of Intel® 64 and IA-32 architecture processors. Other volumes in this set are:

- *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*: *Basic Architecture* (order number 253665).
- *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volumes 2A, 2B, 2C & 2D: Instruction Set Reference* (order numbers 253666, 253667, 326018 and 334569).
- The *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volumes 3A, 3B, 3C & 3D: System Programming Guide* (order numbers 253668, 253669, 326019 and 332831).

The *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*, describes the basic architecture and programming environment of Intel 64 and IA-32 processors. The *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volumes 2A, 2B, 2C & 2D*, describe the instruction set of the processor and the opcode structure. These volumes apply to application programmers and to programmers who write operating systems or executives. The *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volumes 3A, 3B, 3C & 3D*, describe the operating-system support environment of Intel 64 and IA-32 processors. These volumes target operating-system and BIOS designers. In addition, *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3B*, and *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3C* address the programming environment for classes of software that host operating systems. The *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 4*, describes the model-specific registers of Intel 64 and IA-32 processors.

1.1 INTEL® 64 AND IA-32 PROCESSORS COVERED IN THIS MANUAL

This manual set includes information pertaining primarily to the most recent Intel 64 and IA-32 processors, which include:

- Pentium® processors
- P6 family processors
- Pentium® 4 processors
- Pentium® M processors
- Intel® Xeon® processors
- Pentium® D processors
- Pentium® processor Extreme Editions
- 64-bit Intel® Xeon® processors
- Intel® Core™ Duo processor
- Intel® Core™ Solo processor
- Dual-Core Intel® Xeon® processor LV
- Intel® Core™2 Duo processor
- Intel® Core™2 Quad processor Q6000 series
- Intel® Xeon® processor 3000, 3200 series
- Intel® Xeon® processor 5000 series
- Intel® Xeon® processor 5100, 5300 series
- Intel® Core™2 Extreme processor X7000 and X6800 series
- Intel® Core™2 Extreme QX6000 series
- Intel® Xeon® processor 7100 series

- Intel® Pentium® Dual-Core processor
- Intel® Xeon® processor 7200, 7300 series
- Intel® Core™2 Extreme QX9000 series
- Intel® Xeon® processor 5200, 5400, 7400 series
- Intel® Core™2 Extreme processor QX9000 and X9000 series
- Intel® Core™2 Quad processor Q9000 series
- Intel® Core™2 Duo processor E8000, T9000 series
- Intel® Atom™ processors 200, 300, D400, D500, D2000, N200, N400, N2000, E2000, Z500, Z600, Z2000, C1000 series are built from 45 nm and 32 nm processes.
- Intel® Core™ i7 processor
- Intel® Core™ i5 processor
- Intel® Xeon® processor E7-8800/4800/2800 product families
- Intel® Core™ i7-3930K processor
- 2nd generation Intel® Core™ i7-2xxx, Intel® Core™ i5-2xxx, Intel® Core™ i3-2xxx processor series
- Intel® Xeon® processor E3-1200 product family
- Intel® Xeon® processor E5-2400/1400 product family
- Intel® Xeon® processor E5-4600/2600/1600 product family
- 3rd generation Intel® Core™ processors
- Intel® Xeon® processor E3-1200 v2 product family
- Intel® Xeon® processor E5-2400/1400 v2 product families
- Intel® Xeon® processor E5-4600/2600/1600 v2 product families
- Intel® Xeon® processor E7-8800/4800/2800 v2 product families
- 4th generation Intel® Core™ processors
- The Intel® Core™ M processor family
- Intel® Core™ i7-59xx Processor Extreme Edition
- Intel® Core™ i7-49xx Processor Extreme Edition
- Intel® Xeon® processor E3-1200 v3 product family
- Intel® Xeon® processor E5-2600/1600 v3 product families
- 5th generation Intel® Core™ processors
- Intel® Xeon® processor D-1500 product family
- Intel® Xeon® processor E5 v4 family
- Intel® Atom™ processor X7-Z8000 and X5-Z8000 series
- Intel® Atom™ processor Z3400 series
- Intel® Atom™ processor Z3500 series
- 6th generation Intel® Core™ processors
- Intel® Xeon® processor E3-1500m v5 product family
- 7th generation Intel® Core™ processors
- Intel® Xeon Phi™ Processor 3200, 5200, 7200 Series
- Intel® Xeon® Processor Scalable Family

The Pentium® 4, Pentium® D, and Pentium® processor Extreme Editions are based on the Intel NetBurst® micro-architecture. Most early Intel® Xeon® processors are based on the Intel NetBurst® microarchitecture. Intel Xeon processor 5000, 7100 series are based on the Intel NetBurst® microarchitecture.

The Intel® Core™ Duo, Intel® Core™ Solo and dual-core Intel® Xeon® processor LV are based on an improved Pentium® M processor microarchitecture.

The Intel® Xeon® processor 3000, 3200, 5100, 5300, 7200, and 7300 series, Intel® Pentium® dual-core, Intel® Core™2 Duo, Intel® Core™2 Quad, and Intel® Core™2 Extreme processors are based on Intel® Core™ microarchitecture.

The Intel® Xeon® processor 5200, 5400, 7400 series, Intel® Core™2 Quad processor Q9000 series, and Intel® Core™2 Extreme processors QX9000, X9000 series, Intel® Core™2 processor E8000 series are based on Enhanced Intel® Core™ microarchitecture.

The Intel® Atom™ processors 200, 300, D400, D500, D2000, N200, N400, N2000, E2000, Z500, Z600, Z2000, C1000 series are based on the Intel® Atom™ microarchitecture and supports Intel 64 architecture.

P6 family, Pentium® M, Intel® Core™ Solo, Intel® Core™ Duo processors, dual-core Intel® Xeon® processor LV, and early generations of Pentium 4 and Intel Xeon processors support IA-32 architecture. The Intel® Atom™ processor Z5xx series support IA-32 architecture.

The Intel® Xeon® processor 3000, 3200, 5000, 5100, 5200, 5300, 5400, 7100, 7200, 7300, 7400 series, Intel® Core™2 Duo, Intel® Core™2 Extreme, Intel® Core™2 Quad processors, Pentium® D processors, Pentium® Dual-Core processor, newer generations of Pentium 4 and Intel Xeon processor family support Intel® 64 architecture.

The Intel® Core™ i7 processor and Intel® Xeon® processor 3400, 5500, 7500 series are based on 45 nm Intel® microarchitecture code name Nehalem. Intel® microarchitecture code name Westmere is a 32 nm version of Intel® microarchitecture code name Nehalem. Intel® Xeon® processor 5600 series, Intel Xeon processor E7 and various Intel Core i7, i5, i3 processors are based on Intel® microarchitecture code name Westmere. These processors support Intel 64 architecture.

The Intel® Xeon® processor E5 family, Intel® Xeon® processor E3-1200 family, Intel® Xeon® processor E7-8800/4800/2800 product families, Intel® Core™ i7-3930K processor, and 2nd generation Intel® Core™ i7-2xxx, Intel® Core™ i5-2xxx, Intel® Core™ i3-2xxx processor series are based on the Intel® microarchitecture code name Sandy Bridge and support Intel 64 architecture.

The Intel® Xeon® processor E7-8800/4800/2800 v2 product families, Intel® Xeon® processor E3-1200 v2 product family and 3rd generation Intel® Core™ processors are based on the Intel® microarchitecture code name Ivy Bridge and support Intel 64 architecture.

The Intel® Xeon® processor E5-4600/2600/1600 v2 product families, Intel® Xeon® processor E5-2400/1400 v2 product families and Intel® Core™ i7-49xx Processor Extreme Edition are based on the Intel® microarchitecture code name Ivy Bridge-E and support Intel 64 architecture.

The Intel® Xeon® processor E3-1200 v3 product family and 4th Generation Intel® Core™ processors are based on the Intel® microarchitecture code name Haswell and support Intel 64 architecture.

The Intel® Xeon® processor E5-2600/1600 v3 product families and the Intel® Core™ i7-59xx Processor Extreme Edition are based on the Intel® microarchitecture code name Haswell-E and support Intel 64 architecture.

The Intel® Atom™ processor Z8000 series is based on the Intel microarchitecture code name Airmont.

The Intel® Atom™ processor Z3400 series and the Intel® Atom™ processor Z3500 series are based on the Intel microarchitecture code name Silvermont.

The Intel® Core™ M processor family, 5th generation Intel® Core™ processors, Intel® Xeon® processor D-1500 product family and the Intel® Xeon® processor E5 v4 family are based on the Intel® microarchitecture code name Broadwell and support Intel 64 architecture.

The Intel® Xeon® Processor Scalable Family, Intel® Xeon® processor E3-1500m v5 product family and 6th generation Intel® Core™ processors are based on the Intel® microarchitecture code name Skylake and support Intel 64 architecture.

The 7th generation Intel® Core™ processors are based on the Intel® microarchitecture code name Kaby Lake and support Intel 64 architecture.

The Intel® Xeon Phi™ Processor 3200, 5200, 7200 Series is based on the Intel® microarchitecture code name Knights Landing and supports Intel 64 architecture.

IA-32 architecture is the instruction set architecture and programming environment for Intel's 32-bit microprocessors. Intel® 64 architecture is the instruction set architecture and programming environment which is the superset of Intel’s 32-bit and 64-bit architectures. It is compatible with the IA-32 architecture.

1.2 OVERVIEW OF THE SYSTEM PROGRAMMING GUIDE

A description of this manual's content follows:

Chapter 1 — About This Manual. Gives an overview of all eight volumes of the *Intel® 64 and IA-32 Architectures Software Developer's Manual*. It also describes the notational conventions in these manuals and lists related Intel manuals and documentation of interest to programmers and hardware designers.

Chapter 2 — Model-Specific Registers (MSRs). Lists the MSRs available in the Pentium processors, the P6 family processors, the Pentium 4, Intel Xeon, Intel Core Solo, Intel Core Duo processors, Intel Core 2 processor family, and Intel Atom processors, and describes their functions.

1.3 NOTATIONAL CONVENTIONS

This manual uses specific notation for data-structure formats, for symbolic representation of instructions, and for hexadecimal and binary numbers. A review of this notation makes the manual easier to read.

1.3.1 Bit and Byte Order

In illustrations of data structures in memory, smaller addresses appear toward the bottom of the figure; addresses increase toward the top. Bit positions are numbered from right to left. The numerical value of a set bit is equal to two raised to the power of the bit position. Intel 64 and IA-32 processors are "little endian" machines; this means the bytes of a word are numbered starting from the least significant byte. Figure 1-1 illustrates these conventions.

1.3.2 Reserved Bits and Software Compatibility

In many register and memory layout descriptions, certain bits are marked as **reserved**. When bits are marked as reserved, it is essential for compatibility with future processors that software treat these bits as having a future, though unknown, effect. The behavior of reserved bits should be regarded as not only undefined, but unpredictable. Software should follow these guidelines in dealing with reserved bits:

- Do not depend on the states of any reserved bits when testing the values of registers which contain such bits. Mask out the reserved bits before testing.
- Do not depend on the states of any reserved bits when storing to memory or to a register.
- Do not depend on the ability to retain information written into any reserved bits.
- When loading a register, always load the reserved bits with the values indicated in the documentation, if any, or reload them with values previously read from the same register.

NOTE

Avoid any software dependence upon the state of reserved bits in Intel 64 and IA-32 registers. Depending upon the values of reserved register bits will make software dependent upon the unspecified manner in which the processor handles these bits. Programs that depend upon reserved values risk incompatibility with future processors.

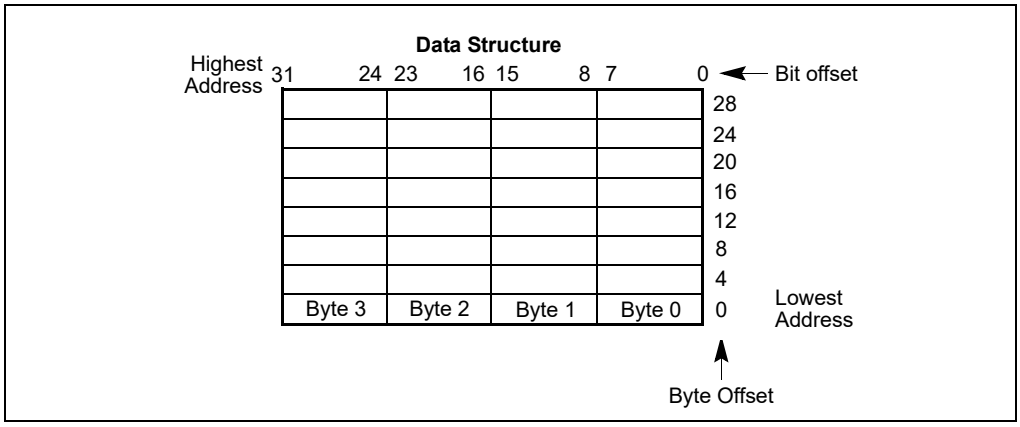


Figure 1-1. Bit and Byte Order

1.3.3 Instruction Operands

When instructions are represented symbolically, a subset of assembly language is used. In this subset, an instruction has the following format:

```
label: mnemonic argument1, argument2, argument3
```

where:

- A **label** is an identifier which is followed by a colon.
- A **mnemonic** is a reserved name for a class of instruction opcodes which have the same function.
- The operands **argument1**, **argument2**, and **argument3** are optional. There may be from zero to three operands, depending on the opcode. When present, they take the form of either literals or identifiers for data items. Operand identifiers are either reserved names of registers or are assumed to be assigned to data items declared in another part of the program (which may not be shown in the example).

When two operands are present in an arithmetic or logical instruction, the right operand is the source and the left operand is the destination.

For example:

```
LOADREG: MOV EAX, SUBTOTAL
```

In this example LOADREG is a label, MOV is the mnemonic identifier of an opcode, EAX is the destination operand, and SUBTOTAL is the source operand. Some assembly languages put the source and destination in reverse order.

1.3.4 Hexadecimal and Binary Numbers

Base 16 (hexadecimal) numbers are represented by a string of hexadecimal digits followed by the character H (for example, F82EH). A hexadecimal digit is a character from the following set: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, and F.

Base 2 (binary) numbers are represented by a string of 1s and 0s, sometimes followed by the character B (for example, 1010B). The "B" designation is only used in situations where confusion as to the type of number might arise.

1.3.5 Segmented Addressing

The processor uses byte addressing. This means memory is organized and accessed as a sequence of bytes. Whether one or more bytes are being accessed, a byte address is used to locate the byte or bytes memory. The range of memory that can be addressed is called an **address space**.

The processor also supports segmented addressing. This is a form of addressing where a program may have many independent address spaces, called **segments**. For example, a program can keep its code (instructions) and stack in separate segments. Code addresses would always refer to the code space, and stack addresses would always refer to the stack space. The following notation is used to specify a byte address within a segment:

Segment-register:Byte-address

For example, the following segment address identifies the byte at address FF79H in the segment pointed by the DS register:

DS:FF79H

The following segment address identifies an instruction address in the code segment. The CS register points to the code segment and the EIP register contains the address of the instruction.

CS:EIP

1.3.6 Syntax for CPUID, CR, and MSR Values

Obtain feature flags, status, and system information by using the CPUID instruction, by checking control register bits, and by reading model-specific registers. We are moving toward a single syntax to represent this type of information. See Figure 1-2.

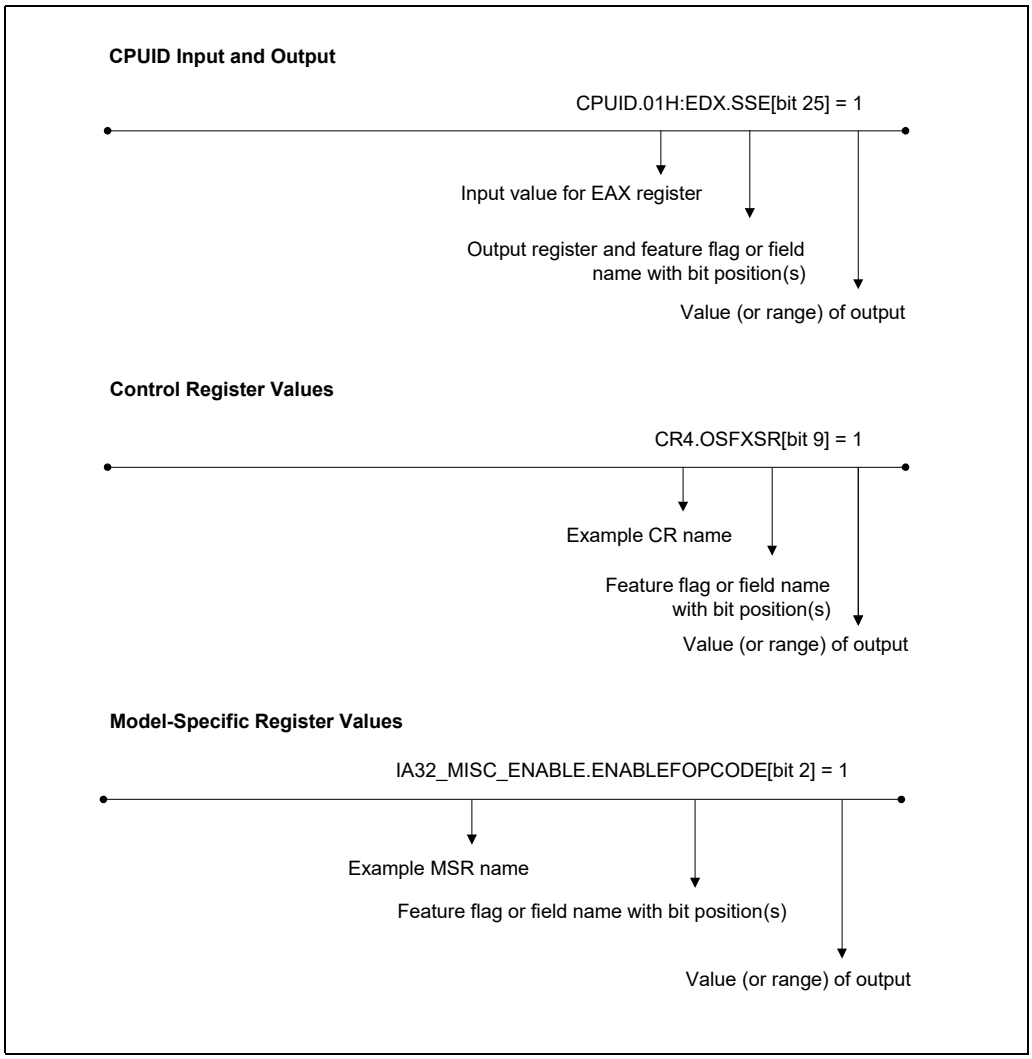


Figure 1-2. Syntax for CPUID, CR, and MSR Data Presentation

1.3.7 Exceptions

An exception is an event that typically occurs when an instruction causes an error. For example, an attempt to divide by zero generates an exception. However, some exceptions, such as breakpoints, occur under other conditions. Some types of exceptions may provide error codes. An error code reports additional information about the error. An example of the notation used to show an exception and error code is shown below:

```
#PF(fault code)
```

This example refers to a page-fault exception under conditions where an error code naming a type of fault is reported. Under some conditions, exceptions which produce error codes may not be able to report an accurate code. In this case, the error code is zero, as shown below for a general-protection exception:

```
#GP(0)
```

1.4 RELATED LITERATURE

Literature related to Intel 64 and IA-32 processors is listed and viewable on-line at:

http://www.intel.com/content/www/us/en/processors/architectures-software-developer-manuals.html

See also:

- The data sheet for a particular Intel 64 or IA-32 processor
- The specification update for a particular Intel 64 or IA-32 processor
- Intel® C++ Compiler documentation and online help:
  http://software.intel.com/en-us/articles/intel-compilers/
- Intel® Fortran Compiler documentation and online help:
  http://software.intel.com/en-us/articles/intel-compilers/
- Intel® Software Development Tools:
  https://software.intel.com/en-us/intel-sdp-home
- Intel® 64 and IA-32 Architectures Software Developer's Manual (in one, four or ten volumes):
  https://software.intel.com/en-us/articles/intel-sdm
- Intel® 64 and IA-32 Architectures Optimization Reference Manual:
  https://software.intel.com/en-us/articles/intel-sdm#optimization
- Intel 64 Architecture x2APIC Specification:

  http://www.intel.com/content/www/us/en/architecture-and-technology/64-architecture-x2apic-specification.html
- Intel® Trusted Execution Technology Measured Launched Environment Programming Guide:

  http://www.intel.com/content/www/us/en/software-developers/intel-txt-software-development-guide.html
- Developing Multi-threaded Applications: A Platform Consistent Approach:
  https://software.intel.com/sites/default/files/article/147714/51534-developing-multithreaded-applications.pdf
- Using Spin-Loops on Intel® Pentium® 4 Processor and Intel® Xeon® Processor:
  https://software.intel.com/sites/default/files/22/30/25602
- Performance Monitoring Unit Sharing Guide
  http://software.intel.com/file/30388

Literature related to selected features in future Intel processors are available at:

- Intel® Architecture Instruction Set Extensions Programming Reference
  https://software.intel.com/en-us/isa-extensions
- Intel® Software Guard Extensions (Intel® SGX) Programming Reference
  https://software.intel.com/en-us/isa-extensions/intel-sgx

More relevant links are:

- Intel® Developer Zone:
  https://software.intel.com/en-us
- Developer centers:
  http://www.intel.com/content/www/us/en/hardware-developers/developer-centers.html
- Processor support general link:
  http://www.intel.com/support/processors/
- Intel® Hyper-Threading Technology (Intel® HT Technology):
  http://www.intel.com/technology/platform-technology/hyper-threading/index.htm

28.Updates to Chapter 2, Volume 4

Change bars show changes to Chapter 2 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 4:* Model-Specific Registers (MSRs).

Changes include: Updates to Table 2-1 "CPUID Signature Values of DisplayFamily_DisplayModel". Added MSR CEH to Table 2-7 "MSRs Common to the Silvermont and Airmont Microarchitectures". Moved MSR CDH from Table 2-7 "MSRs Common to the Silvermont and Airmont Microarchitectures" to Table 2-8 "Specific MSRs Supported by Intel® Atom™ Processors with CPUID Signatures 06_37H, 06_4AH, 06_5AH, 06_5DH" (Section 2.4). Addition of MSRs for Intel Atom Processors based on Goldmont Plus Microarchitecture (Section 2.6). Addition of MSRs for Intel® Core™ processors based on Cannon Lake microarchitecture (Section 2.15). Addition of MSRs for Intel® Xeon® Processor Scalable Family and Processors based on Skylake Microarchitecture (Section 2.16). Addition of MSRs for Future Intel® Xeon Phi™ Processors based on Knights Mill microarchitecture (Section 2.17). Various missing MSR descriptions added.

CHAPTER 2
MODEL-SPECIFIC REGISTERS (MSRS)

This chapter lists MSRs across Intel processor families. All MSRs listed can be read with the RDMSR and written with the WRMSR instructions.

Register addresses are given in both hexadecimal and decimal. The register name is the mnemonic register name and the bit description describes individual bits in registers.

Model specific registers and its bit-fields may be supported for a finite range of processor families/models. To distinguish between different processor family and/or models, software must use CPUID.01H leaf function to query the combination of DisplayFamily and DisplayModel to determine model-specific availability of MSRs (see CPUID instruction in Chapter 3, "Instruction Set Reference, A-L" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A*). Table 2-1 lists the signature values of DisplayFamily and DisplayModel for various processor families or processor number series.

Table 2-1. CPUID Signature Values of DisplayFamily_DisplayModel

DisplayFamily_DisplayModel	Processor Families/Processor Number Series
06_85H	Future Intel® Xeon Phi™ Processor based on Knights Mill microarchitecture
06_57H	Intel® Xeon Phi™ Processor 3200, 5200, 7200 Series based on Knights Landing microarchitecture
06_66H	Future Intel® Core™ processors based on Cannon Lake microarchitecture
06_8EH, 06_9EH	7th generation Intel® Core™ processors based on Kaby Lake microarchitecture
06_55H	Intel® Xeon® Processor Scalable Family based on Skylake microarchitecture
06_4EH, 06_5EH	6th generation Intel Core processors and Intel Xeon processor E3-1500m v5 product family and E3-1200 v5 product family based on Skylake microarchitecture
06_56H	Intel Xeon processor D-1500 product family based on Broadwell microarchitecture
06_4FH	Intel Xeon processor E5 v4 Family based on Broadwell microarchitecture, Intel Xeon processor E7 v4 Family, Intel Core i7-69xx Processor Extreme Edition
06_47H	5th generation Intel Core processors, Intel Xeon processor E3-1200 v4 product family based on Broadwell microarchitecture
06_3DH	Intel Core M-5xxx Processor, 5th generation Intel Core processors based on Broadwell microarchitecture
06_3FH	Intel Xeon processor E5-4600/2600/1600 v3 product families, Intel Xeon processor E7 v3 product families based on Haswell-E microarchitecture, Intel Core i7-59xx Processor Extreme Edition
06_3CH, 06_45H, 06_46H	4th Generation Intel Core processor and Intel Xeon processor E3-1200 v3 product family based on Haswell microarchitecture
06_3EH	Intel Xeon processor E7-8800/4800/2800 v2 product families based on Ivy Bridge-E microarchitecture
06_3EH	Intel Xeon processor E5-2600/1600 v2 product families and Intel Xeon processor E5-2400 v2 product family based on Ivy Bridge-E microarchitecture, Intel Core i7-49xx Processor Extreme Edition
06_3AH	3rd Generation Intel Core Processor and Intel Xeon processor E3-1200 v2 product family based on Ivy Bridge microarchitecture
06_2DH	Intel Xeon processor E5 Family based on Intel microarchitecture code name Sandy Bridge, Intel Core i7-39xx Processor Extreme Edition
06_2FH	Intel Xeon Processor E7 Family
06_2AH	Intel Xeon processor E3-1200 product family; 2nd Generation Intel Core i7, i5, i3 Processors 2xxx Series
06_2EH	Intel Xeon processor 7500, 6500 series

Table 2-1. CPUID Signature (Contd.)Values of DisplayFamily_DisplayModel (Contd.)

DisplayFamily_DisplayModel	Processor Families/Processor Number Series
06_25H, 06_2CH	Intel Xeon processors 3600, 5600 series, Intel Core i7, i5 and i3 Processors
06_1EH, 06_1FH	Intel Core i7 and i5 Processors
06_1AH	Intel Core i7 Processor, Intel Xeon processor 3400, 3500, 5500 series
06_1DH	Intel Xeon processor MP 7400 series
06_17H	Intel Xeon processor 3100, 3300, 5200, 5400 series, Intel Core 2 Quad processors 8000, 9000 series
06_0FH	Intel Xeon processor 3000, 3200, 5100, 5300, 7300 series, Intel Core 2 Quad processor 6000 series, Intel Core 2 Extreme 6000 series, Intel Core 2 Duo 4000, 5000, 6000, 7000 series processors, Intel Pentium dual-core processors
06_0EH	Intel Core Duo, Intel Core Solo processors
06_0DH	Intel Pentium M processor
06_7AH	Future Intel® Atom™ processors based on Goldmont Plus Microarchitecture
06_5FH	Intel Atom processors based on Goldmont Microarchitecture (code name Denverton)
06_5CH	Intel Atom processors based on Goldmont Microarchitecture
06_4CH	Intel Atom processor X7-Z8000 and X5-Z8000 series based on Airmont Microarchitecture
06_5DH	Intel Atom processor X3-C3000 based on Silvermont Microarchitecture
06_5AH	Intel Atom processor Z3500 series
06_4AH	Intel Atom processor Z3400 series
06_37H	Intel Atom processor E3000 series, Z3600 series, Z3700 series
06_4DH	Intel Atom processor C2000 series
06_36H	Intel Atom processor S1000 Series
06_1CH, 06_26H, 06_27H, 06_35H, 06_36H	Intel Atom processor family, Intel Atom processor D2000, N2000, E2000, Z2000, C1000 series
0F_06H	Intel Xeon processor 7100, 5000 Series, Intel Xeon Processor MP, Intel Pentium 4, Pentium D processors
0F_03H, 0F_04H	Intel Xeon processor, Intel Xeon processor MP, Intel Pentium 4, Pentium D processors
06_09H	Intel Pentium M processor
0F_02H	Intel Xeon Processor, Intel Xeon processor MP, Intel Pentium 4 processors
0F_0H, 0F_01H	Intel Xeon Processor, Intel Xeon processor MP, Intel Pentium 4 processors
06_7H, 06_08H, 06_0AH, 06_0BH	Intel Pentium III Xeon processor, Intel Pentium III processor
06_03H, 06_05H	Intel Pentium II Xeon processor, Intel Pentium II processor
06_01H	Intel Pentium Pro processor
05_01H, 05_02H, 05_04H	Intel Pentium processor, Intel Pentium processor with MMX Technology

The Intel® Quark™ SoC X1000 processor can be identified by the signature of DisplayFamily_DisplayModel = 05_09H and SteppingID = 0

2.1 ARCHITECTURAL MSRS

Many MSRs have carried over from one generation of IA-32 processors to the next and to Intel 64 processors. A subset of MSRs and associated bit fields, which do not change on future processor generations, are now considered architectural MSRs. For historical reasons (beginning with the Pentium 4 processor), these "architectural MSRs" were given the prefix "IA32_". Table 2-2 lists the architectural MSRs, their addresses, their current names, their names in previous IA-32 processors, and bit fields that are considered architectural. MSR addresses outside Table 2-2 and certain bit fields in an MSR address that may overlap with architectural MSR addresses are model-specific.

Code that accesses a machine specified MSR and that is executed on a processor that does not support that MSR will generate an exception.

Architectural MSR or individual bit fields in an architectural MSR may be introduced or transitioned at the granularity of certain processor family/model or the presence of certain CPUID feature flags. The right-most column of Table 2-2 provides information on the introduction of each architectural MSR or its individual fields. This information is expressed either as signature values of "DF_DM" (see Table 2-1) or via CPUID flags.

Certain bit field position may be related to the maximum physical address width, the value of which is expressed as "MAXPHYADDR" in Table 2-2. "MAXPHYADDR" is reported by CPUID.8000_0008H leaf.

MSR address range between 40000000H - 400000FFH is marked as a specially reserved range. All existing and future processors will not implement any features using any MSR in this range.

Table 2-2. IA-32 Architectural MSRs

Register Address		Architectural MSR Name and bit fields (Former MSR Name)	MSR/Bit Description	Comment
Hex	Decimal			
0H	0	IA32_P5_MC_ADDR (P5_MC_ADDR)	See Section 2.22, "MSRs in Pentium Processors."	**Pentium Processor (05_01H)**
1H	1	IA32_P5_MC_TYPE (P5_MC_TYPE)	See Section 2.22, "MSRs in Pentium Processors."	DF_DM = 05_01H
6H	6	IA32_MONITOR_FILTER_SIZE	See Section 8.10.5, "Monitor/Mwait Address Range Determination."	0F_03H
10H	16	IA32_TIME_STAMP_ COUNTER (TSC)	See Section 17.17, "Time-Stamp Counter."	05_01H
17H	23	IA32_PLATFORM_ID (MSR_PLATFORM_ID)	**Platform ID (RO)** The operating system can use this MSR to determine "slot" information for the processor and the proper microcode update to load.	06_01H
		49:0	Reserved.	
		52:50	**Platform Id (RO)** Contains information concerning the intended platform for the processor. 52 51 50 0 0 0 Processor Flag 0 0 0 1 Processor Flag 1 0 1 0 Processor Flag 2 0 1 1 Processor Flag 3 1 0 0 Processor Flag 4 1 0 1 Processor Flag 5 1 1 0 Processor Flag 6 1 1 1 Processor Flag 7	
		63:53	Reserved.	
1BH	27	IA32_APIC_BASE (APIC_BASE)	This register holds the APIC base address, permitting the relocation of the APIC memory map. See Section 10.4.4, "Local APIC Status and Location" and Section 10.4.5, "Relocating the Local APIC Registers".	06_01H
		7:0	Reserved	
		8	BSP flag (R/W)	

Table 2-2. IA-32 Architectural MSRs (Contd.)

Register Address		Architectural MSR Name and bit fields (Former MSR Name)	MSR/Bit Description	Comment
Hex	Decimal			
		9	Reserved	
		10	Enable x2APIC mode	06_1AH
		11	APIC Global Enable (R/W)	
		(MAXPHYADDR - 1):12	APIC Base (R/W)	
		63: MAXPHYADDR	Reserved	
3AH	58	IA32_FEATURE_CONTROL	**Control Features in Intel 64 Processor (R/W)**	If any one enumeration condition for defined bit field holds
		0	Lock bit (R/WO): (1 = locked). When set, locks this MSR from being written, writes to this bit will result in GP(0). Note: Once the Lock bit is set, the contents of this register cannot be modified. Therefore the lock bit must be set after configuring support for Intel Virtualization Technology and prior to transferring control to an option ROM or the OS. Hence, once the Lock bit is set, the entire IA32_FEATURE_CONTROL contents are preserved across RESET when PWRGOOD is not deasserted.	If any one enumeration condition for defined bit field position greater than bit 0 holds
		1	Enable VMX inside SMX operation (R/WL): This bit enables a system executive to use VMX in conjunction with SMX to support Intel® Trusted Execution Technology. BIOS must set this bit only when the CPUID function 1 returns VMX feature flag and SMX feature flag set (ECX bits 5 and 6 respectively).	If CPUID.01H:ECX[5] = 1 && CPUID.01H:ECX[6] = 1
		2	Enable VMX outside SMX operation (R/WL): This bit enables VMX for system executive that do not require SMX. BIOS must set this bit only when the CPUID function 1 returns VMX feature flag set (ECX bit 5).	If CPUID.01H:ECX[5] = 1
		7:3	Reserved	
		14:8	SENTER Local Function Enables (R/WL): When set, each bit in the field represents an enable control for a corresponding SENTER function. This bit is supported only if CPUID.1:ECX.[bit 6] is set	If CPUID.01H:ECX[6] = 1
		15	SENTER Global Enable (R/WL): This bit must be set to enable SENTER leaf functions. This bit is supported only if CPUID.1:ECX.[bit 6] is set	If CPUID.01H:ECX[6] = 1
		16	Reserved	

Table 2-2. IA-32 Architectural MSRs (Contd.)

Register Address		Architectural MSR Name and bit fields (Former MSR Name)	MSR/Bit Description	Comment
Hex	Decimal			
		17	SGX Launch Control Enable (R/WL): This bit must be set to enable runtime reconfiguration of SGX Launch Control via IA32_SGXLEPUBKEYHASHn MSR.	If CPUID.(EAX=07H, ECX=0H): ECX[30] = 1
		18	SGX Global Enable (R/WL): This bit must be set to enable SGX leaf functions.	If CPUID.(EAX=07H, ECX=0H): EBX[2] = 1
		19	Reserved	
		20	LMCE On (R/WL): When set, system software can program the MSRs associated with LMCE to configure delivery of some machine check exceptions to a single logical processor.	If IA32_MCG_CAP[27] = 1
		63:21	Reserved	
3BH	59	IA32_TSC_ADJUST	Per Logical Processor TSC Adjust (R/Write to clear)	If CPUID.(EAX=07H, ECX=0H): EBX[1] = 1
		63:0	**THREAD_ADJUST:** Local offset value of the IA32_TSC for a logical processor. Reset value is Zero. A write to IA32_TSC will modify the local offset in IA32_TSC_ADJUST and the content of IA32_TSC, but does not affect the internal invariant TSC hardware.	
79H	121	IA32_BIOS_UPDT_TRIG (BIOS_UPDT_TRIG)	BIOS Update Trigger (W) Executing a WRMSR instruction to this MSR causes a microcode update to be loaded into the processor. See Section 9.11.6, "Microcode Update Loader." A processor may prevent writing to this MSR when loading guest states on VM entries or saving guest states on VM exits.	06_01H
8BH	139	IA32_BIOS_SIGN_ID (BIOS_SIGN/BBL_CR_D3)	BIOS Update Signature (RO) Returns the microcode update signature following the execution of CPUID.01H. A processor may prevent writing to this MSR when loading guest states on VM entries or saving guest states on VM exits.	06_01H
		31:0	Reserved	
		63:32	It is recommended that this field be pre-loaded with 0 prior to executing CPUID. If the field remains 0 following the execution of CPUID; this indicates that no microcode update is loaded. Any non-zero value is the microcode update signature.	

Table 2-2. IA-32 Architectural MSRs (Contd.)

Register Address		Architectural MSR Name and bit fields (Former MSR Name)	MSR/Bit Description	Comment
Hex	Decimal			
8CH	140	IA32_SGXLEPUBKEYHASH0	IA32_SGXLEPUBKEYHASH[63:0] (R/W) Bits 63:0 of the SHA256 digest of the SIGSTRUCT.MODULUS for SGX Launch Enclave. On reset, the default value is the digest of Intel's signing key.	Read permitted If CPUID.(EAX=12H,ECX=0H): EAX[0]=1, Write permitted if CPUID.(EAX=12H,ECX=0H): EAX[0]=1 && IA32_FEATURE_CONTROL[17] = 1 && IA32_FEATURE_CONTROL[0] = 1
8DH	141	IA32_SGXLEPUBKEYHASH1	IA32_SGXLEPUBKEYHASH[127:64] (R/W) Bits 127:64 of the SHA256 digest of the SIGSTRUCT.MODULUS for SGX Launch Enclave. On reset, the default value is the digest of Intel's signing key.	Read permitted If CPUID.(EAX=12H,ECX=0H): EAX[0]=1, Write permitted if CPUID.(EAX=12H,ECX=0H): EAX[0]=1 && IA32_FEATURE_CONTROL[17] = 1 && IA32_FEATURE_CONTROL[0] = 1
8EH	142	IA32_SGXLEPUBKEYHASH2	IA32_SGXLEPUBKEYHASH[191:128] (R/W) Bits 191:128 of the SHA256 digest of the SIGSTRUCT.MODULUS for SGX Launch Enclave. On reset, the default value is the digest of Intel's signing key.	Read permitted If CPUID.(EAX=12H,ECX=0H): EAX[0]=1, Write permitted if CPUID.(EAX=12H,ECX=0H): EAX[0]=1 && IA32_FEATURE_CONTROL[17] = 1 && IA32_FEATURE_CONTROL[0] = 1
8FH	143	IA32_SGXLEPUBKEYHASH3	IA32_SGXLEPUBKEYHASH[255:192] (R/W) Bits 255:192 of the SHA256 digest of the SIGSTRUCT.MODULUS for SGX Launch Enclave. On reset, the default value is the digest of Intel's signing key.	
9BH	155	IA32_SMM_MONITOR_CTL	SMM Monitor Configuration (R/W)	If CPUID.01H: ECX[5]=1 || CPUID.01H: ECX[6] = 1
		0	Valid (R/W)	
		1	Reserved	
		2	Controls SMI unblocking by VMXOFF (see Section 34.14.4)	If IA32_VMX_MISC[28]
		11:3	Reserved	
		31:12	MSEG Base (R/W)	
		63:32	Reserved	
9EH	158	IA32_SMBASE	Base address of the logical processor's SMRAM image (RO, SMM only)	If IA32_VMX_MISC[15]
C1H	193	IA32_PMC0 (PERFCTR0)	General Performance Counter 0 (R/W)	If CPUID.0AH: EAX[15:8] > 0
C2H	194	IA32_PMC1 (PERFCTR1)	General Performance Counter 1 (R/W)	If CPUID.0AH: EAX[15:8] > 1

Table 2-2. IA-32 Architectural MSRs (Contd.)

Register Address		Architectural MSR Name and bit fields (Former MSR Name)	MSR/Bit Description	Comment
Hex	Decimal			
C3H	195	IA32_PMC2	General Performance Counter 2 (R/W)	If CPUID.0AH: EAX[15:8] > 2
C4H	196	IA32_PMC3	General Performance Counter 3 (R/W)	If CPUID.0AH: EAX[15:8] > 3
C5H	197	IA32_PMC4	General Performance Counter 4 (R/W)	If CPUID.0AH: EAX[15:8] > 4
C6H	198	IA32_PMC5	General Performance Counter 5 (R/W)	If CPUID.0AH: EAX[15:8] > 5
C7H	199	IA32_PMC6	General Performance Counter 6 (R/W)	If CPUID.0AH: EAX[15:8] > 6
C8H	200	IA32_PMC7	General Performance Counter 7 (R/W)	If CPUID.0AH: EAX[15:8] > 7
E7H	231	IA32_MPERF	TSC Frequency Clock Counter (R/Write to clear)	If CPUID.06H: ECX[0] = 1
		63:0	**C0_MCNT: C0 TSC Frequency Clock Count** Increments at fixed interval (relative to TSC freq.) when the logical processor is in C0. Cleared upon overflow / wrap-around of IA32_APERF.	
E8H	232	IA32_APERF	Actual Performance Clock Counter (R/Write to clear).	If CPUID.06H: ECX[0] = 1
		63:0	**C0_ACNT: C0 Actual Frequency Clock Count** Accumulates core clock counts at the coordinated clock frequency, when the logical processor is in C0. Cleared upon overflow / wrap-around of IA32_MPERF.	
FEH	254	IA32_MTRRCAP (MTRRcap)	MTRR Capability (RO) Section 11.11.2.1, "IA32_MTRR_DEF_TYPE MSR."	06_01H
		7:0	VCNT: The number of variable memory type ranges in the processor.	
		8	Fixed range MTRRs are supported when set.	
		9	Reserved.	
		10	WC Supported when set.	
		11	SMRR Supported when set.	
		63:12	Reserved.	
174H	372	IA32_SYSENTER_CS	SYSENTER_CS_MSR (R/W)	06_01H
		15:0	CS Selector	
		63:16	Reserved.	
175H	373	IA32_SYSENTER_ESP	SYSENTER_ESP_MSR (R/W)	06_01H
176H	374	IA32_SYSENTER_EIP	SYSENTER_EIP_MSR (R/W)	06_01H

Table 2-2. IA-32 Architectural MSRs (Contd.)

Register Address		Architectural MSR Name and bit fields (Former MSR Name)	MSR/Bit Description	Comment
Hex	Decimal			
179H	377	IA32_MCG_CAP (MCG_CAP)	Global Machine Check Capability (RO)	06_01H
		7:0	Count: Number of reporting banks.	
		8	MCG_CTL_P: IA32_MCG_CTL is present if this bit is set	
		9	MCG_EXT_P: Extended machine check state registers are present if this bit is set	
		10	MCP_CMCI_P: Support for corrected MC error event is present.	06_01H
		11	MCG_TES_P: Threshold-based error status register are present if this bit is set.	
		15:12	Reserved	
		23:16	MCG_EXT_CNT: Number of extended machine check state registers present.	
		24	MCG_SER_P: The processor supports software error recovery if this bit is set.	
		25	Reserved.	
		26	MCG_ELOG_P: Indicates that the processor allows platform firmware to be invoked when an error is detected so that it may provide additional platform specific information in an ACPI format "Generic Error Data Entry" that augments the data included in machine check bank registers.	06_3EH
		27	MCG_LMCE_P: Indicates that the processor support extended state in IA32_MCG_STATUS and associated MSR necessary to configure Local Machine Check Exception (LMCE).	06_3EH
		63:28	Reserved.	
17AH	378	IA32_MCG_STATUS (MCG_STATUS)	Global Machine Check Status (R/W0)	06_01H
		0	RIPV. Restart IP valid	06_01H
		1	EIPV. Error IP valid	06_01H
		2	MCIP. Machine check in progress	06_01H
		3	LMCE_S.	If IA32_MCG_CAP.LMCE_P[27] =1
		63:4	Reserved.	
17BH	379	IA32_MCG_CTL (MCG_CTL)	Global Machine Check Control (R/W)	If IA32_MCG_CAP.CTL_P[8] =1
180H-185H	384-389	Reserved		06_0EH[1]
186H	390	IA32_PERFEVTSEL0 (PERFEVTSEL0)	Performance Event Select Register 0 (R/W)	If CPUID.0AH: EAX[15:8] > 0

Table 2-2. IA-32 Architectural MSRs (Contd.)

Register Address		Architectural MSR Name and bit fields (Former MSR Name)	MSR/Bit Description	Comment
Hex	Decimal			
		7:0	Event Select: Selects a performance event logic unit.	
		15:8	UMask: Qualifies the microarchitectural condition to detect on the selected event logic.	
		16	USR: Counts while in privilege level is not ring 0.	
		17	OS: Counts while in privilege level is ring 0.	
		18	Edge: Enables edge detection if set.	
		19	PC: enables pin control.	
		20	INT: enables interrupt on counter overflow.	
		21	AnyThread: When set to 1, it enables counting the associated event conditions occurring across all logical processors sharing a processor core. When set to 0, the counter only increments the associated event conditions occurring in the logical processor which programmed the MSR.	
		22	EN: enables the corresponding performance counter to commence counting when this bit is set.	
		23	INV: invert the CMASK.	
		31:24	CMASK: When CMASK is not zero, the corresponding performance counter increments each cycle if the event count is greater than or equal to the CMASK.	
		63:32	Reserved.	
187H	391	IA32_PERFEVTSEL1 (PERFEVTSEL1)	Performance Event Select Register 1 (R/W)	If CPUID.0AH: EAX[15:8] > 1
188H	392	IA32_PERFEVTSEL2	Performance Event Select Register 2 (R/W)	If CPUID.0AH: EAX[15:8] > 2
189H	393	IA32_PERFEVTSEL3	Performance Event Select Register 3 (R/W)	If CPUID.0AH: EAX[15:8] > 3
18AH-197H	394-407	Reserved		06_0EH[2]
198H	408	IA32_PERF_STATUS	Current performance status. (RO) See Section 14.1.1, "Software Interface For Initiating Performance State Transitions".	0F_03H
		15:0	Current performance State Value	
		63:16	Reserved.	

Table 2-2. IA-32 Architectural MSRs (Contd.)

Register Address		Architectural MSR Name and bit fields (Former MSR Name)	MSR/Bit Description	Comment
Hex	Decimal			
199H	409	IA32_PERF_CTL	Performance Control MSR. (R/W) Software makes a request for a new Performance state (P-State) by writing this MSR. See Section 14.1.1, "Software Interface For Initiating Performance State Transitions".	0F_03H
		15:0	Target performance State Value	
		31:16	Reserved.	
		32	IDA Engage. (R/W) When set to 1: disengages IDA	06_0FH (Mobile only)
		63:33	Reserved.	
19AH	410	IA32_CLOCK_MODULATION	Clock Modulation Control (R/W) See Section 14.7.3, "Software Controlled Clock Modulation."	If CPUID.01H:EDX[22] = 1
		0	Extended On-Demand Clock Modulation Duty Cycle:	If CPUID.06H:EAX[5] = 1
		3:1	On-Demand Clock Modulation Duty Cycle: Specific encoded values for target duty cycle modulation.	If CPUID.01H:EDX[22] = 1
		4	On-Demand Clock Modulation Enable: Set 1 to enable modulation.	If CPUID.01H:EDX[22] = 1
		63:5	Reserved.	
19BH	411	IA32_THERM_INTERRUPT	Thermal Interrupt Control (R/W) Enables and disables the generation of an interrupt on temperature transitions detected with the processor's thermal sensors and thermal monitor. See Section 14.7.2, "Thermal Monitor."	If CPUID.01H:EDX[22] = 1
		0	High-Temperature Interrupt Enable	If CPUID.01H:EDX[22] = 1
		1	Low-Temperature Interrupt Enable	If CPUID.01H:EDX[22] = 1
		2	PROCHOT# Interrupt Enable	If CPUID.01H:EDX[22] = 1
		3	FORCEPR# Interrupt Enable	If CPUID.01H:EDX[22] = 1
		4	Critical Temperature Interrupt Enable	If CPUID.01H:EDX[22] = 1
		7:5	Reserved.	
		14:8	Threshold #1 Value	If CPUID.01H:EDX[22] = 1
		15	Threshold #1 Interrupt Enable	If CPUID.01H:EDX[22] = 1
		22:16	Threshold #2 Value	If CPUID.01H:EDX[22] = 1
		23	Threshold #2 Interrupt Enable	If CPUID.01H:EDX[22] = 1
		24	Power Limit Notification Enable	If CPUID.06H:EAX[4] = 1
		63:25	Reserved.	

Table 2-2. IA-32 Architectural MSRs (Contd.)

Register Address		Architectural MSR Name and bit fields (Former MSR Name)	MSR/Bit Description	Comment
Hex	Decimal			
19CH	412	IA32_THERM_STATUS	Thermal Status Information (RO) Contains status information about the processor's thermal sensor and automatic thermal monitoring facilities. See Section 14.7.2, "Thermal Monitor"	If CPUID.01H:EDX[22] = 1
		0	Thermal Status (RO):	If CPUID.01H:EDX[22] = 1
		1	Thermal Status Log (R/W):	If CPUID.01H:EDX[22] = 1
		2	PROCHOT # or FORCEPR# event (RO)	If CPUID.01H:EDX[22] = 1
		3	PROCHOT # or FORCEPR# log (R/WC0)	If CPUID.01H:EDX[22] = 1
		4	Critical Temperature Status (RO)	If CPUID.01H:EDX[22] = 1
		5	Critical Temperature Status log (R/WC0)	If CPUID.01H:EDX[22] = 1
		6	Thermal Threshold #1 Status (RO)	If CPUID.01H:ECX[8] = 1
		7	Thermal Threshold #1 log (R/WC0)	If CPUID.01H:ECX[8] = 1
		8	Thermal Threshold #2 Status (RO)	If CPUID.01H:ECX[8] = 1
		9	Thermal Threshold #2 log (R/WC0)	If CPUID.01H:ECX[8] = 1
		10	Power Limitation Status (RO)	If CPUID.06H:EAX[4] = 1
		11	Power Limitation log (R/WC0)	If CPUID.06H:EAX[4] = 1
		12	Current Limit Status (RO)	If CPUID.06H:EAX[7] = 1
		13	Current Limit log (R/WC0)	If CPUID.06H:EAX[7] = 1
		14	Cross Domain Limit Status (RO)	If CPUID.06H:EAX[7] = 1
		15	Cross Domain Limit log (R/WC0)	If CPUID.06H:EAX[7] = 1
		22:16	Digital Readout (RO)	If CPUID.06H:EAX[0] = 1
		26:23	Reserved.	
		30:27	Resolution in Degrees Celsius (RO)	If CPUID.06H:EAX[0] = 1
		31	Reading Valid (RO)	If CPUID.06H:EAX[0] = 1
		63:32	Reserved.	
1A0H	416	IA32_MISC_ENABLE	**Enable Misc. Processor Features (R/W)** Allows a variety of processor functions to be enabled and disabled.	
		0	**Fast-Strings Enable** When set, the fast-strings feature (for REP MOVS and REP STORS) is enabled (default); when clear, fast-strings are disabled.	0F_0H
		2:1	Reserved.	

Table 2-2. IA-32 Architectural MSRs (Contd.)

Register Address		Architectural MSR Name and bit fields (Former MSR Name)	MSR/Bit Description	Comment
Hex	Decimal			
		3	**Automatic Thermal Control Circuit Enable (R/W)** 1 = Setting this bit enables the thermal control circuit (TCC) portion of the Intel Thermal Monitor feature. This allows the processor to automatically reduce power consumption in response to TCC activation. 0 = Disabled. Note: In some products clearing this bit might be ignored in critical thermal conditions, and TM1, TM2 and adaptive thermal throttling will still be activated. The default value of this field varies with product . See respective tables where default value is listed.	0F_0H
		6:4	Reserved	
		7	**Performance Monitoring Available (R)** 1 = Performance monitoring enabled 0 = Performance monitoring disabled	0F_0H
		10:8	Reserved.	
		11	**Branch Trace Storage Unavailable (RO)** 1 = Processor doesn't support branch trace storage (BTS) 0 = BTS is supported	0F_0H
		12	**Processor Event Based Sampling (PEBS) Unavailable (RO)** 1 = PEBS is not supported; 0 = PEBS is supported.	06_0FH
		15:13	Reserved.	
		16	**Enhanced Intel SpeedStep Technology Enable (R/W)** 0= Enhanced Intel SpeedStep Technology disabled 1 = Enhanced Intel SpeedStep Technology enabled	If CPUID.01H: ECX[7] =1
		17	Reserved.	

Table 2-2. IA-32 Architectural MSRs (Contd.)

Register Address		Architectural MSR Name and bit fields (Former MSR Name)	MSR/Bit Description	Comment
Hex	Decimal			
		18	**ENABLE MONITOR FSM (R/W)** When this bit is set to 0, the MONITOR feature flag is not set (CPUID.01H:ECX[bit 3] = 0). This indicates that MONITOR/MWAIT are not supported. Software attempts to execute MONITOR/MWAIT will cause #UD when this bit is 0. When this bit is set to 1 (default), MONITOR/MWAIT are supported (CPUID.01H:ECX[bit 3] = 1). If the SSE3 feature flag ECX[0] is not set (CPUID.01H:ECX[bit 0] = 0), the OS must not attempt to alter this bit. BIOS must leave it in the default state. Writing this bit when the SSE3 feature flag is set to 0 may generate a #GP exception.	0F_03H
		21:19	Reserved.	
		22	**Limit CPUID Maxval (R/W)** When this bit is set to 1, CPUID.00H returns a maximum value in EAX[7:0] of 2. BIOS should contain a setup question that allows users to specify when the installed OS does not support CPUID functions greater than 2. Before setting this bit, BIOS must execute the CPUID.0H and examine the maximum value returned in EAX[7:0]. If the maximum value is greater than 2, this bit is supported. Otherwise, this bit is not supported. Setting this bit when the maximum value is not greater than 2 may generate a #GP exception. Setting this bit may cause unexpected behavior in software that depends on the availability of CPUID leaves greater than 2.	0F_03H
		23	**xTPR Message Disable (R/W)** When set to 1, xTPR messages are disabled. xTPR messages are optional messages that allow the processor to inform the chipset of its priority.	if CPUID.01H:ECX[14] = 1
		33:24	Reserved.	

Table 2-2. IA-32 Architectural MSRs (Contd.)

Register Address		Architectural MSR Name and bit fields (Former MSR Name)	MSR/Bit Description	Comment
Hex	Decimal			
		34	**XD Bit Disable (R/W)** When set to 1, the Execute Disable Bit feature (XD Bit) is disabled and the XD Bit extended feature flag will be clear (CPUID.80000001H: EDX[20]=0). When set to a 0 (default), the Execute Disable Bit feature (if available) allows the OS to enable PAE paging and take advantage of data only pages. BIOS must not alter the contents of this bit location, if XD bit is not supported. Writing this bit to 1 when the XD Bit extended feature flag is set to 0 may generate a #GP exception.	if CPUID.80000001H:EDX[20] = 1
		63:35	Reserved.	
1B0H	432	IA32_ENERGY_PERF_BIAS	Performance Energy Bias Hint (R/W)	if CPUID.6H:ECX[3] = 1
		3:0	Power Policy Preference: 0 indicates preference to highest performance. 15 indicates preference to maximize energy saving.	
		63:4	Reserved.	
1B1H	433	IA32_PACKAGE_THERM_STATUS	Package Thermal Status Information (RO) Contains status information about the package's thermal sensor. See Section 14.8, "Package Level Thermal Management."	If CPUID.06H: EAX[6] = 1
		0	Pkg Thermal Status (RO):	
		1	Pkg Thermal Status Log (R/W):	
		2	Pkg PROCHOT # event (RO)	
		3	Pkg PROCHOT # log (R/WC0)	
		4	Pkg Critical Temperature Status (RO)	
		5	Pkg Critical Temperature Status log (R/WC0)	
		6	Pkg Thermal Threshold #1 Status (RO)	
		7	Pkg Thermal Threshold #1 log (R/WC0)	
		8	Pkg Thermal Threshold #2 Status (RO)	
		9	Pkg Thermal Threshold #1 log (R/WC0)	
		10	Pkg Power Limitation Status (RO)	
		11	Pkg Power Limitation log (R/WC0)	
		15:12	Reserved.	
		22:16	Pkg Digital Readout (RO)	

Table 2-2. IA-32 Architectural MSRs (Contd.)

Register Address		Architectural MSR Name and bit fields (Former MSR Name)	MSR/Bit Description	Comment
Hex	Decimal			
		63:23	Reserved.	
1B2H	434	IA32_PACKAGE_THERM_INTERRUPT	Pkg Thermal Interrupt Control (R/W) Enables and disables the generation of an interrupt on temperature transitions detected with the package's thermal sensor. See Section 14.8, "Package Level Thermal Management."	If CPUID.06H: EAX[6] = 1
		0	Pkg High-Temperature Interrupt Enable	
		1	Pkg Low-Temperature Interrupt Enable	
		2	Pkg PROCHOT# Interrupt Enable	
		3	Reserved.	
		4	Pkg Overheat Interrupt Enable	
		7:5	Reserved.	
		14:8	Pkg Threshold #1 Value	
		15	Pkg Threshold #1 Interrupt Enable	
		22:16	Pkg Threshold #2 Value	
		23	Pkg Threshold #2 Interrupt Enable	
		24	Pkg Power Limit Notification Enable	
		63:25	Reserved.	
1D9H	473	IA32_DEBUGCTL (MSR_DEBUGCTLA, MSR_DEBUGCTLB)	Trace/Profile Resource Control (R/W)	06_0EH
		0	LBR: Setting this bit to 1 enables the processor to record a running trace of the most recent branches taken by the processor in the LBR stack.	06_01H
		1	BTF: Setting this bit to 1 enables the processor to treat EFLAGS.TF as single-step on branches instead of single-step on instructions.	06_01H
		5:2	Reserved.	
		6	TR: Setting this bit to 1 enables branch trace messages to be sent.	06_0EH
		7	BTS: Setting this bit enables branch trace messages (BTMs) to be logged in a BTS buffer.	06_0EH
		8	BTINT: When clear, BTMs are logged in a BTS buffer in circular fashion. When this bit is set, an interrupt is generated by the BTS facility when the BTS buffer is full.	06_0EH
		9	1: BTS_OFF_OS: When set, BTS or BTM is skipped if CPL = 0.	06_0FH

Table 2-2. IA-32 Architectural MSRs (Contd.)

Register Address		Architectural MSR Name and bit fields (Former MSR Name)	MSR/Bit Description	Comment
Hex	Decimal			
		10	BTS_OFF_USR: When set, BTS or BTM is skipped if CPL > 0.	06_0FH
		11	FREEZE_LBRS_ON_PMI: When set, the LBR stack is frozen on a PMI request.	If CPUID.01H: ECX[15] = 1 && CPUID.0AH: EAX[7:0] > 1
		12	FREEZE_PERFMON_ON_PMI: When set, each ENABLE bit of the global counter control MSR are frozen (address 38FH) on a PMI request	If CPUID.01H: ECX[15] = 1 && CPUID.0AH: EAX[7:0] > 1
		13	ENABLE_UNCORE_PMI: When set, enables the logical processor to receive and generate PMI on behalf of the uncore.	06_1AH
		14	FREEZE_WHILE_SMM: When set, freezes perfmon and trace messages while in SMM.	If IA32_PERF_CAPABILITIES[12] = 1
		15	RTM_DEBUG: When set, enables DR7 debug bit on XBEGIN	If (CPUID.(EAX=07H, ECX=0):EBX[11] = 1)
		63:16	Reserved.	
1F2H	498	IA32_SMRR_PHYSBASE	**SMRR Base Address (Writeable only in SMM)** Base address of SMM memory range.	If IA32_MTRRCAP.SMRR[11] = 1
		7:0	Type. Specifies memory type of the range.	
		11:8	Reserved.	
		31:12	**PhysBase.** SMRR physical Base Address.	
		63:32	Reserved.	
1F3H	499	IA32_SMRR_PHYSMASK	**SMRR Range Mask. (Writeable only in SMM)** Range Mask of SMM memory range.	If IA32_MTRRCAP[SMRR] = 1
		10:0	Reserved.	
		11	**Valid** Enable range mask.	
		31:12	**PhysMask** SMRR address range mask.	
		63:32	Reserved.	
1F8H	504	IA32_PLATFORM_DCA_CAP	DCA Capability (R)	If CPUID.01H: ECX[18] = 1
1F9H	505	IA32_CPU_DCA_CAP	If set, CPU supports Prefetch-Hint type.	If CPUID.01H: ECX[18] = 1
1FAH	506	IA32_DCA_0_CAP	DCA type 0 Status and Control register.	If CPUID.01H: ECX[18] = 1
		0	DCA_ACTIVE: Set by HW when DCA is fuse-enabled and no defeatures are set.	
		2:1	TRANSACTION	
		6:3	DCA_TYPE	

Table 2-2. IA-32 Architectural MSRs (Contd.)

Register Address		Architectural MSR Name and bit fields (Former MSR Name)	MSR/Bit Description	Comment
Hex	Decimal			
		10:7	DCA_QUEUE_SIZE	
		12:11	Reserved.	
		16:13	DCA_DELAY: Writes will update the register but have no HW side-effect.	
		23:17	Reserved.	
		24	SW_BLOCK: SW can request DCA block by setting this bit.	
		25	Reserved.	
		26	HW_BLOCK: Set when DCA is blocked by HW (e.g. CR0.CD = 1).	
		31:27	Reserved.	
200H	512	IA32_MTRR_PHYSBASE0 (MTRRphysBase0)	See Section 11.11.2.3, "Variable Range MTRRs."	If CPUID.01H: EDX.MTRR[12] =1
201H	513	IA32_MTRR_PHYSMASK0	MTRRphysMask0	If CPUID.01H: EDX.MTRR[12] =1
202H	514	IA32_MTRR_PHYSBASE1	MTRRphysBase1	If CPUID.01H: EDX.MTRR[12] =1
203H	515	IA32_MTRR_PHYSMASK1	MTRRphysMask1	If CPUID.01H: EDX.MTRR[12] =1
204H	516	IA32_MTRR_PHYSBASE2	MTRRphysBase2	If CPUID.01H: EDX.MTRR[12] =1
205H	517	IA32_MTRR_PHYSMASK2	MTRRphysMask2	If CPUID.01H: EDX.MTRR[12] =1
206H	518	IA32_MTRR_PHYSBASE3	MTRRphysBase3	If CPUID.01H: EDX.MTRR[12] =1
207H	519	IA32_MTRR_PHYSMASK3	MTRRphysMask3	If CPUID.01H: EDX.MTRR[12] =1
208H	520	IA32_MTRR_PHYSBASE4	MTRRphysBase4	If CPUID.01H: EDX.MTRR[12] =1
209H	521	IA32_MTRR_PHYSMASK4	MTRRphysMask4	If CPUID.01H: EDX.MTRR[12] =1
20AH	522	IA32_MTRR_PHYSBASE5	MTRRphysBase5	If CPUID.01H: EDX.MTRR[12] =1
20BH	523	IA32_MTRR_PHYSMASK5	MTRRphysMask5	If CPUID.01H: EDX.MTRR[12] =1
20CH	524	IA32_MTRR_PHYSBASE6	MTRRphysBase6	If CPUID.01H: EDX.MTRR[12] =1
20DH	525	IA32_MTRR_PHYSMASK6	MTRRphysMask6	If CPUID.01H: EDX.MTRR[12] =1
20EH	526	IA32_MTRR_PHYSBASE7	MTRRphysBase7	If CPUID.01H: EDX.MTRR[12] =1
20FH	527	IA32_MTRR_PHYSMASK7	MTRRphysMask7	If CPUID.01H: EDX.MTRR[12] =1

Table 2-2. IA-32 Architectural MSRs (Contd.)

Register Address		Architectural MSR Name and bit fields (Former MSR Name)	MSR/Bit Description	Comment
Hex	Decimal			
210H	528	IA32_MTRR_PHYSBASE8	MTRRphysBase8	if IA32_MTRRCAP[7:0] > 8
211H	529	IA32_MTRR_PHYSMASK8	MTRRphysMask8	if IA32_MTRRCAP[7:0] > 8
212H	530	IA32_MTRR_PHYSBASE9	MTRRphysBase9	if IA32_MTRRCAP[7:0] > 9
213H	531	IA32_MTRR_PHYSMASK9	MTRRphysMask9	if IA32_MTRRCAP[7:0] > 9
250H	592	IA32_MTRR_FIX64K_00000	MTRRfix64K_00000	If CPUID.01H: EDX.MTRR[12] =1
258H	600	IA32_MTRR_FIX16K_80000	MTRRfix16K_80000	If CPUID.01H: EDX.MTRR[12] =1
259H	601	IA32_MTRR_FIX16K_A0000	MTRRfix16K_A0000	If CPUID.01H: EDX.MTRR[12] =1
268H	616	IA32_MTRR_FIX4K_C0000 (MTRRfix4K_C0000)	See Section 11.11.2.2, "Fixed Range MTRRs."	If CPUID.01H: EDX.MTRR[12] =1
269H	617	IA32_MTRR_FIX4K_C8000	MTRRfix4K_C8000	If CPUID.01H: EDX.MTRR[12] =1
26AH	618	IA32_MTRR_FIX4K_D0000	MTRRfix4K_D0000	If CPUID.01H: EDX.MTRR[12] =1
26BH	619	IA32_MTRR_FIX4K_D8000	MTRRfix4K_D8000	If CPUID.01H: EDX.MTRR[12] =1
26CH	620	IA32_MTRR_FIX4K_E0000	MTRRfix4K_E0000	If CPUID.01H: EDX.MTRR[12] =1
26DH	621	IA32_MTRR_FIX4K_E8000	MTRRfix4K_E8000	If CPUID.01H: EDX.MTRR[12] =1
26EH	622	IA32_MTRR_FIX4K_F0000	MTRRfix4K_F0000	If CPUID.01H: EDX.MTRR[12] =1
26FH	623	IA32_MTRR_FIX4K_F8000	MTRRfix4K_F8000	If CPUID.01H: EDX.MTRR[12] =1
277H	631	IA32_PAT	IA32_PAT (R/W)	If CPUID.01H: EDX.MTRR[16] =1
		2:0	PA0	
		7:3	Reserved.	
		10:8	PA1	
		15:11	Reserved.	
		18:16	PA2	
		23:19	Reserved.	
		26:24	PA3	
		31:27	Reserved.	
		34:32	PA4	
		39:35	Reserved.	
		42:40	PA5	
		47:43	Reserved.	
		50:48	PA6	

Table 2-2. IA-32 Architectural MSRs (Contd.)

Register Address		Architectural MSR Name and bit fields (Former MSR Name)	MSR/Bit Description	Comment
Hex	Decimal			
		55:51	Reserved.	
		58:56	PA7	
		63:59	Reserved.	
280H	640	IA32_MC0_CTL2	MSR to enable/disable CMCI capability for bank 0. (R/W) See Section 15.3.2.5, "IA32_MCi_CTL2 MSRs".	If IA32_MCG_CAP[10] = 1 && IA32_MCG_CAP[7:0] > 0
		14:0	Corrected error count threshold.	
		29:15	Reserved.	
		30	CMCI_EN	
		63:31	Reserved.	
281H	641	IA32_MC1_CTL2	(R/W) same fields as IA32_MC0_CTL2.	If IA32_MCG_CAP[10] = 1 && IA32_MCG_CAP[7:0] > 1
282H	642	IA32_MC2_CTL2	(R/W) same fields as IA32_MC0_CTL2.	If IA32_MCG_CAP[10] = 1 && IA32_MCG_CAP[7:0] > 2
283H	643	IA32_MC3_CTL2	(R/W) same fields as IA32_MC0_CTL2.	If IA32_MCG_CAP[10] = 1 && IA32_MCG_CAP[7:0] > 3
284H	644	IA32_MC4_CTL2	(R/W) same fields as IA32_MC0_CTL2.	If IA32_MCG_CAP[10] = 1 && IA32_MCG_CAP[7:0] > 4
285H	645	IA32_MC5_CTL2	(R/W) same fields as IA32_MC0_CTL2.	If IA32_MCG_CAP[10] = 1 && IA32_MCG_CAP[7:0] > 5
286H	646	IA32_MC6_CTL2	(R/W) same fields as IA32_MC0_CTL2.	If IA32_MCG_CAP[10] = 1 && IA32_MCG_CAP[7:0] > 6
287H	647	IA32_MC7_CTL2	(R/W) same fields as IA32_MC0_CTL2.	If IA32_MCG_CAP[10] = 1 && IA32_MCG_CAP[7:0] > 7
288H	648	IA32_MC8_CTL2	(R/W) same fields as IA32_MC0_CTL2.	If IA32_MCG_CAP[10] = 1 && IA32_MCG_CAP[7:0] > 8
289H	649	IA32_MC9_CTL2	(R/W) same fields as IA32_MC0_CTL2.	If IA32_MCG_CAP[10] = 1 && IA32_MCG_CAP[7:0] > 9
28AH	650	IA32_MC10_CTL2	(R/W) same fields as IA32_MC0_CTL2.	If IA32_MCG_CAP[10] = 1 && IA32_MCG_CAP[7:0] > 10
28BH	651	IA32_MC11_CTL2	(R/W) same fields as IA32_MC0_CTL2.	If IA32_MCG_CAP[10] = 1 && IA32_MCG_CAP[7:0] > 11

Table 2-2. IA-32 Architectural MSRs (Contd.)

Register Address		Architectural MSR Name and bit fields (Former MSR Name)	MSR/Bit Description	Comment
Hex	Decimal			
28CH	652	IA32_MC12_CTL2	(R/W) same fields as IA32_MC0_CTL2.	If IA32_MCG_CAP[10] = 1 && IA32_MCG_CAP[7:0] > 12
28DH	653	IA32_MC13_CTL2	(R/W) same fields as IA32_MC0_CTL2.	If IA32_MCG_CAP[10] = 1 && IA32_MCG_CAP[7:0] > 13
28EH	654	IA32_MC14_CTL2	(R/W) same fields as IA32_MC0_CTL2.	If IA32_MCG_CAP[10] = 1 && IA32_MCG_CAP[7:0] > 14
28FH	655	IA32_MC15_CTL2	(R/W) same fields as IA32_MC0_CTL2.	If IA32_MCG_CAP[10] = 1 && IA32_MCG_CAP[7:0] > 15
290H	656	IA32_MC16_CTL2	(R/W) same fields as IA32_MC0_CTL2.	If IA32_MCG_CAP[10] = 1 && IA32_MCG_CAP[7:0] > 16
291H	657	IA32_MC17_CTL2	(R/W) same fields as IA32_MC0_CTL2.	If IA32_MCG_CAP[10] = 1 && IA32_MCG_CAP[7:0] > 17
292H	658	IA32_MC18_CTL2	(R/W) same fields as IA32_MC0_CTL2.	If IA32_MCG_CAP[10] = 1 && IA32_MCG_CAP[7:0] > 18
293H	659	IA32_MC19_CTL2	(R/W) same fields as IA32_MC0_CTL2.	If IA32_MCG_CAP[10] = 1 && IA32_MCG_CAP[7:0] > 19
294H	660	IA32_MC20_CTL2	(R/W) same fields as IA32_MC0_CTL2.	If IA32_MCG_CAP[10] = 1 && IA32_MCG_CAP[7:0] > 20
295H	661	IA32_MC21_CTL2	(R/W) same fields as IA32_MC0_CTL2.	If IA32_MCG_CAP[10] = 1 && IA32_MCG_CAP[7:0] > 21
296H	662	IA32_MC22_CTL2	(R/W) same fields as IA32_MC0_CTL2.	If IA32_MCG_CAP[10] = 1 && IA32_MCG_CAP[7:0] > 22
297H	663	IA32_MC23_CTL2	(R/W) same fields as IA32_MC0_CTL2.	If IA32_MCG_CAP[10] = 1 && IA32_MCG_CAP[7:0] > 23
298H	664	IA32_MC24_CTL2	(R/W) same fields as IA32_MC0_CTL2.	If IA32_MCG_CAP[10] = 1 && IA32_MCG_CAP[7:0] > 24
299H	665	IA32_MC25_CTL2	(R/W) same fields as IA32_MC0_CTL2.	If IA32_MCG_CAP[10] = 1 && IA32_MCG_CAP[7:0] > 25
29AH	666	IA32_MC26_CTL2	(R/W) same fields as IA32_MC0_CTL2.	If IA32_MCG_CAP[10] = 1 && IA32_MCG_CAP[7:0] > 26

Table 2-2. IA-32 Architectural MSRs (Contd.)

Register Address		Architectural MSR Name and bit fields (Former MSR Name)	MSR/Bit Description	Comment
Hex	Decimal			
29BH	667	IA32_MC27_CTL2	(R/W) same fields as IA32_MC0_CTL2.	If IA32_MCG_CAP[10] = 1 && IA32_MCG_CAP[7:0] > 27
29CH	668	IA32_MC28_CTL2	(R/W) same fields as IA32_MC0_CTL2.	If IA32_MCG_CAP[10] = 1 && IA32_MCG_CAP[7:0] > 28
29DH	669	IA32_MC29_CTL2	(R/W) same fields as IA32_MC0_CTL2.	If IA32_MCG_CAP[10] = 1 && IA32_MCG_CAP[7:0] > 29
29EH	670	IA32_MC30_CTL2	(R/W) same fields as IA32_MC0_CTL2.	If IA32_MCG_CAP[10] = 1 && IA32_MCG_CAP[7:0] > 30
29FH	671	IA32_MC31_CTL2	(R/W) same fields as IA32_MC0_CTL2.	If IA32_MCG_CAP[10] = 1 && IA32_MCG_CAP[7:0] > 31
2FFH	767	IA32_MTRR_DEF_TYPE	MTRRdefType (R/W)	If CPUID.01H: EDX.MTRR[12] =1
		2:0	Default Memory Type	
		9:3	Reserved.	
		10	Fixed Range MTRR Enable	
		11	MTRR Enable	
		63:12	Reserved.	
309H	777	IA32_FIXED_CTR0 (MSR_PERF_FIXED_CTR0)	Fixed-Function Performance Counter 0 (R/W): Counts Instr_Retired.Any.	If CPUID.0AH: EDX[4:0] > 0
30AH	778	IA32_FIXED_CTR1 (MSR_PERF_FIXED_CTR1)	Fixed-Function Performance Counter 1 (R/W): Counts CPU_CLK_Unhalted.Core	If CPUID.0AH: EDX[4:0] > 1
30BH	779	IA32_FIXED_CTR2 (MSR_PERF_FIXED_CTR2)	Fixed-Function Performance Counter 2 (R/W): Counts CPU_CLK_Unhalted.Ref	If CPUID.0AH: EDX[4:0] > 2
345H	837	IA32_PERF_CAPABILITIES	Read Only MSR that enumerates the existence of performance monitoring features. (RO)	If CPUID.01H: ECX[15] = 1
		5:0	LBR format	
		6	PEBS Trap	
		7	PEBSSaveArchRegs	
		11:8	PEBS Record Format	
		12	1: Freeze while SMM is supported.	
		13	1: Full width of counter writable via IA32_A_PMCx.	
		63:14	Reserved.	

Table 2-2. IA-32 Architectural MSRs (Contd.)

Register Address		Architectural MSR Name and bit fields (Former MSR Name)	MSR/Bit Description	Comment
Hex	Decimal			
38DH	909	IA32_FIXED_CTR_CTRL	Fixed-Function Performance Counter Control (R/W) Counter increments while the results of ANDing respective enable bit in IA32_PERF_GLOBAL_CTRL with the corresponding OS or USR bits in this MSR is true.	If CPUID.0AH: EAX[7:0] > 1
		0	EN0_OS: Enable Fixed Counter 0 to count while CPL = 0.	
		1	EN0_Usr: Enable Fixed Counter 0 to count while CPL > 0.	
		2	AnyThread: When set to 1, it enables counting the associated event conditions occurring across all logical processors sharing a processor core. When set to 0, the counter only increments the associated event conditions occurring in the logical processor which programmed the MSR.	If CPUID.0AH: EAX[7:0] > 2
		3	EN0_PMI: Enable PMI when fixed counter 0 overflows.	
		4	EN1_OS: Enable Fixed Counter 1to count while CPL = 0.	
		5	EN1_Usr: Enable Fixed Counter 1to count while CPL > 0.	
		6	AnyThread: When set to 1, it enables counting the associated event conditions occurring across all logical processors sharing a processor core. When set to 0, the counter only increments the associated event conditions occurring in the logical processor which programmed the MSR.	If CPUID.0AH: EAX[7:0] > 2
		7	EN1_PMI: Enable PMI when fixed counter 1 overflows.	
		8	EN2_OS: Enable Fixed Counter 2 to count while CPL = 0.	
		9	EN2_Usr: Enable Fixed Counter 2 to count while CPL > 0.	
		10	AnyThread: When set to 1, it enables counting the associated event conditions occurring across all logical processors sharing a processor core. When set to 0, the counter only increments the associated event conditions occurring in the logical processor which programmed the MSR.	If CPUID.0AH: EAX[7:0] > 2
		11	EN2_PMI: Enable PMI when fixed counter 2 overflows.	
		63:12	Reserved.	

Table 2-2. IA-32 Architectural MSRs (Contd.)

Register Address		Architectural MSR Name and bit fields (Former MSR Name)	MSR/Bit Description	Comment
Hex	Decimal			
38EH	910	IA32_PERF_GLOBAL_STATUS	Global Performance Counter Status (RO)	If CPUID.0AH: EAX[7:0] > 0
		0	Ovf_PMC0: Overflow status of IA32_PMC0.	If CPUID.0AH: EAX[15:8] > 0
		1	Ovf_PMC1: Overflow status of IA32_PMC1.	If CPUID.0AH: EAX[15:8] > 1
		2	Ovf_PMC2: Overflow status of IA32_PMC2.	If CPUID.0AH: EAX[15:8] > 2
		3	Ovf_PMC3: Overflow status of IA32_PMC3.	If CPUID.0AH: EAX[15:8] > 3
		31:4	Reserved.	
		32	Ovf_FixedCtr0: Overflow status of IA32_FIXED_CTR0.	If CPUID.0AH: EAX[7:0] > 1
		33	Ovf_FixedCtr1: Overflow status of IA32_FIXED_CTR1.	If CPUID.0AH: EAX[7:0] > 1
		34	Ovf_FixedCtr2: Overflow status of IA32_FIXED_CTR2.	If CPUID.0AH: EAX[7:0] > 1
		54:35	Reserved.	
		55	Trace_ToPA_PMI: A PMI occurred due to a ToPA entry memory buffer was completely filled.	If (CPUID.(EAX=07H, ECX=0):EBX[25] = 1) && IA32_RTIT_CTL.ToPA = 1
		57:56	Reserved.	
		58	LBR_Frz: LBRs are frozen due to ▪ IA32_DEBUGCTL.FREEZE_LBR_ON_PMI=1, ▪ The LBR stack overflowed	If CPUID.0AH: EAX[7:0] > 3
		59	CTR_Frz: Performance counters in the core PMU are frozen due to ▪ IA32_DEBUGCTL.FREEZE_PERFMON_ON_PMI=1, ▪ one or more core PMU counters overflowed.	If CPUID.0AH: EAX[7:0] > 3
		60	ASCI: Data in the performance counters in the core PMU may include contributions from the direct or indirect operation intel SGX to protect an enclave.	If CPUID.(EAX=07H, ECX=0):EBX[2] = 1
		61	Ovf_Uncore: Uncore counter overflow status.	If CPUID.0AH: EAX[7:0] > 2
		62	OvfBuf: DS SAVE area Buffer overflow status.	If CPUID.0AH: EAX[7:0] > 0
		63	CondChgd: status bits of this register has changed.	If CPUID.0AH: EAX[7:0] > 0

Table 2-2. IA-32 Architectural MSRs (Contd.)

Register Address		Architectural MSR Name and bit fields (Former MSR Name)	MSR/Bit Description	Comment
Hex	Decimal			
38FH	911	IA32_PERF_GLOBAL_CTRL	Global Performance Counter Control (R/W) Counter increments while the result of ANDing respective enable bit in this MSR with the corresponding OS or USR bits in the general-purpose or fixed counter control MSR is true.	If CPUID.0AH: EAX[7:0] > 0
		0	EN_PMC0	If CPUID.0AH: EAX[15:8] > 0
		1	EN_PMC1	If CPUID.0AH: EAX[15:8] > 1
		2	EN_PMC2	If CPUID.0AH: EAX[15:8] > 2
		n	EN_PMCn	If CPUID.0AH: EAX[15:8] > n
		31:n+1	Reserved.	
		32	EN_FIXED_CTR0	If CPUID.0AH: EDX[4:0] > 0
		33	EN_FIXED_CTR1	If CPUID.0AH: EDX[4:0] > 1
		34	EN_FIXED_CTR2	If CPUID.0AH: EDX[4:0] > 2
		63:35	Reserved.	
390H	912	IA32_PERF_GLOBAL_OVF_CTRL	Global Performance Counter Overflow Control (R/W)	If CPUID.0AH: EAX[7:0] > 0 && CPUID.0AH: EAX[7:0] <= 3
		0	Set 1 to Clear Ovf_PMC0 bit.	If CPUID.0AH: EAX[15:8] > 0
		1	Set 1 to Clear Ovf_PMC1 bit.	If CPUID.0AH: EAX[15:8] > 1
		2	Set 1 to Clear Ovf_PMC2 bit.	If CPUID.0AH: EAX[15:8] > 2
		n	Set 1 to Clear Ovf_PMCn bit.	If CPUID.0AH: EAX[15:8] > n
		31:n	Reserved.	
		32	Set 1 to Clear Ovf_FIXED_CTR0 bit.	If CPUID.0AH: EDX[4:0] > 0
		33	Set 1 to Clear Ovf_FIXED_CTR1 bit.	If CPUID.0AH: EDX[4:0] > 1
		34	Set 1 to Clear Ovf_FIXED_CTR2 bit.	If CPUID.0AH: EDX[4:0] > 2
		54:35	Reserved.	
		55	Set 1 to Clear Trace_ToPA_PMI bit.	If (CPUID.(EAX=07H, ECX=0):EBX[25] = 1) && IA32_RTIT_CTL.ToPA = 1
		60:56	Reserved.	
		61	Set 1 to Clear Ovf_Uncore bit.	06_2EH
		62	Set 1 to Clear OvfBuf: bit.	If CPUID.0AH: EAX[7:0] > 0
		63	Set to 1to clear CondChgd: bit.	If CPUID.0AH: EAX[7:0] > 0

Table 2-2. IA-32 Architectural MSRs (Contd.)

Register Address		Architectural MSR Name and bit fields (Former MSR Name)	MSR/Bit Description	Comment
Hex	Decimal			
390H	912	IA32_PERF_GLOBAL_STATUS_RESET	Global Performance Counter Overflow Reset Control (R/W)	If CPUID.0AH: EAX[7:0] > 3
		0	Set 1 to Clear Ovf_PMC0 bit.	If CPUID.0AH: EAX[15:8] > 0
		1	Set 1 to Clear Ovf_PMC1 bit.	If CPUID.0AH: EAX[15:8] > 1
		2	Set 1 to Clear Ovf_PMC2 bit.	If CPUID.0AH: EAX[15:8] > 2
		n	Set 1 to Clear Ovf_PMCn bit.	If CPUID.0AH: EAX[15:8] > n
		31:n	Reserved.	
		32	Set 1 to Clear Ovf_FIXED_CTR0 bit.	If CPUID.0AH: EDX[4:0] > 0
		33	Set 1 to Clear Ovf_FIXED_CTR1 bit.	If CPUID.0AH: EDX[4:0] > 1
		34	Set 1 to Clear Ovf_FIXED_CTR2 bit.	If CPUID.0AH: EDX[4:0] > 2
		54:35	Reserved.	
		55	Set 1 to Clear Trace_ToPA_PMI bit.	If (CPUID.(EAX=07H, ECX=0):EBX[25] = 1) && IA32_RTIT_CTL.ToPA[8] = 1
		57:56	Reserved.	
		58	Set 1 to Clear LBR_Frz bit.	If CPUID.0AH: EAX[7:0] > 3
		59	Set 1 to Clear CTR_Frz bit.	If CPUID.0AH: EAX[7:0] > 3
		58	Set 1 to Clear ASCI bit.	If CPUID.0AH: EAX[7:0] > 3
		61	Set 1 to Clear Ovf_Uncore bit.	06_2EH
		62	Set 1 to Clear OvfBuf: bit.	If CPUID.0AH: EAX[7:0] > 0
		63	Set to 1to clear CondChgd: bit.	If CPUID.0AH: EAX[7:0] > 0
391H	913	IA32_PERF_GLOBAL_STATUS_SET	Global Performance Counter Overflow Set Control (R/W)	If CPUID.0AH: EAX[7:0] > 3
		0	Set 1 to cause Ovf_PMC0 = 1.	If CPUID.0AH: EAX[7:0] > 3
		1	Set 1 to cause Ovf_PMC1 = 1	If CPUID.0AH: EAX[15:8] > 1
		2	Set 1 to cause Ovf_PMC2 = 1	If CPUID.0AH: EAX[15:8] > 2
		n	Set 1 to cause Ovf_PMCn = 1	If CPUID.0AH: EAX[15:8] > n
		31:n	Reserved.	
		32	Set 1 to cause Ovf_FIXED_CTR0 = 1.	If CPUID.0AH: EAX[7:0] > 3
		33	Set 1 to cause Ovf_FIXED_CTR1 = 1.	If CPUID.0AH: EAX[7:0] > 3
		34	Set 1 to cause Ovf_FIXED_CTR2 = 1.	If CPUID.0AH: EAX[7:0] > 3
		54:35	Reserved.	
		55	Set 1 to cause Trace_ToPA_PMI = 1.	If CPUID.0AH: EAX[7:0] > 3

Table 2-2. IA-32 Architectural MSRs (Contd.)

Register Address		Architectural MSR Name and bit fields (Former MSR Name)	MSR/Bit Description	Comment
Hex	Decimal			
		57:56	Reserved.	
		58	Set 1 to cause LBR_Frz = 1.	If CPUID.0AH: EAX[7:0] > 3
		59	Set 1 to cause CTR_Frz = 1.	If CPUID.0AH: EAX[7:0] > 3
		58	Set 1 to cause ASCI = 1.	If CPUID.0AH: EAX[7:0] > 3
		61	Set 1 to cause Ovf_Uncore = 1.	If CPUID.0AH: EAX[7:0] > 3
		62	Set 1 to cause OvfBuf = 1.	If CPUID.0AH: EAX[7:0] > 3
		63	Reserved	
392H	914	IA32_PERF_GLOBAL_INUSE	Indicator of core perfmon interface is in use (RO)	If CPUID.0AH: EAX[7:0] > 3
		0	IA32_PERFEVTSEL0 in use	
		1	IA32_PERFEVTSEL1 in use	If CPUID.0AH: EAX[15:8] > 1
		2	IA32_PERFEVTSEL2 in use	If CPUID.0AH: EAX[15:8] > 2
		n	IA32_PERFEVTSELn in use	If CPUID.0AH: EAX[15:8] > n
		31:n	Reserved.	
		32	IA32_FIXED_CTR0 in use	
		33	IA32_FIXED_CTR1 in use	
		34	IA32_FIXED_CTR2 in use	
		62:35	Reserved or Model specific.	
		63	PMI in use.	
3F1H	1009	IA32_PEBS_ENABLE	PEBS Control (R/W)	
		0	Enable PEBS on IA32_PMC0.	06_0FH
		3:1	Reserved or Model specific.	
		31:4	Reserved.	
		35:32	Reserved or Model specific.	
		63:36	Reserved.	
400H	1024	IA32_MC0_CTL	MC0_CTL	If IA32_MCG_CAP.CNT >0
401H	1025	IA32_MC0_STATUS	MC0_STATUS	If IA32_MCG_CAP.CNT >0
402H	1026	IA32_MC0_ADDR[1]	MC0_ADDR	If IA32_MCG_CAP.CNT >0
403H	1027	IA32_MC0_MISC	MC0_MISC	If IA32_MCG_CAP.CNT >0
404H	1028	IA32_MC1_CTL	MC1_CTL	If IA32_MCG_CAP.CNT >1
405H	1029	IA32_MC1_STATUS	MC1_STATUS	If IA32_MCG_CAP.CNT >1
406H	1030	IA32_MC1_ADDR[2]	MC1_ADDR	If IA32_MCG_CAP.CNT >1
407H	1031	IA32_MC1_MISC	MC1_MISC	If IA32_MCG_CAP.CNT >1
408H	1032	IA32_MC2_CTL	MC2_CTL	If IA32_MCG_CAP.CNT >2
409H	1033	IA32_MC2_STATUS	MC2_STATUS	If IA32_MCG_CAP.CNT >2

Table 2-2. IA-32 Architectural MSRs (Contd.)

Register Address		Architectural MSR Name and bit fields (Former MSR Name)	MSR/Bit Description	Comment
Hex	Decimal			
40AH	1034	IA32_MC2_ADDR[1]	MC2_ADDR	If IA32_MCG_CAP.CNT >2
40BH	1035	IA32_MC2_MISC	MC2_MISC	If IA32_MCG_CAP.CNT >2
40CH	1036	IA32_MC3_CTL	MC3_CTL	If IA32_MCG_CAP.CNT >3
40DH	1037	IA32_MC3_STATUS	MC3_STATUS	If IA32_MCG_CAP.CNT >3
40EH	1038	IA32_MC3_ADDR[1]	MC3_ADDR	If IA32_MCG_CAP.CNT >3
40FH	1039	IA32_MC3_MISC	MC3_MISC	If IA32_MCG_CAP.CNT >3
410H	1040	IA32_MC4_CTL	MC4_CTL	If IA32_MCG_CAP.CNT >4
411H	1041	IA32_MC4_STATUS	MC4_STATUS	If IA32_MCG_CAP.CNT >4
412H	1042	IA32_MC4_ADDR[1]	MC4_ADDR	If IA32_MCG_CAP.CNT >4
413H	1043	IA32_MC4_MISC	MC4_MISC	If IA32_MCG_CAP.CNT >4
414H	1044	IA32_MC5_CTL	MC5_CTL	If IA32_MCG_CAP.CNT >5
415H	1045	IA32_MC5_STATUS	MC5_STATUS	If IA32_MCG_CAP.CNT >5
416H	1046	IA32_MC5_ADDR[1]	MC5_ADDR	If IA32_MCG_CAP.CNT >5
417H	1047	IA32_MC5_MISC	MC5_MISC	If IA32_MCG_CAP.CNT >5
418H	1048	IA32_MC6_CTL	MC6_CTL	If IA32_MCG_CAP.CNT >6
419H	1049	IA32_MC6_STATUS	MC6_STATUS	If IA32_MCG_CAP.CNT >6
41AH	1050	IA32_MC6_ADDR[1]	MC6_ADDR	If IA32_MCG_CAP.CNT >6
41BH	1051	IA32_MC6_MISC	MC6_MISC	If IA32_MCG_CAP.CNT >6
41CH	1052	IA32_MC7_CTL	MC7_CTL	If IA32_MCG_CAP.CNT >7
41DH	1053	IA32_MC7_STATUS	MC7_STATUS	If IA32_MCG_CAP.CNT >7
41EH	1054	IA32_MC7_ADDR[1]	MC7_ADDR	If IA32_MCG_CAP.CNT >7
41FH	1055	IA32_MC7_MISC	MC7_MISC	If IA32_MCG_CAP.CNT >7
420H	1056	IA32_MC8_CTL	MC8_CTL	If IA32_MCG_CAP.CNT >8
421H	1057	IA32_MC8_STATUS	MC8_STATUS	If IA32_MCG_CAP.CNT >8
422H	1058	IA32_MC8_ADDR[1]	MC8_ADDR	If IA32_MCG_CAP.CNT >8
423H	1059	IA32_MC8_MISC	MC8_MISC	If IA32_MCG_CAP.CNT >8
424H	1060	IA32_MC9_CTL	MC9_CTL	If IA32_MCG_CAP.CNT >9
425H	1061	IA32_MC9_STATUS	MC9_STATUS	If IA32_MCG_CAP.CNT >9
426H	1062	IA32_MC9_ADDR[1]	MC9_ADDR	If IA32_MCG_CAP.CNT >9
427H	1063	IA32_MC9_MISC	MC9_MISC	If IA32_MCG_CAP.CNT >9
428H	1064	IA32_MC10_CTL	MC10_CTL	If IA32_MCG_CAP.CNT >10
429H	1065	IA32_MC10_STATUS	MC10_STATUS	If IA32_MCG_CAP.CNT >10
42AH	1066	IA32_MC10_ADDR[1]	MC10_ADDR	If IA32_MCG_CAP.CNT >10
42BH	1067	IA32_MC10_MISC	MC10_MISC	If IA32_MCG_CAP.CNT >10
42CH	1068	IA32_MC11_CTL	MC11_CTL	If IA32_MCG_CAP.CNT >11
42DH	1069	IA32_MC11_STATUS	MC11_STATUS	If IA32_MCG_CAP.CNT >11
42EH	1070	IA32_MC11_ADDR[1]	MC11_ADDR	If IA32_MCG_CAP.CNT >11

Table 2-2. IA-32 Architectural MSRs (Contd.)

Register Address		Architectural MSR Name and bit fields (Former MSR Name)	MSR/Bit Description	Comment
Hex	Decimal			
42FH	1071	IA32_MC11_MISC	MC11_MISC	If IA32_MCG_CAP.CNT >11
430H	1072	IA32_MC12_CTL	MC12_CTL	If IA32_MCG_CAP.CNT >12
431H	1073	IA32_MC12_STATUS	MC12_STATUS	If IA32_MCG_CAP.CNT >12
432H	1074	IA32_MC12_ADDR[1]	MC12_ADDR	If IA32_MCG_CAP.CNT >12
433H	1075	IA32_MC12_MISC	MC12_MISC	If IA32_MCG_CAP.CNT >12
434H	1076	IA32_MC13_CTL	MC13_CTL	If IA32_MCG_CAP.CNT >13
435H	1077	IA32_MC13_STATUS	MC13_STATUS	If IA32_MCG_CAP.CNT >13
436H	1078	IA32_MC13_ADDR[1]	MC13_ADDR	If IA32_MCG_CAP.CNT >13
437H	1079	IA32_MC13_MISC	MC13_MISC	If IA32_MCG_CAP.CNT >13
438H	1080	IA32_MC14_CTL	MC14_CTL	If IA32_MCG_CAP.CNT >14
439H	1081	IA32_MC14_STATUS	MC14_STATUS	If IA32_MCG_CAP.CNT >14
43AH	1082	IA32_MC14_ADDR[1]	MC14_ADDR	If IA32_MCG_CAP.CNT >14
43BH	1083	IA32_MC14_MISC	MC14_MISC	If IA32_MCG_CAP.CNT >14
43CH	1084	IA32_MC15_CTL	MC15_CTL	If IA32_MCG_CAP.CNT >15
43DH	1085	IA32_MC15_STATUS	MC15_STATUS	If IA32_MCG_CAP.CNT >15
43EH	1086	IA32_MC15_ADDR[1]	MC15_ADDR	If IA32_MCG_CAP.CNT >15
43FH	1087	IA32_MC15_MISC	MC15_MISC	If IA32_MCG_CAP.CNT >15
440H	1088	IA32_MC16_CTL	MC16_CTL	If IA32_MCG_CAP.CNT >16
441H	1089	IA32_MC16_STATUS	MC16_STATUS	If IA32_MCG_CAP.CNT >16
442H	1090	IA32_MC16_ADDR[1]	MC16_ADDR	If IA32_MCG_CAP.CNT >16
443H	1091	IA32_MC16_MISC	MC16_MISC	If IA32_MCG_CAP.CNT >16
444H	1092	IA32_MC17_CTL	MC17_CTL	If IA32_MCG_CAP.CNT >17
445H	1093	IA32_MC17_STATUS	MC17_STATUS	If IA32_MCG_CAP.CNT >17
446H	1094	IA32_MC17_ADDR[1]	MC17_ADDR	If IA32_MCG_CAP.CNT >17
447H	1095	IA32_MC17_MISC	MC17_MISC	If IA32_MCG_CAP.CNT >17
448H	1096	IA32_MC18_CTL	MC18_CTL	If IA32_MCG_CAP.CNT >18
449H	1097	IA32_MC18_STATUS	MC18_STATUS	If IA32_MCG_CAP.CNT >18
44AH	1098	IA32_MC18_ADDR[1]	MC18_ADDR	If IA32_MCG_CAP.CNT >18
44BH	1099	IA32_MC18_MISC	MC18_MISC	If IA32_MCG_CAP.CNT >18
44CH	1100	IA32_MC19_CTL	MC19_CTL	If IA32_MCG_CAP.CNT >19
44DH	1101	IA32_MC19_STATUS	MC19_STATUS	If IA32_MCG_CAP.CNT >19
44EH	1102	IA32_MC19_ADDR[1]	MC19_ADDR	If IA32_MCG_CAP.CNT >19
44FH	1103	IA32_MC19_MISC	MC19_MISC	If IA32_MCG_CAP.CNT >19
450H	1104	IA32_MC20_CTL	MC20_CTL	If IA32_MCG_CAP.CNT >20
451H	1105	IA32_MC20_STATUS	MC20_STATUS	If IA32_MCG_CAP.CNT >20
452H	1106	IA32_MC20_ADDR[1]	MC20_ADDR	If IA32_MCG_CAP.CNT >20
453H	1107	IA32_MC20_MISC	MC20_MISC	If IA32_MCG_CAP.CNT >20

Table 2-2. IA-32 Architectural MSRs (Contd.)

Register Address		Architectural MSR Name and bit fields (Former MSR Name)	MSR/Bit Description	Comment
Hex	Decimal			
454H	1108	IA32_MC21_CTL	MC21_CTL	If IA32_MCG_CAP.CNT >21
455H	1109	IA32_MC21_STATUS	MC21_STATUS	If IA32_MCG_CAP.CNT >21
456H	1110	IA32_MC21_ADDR[1]	MC21_ADDR	If IA32_MCG_CAP.CNT >21
457H	1111	IA32_MC21_MISC	MC21_MISC	If IA32_MCG_CAP.CNT >21
458H		IA32_MC22_CTL	MC22_CTL	If IA32_MCG_CAP.CNT >22
459H		IA32_MC22_STATUS	MC22_STATUS	If IA32_MCG_CAP.CNT >22
45AH		IA32_MC22_ADDR[1]	MC22_ADDR	If IA32_MCG_CAP.CNT >22
45BH		IA32_MC22_MISC	MC22_MISC	If IA32_MCG_CAP.CNT >22
45CH		IA32_MC23_CTL	MC23_CTL	If IA32_MCG_CAP.CNT >23
45DH		IA32_MC23_STATUS	MC23_STATUS	If IA32_MCG_CAP.CNT >23
45EH		IA32_MC23_ADDR[1]	MC23_ADDR	If IA32_MCG_CAP.CNT >23
45FH		IA32_MC23_MISC	MC23_MISC	If IA32_MCG_CAP.CNT >23
460H		IA32_MC24_CTL	MC24_CTL	If IA32_MCG_CAP.CNT >24
461H		IA32_MC24_STATUS	MC24_STATUS	If IA32_MCG_CAP.CNT >24
462H		IA32_MC24_ADDR[1]	MC24_ADDR	If IA32_MCG_CAP.CNT >24
463H		IA32_MC24_MISC	MC24_MISC	If IA32_MCG_CAP.CNT >24
464H		IA32_MC25_CTL	MC25_CTL	If IA32_MCG_CAP.CNT >25
465H		IA32_MC25_STATUS	MC25_STATUS	If IA32_MCG_CAP.CNT >25
466H		IA32_MC25_ADDR[1]	MC25_ADDR	If IA32_MCG_CAP.CNT >25
467H		IA32_MC25_MISC	MC25_MISC	If IA32_MCG_CAP.CNT >25
468H		IA32_MC26_CTL	MC26_CTL	If IA32_MCG_CAP.CNT >26
469H		IA32_MC26_STATUS	MC26_STATUS	If IA32_MCG_CAP.CNT >26
46AH		IA32_MC26_ADDR[1]	MC26_ADDR	If IA32_MCG_CAP.CNT >26
46BH		IA32_MC26_MISC	MC26_MISC	If IA32_MCG_CAP.CNT >26
46CH		IA32_MC27_CTL	MC27_CTL	If IA32_MCG_CAP.CNT >27
46DH		IA32_MC27_STATUS	MC27_STATUS	If IA32_MCG_CAP.CNT >27
46EH		IA32_MC27_ADDR[1]	MC27_ADDR	If IA32_MCG_CAP.CNT >27
46FH		IA32_MC27_MISC	MC27_MISC	If IA32_MCG_CAP.CNT >27
470H		IA32_MC28_CTL	MC28_CTL	If IA32_MCG_CAP.CNT >28
471H		IA32_MC28_STATUS	MC28_STATUS	If IA32_MCG_CAP.CNT >28
472H		IA32_MC28_ADDR[1]	MC28_ADDR	If IA32_MCG_CAP.CNT >28
473H		IA32_MC28_MISC	MC28_MISC	If IA32_MCG_CAP.CNT >28
480H	1152	IA32_VMX_BASIC	**Reporting Register of Basic VMX Capabilities (R/O)** See Appendix A.1, "Basic VMX Information."	If CPUID.01H:ECX.[5] = 1

Table 2-2. IA-32 Architectural MSRs (Contd.)

Register Address		Architectural MSR Name and bit fields (Former MSR Name)	MSR/Bit Description	Comment
Hex	Decimal			
481H	1153	IA32_VMX_PINBASED_CTLS	**Capability Reporting Register of Pin-based VM-execution Controls (R/O)** See Appendix A.3.1, "Pin-Based VM-Execution Controls."	If CPUID.01H:ECX.[5] = 1
482H	1154	IA32_VMX_PROCBASED_CTLS	**Capability Reporting Register of Primary Processor-based VM-execution Controls (R/O)** See Appendix A.3.2, "Primary Processor-Based VM-Execution Controls."	If CPUID.01H:ECX.[5] = 1
483H	1155	IA32_VMX_EXIT_CTLS	**Capability Reporting Register of VM-exit Controls (R/O)** See Appendix A.4, "VM-Exit Controls."	If CPUID.01H:ECX.[5] = 1
484H	1156	IA32_VMX_ENTRY_CTLS	**Capability Reporting Register of VM-entry Controls (R/O)** See Appendix A.5, "VM-Entry Controls."	If CPUID.01H:ECX.[5] = 1
485H	1157	IA32_VMX_MISC	**Reporting Register of Miscellaneous VMX Capabilities (R/O)** See Appendix A.6, "Miscellaneous Data."	If CPUID.01H:ECX.[5] = 1
486H	1158	IA32_VMX_CR0_FIXED0	**Capability Reporting Register of CR0 Bits Fixed to 0 (R/O)** See Appendix A.7, "VMX-Fixed Bits in CR0."	If CPUID.01H:ECX.[5] = 1
487H	1159	IA32_VMX_CR0_FIXED1	**Capability Reporting Register of CR0 Bits Fixed to 1 (R/O)** See Appendix A.7, "VMX-Fixed Bits in CR0."	If CPUID.01H:ECX.[5] = 1
488H	1160	IA32_VMX_CR4_FIXED0	**Capability Reporting Register of CR4 Bits Fixed to 0 (R/O)** See Appendix A.8, "VMX-Fixed Bits in CR4."	If CPUID.01H:ECX.[5] = 1
489H	1161	IA32_VMX_CR4_FIXED1	**Capability Reporting Register of CR4 Bits Fixed to 1 (R/O)** See Appendix A.8, "VMX-Fixed Bits in CR4."	If CPUID.01H:ECX.[5] = 1
48AH	1162	IA32_VMX_VMCS_ENUM	**Capability Reporting Register of VMCS Field Enumeration (R/O)** See Appendix A.9, "VMCS Enumeration."	If CPUID.01H:ECX.[5] = 1
48BH	1163	IA32_VMX_PROCBASED_CTLS2	**Capability Reporting Register of Secondary Processor-based VM-execution Controls (R/O)** See Appendix A.3.3, "Secondary Processor-Based VM-Execution Controls."	If (CPUID.01H:ECX.[5] && IA32_VMX_PROCBASED_CTLS[63])
48CH	1164	IA32_VMX_EPT_VPID_CAP	**Capability Reporting Register of EPT and VPID (R/O)** See Appendix A.10, "VPID and EPT Capabilities."	If (CPUID.01H:ECX.[5] && IA32_VMX_PROCBASED_CTLS[63] && (IA32_VMX_PROCBASED_CTLS2[33] || IA32_VMX_PROCBASED_CTLS2[37]))

Table 2-2. IA-32 Architectural MSRs (Contd.)

Register Address		Architectural MSR Name and bit fields (Former MSR Name)	MSR/Bit Description	Comment
Hex	Decimal			
48DH	1165	IA32_VMX_TRUE_PINBASED_CTLS	**Capability Reporting Register of Pin-based VM-execution Flex Controls (R/O)** See Appendix A.3.1, "Pin-Based VM-Execution Controls."	If (CPUID.01H:ECX.[5] = 1 && IA32_VMX_BASIC[55])
48EH	1166	IA32_VMX_TRUE_PROCBASED_CTLS	**Capability Reporting Register of Primary Processor-based VM-execution Flex Controls (R/O)** See Appendix A.3.2, "Primary Processor-Based VM-Execution Controls."	If(CPUID.01H:ECX.[5] = 1 && IA32_VMX_BASIC[55])
48FH	1167	IA32_VMX_TRUE_EXIT_CTLS	**Capability Reporting Register of VM-exit Flex Controls (R/O)** See Appendix A.4, "VM-Exit Controls."	If(CPUID.01H:ECX.[5] = 1 && IA32_VMX_BASIC[55])
490H	1168	IA32_VMX_TRUE_ENTRY_CTLS	**Capability Reporting Register of VM-entry Flex Controls (R/O)** See Appendix A.5, "VM-Entry Controls."	If(CPUID.01H:ECX.[5] = 1 && IA32_VMX_BASIC[55])
491H	1169	IA32_VMX_VMFUNC	**Capability Reporting Register of VM-function Controls (R/O)**	If(CPUID.01H:ECX.[5] = 1 && IA32_VMX_BASIC[55])
4C1H	1217	IA32_A_PMC0	Full Width Writable IA32_PMC0 Alias (R/W)	(If CPUID.0AH: EAX[15:8] > 0) && IA32_PERF_CAPABILITIES[13] = 1
4C2H	1218	IA32_A_PMC1	Full Width Writable IA32_PMC1 Alias (R/W)	(If CPUID.0AH: EAX[15:8] > 1) && IA32_PERF_CAPABILITIES[13] = 1
4C3H	1219	IA32_A_PMC2	Full Width Writable IA32_PMC2 Alias (R/W)	(If CPUID.0AH: EAX[15:8] > 2) && IA32_PERF_CAPABILITIES[13] = 1
4C4H	1220	IA32_A_PMC3	Full Width Writable IA32_PMC3 Alias (R/W)	(If CPUID.0AH: EAX[15:8] > 3) && IA32_PERF_CAPABILITIES[13] = 1
4C5H	1221	IA32_A_PMC4	Full Width Writable IA32_PMC4 Alias (R/W)	(If CPUID.0AH: EAX[15:8] > 4) && IA32_PERF_CAPABILITIES[13] = 1
4C6H	1222	IA32_A_PMC5	Full Width Writable IA32_PMC5 Alias (R/W)	(If CPUID.0AH: EAX[15:8] > 5) && IA32_PERF_CAPABILITIES[13] = 1

Table 2-2. IA-32 Architectural MSRs (Contd.)

Register Address		Architectural MSR Name and bit fields (Former MSR Name)	MSR/Bit Description	Comment
Hex	Decimal			
4C7H	1223	IA32_A_PMC6	Full Width Writable IA32_PMC6 Alias (R/W)	(If CPUID.0AH: EAX[15:8] > 6) && IA32_PERF_CAPABILITIES[13] = 1
4C8H	1224	IA32_A_PMC7	Full Width Writable IA32_PMC7 Alias (R/W)	(If CPUID.0AH: EAX[15:8] > 7) && IA32_PERF_CAPABILITIES[13] = 1
4D0H	1232	IA32_MCG_EXT_CTL	Allows software to signal some MCEs to only single logical processor in the system. (R/W) See Section 15.3.1.4, "IA32_MCG_EXT_CTL MSR".	If IA32_MCG_CAP.LMCE_P =1
		0	LMCE_EN.	
		63:1	Reserved.	
500H	1280	IA32_SGX_SVN_STATUS	Status and SVN Threshold of SGX Support for ACM (RO).	If CPUID.(EAX=07H, ECX=0H): EBX[2] = 1
		0	Lock.	See Section 41.11.3, "Interactions with Authenticated Code Modules (ACMs)".
		15:1	Reserved.	
		23:16	SGX_SVN_SINIT.	See Section 41.11.3, "Interactions with Authenticated Code Modules (ACMs)".
		63:24	Reserved.	
560H	1376	IA32_RTIT_OUTPUT_BASE	**Trace Output Base Register (R/W)**	If ((CPUID.(EAX=07H, ECX=0):EBX[25] = 1) && ((CPUID.(EAX=14H,ECX=0): ECX[0] = 1) || (CPUID.(EAX=14H,ECX=0): ECX[2] = 1)))
		6:0	Reserved	
		MAXPHYADDR[3]-1:7	Base physical address	
		63:MAXPHYADDR	**Reserved.**	
561H	1377	IA32_RTIT_OUTPUT_MASK_PTRS	**Trace Output Mask Pointers Register (R/W)**	If ((CPUID.(EAX=07H, ECX=0):EBX[25] = 1) && ((CPUID.(EAX=14H,ECX=0): ECX[0] = 1) || (CPUID.(EAX=14H,ECX=0): ECX[2] = 1)))
		6:0	Reserved	
		31:7	MaskOrTableOffset	
		63:32	**Output Offset.**	

Table 2-2. IA-32 Architectural MSRs (Contd.)

Register Address		Architectural MSR Name and bit fields (Former MSR Name)	MSR/Bit Description	Comment
Hex	Decimal			
570H	1392	IA32_RTIT_CTL	**Trace Control Register (R/W)**	If (CPUID.(EAX=07H, ECX=0):EBX[25] = 1)
		0	**TraceEn**	
		1	**CYCEn**	If (CPUID.(EAX=07H, ECX=0):EBX[1] = 1)
		2	**OS**	
		3	**User**	
		4	**PwrEvtEn**	
		5	**FUPonPTW**	
		6	**FabricEn**	If (CPUID.(EAX=07H, ECX=0):ECX[3] = 1)
		7	**CR3 filter**	
		8	**ToPA**	
		9	**MTCEn**	If (CPUID.(EAX=07H, ECX=0):EBX[3] = 1)
		10	**TSCEn**	
		11	**DisRETC**	
		12	**PTWEn**	
		13	**BranchEn**	
		17:14	**MTCFreq**	If (CPUID.(EAX=07H, ECX=0):EBX[3] = 1)
		18	Reserved, MBZ	
		22:19	**CYCThresh**	If (CPUID.(EAX=07H, ECX=0):EBX[1] = 1)
		23	Reserved, MBZ	
		27:24	**PSBFreq**	If (CPUID.(EAX=07H, ECX=0):EBX[1] = 1)
		31:28	Reserved, MBZ	
		35:32	**ADDR0_CFG**	If (CPUID.(EAX=07H, ECX=1):EAX[2:0] > 0)
		39:36	**ADDR1_CFG**	If (CPUID.(EAX=07H, ECX=1):EAX[2:0] > 1)
		43:40	**ADDR2_CFG**	If (CPUID.(EAX=07H, ECX=1):EAX[2:0] > 2)
		47:44	**ADDR3_CFG**	If (CPUID.(EAX=07H, ECX=1):EAX[2:0] > 3)
		63:48	Reserved, MBZ.	
571H	1393	IA32_RTIT_STATUS	**Tracing Status Register (R/W)**	If (CPUID.(EAX=07H, ECX=0):EBX[25] = 1)
		0	**FilterEn** (writes ignored)	If (CPUID.(EAX=07H, ECX=0):EBX[2] = 1)

Table 2-2. IA-32 Architectural MSRs (Contd.)

Register Address		Architectural MSR Name and bit fields (Former MSR Name)	MSR/Bit Description	Comment
Hex	Decimal			
		1	**ContexEn** (writes ignored)	
		2	**TriggerEn** (writes ignored)	
		3	Reserved	
		4	**Error**	
		5	**Stopped**	
		31:6	Reserved, MBZ	
		48:32	**PacketByteCnt**	If (CPUID.(EAX=07H, ECX=0):EBX[1] > 3)
		63:49	**Reserved**	
572H	1394	IA32_RTIT_CR3_MATCH	**Trace Filter CR3 Match Register (R/W)**	If (CPUID.(EAX=07H, ECX=0):EBX[25] = 1)
		4:0	Reserved	
		63:5	CR3[63:5] value to match	
580H	1408	IA32_RTIT_ADDR0_A	**Region 0 Start Address (R/W)**	If (CPUID.(EAX=07H, ECX=1):EAX[2:0] > 0)
		47:0	Virtual Address	
		63:48	SignExt_VA	
581H	1409	IA32_RTIT_ADDR0_B	**Region 0 End Address (R/W)**	If (CPUID.(EAX=07H, ECX=1):EAX[2:0] > 0)
		47:0	Virtual Address	
		63:48	SignExt_VA	
582H	1410	IA32_RTIT_ADDR1_A	**Region 1 Start Address (R/W)**	If (CPUID.(EAX=07H, ECX=1):EAX[2:0] > 1)
		47:0	Virtual Address	
		63:48	SignExt_VA	
583H	1411	IA32_RTIT_ADDR1_B	**Region 1 End Address (R/W)**	If (CPUID.(EAX=07H, ECX=1):EAX[2:0] > 1)
		47:0	Virtual Address	
		63:48	SignExt_VA	
584H	1412	IA32_RTIT_ADDR2_A	**Region 2 Start Address (R/W)**	If (CPUID.(EAX=07H, ECX=1):EAX[2:0] > 2)
		47:0	Virtual Address	
		63:48	SignExt_VA	
585H	1413	IA32_RTIT_ADDR2_B	**Region 2 End Address (R/W)**	If (CPUID.(EAX=07H, ECX=1):EAX[2:0] > 2)
		47:0	Virtual Address	
		63:48	SignExt_VA	
586H	1414	IA32_RTIT_ADDR3_A	**Region 3 Start Address (R/W)**	If (CPUID.(EAX=07H, ECX=1):EAX[2:0] > 3)
		47:0	Virtual Address	

Table 2-2. IA-32 Architectural MSRs (Contd.)

Register Address		Architectural MSR Name and bit fields (Former MSR Name)	MSR/Bit Description	Comment
Hex	Decimal			
		63:48	SignExt_VA	
587H	1415	IA32_RTIT_ADDR3_B	**Region 3 End Address (R/W)**	If (CPUID.(EAX=07H, ECX=1):EAX[2:0] > 3)
		47:0	Virtual Address	
		63:48	SignExt_VA	
600H	1536	IA32_DS_AREA	**DS Save Area (R/W)** Points to the linear address of the first byte of the DS buffer management area, which is used to manage the BTS and PEBS buffers. See Section 18.6.3.4, "Debug Store (DS) Mechanism."	If(CPUID.01H:EDX.DS[21] = 1
		63:0	The linear address of the first byte of the DS buffer management area, if IA-32e mode is active.	
		31:0	The linear address of the first byte of the DS buffer management area, if not in IA-32e mode.	
		63:32	Reserved if not in IA-32e mode.	
6E0H	1760	IA32_TSC_DEADLINE	**TSC Target of Local APIC's TSC Deadline Mode (R/W)**	If CPUID.01H:ECX.[24] = 1
770H	1904	IA32_PM_ENABLE	**Enable/disable HWP (R/W)**	If CPUID.06H:EAX.[7] = 1
		0	**HWP_ENABLE (R/W1-Once)** See Section 14.4.2, "Enabling HWP"	If CPUID.06H:EAX.[7] = 1
		63:1	Reserved.	
771H	1905	IA32_HWP_CAPABILITIES	**HWP Performance Range Enumeration (RO)**	If CPUID.06H:EAX.[7] = 1
		7:0	**Highest_Performance** See Section 14.4.3, "HWP Performance Range and Dynamic Capabilities"	If CPUID.06H:EAX.[7] = 1
		15:8	**Guaranteed_Performance** See Section 14.4.3, "HWP Performance Range and Dynamic Capabilities"	If CPUID.06H:EAX.[7] = 1
		23:16	**Most_Efficient_Performance** See Section 14.4.3, "HWP Performance Range and Dynamic Capabilities"	If CPUID.06H:EAX.[7] = 1
		31:24	**Lowest_Performance** See Section 14.4.3, "HWP Performance Range and Dynamic Capabilities"	If CPUID.06H:EAX.[7] = 1
		63:32	Reserved.	
772H	1906	IA32_HWP_REQUEST_PKG	**Power Management Control Hints for All Logical Processors in a Package (R/W)**	If CPUID.06H:EAX.[11] = 1

Table 2-2. IA-32 Architectural MSRs (Contd.)

Register Address		Architectural MSR Name and bit fields (Former MSR Name)	MSR/Bit Description	Comment
Hex	Decimal			
		7:0	**Minimum_Performance** See Section 14.4.4, "Managing HWP"	If CPUID.06H:EAX.[11] = 1
		15:8	**Maximum_Performance** See Section 14.4.4, "Managing HWP"	If CPUID.06H:EAX.[11] = 1
		23:16	**Desired_Performance** See Section 14.4.4, "Managing HWP"	If CPUID.06H:EAX.[11] = 1
		31:24	**Energy_Performance_Preference** See Section 14.4.4, "Managing HWP"	If CPUID.06H:EAX.[11] = 1 && CPUID.06H:EAX.[10] = 1
		41:32	**Activity_Window** See Section 14.4.4, "Managing HWP"	If CPUID.06H:EAX.[11] = 1 && CPUID.06H:EAX.[9] = 1
		63:42	Reserved.	
773H	1907	IA32_HWP_INTERRUPT	**Control HWP Native Interrupts (R/W)**	If CPUID.06H:EAX.[8] = 1
		0	**EN_Guaranteed_Performance_Change** See Section 14.4.6, "HWP Notifications"	If CPUID.06H:EAX.[8] = 1
		1	**EN_Excursion_Minimum** See Section 14.4.6, "HWP Notifications"	If CPUID.06H:EAX.[8] = 1
		63:2	Reserved.	
774H	1908	IA32_HWP_REQUEST	**Power Management Control Hints to a Logical Processor (R/W)**	If CPUID.06H:EAX.[7] = 1
		7:0	**Minimum_Performance** See Section 14.4.4, "Managing HWP"	If CPUID.06H:EAX.[7] = 1
		15:8	**Maximum_Performance** See Section 14.4.4, "Managing HWP"	If CPUID.06H:EAX.[7] = 1
		23:16	**Desired_Performance** See Section 14.4.4, "Managing HWP"	If CPUID.06H:EAX.[7] = 1
		31:24	**Energy_Performance_Preference** See Section 14.4.4, "Managing HWP"	If CPUID.06H:EAX.[7] = 1 && CPUID.06H:EAX.[10] = 1
		41:32	**Activity_Window** See Section 14.4.4, "Managing HWP"	If CPUID.06H:EAX.[7] = 1 && CPUID.06H:EAX.[9] = 1
		42	**Package_Control** See Section 14.4.4, "Managing HWP"	If CPUID.06H:EAX.[7] = 1 && CPUID.06H:EAX.[11] = 1
		63:43	Reserved.	
777H	1911	IA32_HWP_STATUS	**Log bits indicating changes to Guaranteed & excursions to Minimum (R/W)**	If CPUID.06H:EAX.[7] = 1

Table 2-2. IA-32 Architectural MSRs (Contd.)

Register Address		Architectural MSR Name and bit fields (Former MSR Name)	MSR/Bit Description	Comment
Hex	Decimal			
		0	**Guaranteed_Performance_Change (R/WC0)** See Section 14.4.5, "HWP Feedback"	If CPUID.06H:EAX.[7] = 1
		1	Reserved.	
		2	**Excursion_To_Minimum (R/WC0)** See Section 14.4.5, "HWP Feedback"	If CPUID.06H:EAX.[7] = 1
		63:3	Reserved.	
802H	2050	IA32_X2APIC_APICID	**x2APIC ID Register (R/O)** See x2APIC Specification	If CPUID.01H:ECX[21] = 1 && IA32_APIC_BASE.[10] = 1
803H	2051	IA32_X2APIC_VERSION	**x2APIC Version Register (R/O)**	If CPUID.01H:ECX.[21] = 1 && IA32_APIC_BASE.[10] = 1
808H	2056	IA32_X2APIC_TPR	**x2APIC Task Priority Register (R/W)**	If CPUID.01H:ECX.[21] = 1 && IA32_APIC_BASE.[10] = 1
80AH	2058	IA32_X2APIC_PPR	**x2APIC Processor Priority Register (R/O)**	If CPUID.01H:ECX.[21] = 1 && IA32_APIC_BASE.[10] = 1
80BH	2059	IA32_X2APIC_EOI	**x2APIC EOI Register (W/O)**	If CPUID.01H:ECX.[21] = 1 && IA32_APIC_BASE.[10] = 1
80DH	2061	IA32_X2APIC_LDR	**x2APIC Logical Destination Register (R/O)**	If CPUID.01H:ECX.[21] = 1 && IA32_APIC_BASE.[10] = 1
80FH	2063	IA32_X2APIC_SIVR	**x2APIC Spurious Interrupt Vector Register (R/W)**	If CPUID.01H:ECX.[21] = 1 && IA32_APIC_BASE.[10] = 1
810H	2064	IA32_X2APIC_ISR0	**x2APIC In-Service Register Bits 31:0 (R/O)**	If CPUID.01H:ECX.[21] = 1 && IA32_APIC_BASE.[10] = 1
811H	2065	IA32_X2APIC_ISR1	**x2APIC In-Service Register Bits 63:32 (R/O)**	If CPUID.01H:ECX.[21] = 1 && IA32_APIC_BASE.[10] = 1
812H	2066	IA32_X2APIC_ISR2	**x2APIC In-Service Register Bits 95:64 (R/O)**	If CPUID.01H:ECX.[21] = 1 && IA32_APIC_BASE.[10] = 1
813H	2067	IA32_X2APIC_ISR3	**x2APIC In-Service Register Bits 127:96 (R/O)**	If CPUID.01H:ECX.[21] = 1 && IA32_APIC_BASE.[10] = 1
814H	2068	IA32_X2APIC_ISR4	**x2APIC In-Service Register Bits 159:128 (R/O)**	If CPUID.01H:ECX.[21] = 1 && IA32_APIC_BASE.[10] = 1

Table 2-2. IA-32 Architectural MSRs (Contd.)

Register Address		Architectural MSR Name and bit fields (Former MSR Name)	MSR/Bit Description	Comment
Hex	Decimal			
815H	2069	IA32_X2APIC_ISR5	**x2APIC In-Service Register Bits 191:160 (R/O)**	If CPUID.01H:ECX.[21] = 1 && IA32_APIC_BASE.[10] = 1
816H	2070	IA32_X2APIC_ISR6	**x2APIC In-Service Register Bits 223:192 (R/O)**	If CPUID.01H:ECX.[21] = 1 && IA32_APIC_BASE.[10] = 1
817H	2071	IA32_X2APIC_ISR7	**x2APIC In-Service Register Bits 255:224 (R/O)**	If CPUID.01H:ECX.[21] = 1 && IA32_APIC_BASE.[10] = 1
818H	2072	IA32_X2APIC_TMR0	**x2APIC Trigger Mode Register Bits 31:0 (R/O)**	If CPUID.01H:ECX.[21] = 1 && IA32_APIC_BASE.[10] = 1
819H	2073	IA32_X2APIC_TMR1	**x2APIC Trigger Mode Register Bits 63:32 (R/O)**	If CPUID.01H:ECX.[21] = 1 && IA32_APIC_BASE.[10] = 1
81AH	2074	IA32_X2APIC_TMR2	**x2APIC Trigger Mode Register Bits 95:64 (R/O)**	If CPUID.01H:ECX.[21] = 1 && IA32_APIC_BASE.[10] = 1
81BH	2075	IA32_X2APIC_TMR3	**x2APIC Trigger Mode Register Bits 127:96 (R/O)**	If CPUID.01H:ECX.[21] = 1 && IA32_APIC_BASE.[10] = 1
81CH	2076	IA32_X2APIC_TMR4	**x2APIC Trigger Mode Register Bits 159:128 (R/O)**	If CPUID.01H:ECX.[21] = 1 && IA32_APIC_BASE.[10] = 1
81DH	2077	IA32_X2APIC_TMR5	**x2APIC Trigger Mode Register Bits 191:160 (R/O)**	If CPUID.01H:ECX.[21] = 1 && IA32_APIC_BASE.[10] = 1
81EH	2078	IA32_X2APIC_TMR6	**x2APIC Trigger Mode Register Bits 223:192 (R/O)**	If (CPUID.01H:ECX.[21] = 1 && IA32_APIC_BASE.[10] = 1)
81FH	2079	IA32_X2APIC_TMR7	**x2APIC Trigger Mode Register Bits 255:224 (R/O)**	If CPUID.01H:ECX.[21] = 1 && IA32_APIC_BASE.[10] = 1
820H	2080	IA32_X2APIC_IRR0	**x2APIC Interrupt Request Register Bits 31:0 (R/O)**	If CPUID.01H:ECX.[21] = 1 && IA32_APIC_BASE.[10] = 1
821H	2081	IA32_X2APIC_IRR1	**x2APIC Interrupt Request Register Bits 63:32 (R/O)**	If CPUID.01H:ECX.[21] = 1 && IA32_APIC_BASE.[10] = 1
822H	2082	IA32_X2APIC_IRR2	**x2APIC Interrupt Request Register Bits 95:64 (R/O)**	If CPUID.01H:ECX.[21] = 1 && IA32_APIC_BASE.[10] = 1
823H	2083	IA32_X2APIC_IRR3	**x2APIC Interrupt Request Register Bits 127:96 (R/O)**	If CPUID.01H:ECX.[21] = 1 && IA32_APIC_BASE.[10] = 1

Table 2-2. IA-32 Architectural MSRs (Contd.)

Register Address		Architectural MSR Name and bit fields (Former MSR Name)	MSR/Bit Description	Comment
Hex	Decimal			
824H	2084	IA32_X2APIC_IRR4	**x2APIC Interrupt Request Register Bits 159:128 (R/O)**	If CPUID.01H:ECX.[21] = 1 && IA32_APIC_BASE.[10] = 1
825H	2085	IA32_X2APIC_IRR5	**x2APIC Interrupt Request Register Bits 191:160 (R/O)**	If CPUID.01H:ECX.[21] = 1 && IA32_APIC_BASE.[10] = 1
826H	2086	IA32_X2APIC_IRR6	**x2APIC Interrupt Request Register Bits 223:192 (R/O)**	If CPUID.01H:ECX.[21] = 1 && IA32_APIC_BASE.[10] = 1
827H	2087	IA32_X2APIC_IRR7	**x2APIC Interrupt Request Register Bits 255:224 (R/O)**	If CPUID.01H:ECX.[21] = 1 && IA32_APIC_BASE.[10] = 1
828H	2088	IA32_X2APIC_ESR	**x2APIC Error Status Register (R/W)**	If CPUID.01H:ECX.[21] = 1 && IA32_APIC_BASE.[10] = 1
82FH	2095	IA32_X2APIC_LVT_CMCI	**x2APIC LVT Corrected Machine Check Interrupt Register (R/W)**	If CPUID.01H:ECX.[21] = 1 && IA32_APIC_BASE.[10] = 1
830H	2096	IA32_X2APIC_ICR	**x2APIC Interrupt Command Register (R/W)**	If CPUID.01H:ECX.[21] = 1 && IA32_APIC_BASE.[10] = 1
832H	2098	IA32_X2APIC_LVT_TIMER	**x2APIC LVT Timer Interrupt Register (R/W)**	If CPUID.01H:ECX.[21] = 1 && IA32_APIC_BASE.[10] = 1
833H	2099	IA32_X2APIC_LVT_THERMAL	**x2APIC LVT Thermal Sensor Interrupt Register (R/W)**	If CPUID.01H:ECX.[21] = 1 && IA32_APIC_BASE.[10] = 1
834H	2100	IA32_X2APIC_LVT_PMI	**x2APIC LVT Performance Monitor Interrupt Register (R/W)**	If CPUID.01H:ECX.[21] = 1 && IA32_APIC_BASE.[10] = 1
835H	2101	IA32_X2APIC_LVT_LINT0	**x2APIC LVT LINT0 Register (R/W)**	If CPUID.01H:ECX.[21] = 1 && IA32_APIC_BASE.[10] = 1
836H	2102	IA32_X2APIC_LVT_LINT1	**x2APIC LVT LINT1 Register (R/W)**	If CPUID.01H:ECX.[21] = 1 && IA32_APIC_BASE.[10] = 1
837H	2103	IA32_X2APIC_LVT_ERROR	**x2APIC LVT Error Register (R/W)**	If CPUID.01H:ECX.[21] = 1 && IA32_APIC_BASE.[10] = 1
838H	2104	IA32_X2APIC_INIT_COUNT	**x2APIC Initial Count Register (R/W)**	If CPUID.01H:ECX.[21] = 1 && IA32_APIC_BASE.[10] = 1
839H	2105	IA32_X2APIC_CUR_COUNT	**x2APIC Current Count Register (R/O)**	If CPUID.01H:ECX.[21] = 1 && IA32_APIC_BASE.[10] = 1

Table 2-2. IA-32 Architectural MSRs (Contd.)

Register Address		Architectural MSR Name and bit fields (Former MSR Name)	MSR/Bit Description	Comment
Hex	Decimal			
83EH	2110	IA32_X2APIC_DIV_CONF	**x2APIC Divide Configuration Register (R/W)**	If CPUID.01H:ECX.[21] = 1 && IA32_APIC_BASE.[10] = 1
83FH	2111	IA32_X2APIC_SELF_IPI	**x2APIC Self IPI Register (W/O)**	If CPUID.01H:ECX.[21] = 1 && IA32_APIC_BASE.[10] = 1
C80H	3200	IA32_DEBUG_INTERFACE	**Silicon Debug Feature Control (R/W)**	If CPUID.01H:ECX.[11] = 1
		0	**Enable (R/W)** BIOS set 1 to enable Silicon debug features. Default is 0	If CPUID.01H:ECX.[11] = 1
		29:1	Reserved.	
		30	**Lock (R/W)**: If 1, locks any further change to the MSR. The lock bit is set automatically on the first SMI assertion even if not explicitly set by BIOS. Default is 0.	If CPUID.01H:ECX.[11] = 1
		31	**Debug Occurred (R/O)**: This "sticky bit" is set by hardware to indicate the status of bit 0. Default is 0.	If CPUID.01H:ECX.[11] = 1
		63:32	Reserved.	
C81H	3201	IA32_L3_QOS_CFG	**L3 QOS Configuration (R/W)**	If (CPUID.(EAX=10H, ECX=1):ECX.[2] = 1)
		0	**Enable (R/W)** Set 1 to enable L3 CAT masks and COS to operate in Code and Data Prioritization (CDP) mode	
		63:1	Reserved.	
C8DH	3213	IA32_QM_EVTSEL	**Monitoring Event Select Register (R/W)**	If (CPUID.(EAX=07H, ECX=0):EBX.[12] = 1)
		7:0	**Event ID:** ID of a supported monitoring event to report via IA32_QM_CTR.	
		31: 8	**Reserved**.	
		N+31:32	**Resource Monitoring ID:** ID for monitoring hardware to report monitored data via IA32_QM_CTR.	N = Ceil (Log_2 (CPUID.(EAX= 0FH, ECX=0H).EBX[31:0] +1))
		63:N+32	**Reserved.**	
C8EH	3214	IA32_QM_CTR	**Monitoring Counter Register (R/O)**	If (CPUID.(EAX=07H, ECX=0):EBX.[12] = 1)
		61:0	**Resource Monitored Data**	
		62	**Unavailable**: If 1, indicates data for this RMID is not available or not monitored for this resource or RMID.	
		63	**Error:** If 1, indicates and unsupported RMID or event type was written to IA32_PQR_QM_EVTSEL.	

Table 2-2. IA-32 Architectural MSRs (Contd.)

Register Address		Architectural MSR Name and bit fields (Former MSR Name)	MSR/Bit Description	Comment
Hex	Decimal			
C8FH	3215	IA32_PQR_ASSOC	**Resource Association Register (R/W)**	If ((CPUID.(EAX=07H, ECX=0):EBX[12] =1) or (CPUID.(EAX=07H, ECX=0):EBX[15] =1))
		N-1:0	**Resource Monitoring ID (R/W):** ID for monitoring hardware to track internal operation, e.g. memory access.	N = Ceil (Log_2 (CPUID.(EAX= 0FH, ECX=0H).EBX[31:0] +1))
		31:N	**Reserved**	
		63:32	**COS (R/W).** The class of service (COS) to enforce (on writes); returns the current COS when read.	If (CPUID.(EAX=07H, ECX=0):EBX.[15] = 1)
C90H - D8FH		Reserved MSR Address Space for CAT Mask Registers	See Section 17.19.4.1, "Enumeration and Detection Support of Cache Allocation Technology".	
C90H	3216	IA32_L3_MASK_0	**L3 CAT Mask for COS0 (R/W)**	If (CPUID.(EAX=10H, ECX=0H):EBX[1] != 0)
		31:0	**Capacity Bit Mask (R/W)**	
		63:32	Reserved.	
C90H+ n	3216+n	IA32_L3_MASK_n	**L3 CAT Mask for COSn (R/W)**	n = CPUID.(EAX=10H, ECX=1H):EDX[15:0]
		31:0	**Capacity Bit Mask (R/W)**	
		63:32	Reserved.	
D10H - D4FH		Reserved MSR Address Space for L2 CAT Mask Registers	See Section 17.19.4.1, "Enumeration and Detection Support of Cache Allocation Technology".	
D10H	3344	IA32_L2_MASK_0	**L2 CAT Mask for COS0 (R/W)**	If (CPUID.(EAX=10H, ECX=0H):EBX[2] != 0)
		31:0	**Capacity Bit Mask (R/W)**	
		63:32	Reserved.	
D10H+ n	3344+n	IA32_L2_MASK_n	**L2 CAT Mask for COSn (R/W)**	n = CPUID.(EAX=10H, ECX=2H):EDX[15:0]
		31:0	**Capacity Bit Mask (R/W)**	
		63:32	Reserved.	
D90H	3472	IA32_BNDCFGS	**Supervisor State of MPX Configuration. (R/W)**	If (CPUID.(EAX=07H, ECX=0H):EBX[14] = 1)
		0	**EN:** Enable Intel MPX in supervisor mode	

Table 2-2. IA-32 Architectural MSRs (Contd.)

Register Address		Architectural MSR Name and bit fields (Former MSR Name)	MSR/Bit Description	Comment
Hex	Decimal			
		1	**BNDPRESERVE:** Preserve the bounds registers for near branch instructions in the absence of the BND prefix	
		11:2	Reserved, must be 0	
		63:12	**Base Address of Bound Directory.**	
DA0H	3488	IA32_XSS	**Extended Supervisor State Mask (R/W)**	If(CPUID.(0DH, 1):EAX.[3] = 1
		7:0	**Reserved**	
		8	**Trace Packet Configuration State (R/W)**	
		63:9	Reserved.	
DB0H	3504	IA32_PKG_HDC_CTL	**Package Level Enable/disable HDC (R/W)**	If CPUID.06H:EAX.[13] = 1
		0	**HDC_Pkg_Enable (R/W)** Force HDC idling or wake up HDC-idled logical processors in the package. See Section 14.5.2, "Package level Enabling HDC"	If CPUID.06H:EAX.[13] = 1
		63:1	Reserved.	
DB1H	3505	IA32_PM_CTL1	**Enable/disable HWP (R/W)**	If CPUID.06H:EAX.[13] = 1
		0	**HDC_Allow_Block (R/W)** Allow/Block this logical processor for package level HDC control. See Section 14.5.3	If CPUID.06H:EAX.[13] = 1
		63:1	Reserved.	
DB2H	3506	IA32_THREAD_STALL	**Per-Logical_Processor HDC Idle Residency (R/0)**	If CPUID.06H:EAX.[13] = 1
		63:0	**Stall_Cycle_Cnt (R/W)** Stalled cycles due to HDC forced idle on this logical processor. See Section 14.5.4.1	If CPUID.06H:EAX.[13] = 1
4000_0000H - 4000_00FFH		Reserved MSR Address Space	All existing and future processors will not implement MSR in this range.	
C000_0080H		IA32_EFER	**Extended Feature Enables**	If (CPUID.80000001H:EDX.[20] || CPUID.80000001H:EDX.[29])
		0	**SYSCALL Enable: IA32_EFER.SCE (R/W)** Enables SYSCALL/SYSRET instructions in 64-bit mode.	
		7:1	Reserved.	

Table 2-2. IA-32 Architectural MSRs (Contd.)

Register Address		Architectural MSR Name and bit fields (Former MSR Name)	MSR/Bit Description	Comment
Hex	Decimal			
		8	**IA-32e Mode Enable: IA32_EFER.LME (R/W)** Enables IA-32e mode operation.	
		9	Reserved.	
		10	**IA-32e Mode Active: IA32_EFER.LMA (R)** Indicates IA-32e mode is active when set.	
		11	**Execute Disable Bit Enable: IA32_EFER.NXE (R/W)**	
		63:12	Reserved.	
C000_0081H		IA32_STAR	**System Call Target Address (R/W)**	If CPUID.80000001:EDX.[29] = 1
C000_0082H		IA32_LSTAR	**IA-32e Mode System Call Target Address (R/W)**	If CPUID.80000001:EDX.[29] = 1
C000_0084H		IA32_FMASK	**System Call Flag Mask (R/W)**	If CPUID.80000001:EDX.[29] = 1
C000_0100H		IA32_FS_BASE	**Map of BASE Address of FS (R/W)**	If CPUID.80000001:EDX.[29] = 1
C000_0101H		IA32_GS_BASE	**Map of BASE Address of GS (R/W)**	If CPUID.80000001:EDX.[29] = 1
C000_0102H		IA32_KERNEL_GS_BASE	**Swap Target of BASE Address of GS (R/W)**	If CPUID.80000001:EDX.[29] = 1
C000_0103H		IA32_TSC_AUX	Auxiliary TSC (RW)	If CPUID.80000001H: EDX[27] = 1
		31:0	AUX: Auxiliary signature of TSC	
		63:32	Reserved.	

NOTES:

1. In processors based on Intel NetBurst® microarchitecture, MSR addresses 180H-197H are supported, software must treat them as model-specific. Starting with Intel Core Duo processors, MSR addresses 180H-185H, 188H-197H are reserved.
2. The *_ADDR MSRs may or may not be present; this depends on flag settings in IA32_MC*i*_STATUS. See Section 15.3.2.3 and Section 15.3.2.4 for more information.
3. MAXPHYADDR is reported by CPUID.80000008H:EAX[7:0].

2.2 MSRS IN THE INTEL® CORE™ 2 PROCESSOR FAMILY

Table 2-3 lists model-specific registers (MSRs) for Intel Core 2 processor family and for Intel Xeon processors based on Intel Core microarchitecture, architectural MSR addresses are also included in Table 2-3. These processors have a CPUID signature with DisplayFamily_DisplayModel of 06_0FH, see Table 2-1.

MSRs listed in Table 2-2 and Table 2-3 are also supported by processors based on the Enhanced Intel Core micro-architecture. Processors based on the Enhanced Intel Core microarchitecture have the CPUID signature DisplayFamily_DisplayModel of 06_17H.

The column "Shared/Unique" applies to multi-core processors based on Intel Core microarchitecture. "Unique" means each processor core has a separate MSR, or a bit field in an MSR governs only a core independently. "Shared" means the MSR or the bit field in an MSR address governs the operation of both processor cores.

Table 2-3. MSRs in Processors Based on Intel® Core™ Microarchitecture

Register Address		Register Name	Shared/ Unique	Bit Description
Hex	Dec			
0H	0	IA32_P5_MC_ADDR	Unique	See Section 2.22, "MSRs in Pentium Processors."
1H	1	IA32_P5_MC_TYPE	Unique	See Section 2.22, "MSRs in Pentium Processors."
6H	6	IA32_MONITOR_FILTER_SIZE	Unique	See Section 8.10.5, "Monitor/Mwait Address Range Determination." andTable 2-2.
10H	16	IA32_TIME_STAMP_COUNTER	Unique	See Section 17.17, "Time-Stamp Counter," and see Table 2-2.
17H	23	IA32_PLATFORM_ID	Shared	**Platform ID (R)** See Table 2-2.
17H	23	MSR_PLATFORM_ID	Shared	**Model Specific Platform ID (R)**
		7:0		Reserved.
		12:8		**Maximum Qualified Ratio (R)** The maximum allowed bus ratio.
		49:13		Reserved.
		52:50		See Table 2-2.
		63:53		Reserved.
1BH	27	IA32_APIC_BASE	Unique	See Section 10.4.4, "Local APIC Status and Location." and Table 2-2.
2AH	42	MSR_EBL_CR_POWERON	Shared	**Processor Hard Power-On Configuration (R/W)** Enables and disables processor features; (R) indicates current processor configuration.
		0		Reserved.
		1		**Data Error Checking Enable (R/W)** 1 = Enabled; 0 = Disabled Note: Not all processor implements R/W.
		2		**Response Error Checking Enable (R/W)** 1 = Enabled; 0 = Disabled Note: Not all processor implements R/W.
		3		**MCERR# Drive Enable (R/W)** 1 = Enabled; 0 = Disabled Note: Not all processor implements R/W.
		4		**Address Parity Enable (R/W)** 1 = Enabled; 0 = Disabled Note: Not all processor implements R/W.
		5		Reserved.

Table 2-3. MSRs in Processors Based on Intel® Core™ Microarchitecture (Contd.)

Register Address		Register Name	Shared/ Unique	Bit Description
Hex	Dec			
		6		Reserved.
		7		**BINIT# Driver Enable (R/W)** 1 = Enabled; 0 = Disabled Note: Not all processor implements R/W.
		8		**Output Tri-state Enabled (R/O)** 1 = Enabled; 0 = Disabled
		9		**Execute BIST (R/O)** 1 = Enabled; 0 = Disabled
		10		**MCERR# Observation Enabled (R/O)** 1 = Enabled; 0 = Disabled
		11		Intel TXT Capable Chipset. (R/O) 1 = Present; 0 = Not Present
		12		**BINIT# Observation Enabled (R/O)** 1 = Enabled; 0 = Disabled
		13		Reserved.
		14		**1 MByte Power on Reset Vector (R/O)** 1 = 1 MByte; 0 = 4 GBytes
		15		Reserved.
		17:16		**APIC Cluster ID (R/O)**
		18		**N/2 Non-Integer Bus Ratio (R/O)** 0 = Integer ratio; 1 = Non-integer ratio
		19		Reserved.
		21: 20		**Symmetric Arbitration ID (R/O)**
		26:22		**Integer Bus Frequency Ratio (R/O)**
3AH	58	MSR_FEATURE_CONTROL	Unique	**Control Features in Intel 64Processor (R/W)** See Table 2-2.
		3	Unique	**SMRR Enable (R/WL)** When this bit is set and the lock bit is set makes the SMRR_PHYS_BASE and SMRR_PHYS_MASK registers read visible and writeable while in SMM.
40H	64	MSR_ LASTBRANCH_0_FROM_IP	Unique	**Last Branch Record 0 From IP (R/W)** One of four pairs of last branch record registers on the last branch record stack. The From_IP part of the stack contains pointers to the **source instruction**. See also: ▪ Last Branch Record Stack TOS at 1C9H ▪ Section 17.5
41H	65	MSR_ LASTBRANCH_1_FROM_IP	Unique	**Last Branch Record 1 From IP (R/W)** See description of MSR_LASTBRANCH_0_FROM_IP.

Table 2-3. MSRs in Processors Based on Intel® Core™ Microarchitecture (Contd.)

Register Address		Register Name	Shared/ Unique	Bit Description
Hex	Dec			
42H	66	MSR_ LASTBRANCH_2_FROM_IP	Unique	**Last Branch Record 2 From IP (R/W)** See description of MSR_LASTBRANCH_0_FROM_IP.
43H	67	MSR_ LASTBRANCH_3_FROM_IP	Unique	**Last Branch Record 3 From IP (R/W)** See description of MSR_LASTBRANCH_0_FROM_IP.
60H	96	MSR_ LASTBRANCH_0_TO_IP	Unique	**Last Branch Record 0 To IP (R/W)** One of four pairs of last branch record registers on the last branch record stack. This To_IP part of the stack contains pointers to the destination instruction.
61H	97	MSR_ LASTBRANCH_1_TO_IP	Unique	**Last Branch Record 1 To IP (R/W)** See description of MSR_LASTBRANCH_0_TO_IP.
62H	98	MSR_ LASTBRANCH_2_TO_IP	Unique	**Last Branch Record 2 To IP (R/W)** See description of MSR_LASTBRANCH_0_TO_IP.
63H	99	MSR_ LASTBRANCH_3_TO_IP	Unique	**Last Branch Record 3 To IP (R/W)** See description of MSR_LASTBRANCH_0_TO_IP.
79H	121	IA32_BIOS_UPDT_TRIG	Unique	**BIOS Update Trigger Register (W)** See Table 2-2.
8BH	139	IA32_BIOS_SIGN_ID	Unique	**BIOS Update Signature ID (RO)** See Table 2-2.
A0H	160	MSR_SMRR_PHYSBASE	Unique	**System Management Mode Base Address register (WO in SMM)** Model-specific implementation of SMRR-like interface, read visible and write only in SMM.
		11:0		Reserved.
		31:12		PhysBase. SMRR physical Base Address.
		63:32		Reserved.
A1H	161	MSR_SMRR_PHYSMASK	Unique	**System Management Mode Physical Address Mask register (WO in SMM)** Model-specific implementation of SMRR-like interface, read visible and write only in SMM.
		10:0		Reserved.
		11		Valid. Physical address base and range mask are valid.
		31:12		PhysMask. SMRR physical address range mask.
		63:32		Reserved.
C1H	193	IA32_PMC0	Unique	**Performance Counter Register** See Table 2-2.
C2H	194	IA32_PMC1	Unique	**Performance Counter Register** See Table 2-2.
CDH	205	MSR_FSB_FREQ	Shared	**Scaleable Bus Speed(RO)** This field indicates the intended scaleable bus clock speed for processors based on Intel Core microarchitecture:

Table 2-3. MSRs in Processors Based on Intel® Core™ Microarchitecture (Contd.)

Register Address		Register Name	Shared/ Unique	Bit Description
Hex	Dec			
		2:0		▪ 101B: 100 MHz (FSB 400) ▪ 001B: 133 MHz (FSB 533) ▪ 011B: 167 MHz (FSB 667) ▪ 010B: 200 MHz (FSB 800) ▪ 000B: 267 MHz (FSB 1067) ▪ 100B: 333 MHz (FSB 1333)
				133.33 MHz should be utilized if performing calculation with System Bus Speed when encoding is 001B. 166.67 MHz should be utilized if performing calculation with System Bus Speed when encoding is 011B.
				266.67 MHz should be utilized if performing calculation with System Bus Speed when encoding is 000B. 333.33 MHz should be utilized if performing calculation with System Bus Speed when encoding is 100B.
		63:3		Reserved.
CDH	205	MSR_FSB_FREQ	Shared	**Scaleable Bus Speed(RO)** This field indicates the intended scaleable bus clock speed for processors based on Enhanced Intel Core microarchitecture:
		2:0		▪ 101B: 100 MHz (FSB 400) ▪ 001B: 133 MHz (FSB 533) ▪ 011B: 167 MHz (FSB 667) ▪ 010B: 200 MHz (FSB 800) ▪ 000B: 267 MHz (FSB 1067) ▪ 100B: 333 MHz (FSB 1333) ▪ 110B: 400 MHz (FSB 1600)
				133.33 MHz should be utilized if performing calculation with System Bus Speed when encoding is 001B. 166.67 MHz should be utilized if performing calculation with System Bus Speed when encoding is 011B.
				266.67 MHz should be utilized if performing calculation with System Bus Speed when encoding is 110B. 333.33 MHz should be utilized if performing calculation with System Bus Speed when encoding is 111B.
		63:3		Reserved.
E7H	231	IA32_MPERF	Unique	**Maximum Performance Frequency Clock Count (RW)** See Table 2-2.
E8H	232	IA32_APERF	Unique	**Actual Performance Frequency Clock Count (RW)** See Table 2-2.
FEH	254	IA32_MTRRCAP	Unique	See Table 2-2.
		11	Unique	**SMRR Capability Using MSR 0A0H and 0A1H (R)**
174H	372	IA32_SYSENTER_CS	Unique	See Table 2-2.
175H	373	IA32_SYSENTER_ESP	Unique	See Table 2-2.
176H	374	IA32_SYSENTER_EIP	Unique	See Table 2-2.

Table 2-3. MSRs in Processors Based on Intel® Core™ Microarchitecture (Contd.)

Register Address		Register Name	Shared/ Unique	Bit Description
Hex	Dec			
179H	377	IA32_MCG_CAP	Unique	See Table 2-2.
17AH	378	IA32_MCG_STATUS	Unique	Global Machine Check Status
		0		**RIPV** When set, bit indicates that the instruction addressed by the instruction pointer pushed on the stack (when the machine check was generated) can be used to restart the program. If cleared, the program cannot be reliably restarted.
		1		**EIPV** When set, bit indicates that the instruction addressed by the instruction pointer pushed on the stack (when the machine check was generated) is directly associated with the error.
		2		**MCIP** When set, bit indicates that a machine check has been generated. If a second machine check is detected while this bit is still set, the processor enters a shutdown state. Software should write this bit to 0 after processing a machine check exception.
		63:3		Reserved.
186H	390	IA32_PERFEVTSEL0	Unique	See Table 2-2.
187H	391	IA32_PERFEVTSEL1	Unique	See Table 2-2.
198H	408	IA32_PERF_STATUS	Shared	See Table 2-2.
198H	408	MSR_PERF_STATUS	Shared	Current performance status. See Section 14.1.1, “Software Interface For Initiating Performance State Transitions”.
		15:0		Current Performance State Value.
		30:16		Reserved.
		31		XE Operation (R/O). If set, XE operation is enabled. Default is cleared.
		39:32		Reserved.
		44:40		Maximum Bus Ratio (R/O) Indicates maximum bus ratio configured for the processor.
		45		Reserved.
		46		Non-Integer Bus Ratio (R/O) Indicates non-integer bus ratio is enabled. Applies processors based on Enhanced Intel Core microarchitecture.
		63:47		Reserved.
199H	409	IA32_PERF_CTL	Unique	See Table 2-2.
19AH	410	IA32_CLOCK_MODULATION	Unique	**Clock Modulation (R/W)** See Table 2-2. IA32_CLOCK_MODULATION MSR was originally named IA32_THERM_CONTROL MSR.

Table 2-3. MSRs in Processors Based on Intel® Core™ Microarchitecture (Contd.)

Register Address		Register Name	Shared/ Unique	Bit Description
Hex	Dec			
19BH	411	IA32_THERM_INTERRUPT	Unique	**Thermal Interrupt Control (R/W)** See Table 2-2.
19CH	412	IA32_THERM_STATUS	Unique	**Thermal Monitor Status (R/W)** See Table 2-2.
19DH	413	MSR_THERM2_CTL	Unique	Thermal Monitor 2 Control
		15:0		Reserved.
		16		**TM_SELECT (R/W)** Mode of automatic thermal monitor: 0 = Thermal Monitor 1 (thermally-initiated on-die modulation of the stop-clock duty cycle) 1 = Thermal Monitor 2 (thermally-initiated frequency transitions) If bit 3 of the IA32_MISC_ENABLE register is cleared, TM_SELECT has no effect. Neither TM1 nor TM2 are enabled.
		63:16		Reserved.
1A0H	416	IA32_MISC_ENABLE		**Enable Misc. Processor Features (R/W)** Allows a variety of processor functions to be enabled and disabled.
		0		**Fast-Strings Enable** See Table 2-2.
		2:1		Reserved.
		3	Unique	**Automatic Thermal Control Circuit Enable (R/W)** See Table 2-2.
		6:4		Reserved.
		7	Shared	**Performance Monitoring Available (R)** See Table 2-2.
		8		Reserved.
		9		**Hardware Prefetcher Disable (R/W)** When set, disables the hardware prefetcher operation on streams of data. When clear (default), enables the prefetch queue. Disabling of the hardware prefetcher may impact processor performance.
		10	Shared	**FERR# Multiplexing Enable (R/W)** 1 = FERR# asserted by the processor to indicate a pending break event within the processor 0 = Indicates compatible FERR# signaling behavior This bit must be set to 1 to support XAPIC interrupt model usage.
		11	Shared	**Branch Trace Storage Unavailable (RO)** See Table 2-2.
		12	Shared	**Processor Event Based Sampling Unavailable (RO)** See Table 2-2.

Table 2-3. MSRs in Processors Based on Intel® Core™ Microarchitecture (Contd.)

Register Address		Register Name	Shared/ Unique	Bit Description
Hex	Dec			
		13	Shared	**TM2 Enable (R/W)** When this bit is set (1) and the thermal sensor indicates that the die temperature is at the pre-determined threshold, the Thermal Monitor 2 mechanism is engaged. TM2 will reduce the bus to core ratio and voltage according to the value last written to MSR_THERM2_CTL bits 15:0.
				When this bit is clear (0, default), the processor does not change the VID signals or the bus to core ratio when the processor enters a thermally managed state. The BIOS must enable this feature if the TM2 feature flag (CPUID.1:ECX[8]) is set; if the TM2 feature flag is not set, this feature is not supported and BIOS must not alter the contents of the TM2 bit location. The processor is operating out of specification if both this bit and the TM1 bit are set to 0.
		15:14		Reserved.
		16	Shared	**Enhanced Intel SpeedStep Technology Enable (R/W)** See Table 2-2.
		18	Shared	**ENABLE MONITOR FSM (R/W)** See Table 2-2.
		19	Shared	**Adjacent Cache Line Prefetch Disable (R/W)** When set to 1, the processor fetches the cache line that contains data currently required by the processor. When set to 0, the processor fetches cache lines that comprise a cache line pair (128 bytes). Single processor platforms should not set this bit. Server platforms should set or clear this bit based on platform performance observed in validation and testing. BIOS may contain a setup option that controls the setting of this bit.
		20	Shared	**Enhanced Intel SpeedStep Technology Select Lock (R/WO)** When set, this bit causes the following bits to become read-only: ▪ Enhanced Intel SpeedStep Technology Select Lock (this bit), ▪ Enhanced Intel SpeedStep Technology Enable bit. The bit must be set before an Enhanced Intel SpeedStep Technology transition is requested. This bit is cleared on reset.
		21		Reserved.
		22	Shared	**Limit CPUID Maxval (R/W)** See Table 2-2.
		23	Shared	**xTPR Message Disable (R/W)** See Table 2-2.
		33:24		Reserved.

Table 2-3. MSRs in Processors Based on Intel® Core™ Microarchitecture (Contd.)

Register Address		Register Name	Shared/ Unique	Bit Description
Hex	Dec			
		34	Unique	**XD Bit Disable (R/W)** See Table 2-2.
		36:35		Reserved.
		37	Unique	**DCU Prefetcher Disable (R/W)** When set to 1, The DCU L1 data cache prefetcher is disabled. The default value after reset is 0. BIOS may write '1' to disable this feature. The DCU prefetcher is an L1 data cache prefetcher. When the DCU prefetcher detects multiple loads from the same line done within a time limit, the DCU prefetcher assumes the next line will be required. The next line is prefetched in to the L1 data cache from memory or L2.
		38	Shared	**IDA Disable (R/W)** When set to 1 on processors that support IDA, the Intel Dynamic Acceleration feature (IDA) is disabled and the IDA_Enable feature flag will be clear (CPUID.06H: EAX[1]=0). When set to a 0 on processors that support IDA, CPUID.06H: EAX[1] reports the processor's support of IDA is enabled. **Note:** the power-on default value is used by BIOS to detect hardware support of IDA. If power-on default value is 1, IDA is available in the processor. If power-on default value is 0, IDA is not available.
		39	Unique	**IP Prefetcher Disable (R/W)** When set to 1, The IP prefetcher is disabled. The default value after reset is 0. BIOS may write '1' to disable this feature. The IP prefetcher is an L1 data cache prefetcher. The IP prefetcher looks for sequential load history to determine whether to prefetch the next expected data into the L1 cache from memory or L2.
		63:40		Reserved.
1C9H	457	MSR_LASTBRANCH_TOS	Unique	**Last Branch Record Stack TOS (R/W)** Contains an index (bits 0-3) that points to the MSR containing the most recent branch record. See MSR_LASTBRANCH_0_FROM_IP (at 40H).
1D9H	473	IA32_DEBUGCTL	Unique	**Debug Control (R/W)** See Table 2-2
1DDH	477	MSR_LER_FROM_LIP	Unique	**Last Exception Record From Linear IP (R)** Contains a pointer to the last branch instruction that the processor executed prior to the last exception that was generated or the last interrupt that was handled.
1DEH	478	MSR_LER_TO_LIP	Unique	**Last Exception Record To Linear IP (R)** This area contains a pointer to the target of the last branch instruction that the processor executed prior to the last exception that was generated or the last interrupt that was handled.

Table 2-3. MSRs in Processors Based on Intel® Core™ Microarchitecture (Contd.)

Register Address		Register Name	Shared/ Unique	Bit Description
Hex	Dec			
200H	512	IA32_MTRR_PHYSBASE0	Unique	See Table 2-2.
201H	513	IA32_MTRR_PHYSMASK0	Unique	See Table 2-2.
202H	514	IA32_MTRR_PHYSBASE1	Unique	See Table 2-2.
203H	515	IA32_MTRR_PHYSMASK1	Unique	See Table 2-2.
204H	516	IA32_MTRR_PHYSBASE2	Unique	See Table 2-2.
205H	517	IA32_MTRR_PHYSMASK2	Unique	See Table 2-2.
206H	518	IA32_MTRR_PHYSBASE3	Unique	See Table 2-2.
207H	519	IA32_MTRR_PHYSMASK3	Unique	See Table 2-2.
208H	520	IA32_MTRR_PHYSBASE4	Unique	See Table 2-2.
209H	521	IA32_MTRR_PHYSMASK4	Unique	See Table 2-2.
20AH	522	IA32_MTRR_PHYSBASE5	Unique	See Table 2-2.
20BH	523	IA32_MTRR_PHYSMASK5	Unique	See Table 2-2.
20CH	524	IA32_MTRR_PHYSBASE6	Unique	See Table 2-2.
20DH	525	IA32_MTRR_PHYSMASK6	Unique	See Table 2-2.
20EH	526	IA32_MTRR_PHYSBASE7	Unique	See Table 2-2.
20FH	527	IA32_MTRR_PHYSMASK7	Unique	See Table 2-2.
250H	592	IA32_MTRR_FIX64K_ 00000	Unique	See Table 2-2.
258H	600	IA32_MTRR_FIX16K_ 80000	Unique	See Table 2-2.
259H	601	IA32_MTRR_FIX16K_ A0000	Unique	See Table 2-2.
268H	616	IA32_MTRR_FIX4K_C0000	Unique	See Table 2-2.
269H	617	IA32_MTRR_FIX4K_C8000	Unique	See Table 2-2.
26AH	618	IA32_MTRR_FIX4K_D0000	Unique	See Table 2-2.
26BH	619	IA32_MTRR_FIX4K_D8000	Unique	See Table 2-2.
26CH	620	IA32_MTRR_FIX4K_E0000	Unique	See Table 2-2.
26DH	621	IA32_MTRR_FIX4K_E8000	Unique	See Table 2-2.
26EH	622	IA32_MTRR_FIX4K_F0000	Unique	See Table 2-2.
26FH	623	IA32_MTRR_FIX4K_F8000	Unique	See Table 2-2.
277H	631	IA32_PAT	Unique	See Table 2-2.
2FFH	767	IA32_MTRR_DEF_TYPE	Unique	**Default Memory Types (R/W)** See Table 2-2.
309H	777	IA32_FIXED_CTR0	Unique	**Fixed-Function Performance Counter Register 0 (R/W)** See Table 2-2.
309H	777	MSR_PERF_FIXED_CTR0	Unique	**Fixed-Function Performance Counter Register 0 (R/W)**

Table 2-3. MSRs in Processors Based on Intel® Core™ Microarchitecture (Contd.)

Register Address		Register Name	Shared/ Unique	Bit Description
Hex	Dec			
30AH	778	IA32_FIXED_CTR1	Unique	**Fixed-Function Performance Counter Register 1 (R/W)** See Table 2-2.
30AH	778	MSR_PERF_FIXED_CTR1	Unique	**Fixed-Function Performance Counter Register 1 (R/W)**
30BH	779	IA32_FIXED_CTR2	Unique	**Fixed-Function Performance Counter Register 2 (R/W)** See Table 2-2.
30BH	779	MSR_PERF_FIXED_CTR2	Unique	**Fixed-Function Performance Counter Register 2 (R/W)**
345H	837	IA32_PERF_CAPABILITIES	Unique	See Table 2-2. See Section 17.4.1, "IA32_DEBUGCTL MSR."
345H	837	MSR_PERF_CAPABILITIES	Unique	RO. This applies to processors that do not support architectural perfmon version 2.
		5:0		LBR Format. See Table 2-2.
		6		PEBS Record Format.
		7		PEBSSaveArchRegs. See Table 2-2.
		63:8		Reserved.
38DH	909	IA32_FIXED_CTR_CTRL	Unique	**Fixed-Function-Counter Control Register (R/W)** See Table 2-2.
38DH	909	MSR_PERF_FIXED_CTR_ CTRL	Unique	**Fixed-Function-Counter Control Register (R/W)**
38EH	910	IA32_PERF_GLOBAL_ STATUS	Unique	See Table 2-2. See Section 18.6.2.2, "Global Counter Control Facilities."
38EH	910	MSR_PERF_GLOBAL_STATU S	Unique	See Section 18.6.2.2, "Global Counter Control Facilities."
38FH	911	IA32_PERF_GLOBAL_CTRL	Unique	See Table 2-2. See Section 18.6.2.2, "Global Counter Control Facilities."
38FH	911	MSR_PERF_GLOBAL_CTRL	Unique	See Section 18.6.2.2, "Global Counter Control Facilities."
390H	912	IA32_PERF_GLOBAL_OVF_ CTRL	Unique	See Table 2-2. See Section 18.6.2.2, "Global Counter Control Facilities."
390H	912	MSR_PERF_GLOBAL_OVF_ CTRL	Unique	See Section 18.6.2.2, "Global Counter Control Facilities."
3F1H	1009	MSR_PEBS_ENABLE	Unique	See Table 2-2. See Section 18.6.2.4, "Processor Event Based Sampling (PEBS)."
		0		Enable PEBS on IA32_PMC0. (R/W)
400H	1024	IA32_MC0_CTL	Unique	See Section 15.3.2.1, "IA32_MCi_CTL MSRs."
401H	1025	IA32_MC0_STATUS	Unique	See Section 15.3.2.2, "IA32_MCi_STATUS MSRS."
402H	1026	IA32_MC0_ADDR	Unique	See Section 15.3.2.3, "IA32_MCi_ADDR MSRs." The IA32_MC0_ADDR register is either not implemented or contains no address if the ADDRV flag in the IA32_MC0_STATUS register is clear. When not implemented in the processor, all reads and writes to this MSR will cause a general-protection exception.

Table 2-3. MSRs in Processors Based on Intel® Core™ Microarchitecture (Contd.)

Register Address		Register Name	Shared/ Unique	Bit Description
Hex	Dec			
404H	1028	IA32_MC1_CTL	Unique	See Section 15.3.2.1, "IA32_MCi_CTL MSRs."
405H	1029	IA32_MC1_STATUS	Unique	See Section 15.3.2.2, "IA32_MCi_STATUS MSRS."
406H	1030	IA32_MC1_ADDR	Unique	See Section 15.3.2.3, "IA32_MCi_ADDR MSRs." The IA32_MC1_ADDR register is either not implemented or contains no address if the ADDRV flag in the IA32_MC1_STATUS register is clear. When not implemented in the processor, all reads and writes to this MSR will cause a general-protection exception.
408H	1032	IA32_MC2_CTL	Unique	See Section 15.3.2.1, "IA32_MCi_CTL MSRs."
409H	1033	IA32_MC2_STATUS	Unique	See Section 15.3.2.2, "IA32_MCi_STATUS MSRS."
40AH	1034	IA32_MC2_ADDR	Unique	See Section 15.3.2.3, "IA32_MCi_ADDR MSRs." The IA32_MC2_ADDR register is either not implemented or contains no address if the ADDRV flag in the IA32_MC2_STATUS register is clear. When not implemented in the processor, all reads and writes to this MSR will cause a general-protection exception.
40CH	1036	IA32_MC4_CTL	Unique	See Section 15.3.2.1, "IA32_MCi_CTL MSRs."
40DH	1037	IA32_MC4_STATUS	Unique	See Section 15.3.2.2, "IA32_MCi_STATUS MSRS."
40EH	1038	IA32_MC4_ADDR	Unique	See Section 15.3.2.3, "IA32_MCi_ADDR MSRs." The MSR_MC4_ADDR register is either not implemented or contains no address if the ADDRV flag in the MSR_MC4_STATUS register is clear. When not implemented in the processor, all reads and writes to this MSR will cause a general-protection exception.
410H	1040	IA32_MC3_CTL		See Section 15.3.2.1, "IA32_MCi_CTL MSRs."
411H	1041	IA32_MC3_STATUS		See Section 15.3.2.2, "IA32_MCi_STATUS MSRS."
412H	1042	IA32_MC3_ADDR	Unique	See Section 15.3.2.3, "IA32_MCi_ADDR MSRs." The MSR_MC3_ADDR register is either not implemented or contains no address if the ADDRV flag in the MSR_MC3_STATUS register is clear. When not implemented in the processor, all reads and writes to this MSR will cause a general-protection exception.
413H	1043	IA32_MC3_MISC	Unique	Machine Check Error Reporting Register - contains additional information describing the machine-check error if the MISCV flag in the IA32_MCi_STATUS register is set.
414H	1044	IA32_MC5_CTL	Unique	Machine Check Error Reporting Register - controls signaling of #MC for errors produced by a particular hardware unit (or group of hardware units).

Table 2-3. MSRs in Processors Based on Intel® Core™ Microarchitecture (Contd.)

Register Address		Register Name	Shared/ Unique	Bit Description
Hex	Dec			
415H	1045	IA32_MC5_STATUS	Unique	Machine Check Error Reporting Register - contains information related to a machine-check error if its VAL (valid) flag is set. Software is responsible for clearing IA32_MCi_STATUS MSRs by explicitly writing 0s to them; writing 1s to them causes a general-protection exception.
416H	1046	IA32_MC5_ADDR	Unique	Machine Check Error Reporting Register - contains the address of the code or data memory location that produced the machine-check error if the ADDRV flag in the IA32_MCi_STATUS register is set.
417H	1047	IA32_MC5_MISC	Unique	Machine Check Error Reporting Register - contains additional information describing the machine-check error if the MISCV flag in the IA32_MCi_STATUS register is set.
419H	1045	IA32_MC6_STATUS	Unique	Apply to Intel Xeon processor 7400 series (processor signature 06_1D) only. See Section 15.3.2.2, "IA32_MCi_STATUS MSRS." and Chapter 23.
480H	1152	IA32_VMX_BASIC	Unique	**Reporting Register of Basic VMX Capabilities (R/O)** See Table 2-2. See Appendix A.1, "Basic VMX Information."
481H	1153	IA32_VMX_PINBASED_ CTLS	Unique	**Capability Reporting Register of Pin-based VM-execution Controls (R/O)** See Table 2-2. See Appendix A.3, "VM-Execution Controls."
482H	1154	IA32_VMX_PROCBASED_ CTLS	Unique	**Capability Reporting Register of Primary Processor-based VM-execution Controls (R/O)** See Appendix A.3, "VM-Execution Controls."
483H	1155	IA32_VMX_EXIT_CTLS	Unique	**Capability Reporting Register of VM-exit Controls (R/O)** See Table 2-2. See Appendix A.4, "VM-Exit Controls."
484H	1156	IA32_VMX_ENTRY_CTLS	Unique	**Capability Reporting Register of VM-entry Controls (R/O)** See Table 2-2. See Appendix A.5, "VM-Entry Controls."
485H	1157	IA32_VMX_MISC	Unique	**Reporting Register of Miscellaneous VMX Capabilities (R/O)** See Table 2-2. See Appendix A.6, "Miscellaneous Data."
486H	1158	IA32_VMX_CR0_FIXED0	Unique	**Capability Reporting Register of CR0 Bits Fixed to 0 (R/O)** See Table 2-2. See Appendix A.7, "VMX-Fixed Bits in CR0."
487H	1159	IA32_VMX_CR0_FIXED1	Unique	**Capability Reporting Register of CR0 Bits Fixed to 1 (R/O)** See Table 2-2. See Appendix A.7, "VMX-Fixed Bits in CR0."

Table 2-3. MSRs in Processors Based on Intel® Core™ Microarchitecture (Contd.)

Register Address		Register Name	Shared/ Unique	Bit Description
Hex	Dec			
488H	1160	IA32_VMX_CR4_FIXED0	Unique	**Capability Reporting Register of CR4 Bits Fixed to 0 (R/O)** See Table 2-2. See Appendix A.8, "VMX-Fixed Bits in CR4."
489H	1161	IA32_VMX_CR4_FIXED1	Unique	**Capability Reporting Register of CR4 Bits Fixed to 1 (R/O)** See Table 2-2. See Appendix A.8, "VMX-Fixed Bits in CR4."
48AH	1162	IA32_VMX_VMCS_ENUM	Unique	**Capability Reporting Register of VMCS Field Enumeration (R/O)** See Table 2-2. See Appendix A.9, "VMCS Enumeration."
48BH	1163	IA32_VMX_PROCBASED_ CTLS2	Unique	**Capability Reporting Register of Secondary Processor-based VM-execution Controls (R/O)** See Appendix A.3, "VM-Execution Controls."
600H	1536	IA32_DS_AREA	Unique	**DS Save Area (R/W)** See Table 2-2. See Section 18.6.3.4, "Debug Store (DS) Mechanism."
107CC H		MSR_EMON_L3_CTR_CTL0	Unique	**GBUSQ Event Control/Counter Register (R/W)** Apply to Intel Xeon processor 7400 series (processor signature 06_1D) only. See Section 17.2.2
107CD H		MSR_EMON_L3_CTR_CTL1	Unique	**GBUSQ Event Control/Counter Register (R/W)** Apply to Intel Xeon processor 7400 series (processor signature 06_1D) only. See Section 17.2.2
107CE H		MSR_EMON_L3_CTR_CTL2	Unique	**GSNPQ Event Control/Counter Register (R/W)** Apply to Intel Xeon processor 7400 series (processor signature 06_1D) only. See Section 17.2.2
107CF H		MSR_EMON_L3_CTR_CTL3	Unique	**GSNPQ Event Control/Counter Register (R/W)** Apply to Intel Xeon processor 7400 series (processor signature 06_1D) only. See Section 17.2.2
107D0 H		MSR_EMON_L3_CTR_CTL4	Unique	**FSB Event Control/Counter Register (R/W)** Apply to Intel Xeon processor 7400 series (processor signature 06_1D) only. See Section 17.2.2
107D1 H		MSR_EMON_L3_CTR_CTL5	Unique	**FSB Event Control/Counter Register (R/W)** Apply to Intel Xeon processor 7400 series (processor signature 06_1D) only. See Section 17.2.2
107D2 H		MSR_EMON_L3_CTR_CTL6	Unique	**FSB Event Control/Counter Register (R/W)** Apply to Intel Xeon processor 7400 series (processor signature 06_1D) only. See Section 17.2.2
107D3 H		MSR_EMON_L3_CTR_CTL7	Unique	**FSB Event Control/Counter Register (R/W)** Apply to Intel Xeon processor 7400 series (processor signature 06_1D) only. See Section 17.2.2

Table 2-3. MSRs in Processors Based on Intel® Core™ Microarchitecture (Contd.)

Register Address		Register Name	Shared/ Unique	Bit Description
Hex	Dec			
107D8H		MSR_EMON_L3_GL_CTL	Unique	**L3/FSB Common Control Register (R/W)** Apply to Intel Xeon processor 7400 series (processor signature 06_1D) only. See Section 17.2.2
C000_0080H		IA32_EFER	Unique	**Extended Feature Enables** See Table 2-2.
C000_0081H		IA32_STAR	Unique	**System Call Target Address (R/W)** See Table 2-2.
C000_0082H		IA32_LSTAR	Unique	**IA-32e Mode System Call Target Address (R/W)** See Table 2-2.
C000_0084H		IA32_FMASK	Unique	**System Call Flag Mask (R/W)** See Table 2-2.
C000_0100H		IA32_FS_BASE	Unique	**Map of BASE Address of FS (R/W)** See Table 2-2.
C000_0101H		IA32_GS_BASE	Unique	**Map of BASE Address of GS (R/W)** See Table 2-2.
C000_0102H		IA32_KERNEL_GS_BASE	Unique	**Swap Target of BASE Address of GS (R/W)** See Table 2-2.

2.3 MSRS IN THE 45 NM AND 32 NM INTEL® ATOM™ PROCESSOR FAMILY

Table 2-4 lists model-specific registers (MSRs) for 45 nm and 32 nm Intel Atom processors, architectural MSR addresses are also included in Table 2-4. These processors have a CPUID signature with DisplayFamily_DisplayModel of 06_1CH, 06_26H, 06_27H, 06_35H and 06_36H; see Table 2-1.

The column "Shared/Unique" applies to logical processors sharing the same core in processors based on the Intel Atom microarchitecture. "Unique" means each logical processor has a separate MSR, or a bit field in an MSR governs only a logical processor. "Shared" means the MSR or the bit field in an MSR address governs the operation of both logical processors in the same core.

Table 2-4. MSRs in 45 nm and 32 nm Intel® Atom™ Processor Family

Register Address		Register Name	Shared/ Unique	Bit Description
Hex	Dec			
0H	0	IA32_P5_MC_ADDR	Shared	See Section 2.22, "MSRs in Pentium Processors."
1H	1	IA32_P5_MC_TYPE	Shared	See Section 2.22, "MSRs in Pentium Processors."
6H	6	IA32_MONITOR_FILTER_SIZE	Unique	See Section 8.10.5, "Monitor/Mwait Address Range Determination." andTable 2-2
10H	16	IA32_TIME_STAMP_COUNTER	Unique	See Section 17.17, "Time-Stamp Counter," and see Table 2-2.
17H	23	IA32_PLATFORM_ID	Shared	**Platform ID (R)** See Table 2-2.

Table 2-4. MSRs in 45 nm and 32 nm Intel® Atom™ Processor Family (Contd.)

Register Address		Register Name	Shared/ Unique	Bit Description
Hex	Dec			
17H	23	MSR_PLATFORM_ID	Shared	**Model Specific Platform ID (R)**
		7:0		Reserved.
		12:8		**Maximum Qualified Ratio (R)** The maximum allowed bus ratio.
		63:13		Reserved.
1BH	27	IA32_APIC_BASE	Unique	See Section 10.4.4, "Local APIC Status and Location," and Table 2-2.
2AH	42	MSR_EBL_CR_POWERON	Shared	**Processor Hard Power-On Configuration (R/W)** Enables and disables processor features; **(R)** indicates current processor configuration.
		0		Reserved.
		1		**Data Error Checking Enable (R/W)** 1 = Enabled; 0 = Disabled Always 0.
		2		**Response Error Checking Enable (R/W)** 1 = Enabled; 0 = Disabled Always 0.
		3		**AERR# Drive Enable (R/W)** 1 = Enabled; 0 = Disabled Always 0.
		4		**BERR# Enable for initiator bus requests (R/W)** 1 = Enabled; 0 = Disabled Always 0.
		5		Reserved.
		6		Reserved.
		7		**BINIT# Driver Enable (R/W)** 1 = Enabled; 0 = Disabled Always 0.
		8		Reserved.
		9		**Execute BIST (R/O)** 1 = Enabled; 0 = Disabled
		10		**AERR# Observation Enabled (R/O)** 1 = Enabled; 0 = Disabled Always 0.
		11		Reserved.
		12		**BINIT# Observation Enabled (R/O)** 1 = Enabled; 0 = Disabled Always 0.
		13		**Reserved.**

Table 2-4. MSRs in 45 nm and 32 nm Intel® Atom™ Processor Family (Contd.)

Register Address		Register Name	Shared/ Unique	Bit Description
Hex	Dec			
		14		**1 MByte Power on Reset Vector (R/O)** 1 = 1 MByte; 0 = 4 GBytes
		15		Reserved
		17:16		**APIC Cluster ID (R/O)** Always 00B.
		19: 18		Reserved.
		21: 20		**Symmetric Arbitration ID (R/O)** Always 00B.
		26:22		**Integer Bus Frequency Ratio (R/O)**
3AH	58	IA32_FEATURE_CONTROL	Unique	**Control Features in Intel 64Processor (R/W)** See Table 2-2.
40H	64	MSR_ LASTBRANCH_0_FROM_IP	Unique	**Last Branch Record 0 From IP (R/W)** One of eight pairs of last branch record registers on the last branch record stack. The From_IP part of the stack contains pointers to the **source instruction** . See also: ▪ Last Branch Record Stack TOS at 1C9H ▪ Section 17.5
41H	65	MSR_ LASTBRANCH_1_FROM_IP	Unique	**Last Branch Record 1 From IP (R/W)** See description of MSR_LASTBRANCH_0_FROM_IP.
42H	66	MSR_ LASTBRANCH_2_FROM_IP	Unique	**Last Branch Record 2 From IP (R/W)** See description of MSR_LASTBRANCH_0_FROM_IP.
43H	67	MSR_ LASTBRANCH_3_FROM_IP	Unique	**Last Branch Record 3 From IP (R/W)** See description of MSR_LASTBRANCH_0_FROM_IP.
44H	68	MSR_ LASTBRANCH_4_FROM_IP	Unique	**Last Branch Record 4 From IP (R/W)** See description of MSR_LASTBRANCH_0_FROM_IP.
45H	69	MSR_ LASTBRANCH_5_FROM_IP	Unique	**Last Branch Record 5 From IP (R/W)** See description of MSR_LASTBRANCH_0_FROM_IP.
46H	70	MSR_ LASTBRANCH_6_FROM_IP	Unique	**Last Branch Record 6 From IP (R/W)** See description of MSR_LASTBRANCH_0_FROM_IP.
47H	71	MSR_ LASTBRANCH_7_FROM_IP	Unique	**Last Branch Record 7 From IP (R/W)** See description of MSR_LASTBRANCH_0_FROM_IP.
60H	96	MSR_ LASTBRANCH_0_TO_IP	Unique	**Last Branch Record 0 To IP (R/W)** One of eight pairs of last branch record registers on the last branch record stack. The To_IP part of the stack contains pointers to the destination instruction.
61H	97	MSR_ LASTBRANCH_1_TO_IP	Unique	**Last Branch Record 1 To IP (R/W)** See description of MSR_LASTBRANCH_0_TO_IP.
62H	98	MSR_ LASTBRANCH_2_TO_IP	Unique	**Last Branch Record 2 To IP (R/W)** See description of MSR_LASTBRANCH_0_TO_IP.
63H	99	MSR_ LASTBRANCH_3_TO_IP	Unique	**Last Branch Record 3 To IP (R/W)** See description of MSR_LASTBRANCH_0_TO_IP.

Table 2-4. MSRs in 45 nm and 32 nm Intel® Atom™ Processor Family (Contd.)

Register Address		Register Name	Shared/ Unique	Bit Description
Hex	Dec			
64H	100	MSR_ LASTBRANCH_4_TO_IP	Unique	**Last Branch Record 4 To IP (R/W)** See description of MSR_LASTBRANCH_0_TO_IP.
65H	101	MSR_ LASTBRANCH_5_TO_IP	Unique	**Last Branch Record 5 To IP (R/W)** See description of MSR_LASTBRANCH_0_TO_IP.
66H	102	MSR_ LASTBRANCH_6_TO_IP	Unique	**Last Branch Record 6 To IP (R/W)** See description of MSR_LASTBRANCH_0_TO_IP.
67H	103	MSR_ LASTBRANCH_7_TO_IP	Unique	**Last Branch Record 7 To IP (R/W)** See description of MSR_LASTBRANCH_0_TO_IP.
79H	121	IA32_BIOS_UPDT_TRIG	Shared	**BIOS Update Trigger Register (W)** See Table 2-2.
8BH	139	IA32_BIOS_SIGN_ID	Unique	**BIOS Update Signature ID (RO)** See Table 2-2.
C1H	193	IA32_PMC0	Unique	**Performance counter register** See Table 2-2.
C2H	194	IA32_PMC1	Unique	**Performance Counter Register** See Table 2-2.
CDH	205	MSR_FSB_FREQ	Shared	**Scaleable Bus Speed(RO)** This field indicates the intended scaleable bus clock speed for processors based on Intel Atom microarchitecture:
		2:0		▪ 111B: 083 MHz (FSB 333) ▪ 101B: 100 MHz (FSB 400) ▪ 001B: 133 MHz (FSB 533) ▪ 011B: 167 MHz (FSB 667) 133.33 MHz should be utilized if performing calculation with System Bus Speed when encoding is 001B. 166.67 MHz should be utilized if performing calculation with System Bus Speed when encoding is 011B.
		63:3		Reserved.
E7H	231	IA32_MPERF	Unique	**Maximum Performance Frequency Clock Count (RW)** See Table 2-2.
E8H	232	IA32_APERF	Unique	**Actual Performance Frequency Clock Count (RW)** See Table 2-2.
FEH	254	IA32_MTRRCAP	Shared	**Memory Type Range Register (R)** See Table 2-2.
11EH	281	MSR_BBL_CR_CTL3	Shared	Control register 3. Used to configure the L2 Cache.
		0		**L2 Hardware Enabled (RO)** 1 = If the L2 is hardware-enabled 0 = Indicates if the L2 is hardware-disabled
		7:1		Reserved.

Table 2-4. MSRs in 45 nm and 32 nm Intel® Atom™ Processor Family (Contd.)

Register Address		Register Name	Shared/ Unique	Bit Description
Hex	Dec			
		8		**L2 Enabled. (R/W)** 1 = L2 cache has been initialized 0 = Disabled (default) Until this bit is set the processor will not respond to the WBINVD instruction or the assertion of the FLUSH# input.
		22:9		Reserved.
		23		**L2 Not Present (RO)** 0 = L2 Present 1 = L2 Not Present
		63:24		Reserved.
174H	372	IA32_SYSENTER_CS	Unique	See Table 2-2.
175H	373	IA32_SYSENTER_ESP	Unique	See Table 2-2.
176H	374	IA32_SYSENTER_EIP	Unique	See Table 2-2.
179H	377	IA32_MCG_CAP	Unique	See Table 2-2.
17AH	378	IA32_MCG_STATUS	Unique	Global Machine Check Status
		0		**RIPV** When set, bit indicates that the instruction addressed by the instruction pointer pushed on the stack (when the machine check was generated) can be used to restart the program. If cleared, the program cannot be reliably restarted
		1		**EIPV** When set, bit indicates that the instruction addressed by the instruction pointer pushed on the stack (when the machine check was generated) is directly associated with the error.
		2		**MCIP** When set, bit indicates that a machine check has been generated. If a second machine check is detected while this bit is still set, the processor enters a shutdown state. Software should write this bit to 0 after processing a machine check exception.
		63:3		Reserved.
186H	390	IA32_PERFEVTSEL0	Unique	See Table 2-2.
187H	391	IA32_PERFEVTSEL1	Unique	See Table 2-2.
198H	408	IA32_PERF_STATUS	Shared	See Table 2-2.
198H	408	MSR_PERF_STATUS	Shared	Performance Status
		15:0		Current Performance State Value.
		39:16		Reserved.
		44:40		Maximum Bus Ratio (R/O) Indicates maximum bus ratio configured for the processor.
		63:45		Reserved.
199H	409	IA32_PERF_CTL	Unique	See Table 2-2.

Table 2-4. MSRs in 45 nm and 32 nm Intel® Atom™ Processor Family (Contd.)

Register Address		Register Name	Shared/ Unique	Bit Description
Hex	Dec			
19AH	410	IA32_CLOCK_MODULATION	Unique	**Clock Modulation (R/W)** See Table 2-2. IA32_CLOCK_MODULATION MSR was originally named IA32_THERM_CONTROL MSR.
19BH	411	IA32_THERM_INTERRUPT	Unique	**Thermal Interrupt Control (R/W)** See Table 2-2.
19CH	412	IA32_THERM_STATUS	Unique	**Thermal Monitor Status (R/W)** See Table 2-2.
19DH	413	MSR_THERM2_CTL	Shared	Thermal Monitor 2 Control
		15:0		Reserved.
		16		**TM_SELECT (R/W)** Mode of automatic thermal monitor: 0 = Thermal Monitor 1 (thermally-initiated on-die modulation of the stop-clock duty cycle) 1 = Thermal Monitor 2 (thermally-initiated frequency transitions) If bit 3 of the IA32_MISC_ENABLE register is cleared, TM_SELECT has no effect. Neither TM1 nor TM2 are enabled.
		63:17		Reserved.
1A0H	416	IA32_MISC_ENABLE	Unique	**Enable Misc. Processor Features (R/W)** Allows a variety of processor functions to be enabled and disabled.
		0		**Fast-Strings Enable** See Table 2-2.
		2:1		Reserved.
		3	Unique	**Automatic Thermal Control Circuit Enable (R/W)** See Table 2-2. Default value is 0.
		6:4		Reserved.
		7	Shared	**Performance Monitoring Available (R)** See Table 2-2.
		8		Reserved.
		9		Reserved.
		10	Shared	**FERR# Multiplexing Enable (R/W)** 1 = FERR# asserted by the processor to indicate a pending break event within the processor 0 = Indicates compatible FERR# signaling behavior This bit must be set to 1 to support XAPIC interrupt model usage.
		11	Shared	**Branch Trace Storage Unavailable (RO)** See Table 2-2.
		12	Shared	**Processor Event Based Sampling Unavailable (RO)** See Table 2-2.

Table 2-4. MSRs in 45 nm and 32 nm Intel® Atom™ Processor Family (Contd.)

Register Address		Register Name	Shared/ Unique	Bit Description
Hex	Dec			
		13	Shared	**TM2 Enable (R/W)** When this bit is set (1) and the thermal sensor indicates that the die temperature is at the pre-determined threshold, the Thermal Monitor 2 mechanism is engaged. TM2 will reduce the bus to core ratio and voltage according to the value last written to MSR_THERM2_CTL bits 15:0.
				When this bit is clear (0, default), the processor does not change the VID signals or the bus to core ratio when the processor enters a thermally managed state. The BIOS must enable this feature if the TM2 feature flag (CPUID.1:ECX[8]) is set; if the TM2 feature flag is not set, this feature is not supported and BIOS must not alter the contents of the TM2 bit location. The processor is operating out of specification if both this bit and the TM1 bit are set to 0.
		15:14		Reserved.
		16	Shared	**Enhanced Intel SpeedStep Technology Enable (R/W)** See Table 2-2.
		18	Shared	**ENABLE MONITOR FSM (R/W)** See Table 2-2.
		19		Reserved.
		20	Shared	**Enhanced Intel SpeedStep Technology Select Lock (R/WO)** When set, this bit causes the following bits to become read-only: ▪ Enhanced Intel SpeedStep Technology Select Lock (this bit), ▪ Enhanced Intel SpeedStep Technology Enable bit. The bit must be set before an Enhanced Intel SpeedStep Technology transition is requested. This bit is cleared on reset.
		21		Reserved.
		22	Unique	**Limit CPUID Maxval (R/W)** See Table 2-2.
		23	Shared	**xTPR Message Disable (R/W)** See Table 2-2.
		33:24		Reserved.
		34	Unique	**XD Bit Disable (R/W)** See Table 2-2.
		63:35		Reserved.
1C9H	457	MSR_LASTBRANCH_TOS	Unique	**Last Branch Record Stack TOS (R/W)** Contains an index (bits 0-2) that points to the MSR containing the most recent branch record. See MSR_LASTBRANCH_0_FROM_IP (at 40H).
1D9H	473	IA32_DEBUGCTL	Unique	**Debug Control (R/W)** See Table 2-2.

Table 2-4. MSRs in 45 nm and 32 nm Intel® Atom™ Processor Family (Contd.)

Register Address		Register Name	Shared/ Unique	Bit Description
Hex	Dec			
1DDH	477	MSR_LER_FROM_LIP	Unique	**Last Exception Record From Linear IP (R)** Contains a pointer to the last branch instruction that the processor executed prior to the last exception that was generated or the last interrupt that was handled.
1DEH	478	MSR_LER_TO_LIP	Unique	**Last Exception Record To Linear IP (R)** This area contains a pointer to the target of the last branch instruction that the processor executed prior to the last exception that was generated or the last interrupt that was handled.
200H	512	IA32_MTRR_PHYSBASE0	Shared	See Table 2-2.
201H	513	IA32_MTRR_PHYSMASK0	Shared	See Table 2-2.
202H	514	IA32_MTRR_PHYSBASE1	Shared	See Table 2-2.
203H	515	IA32_MTRR_PHYSMASK1	Shared	See Table 2-2.
204H	516	IA32_MTRR_PHYSBASE2	Shared	See Table 2-2.
205H	517	IA32_MTRR_PHYSMASK2	Shared	See Table 2-2.
206H	518	IA32_MTRR_PHYSBASE3	Shared	See Table 2-2.
207H	519	IA32_MTRR_PHYSMASK3	Shared	See Table 2-2.
208H	520	IA32_MTRR_PHYSBASE4	Shared	See Table 2-2.
209H	521	IA32_MTRR_PHYSMASK4	Shared	See Table 2-2.
20AH	522	IA32_MTRR_PHYSBASE5	Shared	See Table 2-2.
20BH	523	IA32_MTRR_PHYSMASK5	Shared	See Table 2-2.
20CH	524	IA32_MTRR_PHYSBASE6	Shared	See Table 2-2.
20DH	525	IA32_MTRR_PHYSMASK6	Shared	See Table 2-2.
20EH	526	IA32_MTRR_PHYSBASE7	Shared	See Table 2-2.
20FH	527	IA32_MTRR_PHYSMASK7	Shared	See Table 2-2.
250H	592	IA32_MTRR_FIX64K_ 00000	Shared	See Table 2-2.
258H	600	IA32_MTRR_FIX16K_ 80000	Shared	See Table 2-2.
259H	601	IA32_MTRR_FIX16K_ A0000	Shared	See Table 2-2.
268H	616	IA32_MTRR_FIX4K_C0000	Shared	See Table 2-2.
269H	617	IA32_MTRR_FIX4K_C8000	Shared	See Table 2-2.
26AH	618	IA32_MTRR_FIX4K_D0000	Shared	See Table 2-2.
26BH	619	IA32_MTRR_FIX4K_D8000	Shared	See Table 2-2.
26CH	620	IA32_MTRR_FIX4K_E0000	Shared	See Table 2-2.
26DH	621	IA32_MTRR_FIX4K_E8000	Shared	See Table 2-2.
26EH	622	IA32_MTRR_FIX4K_F0000	Shared	See Table 2-2.
26FH	623	IA32_MTRR_FIX4K_F8000	Shared	See Table 2-2.
277H	631	IA32_PAT	Unique	See Table 2-2.

Table 2-4. MSRs in 45 nm and 32 nm Intel® Atom™ Processor Family (Contd.)

Register Address		Register Name	Shared/ Unique	Bit Description
Hex	Dec			
309H	777	IA32_FIXED_CTR0	Unique	**Fixed-Function Performance Counter Register 0 (R/W)** See Table 2-2.
30AH	778	IA32_FIXED_CTR1	Unique	**Fixed-Function Performance Counter Register 1 (R/W)** See Table 2-2.
30BH	779	IA32_FIXED_CTR2	Unique	**Fixed-Function Performance Counter Register 2 (R/W)** See Table 2-2.
345H	837	IA32_PERF_CAPABILITIES	Shared	See Table 2-2. See Section 17.4.1, "IA32_DEBUGCTL MSR."
38DH	909	IA32_FIXED_CTR_CTRL	Unique	**Fixed-Function-Counter Control Register (R/W)** See Table 2-2.
38EH	910	IA32_PERF_GLOBAL_ STATUS	Unique	See Table 2-2. See Section 18.6.2.2, "Global Counter Control Facilities."
38FH	911	IA32_PERF_GLOBAL_CTRL	Unique	See Table 2-2. See Section 18.6.2.2, "Global Counter Control Facilities."
390H	912	IA32_PERF_GLOBAL_OVF_ CTRL	Unique	See Table 2-2. See Section 18.6.2.2, "Global Counter Control Facilities."
3F1H	1009	MSR_PEBS_ENABLE	Unique	See Table 2-2. See Section 18.6.2.4, "Processor Event Based Sampling (PEBS)."
		0		Enable PEBS on IA32_PMC0. (R/W)
400H	1024	IA32_MC0_CTL	Shared	See Section 15.3.2.1, "IA32_MCi_CTL MSRs."
401H	1025	IA32_MC0_STATUS	Shared	See Section 15.3.2.2, "IA32_MCi_STATUS MSRS."
402H	1026	IA32_MC0_ADDR	Shared	See Section 15.3.2.3, "IA32_MCi_ADDR MSRs." The IA32_MC0_ADDR register is either not implemented or contains no address if the ADDRV flag in the IA32_MC0_STATUS register is clear. When not implemented in the processor, all reads and writes to this MSR will cause a general-protection exception.
404H	1028	IA32_MC1_CTL	Shared	See Section 15.3.2.1, "IA32_MCi_CTL MSRs."
405H	1029	IA32_MC1_STATUS	Shared	See Section 15.3.2.2, "IA32_MCi_STATUS MSRS."
408H	1032	IA32_MC2_CTL	Shared	See Section 15.3.2.1, "IA32_MCi_CTL MSRs."
409H	1033	IA32_MC2_STATUS	Shared	See Section 15.3.2.2, "IA32_MCi_STATUS MSRS."
40AH	1034	IA32_MC2_ADDR	Shared	See Section 15.3.2.3, "IA32_MCi_ADDR MSRs." The IA32_MC2_ADDR register is either not implemented or contains no address if the ADDRV flag in the IA32_MC2_STATUS register is clear. When not implemented in the processor, all reads and writes to this MSR will cause a general-protection exception.
40CH	1036	IA32_MC3_CTL	Shared	See Section 15.3.2.1, "IA32_MCi_CTL MSRs."
40DH	1037	IA32_MC3_STATUS	Shared	See Section 15.3.2.2, "IA32_MCi_STATUS MSRS."

Table 2-4. MSRs in 45 nm and 32 nm Intel® Atom™ Processor Family (Contd.)

Register Address		Register Name	Shared/ Unique	Bit Description
Hex	Dec			
40EH	1038	IA32_MC3_ADDR	Shared	See Section 15.3.2.3, "IA32_MCi_ADDR MSRs." The MSR_MC3_ADDR register is either not implemented or contains no address if the ADDRV flag in the MSR_MC3_STATUS register is clear. When not implemented in the processor, all reads and writes to this MSR will cause a general-protection exception.
410H	1040	IA32_MC4_CTL	Shared	See Section 15.3.2.1, "IA32_MCi_CTL MSRs."
411H	1041	IA32_MC4_STATUS	Shared	See Section 15.3.2.2, "IA32_MCi_STATUS MSRS."
412H	1042	IA32_MC4_ADDR	Shared	See Section 15.3.2.3, "IA32_MCi_ADDR MSRs." The MSR_MC4_ADDR register is either not implemented or contains no address if the ADDRV flag in the MSR_MC4_STATUS register is clear. When not implemented in the processor, all reads and writes to this MSR will cause a general-protection exception.
480H	1152	IA32_VMX_BASIC	Unique	**Reporting Register of Basic VMX Capabilities (R/O)** See Table 2-2. See Appendix A.1, "Basic VMX Information."
481H	1153	IA32_VMX_PINBASED_ CTLS	Unique	**Capability Reporting Register of Pin-based VM-execution Controls (R/O)** See Table 2-2. See Appendix A.3, "VM-Execution Controls."
482H	1154	IA32_VMX_PROCBASED_ CTLS	Unique	**Capability Reporting Register of Primary Processor-based VM-execution Controls (R/O)** See Appendix A.3, "VM-Execution Controls."
483H	1155	IA32_VMX_EXIT_CTLS	Unique	**Capability Reporting Register of VM-exit Controls (R/O)** See Table 2-2. See Appendix A.4, "VM-Exit Controls."
484H	1156	IA32_VMX_ENTRY_CTLS	Unique	**Capability Reporting Register of VM-entry Controls (R/O)** See Table 2-2. See Appendix A.5, "VM-Entry Controls."
485H	1157	IA32_VMX_MISC	Unique	**Reporting Register of Miscellaneous VMX Capabilities (R/O)** See Table 2-2. See Appendix A.6, "Miscellaneous Data."
486H	1158	IA32_VMX_CR0_FIXED0	Unique	**Capability Reporting Register of CR0 Bits Fixed to 0 (R/O)** See Table 2-2. See Appendix A.7, "VMX-Fixed Bits in CR0."
487H	1159	IA32_VMX_CR0_FIXED1	Unique	**Capability Reporting Register of CR0 Bits Fixed to 1 (R/O)** See Table 2-2. See Appendix A.7, "VMX-Fixed Bits in CR0."
488H	1160	IA32_VMX_CR4_FIXED0	Unique	**Capability Reporting Register of CR4 Bits Fixed to 0 (R/O)** See Table 2-2. See Appendix A.8, "VMX-Fixed Bits in CR4."

Table 2-4. MSRs in 45 nm and 32 nm Intel® Atom™ Processor Family (Contd.)

Register Address		Register Name	Shared/ Unique	Bit Description
Hex	Dec			
489H	1161	IA32_VMX_CR4_FIXED1	Unique	**Capability Reporting Register of CR4 Bits Fixed to 1 (R/O)** See Table 2-2. See Appendix A.8, "VMX-Fixed Bits in CR4."
48AH	1162	IA32_VMX_VMCS_ENUM	Unique	**Capability Reporting Register of VMCS Field Enumeration (R/O)** See Table 2-2. See Appendix A.9, "VMCS Enumeration."
48BH	1163	IA32_VMX_PROCBASED_ CTLS2	Unique	**Capability Reporting Register of Secondary Processor-based VM-execution Controls (R/O)** See Appendix A.3, "VM-Execution Controls."
600H	1536	IA32_DS_AREA	Unique	**DS Save Area (R/W)** See Table 2-2. See Section 18.6.3.4, "Debug Store (DS) Mechanism."
C000_ 0080H		IA32_EFER	Unique	**Extended Feature Enables** See Table 2-2.
C000_ 0081H		IA32_STAR	Unique	**System Call Target Address (R/W)** See Table 2-2.
C000_ 0082H		IA32_LSTAR	Unique	**IA-32e Mode System Call Target Address (R/W)** See Table 2-2.
C000_ 0084H		IA32_FMASK	Unique	**System Call Flag Mask (R/W)** See Table 2-2.
C000_ 0100H		IA32_FS_BASE	Unique	**Map of BASE Address of FS (R/W)** See Table 2-2.
C000_ 0101H		IA32_GS_BASE	Unique	**Map of BASE Address of GS (R/W)** See Table 2-2.
C000_ 0102H		IA32_KERNEL_GS_BASE	Unique	**Swap Target of BASE Address of GS (R/W)** See Table 2-2.

Table 2-5 lists model-specific registers (MSRs) that are specific to Intel® Atom™ processor with the CPUID signature with DisplayFamily_DisplayModel of 06_27H.

Table 2-5. MSRs Supported by Intel® Atom™ Processors with CPUID Signature 06_27H

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
3F8H	1016	MSR_PKG_C2_RESIDENCY	Package	**Package C2 Residency** Note: C-state values are processor specific C-state code names, unrelated to MWAIT extension C-state parameters or ACPI C-States
		63:0	Package	Package C2 Residency Counter. (R/O) Time that this package is in processor-specific C2 states since last reset. Counts at 1 Mhz frequency.

Table 2-5. MSRs Supported by Intel® Atom™ Processors (Contd.)with CPUID Signature 06_27H

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
3F9H	1017	MSR_PKG_C4_RESIDENCY	Package	**Package C4 Residency** Note: C-state values are processor specific C-state code names, unrelated to MWAIT extension C-state parameters or ACPI C-States
		63:0	Package	Package C4 Residency Counter. (R/O) Time that this package is in processor-specific C4 states since last reset. Counts at 1 Mhz frequency.
3FAH	1018	MSR_PKG_C6_RESIDENCY	Package	**Package C6 Residency** Note: C-state values are processor specific C-state code names, unrelated to MWAIT extension C-state parameters or ACPI C-States
		63:0	Package	Package C6 Residency Counter. (R/O) Time that this package is in processor-specific C6 states since last reset. Counts at 1 Mhz frequency.

2.4 MSRS IN INTEL PROCESSORS BASED ON SILVERMONT MICROARCHITECTURE

Table 2-6 lists model-specific registers (MSRs) common to Intel processors based on the Silvermont microarchitecture. These processors have a CPUID signature with DisplayFamily_DisplayModel of 06_37H, 06_4AH, 06_4DH, 06_5AH, and 06_5DH; see Table 2-1. The MSRs listed in Table 2-6 are also common to processors based on the Airmont microarchitecture and newer microarchitectures for next generation Intel Atom processors.

Table 2-7 lists MSRs common to processors based on the Silvermont and Airmont microarchitectures, but not newer microarchitectures.

Table 2-8, Table 2-9, and Table 2-10 lists MSRs that are model-specific across processors based on the Silvermont microarchitecture.

In the Silvermont microarchitecture, the scope column indicates the following: "Core" means each processor core has a separate MSR, or a bit field not shared with another processor core. "Module" means the MSR or the bit field is shared by a pair of processor cores in the physical package. "Package" means all processor cores in the physical package share the same MSR or bit interface.

Table 2-6. MSRs Common to the Silvermont Microarchitecture and Newer Microarchitectures for Intel Atom Processors

Address		Register Name	Scope	Bit Description
Hex	Dec			
0H	0	IA32_P5_MC_ADDR	Module	See Section 2.22, "MSRs in Pentium Processors."
1H	1	IA32_P5_MC_TYPE	Module	See Section 2.22, "MSRs in Pentium Processors."
6H	6	IA32_MONITOR_FILTER_SIZE	Core	See Section 8.10.5, "Monitor/Mwait Address Range Determination." andTable 2-2
10H	16	IA32_TIME_STAMP_COUNTER	Core	See Section 17.17, "Time-Stamp Counter," and see Table 2-2.
1BH	27	IA32_APIC_BASE	Core	See Section 10.4.4, "Local APIC Status and Location," and Table 2-2.
2AH	42	MSR_EBL_CR_POWERON	Module	**Processor Hard Power-On Configuration (R/W)** Writes ignored.
		63:0		Reserved **(R/O)**

Table 2-6. MSRs Common to the Silvermont Microarchitecture and Newer Microarchitectures for Intel Atom Processors

Address		Register Name	Scope	Bit Description
Hex	Dec			
34H	52	MSR_SMI_COUNT	Core	**SMI Counter (R/O)**
		31:0		**SMI Count (R/O)** Running count of SMI events since last RESET.
		63:32		Reserved.
79H	121	IA32_BIOS_UPDT_TRIG	Core	**BIOS Update Trigger Register (W)** See Table 2-2.
8BH	139	IA32_BIOS_SIGN_ID	Core	**BIOS Update Signature ID (RO)** See Table 2-2.
C1H	193	IA32_PMC0	Core	**Performance counter register** See Table 2-2.
C2H	194	IA32_PMC1	Core	**Performance Counter Register** See Table 2-2.
E4H	228	MSR_PMG_IO_CAPTURE_BASE	Module	**Power Management IO Redirection in C-state (R/W)** See http://biosbits.org.
		15:0		**LVL_2 Base Address (R/W)** Specifies the base address visible to software for IO redirection. If IO MWAIT Redirection is enabled, reads to this address will be consumed by the power management logic and decoded to MWAIT instructions. When IO port address redirection is enabled, this is the IO port address reported to the OS/software.
		18:16		**C-state Range (R/W)** Specifies the encoding value of the maximum C-State code name to be included when IO read to MWAIT redirection is enabled by MSR_PKG_CST_CONFIG_CONTROL[bit10]: 100b - C4 is the max C-State to include 110b - C6 is the max C-State to include 111b - C7 is the max C-State to include
		63:19		Reserved.
E7H	231	IA32_MPERF	Core	**Maximum Performance Frequency Clock Count (RW)** See Table 2-2.
E8H	232	IA32_APERF	Core	**Actual Performance Frequency Clock Count (RW)** See Table 2-2.
FEH	254	IA32_MTRRCAP	Core	**Memory Type Range Register (R)** See Table 2-2.
13CH	52	MSR_FEATURE_CONFIG	Core	**AES Configuration (RW-L)** Privileged post-BIOS agent must provide a #GP handler to handle unsuccessful read of this MSR.

Table 2-6. MSRs Common to the Silvermont Microarchitecture and Newer Microarchitectures for Intel Atom Processors

Address		Register Name	Scope	Bit Description
Hex	Dec			
		1:0		**AES Configuration (RW-L)** Upon a successful read of this MSR, the configuration of AES instruction set availability is as follows: 11b: AES instructions are not available until next RESET. otherwise, AES instructions are available. Note, AES instruction set is not available if read is unsuccessful. If the configuration is not 01b, AES instruction can be mis-configured if a privileged agent unintentionally writes 11b.
		63:2		Reserved.
174H	372	IA32_SYSENTER_CS	Core	See Table 2-2.
175H	373	IA32_SYSENTER_ESP	Core	See Table 2-2.
176H	374	IA32_SYSENTER_EIP	Core	See Table 2-2.
179H	377	IA32_MCG_CAP	Core	See Table 2-2.
17AH	378	IA32_MCG_STATUS	Core	Global Machine Check Status
		0		**RIPV** When set, bit indicates that the instruction addressed by the instruction pointer pushed on the stack (when the machine check was generated) can be used to restart the program. If cleared, the program cannot be reliably restarted
		1		**EIPV** When set, bit indicates that the instruction addressed by the instruction pointer pushed on the stack (when the machine check was generated) is directly associated with the error.
		2		**MCIP** When set, bit indicates that a machine check has been generated. If a second machine check is detected while this bit is still set, the processor enters a shutdown state. Software should write this bit to 0 after processing a machine check exception.
		63:3		Reserved.
186H	390	IA32_PERFEVTSEL0	Core	See Table 2-2.
		7:0		**Event Select**
		15:8		**UMask**
		16		**USR**
		17		**OS**
		18		**Edge**
		19		**PC**
		20		**INT**
		21		**Reserved**
		22		**EN**
		23		**INV**
		31:24		**CMASK**

Table 2-6. MSRs Common to the Silvermont Microarchitecture and Newer Microarchitectures for Intel Atom Processors

Address		Register Name	Scope	Bit Description
Hex	Dec			
		63:32		Reserved.
187H	391	IA32_PERFEVTSEL1	Core	See Table 2-2.
198H	408	IA32_PERF_STATUS	Module	See Table 2-2.
199H	409	IA32_PERF_CTL	Core	See Table 2-2.
19AH	410	IA32_CLOCK_MODULATION	Core	**Clock Modulation (R/W)** See Table 2-2. IA32_CLOCK_MODULATION MSR was originally named IA32_THERM_CONTROL MSR.
19BH	411	IA32_THERM_INTERRUPT	Core	**Thermal Interrupt Control (R/W)** See Table 2-2.
19CH	412	IA32_THERM_STATUS	Core	**Thermal Monitor Status (R/W)** See Table 2-2.
1A2H	418	MSR_ TEMPERATURE_TARGET	Package	Temperature Target
		15:0		Reserved.
		23:16		**Temperature Target (R)** The default thermal throttling or PROCHOT# activation temperature in degree C, The effective temperature for thermal throttling or PROCHOT# activation is “Temperature Target” + “Target Offset”
		29:24		**Target Offset (R/W)** Specifies an offset in degrees C to adjust the throttling and PROCHOT# activation temperature from the default target specified in TEMPERATURE_TARGET (bits 23:16).
		63:30		Reserved.
1A6H	422	MSR_OFFCORE_RSP_0	Module	**Offcore Response Event Select Register (R/W)**
1A7H	423	MSR_OFFCORE_RSP_1	Module	**Offcore Response Event Select Register (R/W)**
1B0H	432	IA32_ENERGY_PERF_BIAS	Core	See Table 2-2.
1D9H	473	IA32_DEBUGCTL	Core	**Debug Control (R/W)** See Table 2-2.
1DDH	477	MSR_LER_FROM_LIP	Core	**Last Exception Record From Linear IP (R)** Contains a pointer to the last branch instruction that the processor executed prior to the last exception that was generated or the last interrupt that was handled.
1DEH	478	MSR_LER_TO_LIP	Core	**Last Exception Record To Linear IP (R)** This area contains a pointer to the target of the last branch instruction that the processor executed prior to the last exception that was generated or the last interrupt that was handled.
1F2H	498	IA32_SMRR_PHYSBASE	Core	See Table 2-2.
1F3H	499	IA32_SMRR_PHYSMASK	Core	See Table 2-2.
200H	512	IA32_MTRR_PHYSBASE0	Core	See Table 2-2.

Table 2-6. MSRs Common to the Silvermont Microarchitecture and Newer Microarchitectures for Intel Atom Processors

Address		Register Name	Scope	Bit Description
Hex	Dec			
201H	513	IA32_MTRR_PHYSMASK0	Core	See Table 2-2.
202H	514	IA32_MTRR_PHYSBASE1	Core	See Table 2-2.
203H	515	IA32_MTRR_PHYSMASK1	Core	See Table 2-2.
204H	516	IA32_MTRR_PHYSBASE2	Core	See Table 2-2.
205H	517	IA32_MTRR_PHYSMASK2	Core	See Table 2-2.
206H	518	IA32_MTRR_PHYSBASE3	Core	See Table 2-2.
207H	519	IA32_MTRR_PHYSMASK3	Core	See Table 2-2.
208H	520	IA32_MTRR_PHYSBASE4	Core	See Table 2-2.
209H	521	IA32_MTRR_PHYSMASK4	Core	See Table 2-2.
20AH	522	IA32_MTRR_PHYSBASE5	Core	See Table 2-2.
20BH	523	IA32_MTRR_PHYSMASK5	Core	See Table 2-2.
20CH	524	IA32_MTRR_PHYSBASE6	Core	See Table 2-2.
20DH	525	IA32_MTRR_PHYSMASK6	Core	See Table 2-2.
20EH	526	IA32_MTRR_PHYSBASE7	Core	See Table 2-2.
20FH	527	IA32_MTRR_PHYSMASK7	Core	See Table 2-2.
250H	592	IA32_MTRR_FIX64K_ 00000	Core	See Table 2-2.
258H	600	IA32_MTRR_FIX16K_ 80000	Core	See Table 2-2.
259H	601	IA32_MTRR_FIX16K_ A0000	Core	See Table 2-2.
268H	616	IA32_MTRR_FIX4K_C0000	Core	See Table 2-2.
269H	617	IA32_MTRR_FIX4K_C8000	Core	See Table 2-2.
26AH	618	IA32_MTRR_FIX4K_D0000	Core	See Table 2-2.
26BH	619	IA32_MTRR_FIX4K_D8000	Core	See Table 2-2.
26CH	620	IA32_MTRR_FIX4K_E0000	Core	See Table 2-2.
26DH	621	IA32_MTRR_FIX4K_E8000	Core	See Table 2-2.
26EH	622	IA32_MTRR_FIX4K_F0000	Core	See Table 2-2.
26FH	623	IA32_MTRR_FIX4K_F8000	Core	See Table 2-2.
277H	631	IA32_PAT	Core	See Table 2-2.
2FFH	767	IA32_MTRR_DEF_TYPE	Core	**Default Memory Types (R/W)** See Table 2-2.
309H	777	IA32_FIXED_CTR0	Core	**Fixed-Function Performance Counter Register 0 (R/W)** See Table 2-2.
30AH	778	IA32_FIXED_CTR1	Core	**Fixed-Function Performance Counter Register 1 (R/W)** See Table 2-2.
30BH	779	IA32_FIXED_CTR2	Core	**Fixed-Function Performance Counter Register 2 (R/W)** See Table 2-2.

Table 2-6. MSRs Common to the Silvermont Microarchitecture and Newer Microarchitectures for Intel Atom Processors

Address		Register Name	Scope	Bit Description
Hex	Dec			
345H	837	IA32_PERF_CAPABILITIES	Core	See Table 2-2. See Section 17.4.1, "IA32_DEBUGCTL MSR."
38DH	909	IA32_FIXED_CTR_CTRL	Core	**Fixed-Function-Counter Control Register (R/W)** See Table 2-2.
38FH	911	IA32_PERF_GLOBAL_CTRL	Core	See Table 2-2. See Section 18.6.2.2, "Global Counter Control Facilities."
3FDH	1021	MSR_CORE_C6_RESIDENCY	Core	Note: C-state values are processor specific C-state code names, unrelated to MWAIT extension C-state parameters or ACPI C-States.
		63:0		CORE C6 Residency Counter. (R/O) Value since last reset that this core is in processor-specific C6 states. Counts at the TSC Frequency.
400H	1024	IA32_MC0_CTL	Module	See Section 15.3.2.1, "IA32_MCi_CTL MSRs."
401H	1025	IA32_MC0_STATUS	Module	See Section 15.3.2.2, "IA32_MCi_STATUS MSRS."
402H	1026	IA32_MC0_ADDR	Module	See Section 15.3.2.3, "IA32_MCi_ADDR MSRs." The IA32_MC0_ADDR register is either not implemented or contains no address if the ADDRV flag in the IA32_MC0_STATUS register is clear. When not implemented in the processor, all reads and writes to this MSR will cause a general-protection exception.
404H	1028	IA32_MC1_CTL	Module	See Section 15.3.2.1, "IA32_MCi_CTL MSRs."
405H	1029	IA32_MC1_STATUS	Module	See Section 15.3.2.2, "IA32_MCi_STATUS MSRS."
408H	1032	IA32_MC2_CTL	Module	See Section 15.3.2.1, "IA32_MCi_CTL MSRs."
409H	1033	IA32_MC2_STATUS	Module	See Section 15.3.2.2, "IA32_MCi_STATUS MSRS."
40AH	1034	IA32_MC2_ADDR	Module	See Section 15.3.2.3, "IA32_MCi_ADDR MSRs." The IA32_MC2_ADDR register is either not implemented or contains no address if the ADDRV flag in the IA32_MC2_STATUS register is clear. When not implemented in the processor, all reads and writes to this MSR will cause a general-protection exception.
40CH	1036	IA32_MC3_CTL	Core	See Section 15.3.2.1, "IA32_MCi_CTL MSRs."
40DH	1037	IA32_MC3_STATUS	Core	See Section 15.3.2.2, "IA32_MCi_STATUS MSRS."
40EH	1038	IA32_MC3_ADDR	Core	See Section 15.3.2.3, "IA32_MCi_ADDR MSRs." The MSR_MC3_ADDR register is either not implemented or contains no address if the ADDRV flag in the MSR_MC3_STATUS register is clear. When not implemented in the processor, all reads and writes to this MSR will cause a general-protection exception.
410H	1040	IA32_MC4_CTL	Core	See Section 15.3.2.1, "IA32_MCi_CTL MSRs."
411H	1041	IA32_MC4_STATUS	Core	See Section 15.3.2.2, "IA32_MCi_STATUS MSRS."

Table 2-6. MSRs Common to the Silvermont Microarchitecture and Newer Microarchitectures for Intel Atom Processors

Address		Register Name	Scope	Bit Description
Hex	Dec			
412H	1042	IA32_MC4_ADDR	Core	See Section 15.3.2.3, "IA32_MCi_ADDR MSRs." The MSR_MC4_ADDR register is either not implemented or contains no address if the ADDRV flag in the MSR_MC4_STATUS register is clear. When not implemented in the processor, all reads and writes to this MSR will cause a general-protection exception.
414H	1044	IA32_MC5_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs."
415H	1045	IA32_MC5_STATUS	Package	See Section 15.3.2.2, "IA32_MCi_STATUS MSRS."
416H	1046	IA32_MC5_ADDR	Package	See Section 15.3.2.3, "IA32_MCi_ADDR MSRs." The MSR_MC4_ADDR register is either not implemented or contains no address if the ADDRV flag in the MSR_MC4_STATUS register is clear. When not implemented in the processor, all reads and writes to this MSR will cause a general-protection exception.
480H	1152	IA32_VMX_BASIC	Core	**Reporting Register of Basic VMX Capabilities (R/O)** See Table 2-2. See Appendix A.1, "Basic VMX Information."
481H	1153	IA32_VMX_PINBASED_CTLS	Core	**Capability Reporting Register of Pin-based VM-execution Controls (R/O)** See Table 2-2. See Appendix A.3, "VM-Execution Controls."
482H	1154	IA32_VMX_PROCBASED_CTLS	Core	**Capability Reporting Register of Primary Processor-based VM-execution Controls (R/O)** See Appendix A.3, "VM-Execution Controls."
483H	1155	IA32_VMX_EXIT_CTLS	Core	**Capability Reporting Register of VM-exit Controls (R/O)** See Table 2-2. See Appendix A.4, "VM-Exit Controls."
484H	1156	IA32_VMX_ENTRY_CTLS	Core	**Capability Reporting Register of VM-entry Controls (R/O)** See Table 2-2. See Appendix A.5, "VM-Entry Controls."
485H	1157	IA32_VMX_MISC	Core	**Reporting Register of Miscellaneous VMX Capabilities (R/O)** See Table 2-2. See Appendix A.6, "Miscellaneous Data."
486H	1158	IA32_VMX_CR0_FIXED0	Core	**Capability Reporting Register of CR0 Bits Fixed to 0 (R/O)** See Table 2-2. See Appendix A.7, "VMX-Fixed Bits in CR0."
487H	1159	IA32_VMX_CR0_FIXED1	Core	**Capability Reporting Register of CR0 Bits Fixed to 1 (R/O)** See Table 2-2. See Appendix A.7, "VMX-Fixed Bits in CR0."
488H	1160	IA32_VMX_CR4_FIXED0	Core	**Capability Reporting Register of CR4 Bits Fixed to 0 (R/O)** See Table 2-2. See Appendix A.8, "VMX-Fixed Bits in CR4."

Table 2-6. MSRs Common to the Silvermont Microarchitecture and Newer Microarchitectures for Intel Atom Processors

Address		Register Name	Scope	Bit Description
Hex	Dec			
489H	1161	IA32_VMX_CR4_FIXED1	Core	**Capability Reporting Register of CR4 Bits Fixed to 1 (R/O)** See Table 2-2. See Appendix A.8, "VMX-Fixed Bits in CR4."
48AH	1162	IA32_VMX_VMCS_ENUM	Core	**Capability Reporting Register of VMCS Field Enumeration (R/O)** See Table 2-2. See Appendix A.9, "VMCS Enumeration."
48BH	1163	IA32_VMX_PROCBASED_CTLS2	Core	**Capability Reporting Register of Secondary Processor-based VM-execution Controls (R/O)** See Appendix A.3, "VM-Execution Controls."
48CH	1164	IA32_VMX_EPT_VPID_ENUM	Core	**Capability Reporting Register of EPT and VPID (R/O)** See Table 2-2
48DH	1165	IA32_VMX_TRUE_PINBASED_CTLS	Core	**Capability Reporting Register of Pin-based VM-execution Flex Controls (R/O)** See Table 2-2
48EH	1166	IA32_VMX_TRUE_PROCBASED_CTLS	Core	**Capability Reporting Register of Primary Processor-based VM-execution Flex Controls (R/O)** See Table 2-2
48FH	1167	IA32_VMX_TRUE_EXIT_CTLS	Core	**Capability Reporting Register of VM-exit Flex Controls (R/O)** See Table 2-2
490H	1168	IA32_VMX_TRUE_ENTRY_CTLS	Core	**Capability Reporting Register of VM-entry Flex Controls (R/O)** See Table 2-2
491H	1169	IA32_VMX_FMFUNC	Core	**Capability Reporting Register of VM-function Controls (R/O)** See Table 2-2
4C1H	1217	IA32_A_PMC0	Core	See Table 2-2.
4C2H	1218	IA32_A_PMC1	Core	See Table 2-2.
600H	1536	IA32_DS_AREA	Core	**DS Save Area (R/W)** See Table 2-2. See Section 18.6.3.4, "Debug Store (DS) Mechanism."
660H	1632	MSR_CORE_C1_RESIDENCY	Core	Note: C-state values are processor specific C-state code names, unrelated to MWAIT extension C-state parameters or ACPI C-States.
		63:0		CORE C1 Residency Counter. (R/O) Value since last reset that this core is in processor-specific C1 states. Counts at the TSC frequency.
6E0H	1760	IA32_TSC_DEADLINE	Core	**TSC Target of Local APIC's TSC Deadline Mode (R/W)** See Table 2-2
C000_0080H		IA32_EFER	Core	**Extended Feature Enables** See Table 2-2.
C000_0081H		IA32_STAR	Core	**System Call Target Address (R/W)** See Table 2-2.

Table 2-6. MSRs Common to the Silvermont Microarchitecture and Newer Microarchitectures for Intel Atom Processors

Address		Register Name	Scope	Bit Description
Hex	Dec			
C000_0082H		IA32_LSTAR	Core	**IA-32e Mode System Call Target Address (R/W)** See Table 2-2.
C000_0084H		IA32_FMASK	Core	**System Call Flag Mask (R/W)** See Table 2-2.
C000_0100H		IA32_FS_BASE	Core	**Map of BASE Address of FS (R/W)** See Table 2-2.
C000_0101H		IA32_GS_BASE	Core	**Map of BASE Address of GS (R/W)** See Table 2-2.
C000_0102H		IA32_KERNEL_GS_BASE	Core	**Swap Target of BASE Address of GS (R/W)** See Table 2-2.
C000_0103H		IA32_TSC_AUX	Core	**AUXILIARY TSC Signature. (R/W)** See Table 2-2

Table 2-7 lists model-specific registers (MSRs) that are common to Intel® Atom™ processors based on the Silvermont and Airmont microarchitectures but not newer microarchitectures.

Table 2-7. MSRs Common to the Silvermont and Airmont Microarchitectures

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
17H	23	MSR_PLATFORM_ID	Module	**Model Specific Platform ID (R)**
		7:0		Reserved.
		13:8		**Maximum Qualified Ratio (R)** The maximum allowed bus ratio.
		49:13		Reserved.
		52:50		**See** Table 2-2
		63:33		Reserved.
3AH	58	IA32_FEATURE_CONTROL	Core	**Control Features in Intel 64Processor (R/W)** See Table 2-2.
		0		**Lock (R/WL)**
		1		**Reserved**
		2		**Enable VMX outside SMX operation (R/WL)**
40H	64	MSR_LASTBRANCH_0_FROM_IP	Core	**Last Branch Record 0 From IP (R/W)** One of eight pairs of last branch record registers on the last branch record stack. The From_IP part of the stack contains pointers to the **source instruction**. See also: ▪ Last Branch Record Stack TOS at 1C9H ▪ Section 17.5 and record format in Section 17.4.8.1
41H	65	MSR_LASTBRANCH_1_FROM_IP	Core	**Last Branch Record 1 From IP (R/W)** See description of MSR_LASTBRANCH_0_FROM_IP.

Table 2-7. MSRs Common to the Silvermont and Airmont Microarchitectures

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
42H	66	MSR_ LASTBRANCH_2_FROM_IP	Core	**Last Branch Record 2 From IP (R/W)** See description of MSR_LASTBRANCH_0_FROM_IP.
43H	67	MSR_ LASTBRANCH_3_FROM_IP	Core	**Last Branch Record 3 From IP (R/W)** See description of MSR_LASTBRANCH_0_FROM_IP.
44H	68	MSR_ LASTBRANCH_4_FROM_IP	Core	**Last Branch Record 4 From IP (R/W)** See description of MSR_LASTBRANCH_0_FROM_IP.
45H	69	MSR_ LASTBRANCH_5_FROM_IP	Core	**Last Branch Record 5 From IP (R/W)** See description of MSR_LASTBRANCH_0_FROM_IP.
46H	70	MSR_ LASTBRANCH_6_FROM_IP	Core	**Last Branch Record 6 From IP (R/W)** See description of MSR_LASTBRANCH_0_FROM_IP.
47H	71	MSR_ LASTBRANCH_7_FROM_IP	Core	**Last Branch Record 7 From IP (R/W)** See description of MSR_LASTBRANCH_0_FROM_IP.
60H	96	MSR_ LASTBRANCH_0_TO_IP	Core	**Last Branch Record 0 To IP (R/W)** One of eight pairs of last branch record registers on the last branch record stack. The To_IP part of the stack contains pointers to the destination instruction.
61H	97	MSR_ LASTBRANCH_1_TO_IP	Core	**Last Branch Record 1 To IP (R/W)** See description of MSR_LASTBRANCH_0_TO_IP.
62H	98	MSR_ LASTBRANCH_2_TO_IP	Core	**Last Branch Record 2 To IP (R/W)** See description of MSR_LASTBRANCH_0_TO_IP.
63H	99	MSR_ LASTBRANCH_3_TO_IP	Core	**Last Branch Record 3 To IP (R/W)** See description of MSR_LASTBRANCH_0_TO_IP.
64H	100	MSR_ LASTBRANCH_4_TO_IP	Core	**Last Branch Record 4 To IP (R/W)** See description of MSR_LASTBRANCH_0_TO_IP.
65H	101	MSR_ LASTBRANCH_5_TO_IP	Core	**Last Branch Record 5 To IP (R/W)** See description of MSR_LASTBRANCH_0_TO_IP.
66H	102	MSR_ LASTBRANCH_6_TO_IP	Core	**Last Branch Record 6 To IP (R/W)** See description of MSR_LASTBRANCH_0_TO_IP.
67H	103	MSR_ LASTBRANCH_7_TO_IP	Core	**Last Branch Record 7 To IP (R/W)** See description of MSR_LASTBRANCH_0_TO_IP.
E2H	226	MSR_PKG_CST_CONFIG_ CONTROL	Module	**C-State Configuration Control (R/W)** Note: C-state values are processor specific C-state code names, unrelated to MWAIT extension C-state parameters or ACPI C-States. See http://biosbits.org.

Table 2-7. MSRs Common to the Silvermont and Airmont Microarchitectures

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
		2:0		**Package C-State Limit (R/W)** Specifies the lowest processor-specific C-state code name (consuming the least power). for the package. The default is set as factory-configured package C-state limit. The following C-state code name encodings are supported: 000b: C0 (no package C-sate support) 001b: C1 (Behavior is the same as 000b) 100b: C4 110b: C6 111b: C7 (Silvermont only).
		9:3		Reserved.
		10		**I/O MWAIT Redirection Enable (R/W)** When set, will map IO_read instructions sent to IO register specified by MSR_PMG_IO_CAPTURE_BASE to MWAIT instructions
		14:11		Reserved.
		15		**CFG Lock (R/WO)** When set, lock bits 15:0 of this register until next reset.
		63:16		Reserved.
11EH	281	MSR_BBL_CR_CTL3	Module	Control register 3. Used to configure the L2 Cache.
		0		**L2 Hardware Enabled (RO)** 1 = If the L2 is hardware-enabled 0 = Indicates if the L2 is hardware-disabled
		7:1		Reserved.
		8		**L2 Enabled. (R/W)** 1 = L2 cache has been initialized 0 = Disabled (default) Until this bit is set the processor will not respond to the WBINVD instruction or the assertion of the FLUSH# input.
		22:9		Reserved.
		23		**L2 Not Present (RO)** 0 = L2 Present 1 = L2 Not Present
		63:24		Reserved.
1A0H	416	IA32_MISC_ENABLE		**Enable Misc. Processor Features (R/W)** Allows a variety of processor functions to be enabled and disabled.
		0	Core	**Fast-Strings Enable** See Table 2-2.
		2:1		Reserved.
		3	Module	**Automatic Thermal Control Circuit Enable (R/W)** See Table 2-2. Default value is 0.

Table 2-7. MSRs Common to the Silvermont and Airmont Microarchitectures

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
		6:4		Reserved.
		7	Core	**Performance Monitoring Available (R)** See Table 2-2.
		10:8		Reserved.
		11	Core	**Branch Trace Storage Unavailable (RO)** See Table 2-2.
		12	Core	**Processor Event Based Sampling Unavailable (RO)** See Table 2-2.
		15:13		Reserved.
		16	Module	**Enhanced Intel SpeedStep Technology Enable (R/W)** See Table 2-2.
		18	Core	**ENABLE MONITOR FSM (R/W)** See Table 2-2.
		21:19		Reserved.
		22	Core	**Limit CPUID Maxval (R/W)** See Table 2-2.
		23	Module	**xTPR Message Disable (R/W)** See Table 2-2.
		33:24		Reserved.
		34	Core	**XD Bit Disable (R/W)** See Table 2-2.
		37:35		Reserved.
		38	Module	**Turbo Mode Disable (R/W)** When set to 1 on processors that support Intel Turbo Boost Technology, the turbo mode feature is disabled and the IDA_Enable feature flag will be clear (CPUID.06H: EAX[1]=0). When set to a 0 on processors that support IDA, CPUID.06H: EAX[1] reports the processor's support of turbo mode is enabled. **Note:** the power-on default value is used by BIOS to detect hardware support of turbo mode. If power-on default value is 1, turbo mode is available in the processor. If power-on default value is 0, turbo mode is not available.
		63:39		Reserved.
1C8H	456	MSR_LBR_SELECT	Core	**Last Branch Record Filtering Select Register (R/W)** See Section 17.9.2, "Filtering of Last Branch Records."
		0		**CPL_EQ_0**
		1		**CPL_NEQ_0**
		2		**JCC**
		3		**NEAR_REL_CALL**
		4		**NEAR_IND_CALL**

Table 2-7. MSRs Common to the Silvermont and Airmont Microarchitectures

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
		5		**NEAR_RET**
		6		**NEAR_IND_JMP**
		7		**NEAR_REL_JMP**
		8		**FAR_BRANCH**
		63:9		Reserved.
1C9H	457	MSR_LASTBRANCH_TOS	Core	**Last Branch Record Stack TOS (R/W)** Contains an index (bits 0-2) that points to the MSR containing the most recent branch record. See MSR_LASTBRANCH_0_FROM_IP.
38EH	910	IA32_PERF_GLOBAL_ STATUS	Core	See Table 2-2. See Section 18.6.2.2, "Global Counter Control Facilities."
390H	912	IA32_PERF_GLOBAL_OVF_ CTRL	Core	See Table 2-2. See Section 18.6.2.2, "Global Counter Control Facilities."
3F1H	1009	MSR_PEBS_ENABLE	Core	See Table 2-2. See Section 18.6.2.4, "Processor Event Based Sampling (PEBS)."
		0		Enable PEBS for precise event on IA32_PMC0. (R/W)
3FAH	1018	MSR_PKG_C6_RESIDENCY	Package	Note: C-state values are processor specific C-state code names, unrelated to MWAIT extension C-state parameters or ACPI C-States.
		63:0		Package C6 Residency Counter. (R/O) Value since last reset that this package is in processor-specific C6 states. Counts at the TSC Frequency.
664H	1636	MSR_MC6_RESIDENCY_COU NTER	Module	**Module C6 Residency Counter (R/O)** Note: C-state values are processor specific C-state code names, unrelated to MWAIT extension C-state parameters or ACPI C-States.
		63:0		Time that this module is in module-specific C6 states since last reset. Counts at 1 Mhz frequency.
CEH	206	MSR_PLATFORM_INFO	Package	Platform Information; contains power management and other model specific features enumeration. See http://biosbits.org.
		7:0		Reserved.
		15:8	Package	**Maximum Non-Turbo Ratio (R/O)** The is the ratio of the maximum frequency that does not require turbo. Frequency = ratio * Scalable Bus Frequency.
		63:16		Reserved.

2.4.1 MSRs with Model-Specific Behavior in the Silvermont Microarchitecture

Table 2-8 lists model-specific registers (MSRs) that are specific to Intel® Atom™ processor E3000 Series (CPUID signature with DisplayFamily_DisplayModel of 06_37H) and Intel Atom processors (CPUID signatures with DisplayFamily_DisplayModel of 06_4AH, 06_5AH, 06_5DH).

Table 2-8. Specific MSRs Supported by Intel® Atom™ Processors with CPUID Signatures 06_37H, 06_4AH, 06_5AH, 06_5DH

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
606H	1542	MSR_RAPL_POWER_UNIT	Package	**Unit Multipliers used in RAPL Interfaces (R/O)** See Section 14.9.1, "RAPL Interfaces."
		3:0		Power Units. Power related information (in milliWatts) is based on the multiplier, 2^PU; where PU is an unsigned integer represented by bits 3:0. Default value is 0101b, indicating power unit is in 32 milliWatts increment.
		7:4		Reserved
		12:8		Energy Status Units. Energy related information (in microJoules) is based on the multiplier, 2^ESU; where ESU is an unsigned integer represented by bits 12:8. Default value is 00101b, indicating energy unit is in 32 microJoules increment.
		15:13		Reserved
		19:16		Time Unit. The value is 0000b, indicating time unit is in one second.
		63:20		Reserved
610H	1552	MSR_PKG_POWER_LIMIT	Package	**PKG RAPL Power Limit Control (R/W)**
		14:0		Package Power Limit #1. (R/W) See Section 14.9.3, "Package RAPL Domain." and MSR_RAPL_POWER_UNIT in Table 2-8.
		15		Enable Power Limit #1. (R/W) See Section 14.9.3, "Package RAPL Domain."
		16		Package Clamping Limitation #1. (R/W) See Section 14.9.3, "Package RAPL Domain."
		23:17		Time Window for Power Limit #1. (R/W) in unit of second. If 0 is specified in bits [23:17], defaults to 1 second window.
		63:24		Reserved
611H	1553	MSR_PKG_ENERGY_STATUS	Package	**PKG Energy Status (R/O)** See Section 14.9.3, "Package RAPL Domain." and MSR_RAPL_POWER_UNIT in Table 2-8
639H	1593	MSR_PP0_ENERGY_STATUS	Package	**PP0 Energy Status (R/O)** See Section 14.9.4, "PP0/PP1 RAPL Domains." and MSR_RAPL_POWER_UNIT in Table 2-8
CDH	205	MSR_FSB_FREQ	Module	**Scaleable Bus Speed(RO)** This field indicates the intended scaleable bus clock speed for processors based on Silvermont microarchitecture.

Table 2-8. Specific MSRs Supported by Intel® Atom™ Processors with CPUID Signatures 06_37H, 06_4AH, 06_5AH, 06_5DH

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
		2:0		▪ 100B: 080.0 MHz ▪ 000B: 083.3 MHz ▪ 001B: 100.0 MHz ▪ 010B: 133.3 MHz ▪ 011B: 116.7 MHz
		63:3		Reserved.

Table 2-9 lists model-specific registers (MSRs) that are specific to Intel® Atom™ processor E3000 Series (CPUID signature with DisplayFamily_DisplayModel of 06_37H).

Table 2-9. Specific MSRs Supported by Intel® Atom™ Processor E3000 Series with CPUID Signature 06_37H

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
668H	1640	MSR_CC6_DEMOTION_POLICY_CONFIG	Package	**Core C6 demotion policy config MSR**
		63:0		Controls per-core C6 demotion policy. Writing a value of 0 disables core level HW demotion policy.
669H	1641	MSR_MC6_DEMOTION_POLICY_CONFIG	Package	**Module C6 demotion policy config MSR**
		63:0		Controls module (i.e. two cores sharing the second-level cache) C6 demotion policy. Writing a value of 0 disables module level HW demotion policy.
664H	1636	MSR_MC6_RESIDENCY_COUNTER	Module	**Module C6 Residency Counter (R/O)** Note: C-state values are processor specific C-state code names, unrelated to MWAIT extension C-state parameters or ACPI C-States.
		63:0		Time that this module is in module-specific C6 states since last reset. Counts at 1 Mhz frequency.

Table 2-10 lists model-specific registers (MSRs) that are specific to Intel® Atom™ processor C2000 Series (CPUID signature with DisplayFamily_DisplayModel of 06_4DH).

Table 2-10. Specific MSRs Supported by Intel® Atom™ Processor C2000 Series with CPUID Signature 06_4DH

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
1A4H	420	MSR_MISC_FEATURE_CONTROL		**Miscellaneous Feature Control (R/W)**
		0	Core	**L2 Hardware Prefetcher Disable (R/W)** If 1, disables the L2 hardware prefetcher, which fetches additional lines of code or data into the L2 cache.
		1		**Reserved**

Table 2-10. Specific MSRs Supported by Intel® Atom™ Processor C2000 Series (Contd.)with CPUID Signature

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
		2	Core	**DCU Hardware Prefetcher Disable (R/W)** If 1, disables the L1 data cache prefetcher, which fetches the next cache line into L1 data cache.
		63:3		Reserved.
1ADH	429	MSR_TURBO_RATIO_LIMIT	Package	**Maximum Ratio Limit of Turbo Mode (RW)**
		7:0	Package	**Maximum Ratio Limit for 1C** Maximum turbo ratio limit of 1 core active.
		15:8	Package	**Maximum Ratio Limit for 2C** Maximum turbo ratio limit of 2 core active.
		23:16	Package	**Maximum Ratio Limit for 3C** Maximum turbo ratio limit of 3 core active.
		31:24	Package	**Maximum Ratio Limit for 4C** Maximum turbo ratio limit of 4 core active.
		39:32	Package	**Maximum Ratio Limit for 5C** Maximum turbo ratio limit of 5 core active.
		47:40	Package	**Maximum Ratio Limit for 6C** Maximum turbo ratio limit of 6 core active.
		55:48	Package	**Maximum Ratio Limit for 7C** Maximum turbo ratio limit of 7 core active.
		63:56	Package	**Maximum Ratio Limit for 8C** Maximum turbo ratio limit of 8 core active.
606H	1542	MSR_RAPL_POWER_UNIT	Package	**Unit Multipliers used in RAPL Interfaces (R/O)** See Section 14.9.1, "RAPL Interfaces."
		3:0		Power Units. Power related information (in milliWatts) is based on the multiplier, 2^PU; where PU is an unsigned integer represented by bits 3:0. Default value is 0101b, indicating power unit is in 32 milliWatts increment.
		7:4		Reserved
		12:8		Energy Status Units. Energy related information (in microJoules) is based on the multiplier, 2^ESU; where ESU is an unsigned integer represented by bits 12:8. Default value is 00101b, indicating energy unit is in 32 microJoules increment.
		15:13		Reserved
		19:16		Time Unit. The value is 0000b, indicating time unit is in one second.
		63:20		Reserved
610H	1552	MSR_PKG_POWER_LIMIT	Package	**PKG RAPL Power Limit Control (R/W)** See Section 14.9.3, "Package RAPL Domain."
66EH	1646	MSR_PKG_POWER_INFO	Package	**PKG RAPL Parameter (R/O)**

Table 2-10. Specific MSRs Supported by Intel® Atom™ Processor C2000 Series (Contd.)with CPUID Signature

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
		14:0		Thermal Spec Power. (R/0) The unsigned integer value is the equivalent of thermal specification power of the package domain. The unit of this field is specified by the "Power Units" field of MSR_RAPL_POWER_UNIT
		63:15		Reserved

2.4.2 MSRs In Intel Atom Processors Based on Airmont Microarchitecture

Intel Atom processor X7-Z8000 and X5-Z8000 series are based on the Airmont microarchitecture. These processors support MSRs listed in Table 2-6, Table 2-7, Table 2-8, and Table 2-11. These processors have a CPUID signature with DisplayFamily_DisplayModel including 06_4CH; see Table 2-1.

Table 2-11. MSRs in Intel Atom Processors Based on the Airmont Microarchitecture

Address		Register Name	Scope	Bit Description
Hex	Dec			
CDH	205	MSR_FSB_FREQ	Module	**Scaleable Bus Speed(RO)** This field indicates the intended scaleable bus clock speed for processors based on Airmont microarchitecture:
		3:0		▪ 0000B: 083.3 MHz ▪ 0001B: 100.0 MHz ▪ 0010B: 133.3 MHz ▪ 0011B: 116.7 MHz ▪ 0100B: 080.0 MHz ▪ 0101B: 093.3 MHz ▪ 0110B: 090.0 MHz ▪ 0111B: 088.9 MHz ▪ 1000B: 087.5 MHz
		63:5		Reserved.
E2H	226	MSR_PKG_CST_CONFIG_ CONTROL	Module	**C-State Configuration Control (R/W)** Note: C-state values are processor specific C-state code names, unrelated to MWAIT extension C-state parameters or ACPI C-States. See http://biosbits.org.
		2:0		**Package C-State Limit (R/W)** Specifies the lowest processor-specific C-state code name (consuming the least power). for the package. The default is set as factory-configured package C-state limit. The following C-state code name encodings are supported: 000b: No limit 001b: C1 010b: C2 110b: C6 111b: C7
		9:3		Reserved.

Table 2-11. MSRs in Intel Atom Processors Based on the Airmont Microarchitecture (Contd.)

Address		Register Name	Scope	Bit Description
Hex	Dec			
		10		**I/O MWAIT Redirection Enable (R/W)** When set, will map IO_read instructions sent to IO register specified by MSR_PMG_IO_CAPTURE_BASE to MWAIT instructions
		14:11		Reserved.
		15		**CFG Lock (R/WO)** When set, lock bits 15:0 of this register until next reset.
		63:16		Reserved.
E4H	228	MSR_PMG_IO_CAPTURE_BASE	Module	**Power Management IO Redirection in C-state (R/W)** See http://biosbits.org.
		15:0		**LVL_2 Base Address (R/W)** Specifies the base address visible to software for IO redirection. If IO MWAIT Redirection is enabled, reads to this address will be consumed by the power management logic and decoded to MWAIT instructions. When IO port address redirection is enabled, this is the IO port address reported to the OS/software.
		18:16		**C-state Range (R/W)** Specifies the encoding value of the maximum C-State code name to be included when IO read to MWAIT redirection is enabled by MSR_PKG_CST_CONFIG_CONTROL[bit10]: 000b - C3 is the max C-State to include 001b - Deep Power Down Technology is the max C-State 010b - C7 is the max C-State to include
		63:19		Reserved.
638H	1592	MSR_PP0_POWER_LIMIT	Package	**PP0 RAPL Power Limit Control (R/W)**
		14:0		PP0 Power Limit #1. (R/W) See Section 14.9.4, "PP0/PP1 RAPL Domains." and MSR_RAPL_POWER_UNIT in Table 2-8.
		15		Enable Power Limit #1. (R/W) See Section 14.9.4, "PP0/PP1 RAPL Domains."
		16		Reserved

Table 2-11. MSRs in Intel Atom Processors Based on the Airmont Microarchitecture (Contd.)

Address		Register Name	Scope	Bit Description
Hex	Dec			
		23:17		Time Window for Power Limit #1. (R/W) Specifies the time duration over which the average power must remain below PP0_POWER_LIMIT #1(14:0). Supported Encodings: 0x0: 1 second time duration. 0x1: 5 second time duration (Default). 0x2: 10 second time duration. 0x3: 15 second time duration. 0x4: 20 second time duration. 0x5: 25 second time duration. 0x6: 30 second time duration. 0x7: 35 second time duration. 0x8: 40 second time duration. 0x9: 45 second time duration. 0xA: 50 second time duration. 0xB-0x7F - reserved.
		63:24		Reserved

2.5 MSRS IN INTEL ATOM PROCESSORS BASED ON GOLDMONT MICROARCHITECTURE

Intel Atom processors based on the Goldmont microarchitecture support MSRs listed in Table 2-6 and Table 2-12. These processors have a CPUID signature with DisplayFamily_DisplayModel of 06_5CH; see Table 2-1.

In the Goldmont microarchitecture, the scope column indicates the following: "Core" means each processor core has a separate MSR, or a bit field not shared with another processor core. "Module" means the MSR or the bit field is shared by a pair of processor cores in the physical package. "Package" means all processor cores in the physical package share the same MSR or bit interface.

Table 2-12. MSRs in Next Generation Intel Atom Processors Based on the Goldmont Microarchitecture

Address		Register Name	Scope	Bit Description
Hex	Dec			
17H	23	MSR_PLATFORM_ID	Module	**Model Specific Platform ID (R)**
		49:0		Reserved.
		52:50		See Table 2-2.
		63:33		Reserved.
3AH	58	IA32_FEATURE_CONTROL	Core	**Control Features in Intel 64Processor (R/W)** See Table 2-2.
		0		**Lock (R/WL)**
		1		**Enable VMX inside SMX operation (R/WL)**
		2		**Enable VMX outside SMX operation (R/WL)**
		14:8		**SENTER local functions enables (R/WL)**

Table 2-12. MSRs in Next Generation Intel Atom Processors Based on the Goldmont Microarchitecture (Contd.)

Address		Register Name	Scope	Bit Description
Hex	Dec			
		15		**SENTER global functions enable (R/WL)**
		18		**SGX global functions enable (R/WL)**
		63:19		Reserved.
3BH	59	IA32_TSC_ADJUST	Core	**Per-Core TSC ADJUST (R/W)** See Table 2-2.
C3H	195	IA32_PMC2	Core	**Performance Counter Register** See Table 2-2.
C4H	196	IA32_PMC3	Core	**Performance Counter Register** See Table 2-2.
CEH	206	MSR_PLATFORM_INFO	Package	Platform Information; contains power management and other model specific features enumeration. See http://biosbits.org.
		7:0		Reserved.
		15:8	Package	**Maximum Non-Turbo Ratio (R/O)** The is the ratio of the maximum frequency that does not require turbo. Frequency = ratio * 100 MHz.
		27:16		Reserved.
		28	Package	**Programmable Ratio Limit for Turbo Mode (R/O)** When set to 1, indicates that Programmable Ratio Limits for Turbo mode is enabled, and when set to 0, indicates Programmable Ratio Limits for Turbo mode is disabled.
		29	Package	**Programmable TDP Limit for Turbo Mode (R/O)** When set to 1, indicates that TDP Limits for Turbo mode are programmable, and when set to 0, indicates TDP Limit for Turbo mode is not programmable.
		30	Package	**Programmable TJ OFFSET (R/O)** When set to 1, indicates that MSR_TEMPERATURE_TARGET.[27:24] is valid and writable to specify an temperature offset.
		39:31		Reserved.
		47:40	Package	**Maximum Efficiency Ratio (R/O)** The is the minimum ratio (maximum efficiency) that the processor can operates, in units of 100MHz.
		63:48		Reserved.
E2H	226	MSR_PKG_CST_CONFIG_ CONTROL	Core	**C-State Configuration Control (R/W)** Note: C-state values are processor specific C-state code names, unrelated to MWAIT extension C-state parameters or ACPI C-States. See http://biosbits.org.

Table 2-12. MSRs in Next Generation Intel Atom Processors Based on the Goldmont Microarchitecture (Contd.)

Address		Register Name	Scope	Bit Description
Hex	Dec			
		3:0		**Package C-State Limit (R/W)** Specifies the lowest processor-specific C-state code name (consuming the least power). for the package. The default is set as factory-configured package C-state limit. The following C-state code name encodings are supported: 0000b: No limit 0001b: C1 0010b: C3 0011b: C6 0100b: C7 0101b: C7S 0110b: C8 0111b: C9 1000b: C10
		9:3		Reserved.
		10		**I/O MWAIT Redirection Enable (R/W)** When set, will map IO_read instructions sent to IO register specified by MSR_PMG_IO_CAPTURE_BASE to MWAIT instructions
		14:11		Reserved.
		15		**CFG Lock (R/WO)** When set, lock bits 15:0 of this register until next reset.
		63:16		Reserved.
17DH	381	MSR_SMM_MCA_CAP	Core	**Enhanced SMM Capabilities (SMM-RO)** Reports SMM capability Enhancement. Accessible only while in SMM.
		57:0		**Reserved**
		58		**SMM_Code_Access_Chk (SMM-RO)** If set to 1 indicates that the SMM code access restriction is supported and the MSR_SMM_FEATURE_CONTROL is supported.
		59		**Long_Flow_Indication (SMM-RO)** If set to 1 indicates that the SMM long flow indicator is supported and the MSR_SMM_DELAYED is supported.
		63:60		Reserved
188H	392	IA32_PERFEVTSEL2	Core	See Table 2-2.
189H	393	IA32_PERFEVTSEL3	Core	See Table 2-2.
1A0H	416	IA32_MISC_ENABLE		**Enable Misc. Processor Features (R/W)** Allows a variety of processor functions to be enabled and disabled.
		0	Core	**Fast-Strings Enable** See Table 2-2.
		2:1		Reserved.
		3	Package	**Automatic Thermal Control Circuit Enable (R/W)** See Table 2-2. Default value is 1.

Table 2-12. MSRs in Next Generation Intel Atom Processors Based on the Goldmont Microarchitecture (Contd.)

Address		Register Name	Scope	Bit Description
Hex	Dec			
		6:4		Reserved.
		7	Core	**Performance Monitoring Available (R)** See Table 2-2.
		10:8		Reserved.
		11	Core	**Branch Trace Storage Unavailable (RO)** See Table 2-2.
		12	Core	**Processor Event Based Sampling Unavailable (RO)** See Table 2-2.
		15:13		Reserved.
		16	Package	**Enhanced Intel SpeedStep Technology Enable (R/W)** See Table 2-2.
		18	Core	**ENABLE MONITOR FSM (R/W)** See Table 2-2.
		21:19		Reserved.
		22	Core	**Limit CPUID Maxval (R/W)** See Table 2-2.
		23	Package	**xTPR Message Disable (R/W)** See Table 2-2.
		33:24		Reserved.
		34	Core	**XD Bit Disable (R/W)** See Table 2-2.
		37:35		Reserved.
		38	Package	**Turbo Mode Disable (R/W)** When set to 1 on processors that support Intel Turbo Boost Technology, the turbo mode feature is disabled and the IDA_Enable feature flag will be clear (CPUID.06H: EAX[1]=0). When set to a 0 on processors that support IDA, CPUID.06H: EAX[1] reports the processor's support of turbo mode is enabled. **Note:** the power-on default value is used by BIOS to detect hardware support of turbo mode. If power-on default value is 1, turbo mode is available in the processor. If power-on default value is 0, turbo mode is not available.
		63:39		Reserved.
1A4H	420	MSR_MISC_FEATURE_ CONTROL		**Miscellaneous Feature Control (R/W)**
		0	Core	**L2 Hardware Prefetcher Disable (R/W)** If 1, disables the L2 hardware prefetcher, which fetches additional lines of code or data into the L2 cache.
		1		**Reserved**
		2	Core	**DCU Hardware Prefetcher Disable (R/W)** If 1, disables the L1 data cache prefetcher, which fetches the next cache line into L1 data cache.

Table 2-12. MSRs in Next Generation Intel Atom Processors Based on the Goldmont Microarchitecture (Contd.)

Address		Register Name	Scope	Bit Description
Hex	Dec			
		63:3		Reserved.
1AAH	426	MSR_MISC_PWR_MGMT	Package	Miscellaneous Power Management Control; various model specific features enumeration. See http://biosbits.org.
		0		**EIST Hardware Coordination Disable (R/W)** When 0, enables hardware coordination of Enhanced Intel Speedstep Technology request from processor cores; When 1, disables hardware coordination of Enhanced Intel Speedstep Technology requests.
		21:1		Reserved.
		22		**Thermal Interrupt Coordination Enable (R/W)** If set, then thermal interrupt on one core is routed to all cores.
		63:23		Reserved.
1ADH	429	MSR_TURBO_RATIO_LIMIT	Package	**Maximum Ratio Limit of Turbo Mode by Core Groups (RW)** **Specifies Maximum Ratio Limit for each Core Group. Max ratio for groups with more cores must decrease monotonically.** **For groups with less than 4 cores, the max ratio must be 32 or less. For groups with 4-5 cores, the max ratio must be 22 or less. For groups with more than 5 cores, the max ratio must be 16 or less.**
		7:0	Package	**Maximum Ratio Limit for Active cores in Group 0** Maximum turbo ratio limit when number of active cores is less or equal to Group 0 threshold.
		15:8	Package	**Maximum Ratio Limit for Active cores in Group 1** Maximum turbo ratio limit when number of active cores is less or equal to Group 1 threshold and greater than Group 0 threshold.
		23:16	Package	**Maximum Ratio Limit for Active cores in Group 2** Maximum turbo ratio limit when number of active cores is less or equal to Group 2 threshold and greater than Group 1 threshold.
		31:24	Package	**Maximum Ratio Limit for Active cores in Group 3** Maximum turbo ratio limit when number of active cores is less or equal to Group 3 threshold and greater than Group 2 threshold.
		39:32	Package	**Maximum Ratio Limit for Active cores in Group 4** Maximum turbo ratio limit when number of active cores is less or equal to Group 4 threshold and greater than Group 3 threshold.
		47:40	Package	**Maximum Ratio Limit for Active cores in Group 5** Maximum turbo ratio limit when number of active cores is less or equal to Group 5 threshold and greater than Group 4 threshold.
		55:48	Package	**Maximum Ratio Limit for Active cores in Group 6** Maximum turbo ratio limit when number of active cores is less or equal to Group 6 threshold and greater than Group 5 threshold.
		63:56	Package	**Maximum Ratio Limit for Active cores in Group 7** Maximum turbo ratio limit when number of active cores is less or equal to Group 7 threshold and greater than Group 6 threshold.

Table 2-12. MSRs in Next Generation Intel Atom Processors Based on the Goldmont Microarchitecture (Contd.)

Address		Register Name	Scope	Bit Description
Hex	Dec			
1AEH	430	MSR_TURBO_GROUP_CORE CNT	Package	**Group Size of Active Cores for Turbo Mode Operation (RW)** **Writes of 0 threshold is ignored**
		7:0	Package	**Group 0 Core Count Threshold** Maximum number of active cores to operate under Group 0 Max Turbo Ratio limit.
		15:8	Package	**Group 1 Core Count Threshold** Maximum number of active cores to operate under Group 1 Max Turbo Ratio limit. Must be greater than Group 0 Core Count.
		23:16	Package	**Group 2 Core Count Threshold** Maximum number of active cores to operate under Group 2 Max Turbo Ratio limit. Must be greater than Group 1 Core Count.
		31:24	Package	**Group 3 Core Count Threshold** Maximum number of active cores to operate under Group 3 Max Turbo Ratio limit. Must be greater than Group 2 Core Count.
		39:32	Package	**Group 4 Core Count Threshold** Maximum number of active cores to operate under Group 4 Max Turbo Ratio limit. Must be greater than Group 3 Core Count.
		47:40	Package	**Group 5 Core Count Threshold** Maximum number of active cores to operate under Group 5 Max Turbo Ratio limit. Must be greater than Group 4 Core Count.
		55:48	Package	**Group 6 Core Count Threshold** Maximum number of active cores to operate under Group 6 Max Turbo Ratio limit. Must be greater than Group 5 Core Count.
		63:56	Package	**Group 7 Core Count Threshold** Maximum number of active cores to operate under Group 7 Max Turbo Ratio limit. Must be greater than Group 6 Core Count and not less than the total number of processor cores in the package. E.g. specify 255.
1C8H	456	MSR_LBR_SELECT	Core	**Last Branch Record Filtering Select Register (R/W)** See Section 17.9.2, "Filtering of Last Branch Records."
		0		**CPL_EQ_0**
		1		**CPL_NEQ_0**
		2		**JCC**
		3		**NEAR_REL_CALL**
		4		**NEAR_IND_CALL**
		5		**NEAR_RET**
		6		**NEAR_IND_JMP**
		7		**NEAR_REL_JMP**
		8		**FAR_BRANCH**
		9		**EN_CALL_STACK**
		63:10		Reserved.

Table 2-12. MSRs in Next Generation Intel Atom Processors Based on the Goldmont Microarchitecture (Contd.)

Address		Register Name	Scope	Bit Description
Hex	Dec			
1C9H	457	MSR_LASTBRANCH_TOS	Core	**Last Branch Record Stack TOS (R/W)** Contains an index (bits 0-4) that points to the MSR containing the most recent branch record. See MSR_LASTBRANCH_0_FROM_IP.
1FCH	508	MSR_POWER_CTL	Core	Power Control Register. See http://biosbits.org.
		0		Reserved.
		1	Package	**C1E Enable (R/W)** When set to '1', will enable the CPU to switch to the Minimum Enhanced Intel SpeedStep Technology operating point when all execution cores enter MWAIT (C1).
		63:2		Reserved.
210H	528	IA32_MTRR_PHYSBASE8	Core	See Table 2-2.
211H	529	IA32_MTRR_PHYSMASK8	Core	See Table 2-2.
212H	530	IA32_MTRR_PHYSBASE9	Core	See Table 2-2.
213H	531	IA32_MTRR_PHYSMASK9	Core	See Table 2-2.
280H	640	IA32_MC0_CTL2	Module	See Table 2-2.
281H	641	IA32_MC1_CTL2	Module	See Table 2-2.
282H	642	IA32_MC2_CTL2	Core	See Table 2-2.
283H	643	IA32_MC3_CTL2	Module	See Table 2-2.
284H	644	IA32_MC4_CTL2	Package	See Table 2-2.
285H	645	IA32_MC5_CTL2	Package	See Table 2-2.
286H	646	IA32_MC6_CTL2	Package	See Table 2-2.
300H	768	MSR_SGXOWNEREPOCH0	Package	**Lower 64 Bit CR_SGXOWNEREPOCH.** Writes do not update CR_SGXOWNEREPOCH if CPUID.(EAX=12H, ECX=0):EAX.SGX1 is 1 on any thread in the package.
		63:0		Lower 64 bits of an 128-bit external entropy value for key derivation of an enclave.
301H	769	MSR_SGXOWNEREPOCH1	Package	**Upper 64 Bit CR_SGXOWNEREPOCH.** Writes do not update CR_SGXOWNEREPOCH if CPUID.(EAX=12H, ECX=0):EAX.SGX1 is 1 on any thread in the package.
		63:0		Upper 64 bits of an 128-bit external entropy value for key derivation of an enclave.
38EH	910	IA32_PERF_GLOBAL_ STATUS	Core	See Table 2-2. See Section 18.2.4, "Architectural Performance Monitoring Version 4."
		0		**Ovf_PMC0**
		1		**Ovf_PMC1**
		2		**Ovf_PMC2**
		3		**Ovf_PMC3**
		31:4		Reserved.
		32		**Ovf_FixedCtr0**
		33		**Ovf_FixedCtr1**

Table 2-12. MSRs in Next Generation Intel Atom Processors Based on the Goldmont Microarchitecture (Contd.)

Address		Register Name	Scope	Bit Description
Hex	Dec			
		34		**Ovf_FixedCtr2**
		54:35		Reserved.
		55		**Trace_ToPA_PMI.**
		57:56		Reserved.
		58		**LBR_Frz.**
		59		**CTR_Frz.**
		60		**ASCI.**
		61		**Ovf_Uncore**
		62		**Ovf_BufDSSAVE**
		63		**CondChgd**
390H	912	IA32_PERF_GLOBAL_STATUS_RESET	Core	See Table 2-2. See Section 18.2.4, "Architectural Performance Monitoring Version 4."
		0		**Set 1 to clear Ovf_PMC0**
		1		**Set 1 to clear Ovf_PMC1**
		2		**Set 1 to clear Ovf_PMC2**
		3		**Set 1 to clear Ovf_PMC3**
		31:4		Reserved.
		32		**Set 1 to clear Ovf_FixedCtr0**
		33		**Set 1 to clear Ovf_FixedCtr1**
		34		**Set 1 to clear Ovf_FixedCtr2**
		54:35		Reserved.
		55		**Set 1 to clear Trace_ToPA_PMI.**
		57:56		Reserved.
		58		**Set 1 to clear LBR_Frz.**
		59		**Set 1 to clear CTR_Frz.**
		60		**Set 1 to clear ASCI.**
		61		**Set 1 to clear Ovf_Uncore**
		62		**Set 1 to clear Ovf_BufDSSAVE**
		63		**Set 1 to clear CondChgd**
391H	913	IA32_PERF_GLOBAL_STATUS_SET	Core	See Table 2-2. See Section 18.2.4, "Architectural Performance Monitoring Version 4."
		0		**Set 1 to cause Ovf_PMC0 = 1**
		1		**Set 1 to cause Ovf_PMC1 = 1**
		2		**Set 1 to cause Ovf_PMC2 = 1**
		3		**Set 1 to cause Ovf_PMC3 = 1**
		31:4		Reserved.
		32		**Set 1 to cause Ovf_FixedCtr0 = 1**
		33		**Set 1 to cause Ovf_FixedCtr1 = 1**

Table 2-12. MSRs in Next Generation Intel Atom Processors Based on the Goldmont Microarchitecture (Contd.)

Address		Register Name	Scope	Bit Description
Hex	Dec			
		34		**Set 1 to cause Ovf_FixedCtr2 = 1**
		54:35		Reserved.
		55		**Set 1 to cause Trace_ToPA_PMI = 1**
		57:56		Reserved.
		58		**Set 1 to cause LBR_Frz = 1**
		59		**Set 1 to cause CTR_Frz = 1**
		60		**Set 1 to cause ASCI = 1**
		61		**Set 1 to cause Ovf_Uncore**
		62		**Set 1 to cause Ovf_BufDSSAVE**
		63		Reserved.
392H	914	IA32_PERF_GLOBAL_INUSE		See Table 2-2.
3F1H	1009	MSR_PEBS_ENABLE	Core	See Table 2-2. See Section 18.6.2.4, "Processor Event Based Sampling (PEBS)."
		0		Enable PEBS trigger and recording for the programmed event (precise or otherwise) on IA32_PMC0. (R/W)
3F8H	1016	MSR_PKG_C3_RESIDENCY	Package	Note: C-state values are processor specific C-state code names, unrelated to MWAIT extension C-state parameters or ACPI C-States.
		63:0		Package C3 Residency Counter. (R/O) Value since last reset that this package is in processor-specific C3 states. Count at the same frequency as the TSC.
3F9H	1017	MSR_PKG_C6_RESIDENCY	Package	Note: C-state values are processor specific C-state code names, unrelated to MWAIT extension C-state parameters or ACPI C-States.
		63:0		Package C6 Residency Counter. (R/O) Value since last reset that this package is in processor-specific C6 states. Count at the same frequency as the TSC.
3FCH	1020	MSR_CORE_C3_RESIDENCY	Core	Note: C-state values are processor specific C-state code names, unrelated to MWAIT extension C-state parameters or ACPI C-States.
		63:0		CORE C3 Residency Counter. (R/O) Value since last reset that this core is in processor-specific C3 states. Count at the same frequency as the TSC.
406H	1030	IA32_MC1_ADDR	Module	See Section 15.3.2.3, "IA32_MCi_ADDR MSRs." The IA32_MC2_ADDR register is either not implemented or contains no address if the ADDRV flag in the IA32_MC2_STATUS register is clear. When not implemented in the processor, all reads and writes to this MSR will cause a general-protection exception.
418H	1048	IA32_MC6_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs."
419H	1049	IA32_MC6_STATUS	Package	See Section 15.3.2.2, "IA32_MCi_STATUS MSRS," and Chapter 16.
41AH	1050	IA32_MC6_ADDR	Package	See Section 15.3.2.3, "IA32_MCi_ADDR MSRs."

Table 2-12. MSRs in Next Generation Intel Atom Processors Based on the Goldmont Microarchitecture (Contd.)

Address		Register Name	Scope	Bit Description
Hex	Dec			
4C3H	1219	IA32_A_PMC2	Core	See Table 2-2.
4C4H	1220	IA32_A_PMC3	Core	See Table 2-2.
4E0H	1248	MSR_SMM_FEATURE_CONTROL	Package	**Enhanced SMM Feature Control (SMM-RW)** Reports SMM capability Enhancement. Accessible only while in SMM.
		0		**Lock (SMM-RWO)** When set to '1' locks this register from further changes
		1		Reserved
		2		**SMM_Code_Chk_En (SMM-RW)** This control bit is available only if MSR_SMM_MCA_CAP[58] == 1. When set to '0' (default) none of the logical processors are prevented from executing SMM code outside the ranges defined by the SMRR. When set to '1' any logical processor in the package that attempts to execute SMM code not within the ranges defined by the SMRR will assert an unrecoverable MCE.
		63:3		Reserved
4E2H	1250	MSR_SMM_DELAYED	Package	**SMM Delayed (SMM-RO)** Reports the interruptible state of all logical processors in the package. Available only while in SMM and MSR_SMM_MCA_CAP[LONG_FLOW_INDICATION] == 1.
		N-1:0		**LOG_PROC_STATE (SMM-RO)** Each bit represents a processor core of its state in a long flow of internal operation which delays servicing an interrupt. The corresponding bit will be set at the start of long events such as: Microcode Update Load, C6, WBINVD, Ratio Change, Throttle. The bit is automatically cleared at the end of each long event. The reset value of this field is 0. Only bit positions below N = CPUID.(EAX=0BH, ECX=PKG_LVL):EBX[15:0] can be updated.
		63:N		Reserved
4E3H	1251	MSR_SMM_BLOCKED	Package	**SMM Blocked (SMM-RO)** Reports the blocked state of all logical processors in the package. Available only while in SMM.
		N-1:0		**LOG_PROC_STATE (SMM-RO)** Each bit represents a processor core of its blocked state to service an SMI. The corresponding bit will be set if the logical processor is in one of the following states: Wait For SIPI or SENTER Sleep. The reset value of this field is 0FFFH. Only bit positions below N = CPUID.(EAX=0BH, ECX=PKG_LVL):EBX[15:0] can be updated.
		63:N		Reserved
500H	1280	IA32_SGX_SVN_STATUS	Core	**Status and SVN Threshold of SGX Support for ACM (RO).**
		0		**Lock.** See Section 41.11.3, "Interactions with Authenticated Code Modules (ACMs)"

Table 2-12. MSRs in Next Generation Intel Atom Processors Based on the Goldmont Microarchitecture (Contd.)

Address		Register Name	Scope	Bit Description
Hex	Dec			
		15:1		Reserved.
		23:16		**SGX_SVN_SINIT.** See Section 41.11.3, "Interactions with Authenticated Code Modules (ACMs)"
		63:24		Reserved.
560H	1376	IA32_RTIT_OUTPUT_BASE	Core	**Trace Output Base Register (R/W).** See Table 2-2.
561H	1377	IA32_RTIT_OUTPUT_MASK_PTRS	Core	**Trace Output Mask Pointers Register (R/W).** See Table 2-2.
570H	1392	IA32_RTIT_CTL	Core	**Trace Control Register (R/W)**
		0		**TraceEn**
		1		**CYCEn**
		2		**OS**
		3		**User**
		6:4		Reserved, MBZ
		7		**CR3 filter**
		8		**ToPA; writing 0 will #GP if also setting TraceEn**
		9		**MTCEn**
		10		**TSCEn**
		11		**DisRETC**
		12		Reserved, MBZ
		13		**BranchEn**
		17:14		**MTCFreq**
		18		Reserved, MBZ
		22:19		**CYCThresh**
		23		Reserved, MBZ
		27:24		**PSBFreq**
		31:28		Reserved, MBZ
		35:32		**ADDR0_CFG**
		39:36		**ADDR1_CFG**
		63:40		Reserved, MBZ.
571H	1393	IA32_RTIT_STATUS	Core	**Tracing Status Register (R/W)**
		0		**FilterEn**, writes ignored.
		1		**ContexEn**, writes ignored.
		2		**TriggerEn**, writes ignored.
		3		Reserved
		4		**Error (R/W)**
		5		**Stopped**
		31:6		Reserved. MBZ
		48:32		**PacketByteCnt**

Table 2-12. MSRs in Next Generation Intel Atom Processors Based on the Goldmont Microarchitecture (Contd.)

Address		Register Name	Scope	Bit Description
Hex	Dec			
		63:49		Reserved, MBZ.
572H	1394	IA32_RTIT_CR3_MATCH	Core	**Trace Filter CR3 Match Register (R/W)**
		4:0		Reserved
		63:5		CR3[63:5] value to match
580H	1408	IA32_RTIT_ADDR0_A	Core	**Region 0 Start Address (R/W)**
		63:0		See Table 2-2.
581H	1409	IA32_RTIT_ADDR0_B	Core	**Region 0 End Address (R/W)**
		63:0		See Table 2-2.
582H	1410	IA32_RTIT_ADDR1_A	Core	**Region 1 Start Address (R/W)**
		63:0		See Table 2-2.
583H	1411	IA32_RTIT_ADDR1_B	Core	**Region 1 End Address (R/W)**
		63:0		See Table 2-2.
606H	1542	MSR_RAPL_POWER_UNIT	Package	**Unit Multipliers used in RAPL Interfaces (R/O)** See Section 14.9.1, "RAPL Interfaces."
		3:0		Power Units. Power related information (in Watts) is in unit of, 1W/2^PU; where PU is an unsigned integer represented by bits 3:0. Default value is 1000b, indicating power unit is in 3.9 milliWatts increment.
		7:4		Reserved
		12:8		Energy Status Units. Energy related information (in Joules) is in unit of, 1Joule/ (2^ESU); where ESU is an unsigned integer represented by bits 12:8. Default value is 01110b, indicating energy unit is in 61 microJoules.
		15:13		Reserved
		19:16		Time Unit. Time related information (in seconds) is in unit of, 1S/2^TU; where TU is an unsigned integer represented by bits 19:16. Default value is 1010b, indicating power unit is in 0.977 millisecond.
		63:20		Reserved
60AH	1546	MSR_PKGC3_IRTL	Package	**Package C3 Interrupt Response Limit (R/W)** Note: C-state values are processor specific C-state code names, unrelated to MWAIT extension C-state parameters or ACPI C-States.
		9:0		**Interrupt response time limit (R/W)** Specifies the limit that should be used to decide if the package should be put into a package C3 state.
		12:10		**Time Unit (R/W)** Specifies the encoding value of time unit of the interrupt response time limit. See Table 2-19 for supported time unit encodings.
		14:13		Reserved.

Table 2-12. MSRs in Next Generation Intel Atom Processors Based on the Goldmont Microarchitecture (Contd.)

Address		Register Name	Scope	Bit Description
Hex	Dec			
		15		**Valid (R/W)** Indicates whether the values in bits 12:0 are valid and can be used by the processor for package C-sate management.
		63:16		Reserved.
60BH	1547	MSR_PKGC_IRTL1	Package	**Package C6/C7S Interrupt Response Limit 1 (R/W)** This MSR defines the interrupt response time limit used by the processor to manage transition to package C6 or C7S state. Note: C-state values are processor specific C-state code names, unrelated to MWAIT extension C-state parameters or ACPI C-States.
		9:0		**Interrupt response time limit (R/W)** Specifies the limit that should be used to decide if the package should be put into a package C6 or C7S state.
		12:10		**Time Unit (R/W)** Specifies the encoding value of time unit of the interrupt response time limit. See Table 2-19 for supported time unit encodings
		14:13		Reserved.
		15		**Valid (R/W)** Indicates whether the values in bits 12:0 are valid and can be used by the processor for package C-sate management.
		63:16		Reserved.
60CH	1548	MSR_PKGC_IRTL2	Package	**Package C7 Interrupt Response Limit 2 (R/W)** This MSR defines the interrupt response time limit used by the processor to manage transition to package C7 state. Note: C-state values are processor specific C-state code names, unrelated to MWAIT extension C-state parameters or ACPI C-States.
		9:0		**Interrupt response time limit (R/W)** Specifies the limit that should be used to decide if the package should be put into a package C7 state.
		12:10		**Time Unit (R/W)** Specifies the encoding value of time unit of the interrupt response time limit. See Table 2-19 for supported time unit encodings
		14:13		Reserved.
		15		**Valid (R/W)** Indicates whether the values in bits 12:0 are valid and can be used by the processor for package C-sate management.
		63:16		Reserved.
60DH	1549	MSR_PKG_C2_RESIDENCY	Package	Note: C-state values are processor specific C-state code names, unrelated to MWAIT extension C-state parameters or ACPI C-States.
		63:0		**Package C2 Residency Counter. (R/O)** Value since last reset that this package is in processor-specific C2 states. Count at the same frequency as the TSC.

Table 2-12. MSRs in Next Generation Intel Atom Processors Based on the Goldmont Microarchitecture (Contd.)

Address		Register Name	Scope	Bit Description
Hex	Dec			
610H	1552	MSR_PKG_POWER_LIMIT	Package	**PKG RAPL Power Limit Control (R/W)** See Section 14.9.3, "Package RAPL Domain."
611H	1553	MSR_PKG_ENERGY_STATUS	Package	**PKG Energy Status (R/O)** See Section 14.9.3, "Package RAPL Domain."
613H	1555	MSR_PKG_PERF_STATUS	Package	**PKG Perf Status (R/O)** See Section 14.9.3, "Package RAPL Domain."
614H	1556	MSR_PKG_POWER_INFO	Package	**PKG RAPL Parameters (R/W)**
		14:0		**Thermal Spec Power (R/W)** See Section 14.9.3, "Package RAPL Domain."
		15		Reserved.
		30:16		**Minimum Power (R/W)** See Section 14.9.3, "Package RAPL Domain."
		31		Reserved.
		46:32		**Maximum Power (R/W)** See Section 14.9.3, "Package RAPL Domain."
		47		Reserved.
		54:48		**Maximum Time Window (R/W)** Specified by 2^Y * (1.0 + Z/4.0) * Time_Unit, where "Y" is the unsigned integer value represented. by bits 52:48, "Z" is an unsigned integer represented by bits 54:53. "Time_Unit" is specified by the "Time Units" field of MSR_RAPL_POWER_UNIT
		63:55		Reserved.
618H	1560	MSR_DRAM_POWER_LIMIT	Package	**DRAM RAPL Power Limit Control (R/W)** See Section 14.9.5, "DRAM RAPL Domain."
619H	1561	MSR_DRAM_ENERGY_ STATUS	Package	**DRAM Energy Status (R/O)** See Section 14.9.5, "DRAM RAPL Domain."
61BH	1563	MSR_DRAM_PERF_STATUS	Package	**DRAM Performance Throttling Status (R/O)** See Section 14.9.5, "DRAM RAPL Domain."
61CH	1564	MSR_DRAM_POWER_INFO	Package	**DRAM RAPL Parameters (R/W)** See Section 14.9.5, "DRAM RAPL Domain."
632H	1586	MSR_PKG_C10_RESIDENCY	Package	Note: C-state values are processor specific C-state code names,
		63:0		**Package C10 Residency Counter. (R/O)** Value since last reset that the entire SOC is in an S0i3 state. Count at the same frequency as the TSC.
639H	1593	MSR_PP0_ENERGY_STATU S	Package	**PP0 Energy Status (R/O)** See Section 14.9.4, "PP0/PP1 RAPL Domains."
641H	1601	MSR_PP1_ENERGY_STATU S	Package	**PP1 Energy Status (R/O)** See Section 14.9.4, "PP0/PP1 RAPL Domains."
64CH	1612	MSR_TURBO_ACTIVATION_ RATIO	Package	**ConfigTDP Control (R/W)**

Table 2-12. MSRs in Next Generation Intel Atom Processors Based on the Goldmont Microarchitecture (Contd.)

Address		Register Name	Scope	Bit Description
Hex	Dec			
		7:0		**MAX_NON_TURBO_RATIO (RW/L)** System BIOS can program this field.
		30:8		Reserved.
		31		**TURBO_ACTIVATION_RATIO_Lock (RW/L)** When this bit is set, the content of this register is locked until a reset.
		63:32		Reserved.
64FH	1615	MSR_CORE_PERF_LIMIT_REASONS	Package	**Indicator of Frequency Clipping in Processor Cores (R/W)** **(frequency refers to processor core frequency)**
		0		**PROCHOT Status (R0)** When set, processor core frequency is reduced below the operating system request due to assertion of external PROCHOT.
		1		**Thermal Status (R0)** When set, frequency is reduced below the operating system request due to a thermal event.
		2		**Package-Level Power Limiting PL1 Status (R0)** When set, frequency is reduced below the operating system request due to package-level power limiting PL1.
		3		**Package-Level PL2 Power Limiting Status (R0)** When set, frequency is reduced below the operating system request due to package-level power limiting PL2.
		8:4		Reserved.
		9		**Core Power Limiting Status (R0)** When set, frequency is reduced below the operating system request due to domain-level power limiting.
		10		**VR Therm Alert Status (R0)** When set, frequency is reduced below the operating system request due to a thermal alert from the Voltage Regulator.
		11		**Max Turbo Limit Status (R0)** When set, frequency is reduced below the operating system request due to multi-core turbo limits.
		12		**Electrical Design Point Status (R0)** When set, frequency is reduced below the operating system request due to electrical design point constraints (e.g. maximum electrical current consumption).
		13		**Turbo Transition Attenuation Status (R0)** When set, frequency is reduced below the operating system request due to Turbo transition attenuation. This prevents performance degradation due to frequent operating ratio changes.
		14		**Maximum Efficiency Frequency Status (R0)** When set, frequency is reduced below the maximum efficiency frequency.
		15		**Reserved**

Table 2-12. MSRs in Next Generation Intel Atom Processors Based on the Goldmont Microarchitecture (Contd.)

Address		Register Name	Scope	Bit Description
Hex	Dec			
		16		**PROCHOT Log** When set, indicates that the PROCHOT Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.
		17		**Thermal Log** When set, indicates that the Thermal Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.
		18		**Package-Level PL1 Power Limiting Log** When set, indicates that the Package Level PL1 Power Limiting Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.
		19		**Package-Level PL2 Power Limiting Log** When set, indicates that the Package Level PL2 Power Limiting Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.
		24:20		Reserved.
		25		**Core Power Limiting Log** When set, indicates that the Core Power Limiting Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.
		26		**VR Therm Alert Log** When set, indicates that the VR Therm Alert Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.
		27		**Max Turbo Limit Log** When set, indicates that the Max Turbo Limit Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.
		28		**Electrical Design Point Log** When set, indicates that the EDP Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.
		29		**Turbo Transition Attenuation Log** When set, indicates that the Turbo Transition Attenuation Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.
		30		**Maximum Efficiency Frequency Log** When set, indicates that the Maximum Efficiency Frequency Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.
		63:31		Reserved.

Table 2-12. MSRs in Next Generation Intel Atom Processors Based on the Goldmont Microarchitecture (Contd.)

Address		Register Name	Scope	Bit Description
Hex	Dec			
680H	1664	MSR_ LASTBRANCH_0_FROM_IP	Core	**Last Branch Record 0 From IP (R/W)** One of 32 pairs of last branch record registers on the last branch record stack. The From_IP part of the stack contains pointers to the **source instruction** . See also: ▪ Last Branch Record Stack TOS at 1C9H ▪ Section 17.6 and record format in Section 17.4.8.1
		0:47		**From Linear Address (R/W)**
		62:48		Signed extension of bits 47:0.
		63		Mispred
681H	1665	MSR_ LASTBRANCH_1_FROM_IP	Core	**Last Branch Record 1 From IP (R/W)** See description of MSR_LASTBRANCH_0_FROM_IP.
682H	1666	MSR_ LASTBRANCH_2_FROM_IP	Core	**Last Branch Record 2 From IP (R/W)** See description of MSR_LASTBRANCH_0_FROM_IP.
683H	1667	MSR_ LASTBRANCH_3_FROM_IP	Core	**Last Branch Record 3 From IP (R/W)** See description of MSR_LASTBRANCH_0_FROM_IP.
684H	1668	MSR_ LASTBRANCH_4_FROM_IP	Core	**Last Branch Record 4 From IP (R/W)** See description of MSR_LASTBRANCH_0_FROM_IP.
685H	1669	MSR_ LASTBRANCH_5_FROM_IP	Core	**Last Branch Record 5 From IP (R/W)** See description of MSR_LASTBRANCH_0_FROM_IP.
686H	1670	MSR_ LASTBRANCH_6_FROM_IP	Core	**Last Branch Record 6 From IP (R/W)** See description of MSR_LASTBRANCH_0_FROM_IP.
687H	1671	MSR_ LASTBRANCH_7_FROM_IP	Core	**Last Branch Record 7 From IP (R/W)** See description of MSR_LASTBRANCH_0_FROM_IP.
688H	1672	MSR_ LASTBRANCH_8_FROM_IP	Core	**Last Branch Record 8 From IP (R/W)** See description of MSR_LASTBRANCH_0_FROM_IP.
689H	1673	MSR_ LASTBRANCH_9_FROM_IP	Core	**Last Branch Record 9 From IP (R/W)** See description of MSR_LASTBRANCH_0_FROM_IP.
68AH	1674	MSR_ LASTBRANCH_10_FROM_IP	Core	**Last Branch Record 10 From IP (R/W)** See description of MSR_LASTBRANCH_0_FROM_IP.
68BH	1675	MSR_ LASTBRANCH_11_FROM_IP	Core	**Last Branch Record 11 From IP (R/W)** See description of MSR_LASTBRANCH_0_FROM_IP.
68CH	1676	MSR_ LASTBRANCH_12_FROM_IP	Core	**Last Branch Record 12 From IP (R/W)** See description of MSR_LASTBRANCH_0_FROM_IP.
68DH	1677	MSR_ LASTBRANCH_13_FROM_IP	Core	**Last Branch Record 13 From IP (R/W)** See description of MSR_LASTBRANCH_0_FROM_IP.
68EH	1678	MSR_ LASTBRANCH_14_FROM_IP	Core	**Last Branch Record 14 From IP (R/W)** See description of MSR_LASTBRANCH_0_FROM_IP.
68FH	1679	MSR_ LASTBRANCH_15_FROM_IP	Core	**Last Branch Record 15 From IP (R/W)** See description of MSR_LASTBRANCH_0_FROM_IP.
690H	1680	MSR_ LASTBRANCH_16_FROM_IP	Core	**Last Branch Record 16 From IP (R/W)** See description of MSR_LASTBRANCH_0_FROM_IP.

Table 2-12. MSRs in Next Generation Intel Atom Processors Based on the Goldmont Microarchitecture (Contd.)

Address		Register Name	Scope	Bit Description
Hex	Dec			
691H	1681	MSR_ LASTBRANCH_17_FROM_IP	Core	**Last Branch Record 17 From IP (R/W)** See description of MSR_LASTBRANCH_0_FROM_IP.
692H	1682	MSR_ LASTBRANCH_18_FROM_IP	Core	**Last Branch Record 18 From IP (R/W)** See description of MSR_LASTBRANCH_0_FROM_IP.
693H	1683	MSR_ LASTBRANCH_19_FROM_IP	Core	**Last Branch Record 19From IP (R/W)** See description of MSR_LASTBRANCH_0_FROM_IP.
694H	1684	MSR_ LASTBRANCH_20_FROM_IP	Core	**Last Branch Record 20 From IP (R/W)** See description of MSR_LASTBRANCH_0_FROM_IP.
695H	1685	MSR_ LASTBRANCH_21_FROM_IP	Core	**Last Branch Record 21 From IP (R/W)** See description of MSR_LASTBRANCH_0_FROM_IP.
696H	1686	MSR_ LASTBRANCH_22_FROM_IP	Core	**Last Branch Record 22 From IP (R/W)** See description of MSR_LASTBRANCH_0_FROM_IP.
697H	1687	MSR_ LASTBRANCH_23_FROM_IP	Core	**Last Branch Record 23 From IP (R/W)** See description of MSR_LASTBRANCH_0_FROM_IP.
698H	1688	MSR_ LASTBRANCH_24_FROM_IP	Core	**Last Branch Record 24 From IP (R/W)** See description of MSR_LASTBRANCH_0_FROM_IP.
699H	1689	MSR_ LASTBRANCH_25_FROM_IP	Core	**Last Branch Record 25 From IP (R/W)** See description of MSR_LASTBRANCH_0_FROM_IP.
69AH	1690	MSR_ LASTBRANCH_26_FROM_IP	Core	**Last Branch Record 26 From IP (R/W)** See description of MSR_LASTBRANCH_0_FROM_IP.
69BH	1691	MSR_ LASTBRANCH_27_FROM_IP	Core	**Last Branch Record 27 From IP (R/W)** See description of MSR_LASTBRANCH_0_FROM_IP.
69CH	1692	MSR_ LASTBRANCH_28_FROM_IP	Core	**Last Branch Record 28 From IP (R/W)** See description of MSR_LASTBRANCH_0_FROM_IP.
69DH	1693	MSR_ LASTBRANCH_29_FROM_IP	Core	**Last Branch Record 29 From IP (R/W)** See description of MSR_LASTBRANCH_0_FROM_IP.
69EH	1694	MSR_ LASTBRANCH_30_FROM_IP	Core	**Last Branch Record 30 From IP (R/W)** See description of MSR_LASTBRANCH_0_FROM_IP.
69FH	1695	MSR_ LASTBRANCH_31_FROM_IP	Core	**Last Branch Record 31 From IP (R/W)** See description of MSR_LASTBRANCH_0_FROM_IP.
6C0H	1728	MSR_ LASTBRANCH_0_TO_IP	Core	**Last Branch Record 0 To IP (R/W)** One of 32 pairs of last branch record registers on the last branch record stack. The To_IP part of the stack contains pointers to the **Destination instruction** and elapsed cycles from last LBR update. See also: ▪ Section 17.6
		0:47		**Target Linear Address (R/W)**
		63:48		Elapsed cycles from last update to the LBR.
6C1H	1729	MSR_ LASTBRANCH_1_TO_IP	Core	**Last Branch Record 1 To IP (R/W)** See description of MSR_LASTBRANCH_0_TO_IP.

Table 2-12. MSRs in Next Generation Intel Atom Processors Based on the Goldmont Microarchitecture (Contd.)

Address		Register Name	Scope	Bit Description
Hex	Dec			
6C2H	1730	MSR_ LASTBRANCH_2_TO_IP	Core	**Last Branch Record 2 To IP (R/W)** See description of MSR_LASTBRANCH_0_TO_IP.
6C3H	1731	MSR_ LASTBRANCH_3_TO_IP	Core	**Last Branch Record 3 To IP (R/W)** See description of MSR_LASTBRANCH_0_TO_IP.
6C4H	1732	MSR_ LASTBRANCH_4_TO_IP	Core	**Last Branch Record 4 To IP (R/W)** See description of MSR_LASTBRANCH_0_TO_IP.
6C5H	1733	MSR_ LASTBRANCH_5_TO_IP	Core	**Last Branch Record 5 To IP (R/W)** See description of MSR_LASTBRANCH_0_TO_IP.
6C6H	1734	MSR_ LASTBRANCH_6_TO_IP	Core	**Last Branch Record 6 To IP (R/W)** See description of MSR_LASTBRANCH_0_TO_IP.
6C7H	1735	MSR_ LASTBRANCH_7_TO_IP	Core	**Last Branch Record 7 To IP (R/W)** See description of MSR_LASTBRANCH_0_TO_IP.
6C8H	1736	MSR_ LASTBRANCH_8_TO_IP	Core	**Last Branch Record 8 To IP (R/W)** See description of MSR_LASTBRANCH_0_TO_IP.
6C9H	1737	MSR_ LASTBRANCH_9_TO_IP	Core	**Last Branch Record 9 To IP (R/W)** See description of MSR_LASTBRANCH_0_TO_IP.
6CAH	1738	MSR_ LASTBRANCH_10_TO_IP	Core	**Last Branch Record 10 To IP (R/W)** See description of MSR_LASTBRANCH_0_TO_IP.
6CBH	1739	MSR_ LASTBRANCH_11_TO_IP	Core	**Last Branch Record 11 To IP (R/W)** See description of MSR_LASTBRANCH_0_TO_IP.
6CCH	1740	MSR_ LASTBRANCH_12_TO_IP	Core	**Last Branch Record 12 To IP (R/W)** See description of MSR_LASTBRANCH_0_TO_IP.
6CDH	1741	MSR_ LASTBRANCH_13_TO_IP	Core	**Last Branch Record 13 To IP (R/W)** See description of MSR_LASTBRANCH_0_TO_IP.
6CEH	1742	MSR_ LASTBRANCH_14_TO_IP	Core	**Last Branch Record 14 To IP (R/W)** See description of MSR_LASTBRANCH_0_TO_IP.
6CFH	1743	MSR_ LASTBRANCH_15_TO_IP	Core	**Last Branch Record 15 To IP (R/W)** See description of MSR_LASTBRANCH_0_TO_IP.
6D0H	1744	MSR_ LASTBRANCH_16_TO_IP	Core	**Last Branch Record 16 To IP (R/W)** See description of MSR_LASTBRANCH_0_TO_IP.
6D1H	1745	MSR_ LASTBRANCH_17_TO_IP	Core	**Last Branch Record 17 To IP (R/W)** See description of MSR_LASTBRANCH_0_TO_IP.
6D2H	1746	MSR_ LASTBRANCH_18_TO_IP	Core	**Last Branch Record 18 To IP (R/W)** See description of MSR_LASTBRANCH_0_TO_IP.
6D3H	1747	MSR_ LASTBRANCH_19_TO_IP	Core	**Last Branch Record 19To IP (R/W)** See description of MSR_LASTBRANCH_0_TO_IP.
6D4H	1748	MSR_ LASTBRANCH_20_TO_IP	Core	**Last Branch Record 20 To IP (R/W)** See description of MSR_LASTBRANCH_0_TO_IP.
6D5H	1749	MSR_ LASTBRANCH_21_TO_IP	Core	**Last Branch Record 21 To IP (R/W)** See description of MSR_LASTBRANCH_0_TO_IP.

Table 2-12. MSRs in Next Generation Intel Atom Processors Based on the Goldmont Microarchitecture (Contd.)

Address		Register Name	Scope	Bit Description
Hex	Dec			
6D6H	1750	MSR_ LASTBRANCH_22_TO_IP	Core	**Last Branch Record 22 To IP (R/W)** See description of MSR_LASTBRANCH_0_TO_IP.
6D7H	1751	MSR_ LASTBRANCH_23_TO_IP	Core	**Last Branch Record 23 To IP (R/W)** See description of MSR_LASTBRANCH_0_TO_IP.
6D8H	1752	MSR_ LASTBRANCH_24_TO_IP	Core	**Last Branch Record 24 To IP (R/W)** See description of MSR_LASTBRANCH_0_TO_IP.
6D9H	1753	MSR_ LASTBRANCH_25_TO_IP	Core	**Last Branch Record 25 To IP (R/W)** See description of MSR_LASTBRANCH_0_TO_IP.
6DAH	1754	MSR_ LASTBRANCH_26_TO_IP	Core	**Last Branch Record 26 To IP (R/W)** See description of MSR_LASTBRANCH_0_TO_IP.
6DBH	1755	MSR_ LASTBRANCH_27_TO_IP	Core	**Last Branch Record 27 To IP (R/W)** See description of MSR_LASTBRANCH_0_TO_IP.
6DCH	1756	MSR_ LASTBRANCH_28_TO_IP	Core	**Last Branch Record 28 To IP (R/W)** See description of MSR_LASTBRANCH_0_TO_IP.
6DDH	1757	MSR_ LASTBRANCH_29_TO_IP	Core	**Last Branch Record 29 To IP (R/W)** See description of MSR_LASTBRANCH_0_TO_IP.
6DEH	1758	MSR_ LASTBRANCH_30_TO_IP	Core	**Last Branch Record 30 To IP (R/W)** See description of MSR_LASTBRANCH_0_TO_IP.
6DFH	1759	MSR_ LASTBRANCH_31_TO_IP	Core	**Last Branch Record 31 To IP (R/W)** See description of MSR_LASTBRANCH_0_TO_IP.
802H	2050	IA32_X2APIC_APICID	Core	x2APIC ID register (R/O) See x2APIC Specification.
803H	2051	IA32_X2APIC_VERSION	Core	x2APIC Version register (R/O)
808H	2056	IA32_X2APIC_TPR	Core	x2APIC Task Priority register (R/W)
80AH	2058	IA32_X2APIC_PPR	Core	x2APIC Processor Priority register (R/O)
80BH	2059	IA32_X2APIC_EOI	Core	x2APIC EOI register (W/O)
80DH	2061	IA32_X2APIC_LDR	Core	x2APIC Logical Destination register (R/O)
80FH	2063	IA32_X2APIC_SIVR	Core	x2APIC Spurious Interrupt Vector register (R/W)
810H	2064	IA32_X2APIC_ISR0	Core	x2APIC In-Service register bits [31:0] (R/O)
811H	2065	IA32_X2APIC_ISR1	Core	x2APIC In-Service register bits [63:32] (R/O)
812H	2066	IA32_X2APIC_ISR2	Core	x2APIC In-Service register bits [95:64] (R/O)
813H	2067	IA32_X2APIC_ISR3	Core	x2APIC In-Service register bits [127:96] (R/O)
814H	2068	IA32_X2APIC_ISR4	Core	x2APIC In-Service register bits [159:128] (R/O)
815H	2069	IA32_X2APIC_ISR5	Core	x2APIC In-Service register bits [191:160] (R/O)
816H	2070	IA32_X2APIC_ISR6	Core	x2APIC In-Service register bits [223:192] (R/O)
817H	2071	IA32_X2APIC_ISR7	Core	x2APIC In-Service register bits [255:224] (R/O)
818H	2072	IA32_X2APIC_TMR0	Core	x2APIC Trigger Mode register bits [31:0] (R/O)
819H	2073	IA32_X2APIC_TMR1	Core	x2APIC Trigger Mode register bits [63:32] (R/O)
81AH	2074	IA32_X2APIC_TMR2	Core	x2APIC Trigger Mode register bits [95:64] (R/O)
81BH	2075	IA32_X2APIC_TMR3	Core	x2APIC Trigger Mode register bits [127:96] (R/O)

Table 2-12. MSRs in Next Generation Intel Atom Processors Based on the Goldmont Microarchitecture (Contd.)

Address		Register Name	Scope	Bit Description
Hex	Dec			
81CH	2076	IA32_X2APIC_TMR4	Core	x2APIC Trigger Mode register bits [159:128] (R/O)
81DH	2077	IA32_X2APIC_TMR5	Core	x2APIC Trigger Mode register bits [191:160] (R/O)
81EH	2078	IA32_X2APIC_TMR6	Core	x2APIC Trigger Mode register bits [223:192] (R/O)
81FH	2079	IA32_X2APIC_TMR7	Core	x2APIC Trigger Mode register bits [255:224] (R/O)
820H	2080	IA32_X2APIC_IRR0	Core	x2APIC Interrupt Request register bits [31:0] (R/O)
821H	2081	IA32_X2APIC_IRR1	Core	x2APIC Interrupt Request register bits [63:32] (R/O)
822H	2082	IA32_X2APIC_IRR2	Core	x2APIC Interrupt Request register bits [95:64] (R/O)
823H	2083	IA32_X2APIC_IRR3	Core	x2APIC Interrupt Request register bits [127:96] (R/O)
824H	2084	IA32_X2APIC_IRR4	Core	x2APIC Interrupt Request register bits [159:128] (R/O)
825H	2085	IA32_X2APIC_IRR5	Core	x2APIC Interrupt Request register bits [191:160] (R/O)
826H	2086	IA32_X2APIC_IRR6	Core	x2APIC Interrupt Request register bits [223:192] (R/O)
827H	2087	IA32_X2APIC_IRR7	Core	x2APIC Interrupt Request register bits [255:224] (R/O)
828H	2088	IA32_X2APIC_ESR	Core	x2APIC Error Status register (R/W)
82FH	2095	IA32_X2APIC_LVT_CMCI	Core	x2APIC LVT Corrected Machine Check Interrupt register (R/W)
830H	2096	IA32_X2APIC_ICR	Core	x2APIC Interrupt Command register (R/W)
832H	2098	IA32_X2APIC_LVT_TIMER	Core	x2APIC LVT Timer Interrupt register (R/W)
833H	2099	IA32_X2APIC_LVT_THERMAL	Core	x2APIC LVT Thermal Sensor Interrupt register (R/W)
834H	2100	IA32_X2APIC_LVT_PMI	Core	x2APIC LVT Performance Monitor register (R/W)
835H	2101	IA32_X2APIC_LVT_LINT0	Core	x2APIC LVT LINT0 register (R/W)
836H	2102	IA32_X2APIC_LVT_LINT1	Core	x2APIC LVT LINT1 register (R/W)
837H	2103	IA32_X2APIC_LVT_ERROR	Core	x2APIC LVT Error register (R/W)
838H	2104	IA32_X2APIC_INIT_COUNT	Core	x2APIC Initial Count register (R/W)
839H	2105	IA32_X2APIC_CUR_COUNT	Core	x2APIC Current Count register (R/O)
83EH	2110	IA32_X2APIC_DIV_CONF	Core	x2APIC Divide Configuration register (R/W)
83FH	2111	IA32_X2APIC_SELF_IPI	Core	x2APIC Self IPI register (W/O)
C8FH	3215	IA32_PQR_ASSOC	Core	**Resource Association Register (R/W)**
		31:0		**Reserved**
		33:32		**COS (R/W).**
		63: 34		**Reserved**
D10H	3344	IA32_L2_QOS_MASK_0	Module	**L2 Class Of Service Mask - COS 0 (R/W)** if CPUID.(EAX=10H, ECX=1):EDX.COS_MAX[15:0] >=0
		0:7		**CBM: Bit vector of available L2 ways for COS 0 enforcement**
		63:8		**Reserved**
D11H	3345	IA32_L2_QOS_MASK_1	Module	**L2 Class Of Service Mask - COS 1 (R/W)** if CPUID.(EAX=10H, ECX=1):EDX.COS_MAX[15:0] >=1
		0:7		**CBM: Bit vector of available L2 ways for COS 0 enforcement**
		63:8		**Reserved**

Table 2-12. MSRs in Next Generation Intel Atom Processors Based on the Goldmont Microarchitecture (Contd.)

Address		Register Name	Scope	Bit Description
Hex	Dec			
D12H	3346	IA32_L2_QOS_MASK_2	Module	**L2 Class Of Service Mask - COS 2 (R/W)** if CPUID.(EAX=10H, ECX=1):EDX.COS_MAX[15:0] >=2
		0:7		**CBM: Bit vector of available L2 ways for COS 0 enforcement**
		63:8		**Reserved**
D13H	3347	IA32_L2_QOS_MASK_3	Package	**L2 Class Of Service Mask - COS 3 (R/W).** if CPUID.(EAX=10H, ECX=1):EDX.COS_MAX[15:0] >=3
		0:19		**CBM: Bit vector of available L2 ways for COS 3 enforcement**
		63:20		**Reserved**
D90H	3472	IA32_BNDCFGS	Core	See Table 2-2.
DA0H	3488	IA32_XSS	Core	See Table 2-2.
See Table 2-6, and Table 2-12 for MSR definitions applicable to processors with CPUID signature 06_5CH.				

2.6 MSRS IN INTEL ATOM PROCESSORS BASED ON GOLDMONT PLUS MICROARCHITECTURE

Intel Atom processors based on the Goldmont Plus microarchitecture support MSRs listed in Table 2-6, Table 2-12 and Table 2-13. These processors have a CPUID signature with DisplayFamily_DisplayModel of 06_7AH; see Table 2-1. For an MSR listed in Table 2-13 that also appears in the model-specific tables of prior generations, Table 2-13 supercede prior generation tables.

In the Goldmont Plus microarchitecture, the scope column indicates the following: "Core" means each processor core has a separate MSR, or a bit field not shared with another processor core. "Module" means the MSR or the bit field is shared by a pair of processor cores in the physical package. "Package" means all processor cores in the physical package share the same MSR or bit interface.

Table 2-13. MSRs in Intel Atom Processors Based on the Goldmont Plus Microarchitecture

Address		Register Name	Scope	Bit Description
Hex	Dec			
3AH	58	IA32_FEATURE_CONTROL	Core	**Control Features in Intel 64Processor (R/W)** See Table 2-2.
		0		**Lock (R/WL)**
		1		**Enable VMX inside SMX operation (R/WL)**
		2		**Enable VMX outside SMX operation (R/WL)**
		14:8		**SENTER local functions enables (R/WL)**
		15		**SENTER global functions enable (R/WL)**
		17		**SGX Launch Control Enable (R/WL)** This bit must be set to enable runtime reconfiguration of SGX Launch Control via IA32_SGXLEPUBKEYHASHn MSR. Valid if CPUID.(EAX=07H, ECX=0H): ECX[30] = 1.
		18		**SGX global functions enable (R/WL)**
		63:19		Reserved.

Table 2-13. MSRs in Intel Atom Processors Based on the Goldmont Plus Microarchitecture (Contd.)

Address		Register Name	Scope	Bit Description
Hex	Dec			
8CH	140	IA32_SGXLEPUBKEYHASH0	Core	See Table 2-2.
8DH	141	IA32_SGXLEPUBKEYHASH1	Core	See Table 2-2.
8EH	142	IA32_SGXLEPUBKEYHASH2	Core	See Table 2-2.
8FH	143	IA32_SGXLEPUBKEYHASH3	Core	See Table 2-2.
3F1H	1009	MSR_PEBS_ENABLE	Core	See Table 2-2. See Section 18.6.2.4, "Processor Event Based Sampling (PEBS)."
		0		Enable PEBS trigger and recording for the programmed event (precise or otherwise) on IA32_PMC0. (R/W)
		1		Enable PEBS trigger and recording for the programmed event (precise or otherwise) on IA32_PMC1.
		2		Enable PEBS trigger and recording for the programmed event (precise or otherwise) on IA32_PMC2.
		3		Enable PEBS trigger and recording for the programmed event (precise or otherwise) on IA32_PMC3.
		31:4		Reserved.
		32		Enable PEBS trigger and recording for IA32_FIXED_CTR0.
		33		Enable PEBS trigger and recording for IA32_FIXED_CTR1.
		34		Enable PEBS trigger and recording for IA32_FIXED_CTR2.
		63:35		Reserved.
570H	1392	IA32_RTIT_CTL	Core	**Trace Control Register (R/W)**
		0		**TraceEn**
		1		**CYCEn**
		2		**OS**
		3		**User**
		4		**PwrEvtEn**
		5		**FUPonPTW**
		6		**FabricEn**
		7		**CR3 filter**
		8		**ToPA; writing 0 will #GP if also setting TraceEn**
		9		**MTCEn**
		10		**TSCEn**
		11		**DisRETC**
		12		**PTWEn**
		13		**BranchEn**
		17:14		**MTCFreq**
		18		Reserved, MBZ
		22:19		**CYCThresh**
		23		Reserved, MBZ
		27:24		**PSBFreq**

Table 2-13. MSRs in Intel Atom Processors Based on the Goldmont Plus Microarchitecture (Contd.)

Address		Register Name	Scope	Bit Description
Hex	Dec			
		31:28		Reserved, MBZ
		35:32		**ADDR0_CFG**
		39:36		**ADDR1_CFG**
		63:40		Reserved, MBZ.
680H	1664	MSR_ LASTBRANCH_0_FROM_IP	Core	**Last Branch Record 0 From IP (R/W)** One of the three MSRs that make up the first entry of the 32-entry LBR stack. The From_IP part of the stack contains pointers to the source instruction . See also: ▪ Last Branch Record Stack TOS at 1C9H ▪ Section 17.7, "Last Branch, Call Stack, Interrupt, and Exception Recording for Processors based on Goldmont Plus Microarchitecture."
681H - 69FH	1665 - 1695	MSR_ LASTBRANCH_*i*_FROM_IP	Core	**Last Branch Record *i* From IP (R/W)** See description of MSR_LASTBRANCH_0_FROM_IP; *i* = 1-31.
6C0H	1728	MSR_ LASTBRANCH_0_TO_IP	Core	**Last Branch Record 0 To IP (R/W)** One of the 3 MSRs that make up the first entry of the 32-entry LBR stack. The To_IP part of the stack contains pointers to the Destination instruction. See also: ▪ Section 17.7, "Last Branch, Call Stack, Interrupt, and Exception Recording for Processors based on Goldmont Plus Microarchitecture."
6C1H - 6DFH	1729 - 1759	MSR_ LASTBRANCH_*i*_TO_IP	Core	**Last Branch Record *i* To IP (R/W)** See description of MSR_LASTBRANCH_0_TO_IP; *i* = 1-31.
DC0H	3520	MSR_LASTBRANCH_INFO_ 0	Core	**Last Branch Record 0 Additional Information (R/W)** One of the 3 MSRs that make up the first entry of the 32-entry LBR stack. This part of the stack contains flag and elapsed cycle information. See also: ▪ Last Branch Record Stack TOS at 1C9H ▪ Section 17.9.1, "LBR Stack."
DC1H	3521	MSR_LASTBRANCH_INFO_ 1	Core	**Last Branch Record 1 Additional Information (R/W)** See description of MSR_LASTBRANCH_INFO_0.
DC2H	3522	MSR_LASTBRANCH_INFO_ 2	Core	**Last Branch Record 2 Additional Information (R/W)** See description of MSR_LASTBRANCH_INFO_0.
DC3H	3523	MSR_LASTBRANCH_INFO_ 3	Core	**Last Branch Record 3 Additional Information (R/W)** See description of MSR_LASTBRANCH_INFO_0.
DC4H	3524	MSR_LASTBRANCH_INFO_ 4	Core	**Last Branch Record 4 Additional Information (R/W)** See description of MSR_LASTBRANCH_INFO_0.
DC5H	3525	MSR_LASTBRANCH_INFO_ 5	Core	**Last Branch Record 5 Additional Information (R/W)** See description of MSR_LASTBRANCH_INFO_0.
DC6H	3526	MSR_LASTBRANCH_INFO_ 6	Core	**Last Branch Record 6 Additional Information (R/W)** See description of MSR_LASTBRANCH_INFO_0.

Table 2-13. MSRs in Intel Atom Processors Based on the Goldmont Plus Microarchitecture (Contd.)

Address		Register Name	Scope	Bit Description
Hex	Dec			
DC7H	3527	MSR_LASTBRANCH_INFO_7	Core	**Last Branch Record 7 Additional Information (R/W)** See description of MSR_LASTBRANCH_INFO_0.
DC8H	3528	MSR_LASTBRANCH_INFO_8	Core	**Last Branch Record 8 Additional Information (R/W)** See description of MSR_LASTBRANCH_INFO_0.
DC9H	3529	MSR_LASTBRANCH_INFO_9	Core	**Last Branch Record 9 Additional Information (R/W)** See description of MSR_LASTBRANCH_INFO_0.
DCAH	3530	MSR_LASTBRANCH_INFO_10	Core	**Last Branch Record 10 Additional Information (R/W)** See description of MSR_LASTBRANCH_INFO_0.
DCBH	3531	MSR_LASTBRANCH_INFO_11	Core	**Last Branch Record 11 Additional Information (R/W)** See description of MSR_LASTBRANCH_INFO_0.
DCCH	3532	MSR_LASTBRANCH_INFO_12	Core	**Last Branch Record 12 Additional Information (R/W)** See description of MSR_LASTBRANCH_INFO_0.
DCDH	3533	MSR_LASTBRANCH_INFO_13	Core	**Last Branch Record 13 Additional Information (R/W)** See description of MSR_LASTBRANCH_INFO_0.
DCEH	3534	MSR_LASTBRANCH_INFO_14	Core	**Last Branch Record 14 Additional Information (R/W)** See description of MSR_LASTBRANCH_INFO_0.
DCFH	3535	MSR_LASTBRANCH_INFO_15	Core	**Last Branch Record 15 Additional Information (R/W)** See description of MSR_LASTBRANCH_INFO_0.
DD0H	3536	MSR_LASTBRANCH_INFO_16	Core	**Last Branch Record 16 Additional Information (R/W)** See description of MSR_LASTBRANCH_INFO_0.
DD1H	3537	MSR_LASTBRANCH_INFO_17	Core	**Last Branch Record 17 Additional Information (R/W)** See description of MSR_LASTBRANCH_INFO_0.
DD2H	3538	MSR_LASTBRANCH_INFO_18	Core	**Last Branch Record 18 Additional Information (R/W)** See description of MSR_LASTBRANCH_INFO_0.
DD3H	3539	MSR_LASTBRANCH_INFO_19	Core	**Last Branch Record 19 Additional Information (R/W)** See description of MSR_LASTBRANCH_INFO_0.
DD4H	3520	MSR_LASTBRANCH_INFO_20	Core	**Last Branch Record 20 Additional Information (R/W)** See description of MSR_LASTBRANCH_INFO_0.
DD5H	3521	MSR_LASTBRANCH_INFO_21	Core	**Last Branch Record 21 Additional Information (R/W)** See description of MSR_LASTBRANCH_INFO_0.
DD6H	3522	MSR_LASTBRANCH_INFO_22	Core	**Last Branch Record 22 Additional Information (R/W)** See description of MSR_LASTBRANCH_INFO_0.
DD7H	3523	MSR_LASTBRANCH_INFO_23	Core	**Last Branch Record 23 Additional Information (R/W)** See description of MSR_LASTBRANCH_INFO_0.
DD8H	3524	MSR_LASTBRANCH_INFO_24	Core	**Last Branch Record 24 Additional Information (R/W)** See description of MSR_LASTBRANCH_INFO_0.
DD9H	3525	MSR_LASTBRANCH_INFO_25	Core	**Last Branch Record 25 Additional Information (R/W)** See description of MSR_LASTBRANCH_INFO_0.
DDAH	3526	MSR_LASTBRANCH_INFO_26	Core	**Last Branch Record 26 Additional Information (R/W)** See description of MSR_LASTBRANCH_INFO_0.

Table 2-13. MSRs in Intel Atom Processors Based on the Goldmont Plus Microarchitecture (Contd.)

Address		Register Name	Scope	Bit Description
Hex	Dec			
DDBH	3527	MSR_LASTBRANCH_INFO_27	Core	**Last Branch Record 27 Additional Information (R/W)** See description of MSR_LASTBRANCH_INFO_0.
DDCH	3528	MSR_LASTBRANCH_INFO_28	Core	**Last Branch Record 28 Additional Information (R/W)** See description of MSR_LASTBRANCH_INFO_0.
DDDH	3529	MSR_LASTBRANCH_INFO_29	Core	**Last Branch Record 29 Additional Information (R/W)** See description of MSR_LASTBRANCH_INFO_0.
DDEH	3530	MSR_LASTBRANCH_INFO_30	Core	**Last Branch Record 30 Additional Information (R/W)** See description of MSR_LASTBRANCH_INFO_0.
DDFH	3531	MSR_LASTBRANCH_INFO_31	Core	**Last Branch Record 31 Additional Information (R/W)** See description of MSR_LASTBRANCH_INFO_0.
See Table 2-6, Table 2-12 and Table 2-13 for MSR definitions applicable to processors with CPUID signature 06_7AH.				

2.7 MSRS IN THE INTEL® MICROARCHITECTURE CODE NAME NEHALEM

Table 2-14 lists model-specific registers (MSRs) that are common for Intel® microarchitecture code name Nehalem. These include Intel Core i7 and i5 processor family. These processors have a CPUID signature with DisplayFamily_DisplayModel of 06_1AH, 06_1EH, 06_1FH, 06_2EH, see Table 2-1. Additional MSRs specific to 06_1AH, 06_1EH, 06_1FH are listed in Table 2-15. Some MSRs listed in these tables are used by BIOS. More information about these MSR can be found at http://biosbits.org.

The column "Scope" represents the package/core/thread scope of individual bit field of an MSR. "Thread" means this bit field must be programmed on each logical processor independently. "Core" means the bit field must be programmed on each processor core independently, logical processors in the same core will be affected by change of this bit on the other logical processor in the same core. "Package" means the bit field must be programmed once for each physical package. Change of a bit filed with a package scope will affect all logical processors in that physical package.

Table 2-14. MSRs in Processors Based on Intel® Microarchitecture Code Name Nehalem

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
0H	0	IA32_P5_MC_ADDR	Thread	See Section 2.22, "MSRs in Pentium Processors."
1H	1	IA32_P5_MC_TYPE	Thread	See Section 2.22, "MSRs in Pentium Processors."
6H	6	IA32_MONITOR_FILTER_SIZE	Thread	See Section 8.10.5, "Monitor/Mwait Address Range Determination," and Table 2-2.
10H	16	IA32_TIME_STAMP_COUNTER	Thread	See Section 17.17, "Time-Stamp Counter," and see Table 2-2.
17H	23	IA32_PLATFORM_ID	Package	**Platform ID (R)** See Table 2-2.
17H	23	MSR_PLATFORM_ID	Package	**Model Specific Platform ID (R)**
		49:0		Reserved.
		52:50		See Table 2-2.

Table 2-14. MSRs in Processors Based on Intel® Microarchitecture Code Name Nehalem (Contd.)

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
		63:53		Reserved.
1BH	27	IA32_APIC_BASE	Thread	See Section 10.4.4, "Local APIC Status and Location," and Table 2-2.
34H	52	MSR_SMI_COUNT	Thread	**SMI Counter (R/O)**
		31:0		**SMI Count (R/O)** Running count of SMI events since last RESET.
		63:32		Reserved.
3AH	58	IA32_FEATURE_CONTROL	Thread	**Control Features in Intel 64Processor (R/W)** See Table 2-2.
79H	121	IA32_BIOS_ UPDT_TRIG	Core	**BIOS Update Trigger Register (W)** See Table 2-2.
8BH	139	IA32_BIOS_ SIGN_ID	Thread	**BIOS Update Signature ID (RO)** See Table 2-2.
C1H	193	IA32_PMC0	Thread	**Performance Counter Register** See Table 2-2.
C2H	194	IA32_PMC1	Thread	**Performance Counter Register** See Table 2-2.
C3H	195	IA32_PMC2	Thread	**Performance Counter Register** See Table 2-2.
C4H	196	IA32_PMC3	Thread	**Performance Counter Register** See Table 2-2.
CEH	206	MSR_PLATFORM_INFO	Package	Platform Information; contains power management and other model specific features enumeration. See http://biosbits.org.
		7:0		Reserved.
		15:8	Package	**Maximum Non-Turbo Ratio (R/O)** The is the ratio of the frequency that invariant TSC runs at. The invariant TSC frequency can be computed by multiplying this ratio by 133.33 MHz.
		27:16		Reserved.
		28	Package	**Programmable Ratio Limit for Turbo Mode (R/O)** When set to 1, indicates that Programmable Ratio Limits for Turbo mode is enabled, and when set to 0, indicates Programmable Ratio Limits for Turbo mode is disabled.
		29	Package	**Programmable TDC-TDP Limit for Turbo Mode (R/O)** When set to 1, indicates that TDC/TDP Limits for Turbo mode are programmable, and when set to 0, indicates TDC and TDP Limits for Turbo mode are not programmable.
		39:30		Reserved.

Table 2-14. MSRs in Processors Based on Intel® Microarchitecture Code Name Nehalem (Contd.)

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
		47:40	Package	**Maximum Efficiency Ratio (R/O)** The is the minimum ratio (maximum efficiency) that the processor can operates, in units of 133.33MHz.
		63:48		Reserved.
E2H	226	MSR_PKG_CST_CONFIG_ CONTROL	Core	**C-State Configuration Control (R/W)** Note: C-state values are processor specific C-state code names, unrelated to MWAIT extension C-state parameters or ACPI C-States. See http://biosbits.org.
		2:0		**Package C-State Limit (R/W)** Specifies the lowest processor-specific C-state code name (consuming the least power). for the package. The default is set as factory-configured package C-state limit. The following C-state code name encodings are supported: 000b: C0 (no package C-sate support) 001b: C1 (Behavior is the same as 000b) 010b: C3 011b: C6 100b: C7 101b and 110b: Reserved 111: No package C-state limit. Note: This field cannot be used to limit package C-state to C3.
		9:3		Reserved.
		10		**I/O MWAIT Redirection Enable (R/W)** When set, will map IO_read instructions sent to IO register specified by MSR_PMG_IO_CAPTURE_BASE to MWAIT instructions.
		14:11		Reserved.
		15		**CFG Lock (R/WO)** When set, lock bits 15:0 of this register until next reset.
		23:16		Reserved.
		24		**Interrupt filtering enable (R/W)** When set, processor cores in a deep C-State will wake only when the event message is destined for that core. When 0, all processor cores in a deep C-State will wake for an event message.
		25		**C3 state auto demotion enable (R/W)** When set, the processor will conditionally demote C6/C7 requests to C3 based on uncore auto-demote information.
		26		**C1 state auto demotion enable (R/W)** When set, the processor will conditionally demote C3/C6/C7 requests to C1 based on uncore auto-demote information.
		27		**Enable C3 Undemotion (R/W)**
		28		**Enable C1 Undemotion (R/W)**

Table 2-14. MSRs in Processors Based on Intel® Microarchitecture Code Name Nehalem (Contd.)

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
		29		**Package C State Demotion Enable (R/W)**
		30		**Package C State UnDemotion Enable (R/W)**
		63:31		Reserved.
E4H	228	MSR_PMG_IO_CAPTURE_BASE	Core	**Power Management IO Redirection in C-state (R/W)** See http://biosbits.org.
		15:0		**LVL_2 Base Address (R/W)** Specifies the base address visible to software for IO redirection. If IO MWAIT Redirection is enabled, reads to this address will be consumed by the power management logic and decoded to MWAIT instructions. When IO port address redirection is enabled, this is the IO port address reported to the OS/software.
		18:16		**C-state Range (R/W)** Specifies the encoding value of the maximum C-State code name to be included when IO read to MWAIT redirection is enabled by MSR_PKG_CST_CONFIG_CONTROL[bit10]: 000b - C3 is the max C-State to include 001b - C6 is the max C-State to include 010b - C7 is the max C-State to include
		63:19		Reserved.
E7H	231	IA32_MPERF	Thread	**Maximum Performance Frequency Clock Count (RW)** See Table 2-2.
E8H	232	IA32_APERF	Thread	**Actual Performance Frequency Clock Count (RW)** See Table 2-2.
FEH	254	IA32_MTRRCAP	Thread	See Table 2-2.
174H	372	IA32_SYSENTER_CS	Thread	See Table 2-2.
175H	373	IA32_SYSENTER_ESP	Thread	See Table 2-2.
176H	374	IA32_SYSENTER_EIP	Thread	See Table 2-2.
179H	377	IA32_MCG_CAP	Thread	See Table 2-2.
17AH	378	IA32_MCG_STATUS	Thread	Global Machine Check Status
		0		**RIPV** When set, bit indicates that the instruction addressed by the instruction pointer pushed on the stack (when the machine check was generated) can be used to restart the program. If cleared, the program cannot be reliably restarted.
		1		**EIPV** When set, bit indicates that the instruction addressed by the instruction pointer pushed on the stack (when the machine check was generated) is directly associated with the error.

Table 2-14. MSRs in Processors Based on Intel® Microarchitecture Code Name Nehalem (Contd.)

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
		2		**MCIP** When set, bit indicates that a machine check has been generated. If a second machine check is detected while this bit is still set, the processor enters a shutdown state. Software should write this bit to 0 after processing a machine check exception.
		63:3		Reserved.
186H	390	IA32_PERFEVTSEL0	Thread	See Table 2-2.
		7:0		**Event Select**
		15:8		**UMask**
		16		**USR**
		17		**OS**
		18		**Edge**
		19		**PC**
		20		**INT**
		21		**AnyThread**
		22		**EN**
		23		**INV**
		31:24		**CMASK**
		63:32		Reserved.
187H	391	IA32_PERFEVTSEL1	Thread	See Table 2-2.
188H	392	IA32_PERFEVTSEL2	Thread	See Table 2-2.
189H	393	IA32_PERFEVTSEL3	Thread	See Table 2-2.
198H	408	IA32_PERF_STATUS	Core	See Table 2-2.
		15:0		Current Performance State Value.
		63:16		Reserved.
199H	409	IA32_PERF_CTL	Thread	See Table 2-2.
19AH	410	IA32_CLOCK_MODULATION	Thread	**Clock Modulation (R/W)** See Table 2-2. IA32_CLOCK_MODULATION MSR was originally named IA32_THERM_CONTROL MSR.
		0		Reserved.
		3:1		**On demand Clock Modulation Duty Cycle (R/W)**
		4		**On demand Clock Modulation Enable (R/W)**
		63:5		Reserved.
19BH	411	IA32_THERM_INTERRUPT	Core	**Thermal Interrupt Control (R/W)** See Table 2-2.
19CH	412	IA32_THERM_STATUS	Core	**Thermal Monitor Status (R/W)** See Table 2-2.

Table 2-14. MSRs in Processors Based on Intel® Microarchitecture Code Name Nehalem (Contd.)

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
1A0H	416	IA32_MISC_ENABLE		**Enable Misc. Processor Features (R/W)** Allows a variety of processor functions to be enabled and disabled.
		0	Thread	**Fast-Strings Enable** See Table 2-2.
		2:1		Reserved.
		3	Thread	**Automatic Thermal Control Circuit Enable (R/W)** See Table 2-2. Default value is 1.
		6:4		Reserved.
		7	Thread	**Performance Monitoring Available (R)** See Table 2-2.
		10:8		Reserved.
		11	Thread	**Branch Trace Storage Unavailable (RO)** See Table 2-2.
		12	Thread	**Processor Event Based Sampling Unavailable (RO)** See Table 2-2.
		15:13		Reserved.
		16	Package	**Enhanced Intel SpeedStep Technology Enable (R/W)** See Table 2-2.
		18	Thread	ENABLE MONITOR FSM. (R/W) See Table 2-2.
		21:19		Reserved.
		22	Thread	**Limit CPUID Maxval (R/W)** See Table 2-2.
		23	Thread	**xTPR Message Disable (R/W)** See Table 2-2.
		33:24		Reserved.
		34	Thread	**XD Bit Disable (R/W)** See Table 2-2.
		37:35		Reserved.
		38	Package	**Turbo Mode Disable (R/W)** When set to 1 on processors that support Intel Turbo Boost Technology, the turbo mode feature is disabled and the IDA_Enable feature flag will be clear (CPUID.06H: EAX[1]=0). When set to a 0 on processors that support IDA, CPUID.06H: EAX[1] reports the processor's support of turbo mode is enabled. **Note:** the power-on default value is used by BIOS to detect hardware support of turbo mode. If power-on default value is 1, turbo mode is available in the processor. If power-on default value is 0, turbo mode is not available.
		63:39		Reserved.

Table 2-14. MSRs in Processors Based on Intel® Microarchitecture Code Name Nehalem (Contd.)

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
1A2H	418	MSR_ TEMPERATURE_TARGET	Thread	Temperature Target
		15:0		Reserved.
		23:16		**Temperature Target (R)** The minimum temperature at which PROCHOT# will be asserted. The value is degree C.
		63:24		Reserved.
1A4H	420	MSR_MISC_FEATURE_ CONTROL		**Miscellaneous Feature Control (R/W)**
		0	Core	**L2 Hardware Prefetcher Disable (R/W)** If 1, disables the L2 hardware prefetcher, which fetches additional lines of code or data into the L2 cache.
		1	Core	**L2 Adjacent Cache Line Prefetcher Disable (R/W)** If 1, disables the adjacent cache line prefetcher, which fetches the cache line that comprises a cache line pair (128 bytes).
		2	Core	**DCU Hardware Prefetcher Disable (R/W)** If 1, disables the L1 data cache prefetcher, which fetches the next cache line into L1 data cache.
		3	Core	**DCU IP Prefetcher Disable (R/W)** If 1, disables the L1 data cache IP prefetcher, which uses sequential load history (based on instruction Pointer of previous loads) to determine whether to prefetch additional lines.
		63:4		Reserved.
1A6H	422	MSR_OFFCORE_RSP_0	Thread	**Offcore Response Event Select Register (R/W)**
1AAH	426	MSR_MISC_PWR_MGMT		Miscellaneous Power Management Control; Various model specific features enumeration. See http://biosbits.org.
		0	Package	**EIST Hardware Coordination Disable (R/W)** When 0, enables hardware coordination of Enhanced Intel Speedstep Technology request from processor cores; When 1, disables hardware coordination of Enhanced Intel Speedstep Technology requests.
		1	Thread	**Energy/Performance Bias Enable (R/W)** This bit makes the IA32_ENERGY_PERF_BIAS register (MSR 1B0h) visible to software with Ring 0 privileges. This bit's status (1 or 0) is also reflected by CPUID.(EAX=06h):ECX[3].
		63:2		Reserved.
1ACH	428	MSR_TURBO_POWER_ CURRENT_LIMIT		See http://biosbits.org.
		14:0	Package	**TDP Limit (R/W)** TDP limit in 1/8 Watt granularity.

Table 2-14. MSRs in Processors Based on Intel® Microarchitecture Code Name Nehalem (Contd.)

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
		15	Package	**TDP Limit Override Enable (R/W)** A value = 0 indicates override is not active, and a value = 1 indicates active.
		30:16	Package	**TDC Limit (R/W)** TDC limit in 1/8 Amp granularity.
		31	Package	**TDC Limit Override Enable (R/W)** A value = 0 indicates override is not active, and a value = 1 indicates active.
		63:32		Reserved.
1ADH	429	MSR_TURBO_RATIO_LIMIT	Package	**Maximum Ratio Limit of Turbo Mode** **RO** if MSR_PLATFORM_INFO.[28] = 0, **RW** if MSR_PLATFORM_INFO.[28] = 1
		7:0	Package	**Maximum Ratio Limit for 1C** Maximum turbo ratio limit of 1 core active.
		15:8	Package	**Maximum Ratio Limit for 2C** Maximum turbo ratio limit of 2 core active.
		23:16	Package	**Maximum Ratio Limit for 3C** Maximum turbo ratio limit of 3 core active.
		31:24	Package	**Maximum Ratio Limit for 4C** Maximum turbo ratio limit of 4 core active.
		63:32		Reserved.
1C8H	456	MSR_LBR_SELECT	Core	**Last Branch Record Filtering Select Register (R/W)** See Section 17.9.2, "Filtering of Last Branch Records."
		0		**CPL_EQ_0**
		1		**CPL_NEQ_0**
		2		**JCC**
		3		**NEAR_REL_CALL**
		4		**NEAR_IND_CALL**
		5		**NEAR_RET**
		6		**NEAR_IND_JMP**
		7		**NEAR_REL_JMP**
		8		**FAR_BRANCH**
		63:9		Reserved.
1C9H	457	MSR_LASTBRANCH_TOS	Thread	**Last Branch Record Stack TOS (R/W)** Contains an index (bits 0-3) that points to the MSR containing the most recent branch record. See MSR_LASTBRANCH_0_FROM_IP (at 680H).

Table 2-14. MSRs in Processors Based on Intel® Microarchitecture Code Name Nehalem (Contd.)

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
1D9H	473	IA32_DEBUGCTL	Thread	**Debug Control (R/W)** See Table 2-2.
1DDH	477	MSR_LER_FROM_LIP	Thread	**Last Exception Record From Linear IP (R)** Contains a pointer to the last branch instruction that the processor executed prior to the last exception that was generated or the last interrupt that was handled.
1DEH	478	MSR_LER_TO_LIP	Thread	**Last Exception Record To Linear IP (R)** This area contains a pointer to the target of the last branch instruction that the processor executed prior to the last exception that was generated or the last interrupt that was handled.
1F2H	498	IA32_SMRR_PHYSBASE	Core	See Table 2-2.
1F3H	499	IA32_SMRR_PHYSMASK	Core	See Table 2-2.
1FCH	508	MSR_POWER_CTL	Core	Power Control Register. See http://biosbits.org.
		0		Reserved.
		1	Package	**C1E Enable (R/W)** When set to '1', will enable the CPU to switch to the Minimum Enhanced Intel SpeedStep Technology operating point when all execution cores enter MWAIT (C1).
		63:2		Reserved.
200H	512	IA32_MTRR_PHYSBASE0	Thread	See Table 2-2.
201H	513	IA32_MTRR_PHYSMASK0	Thread	See Table 2-2.
202H	514	IA32_MTRR_PHYSBASE1	Thread	See Table 2-2.
203H	515	IA32_MTRR_PHYSMASK1	Thread	See Table 2-2.
204H	516	IA32_MTRR_PHYSBASE2	Thread	See Table 2-2.
205H	517	IA32_MTRR_PHYSMASK2	Thread	See Table 2-2.
206H	518	IA32_MTRR_PHYSBASE3	Thread	See Table 2-2.
207H	519	IA32_MTRR_PHYSMASK3	Thread	See Table 2-2.
208H	520	IA32_MTRR_PHYSBASE4	Thread	See Table 2-2.
209H	521	IA32_MTRR_PHYSMASK4	Thread	See Table 2-2.
20AH	522	IA32_MTRR_PHYSBASE5	Thread	See Table 2-2.
20BH	523	IA32_MTRR_PHYSMASK5	Thread	See Table 2-2.
20CH	524	IA32_MTRR_PHYSBASE6	Thread	See Table 2-2.
20DH	525	IA32_MTRR_PHYSMASK6	Thread	See Table 2-2.
20EH	526	IA32_MTRR_PHYSBASE7	Thread	See Table 2-2.
20FH	527	IA32_MTRR_PHYSMASK7	Thread	See Table 2-2.
210H	528	IA32_MTRR_PHYSBASE8	Thread	See Table 2-2.

Table 2-14. MSRs in Processors Based on Intel® Microarchitecture Code Name Nehalem (Contd.)

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
211H	529	IA32_MTRR_PHYSMASK8	Thread	See Table 2-2.
212H	530	IA32_MTRR_PHYSBASE9	Thread	See Table 2-2.
213H	531	IA32_MTRR_PHYSMASK9	Thread	See Table 2-2.
250H	592	IA32_MTRR_FIX64K_ 00000	Thread	See Table 2-2.
258H	600	IA32_MTRR_FIX16K_ 80000	Thread	See Table 2-2.
259H	601	IA32_MTRR_FIX16K_ A0000	Thread	See Table 2-2.
268H	616	IA32_MTRR_FIX4K_C0000	Thread	See Table 2-2.
269H	617	IA32_MTRR_FIX4K_C8000	Thread	See Table 2-2.
26AH	618	IA32_MTRR_FIX4K_D0000	Thread	See Table 2-2.
26BH	619	IA32_MTRR_FIX4K_D8000	Thread	See Table 2-2.
26CH	620	IA32_MTRR_FIX4K_E0000	Thread	See Table 2-2.
26DH	621	IA32_MTRR_FIX4K_E8000	Thread	See Table 2-2.
26EH	622	IA32_MTRR_FIX4K_F0000	Thread	See Table 2-2.
26FH	623	IA32_MTRR_FIX4K_F8000	Thread	See Table 2-2.
277H	631	IA32_PAT	Thread	See Table 2-2.
280H	640	IA32_MC0_CTL2	Package	See Table 2-2.
281H	641	IA32_MC1_CTL2	Package	See Table 2-2.
282H	642	IA32_MC2_CTL2	Core	See Table 2-2.
283H	643	IA32_MC3_CTL2	Core	See Table 2-2.
284H	644	IA32_MC4_CTL2	Core	See Table 2-2.
285H	645	IA32_MC5_CTL2	Core	See Table 2-2.
286H	646	IA32_MC6_CTL2	Package	See Table 2-2.
287H	647	IA32_MC7_CTL2	Package	See Table 2-2.
288H	648	IA32_MC8_CTL2	Package	See Table 2-2.
2FFH	767	IA32_MTRR_DEF_TYPE	Thread	**Default Memory Types (R/W)** See Table 2-2.
309H	777	IA32_FIXED_CTR0	Thread	**Fixed-Function Performance Counter Register 0 (R/W)** See Table 2-2.
30AH	778	IA32_FIXED_CTR1	Thread	**Fixed-Function Performance Counter Register 1 (R/W)** See Table 2-2.
30BH	779	IA32_FIXED_CTR2	Thread	**Fixed-Function Performance Counter Register 2 (R/W)** See Table 2-2.
345H	837	IA32_PERF_CAPABILITIES	Thread	See Table 2-2. See Section 17.4.1, "IA32_DEBUGCTL MSR."
		5:0		LBR Format. See Table 2-2.

Table 2-14. MSRs in Processors Based on Intel® Microarchitecture Code Name Nehalem (Contd.)

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
		6		PEBS Record Format.
		7		PEBSSaveArchRegs. See Table 2-2.
		11:8		PEBS_REC_FORMAT. See Table 2-2.
		12		SMM_FREEZE. See Table 2-2.
		63:13		Reserved.
38DH	909	IA32_FIXED_CTR_CTRL	Thread	**Fixed-Function-Counter Control Register (R/W)** See Table 2-2.
38EH	910	IA32_PERF_GLOBAL_ STATUS	Thread	See Table 2-2. See Section 18.6.2.2, "Global Counter Control Facilities."
38EH	910	MSR_PERF_GLOBAL_STATUS	Thread	Provides single-bit status used by software to query the overflow condition of each performance counter. (RO)
		61		**UNC_Ovf** Uncore overflowed if 1.
38FH	911	IA32_PERF_GLOBAL_CTRL	Thread	See Table 2-2. See Section 18.6.2.2, "Global Counter Control Facilities."
390H	912	IA32_PERF_GLOBAL_OVF_ CTRL	Thread	See Table 2-2. See Section 18.6.2.2, "Global Counter Control Facilities." Allows software to clear counter overflow conditions on any combination of fixed-function PMCs (MSR_PERF_FIXED_CTRx) or general-purpose PMCs via a single WRMSR.
390H	912	MSR_PERF_GLOBAL_OVF_ CTRL	Thread	**(R/W)**
		61		**CLR_UNC_Ovf** Set 1 to clear UNC_Ovf.
3F1H	1009	MSR_PEBS_ENABLE	Thread	See Section 18.3.1.1.1, "Processor Event Based Sampling (PEBS)."
		0		Enable PEBS on IA32_PMC0. (R/W)
		1		Enable PEBS on IA32_PMC1. (R/W)
		2		Enable PEBS on IA32_PMC2. (R/W)
		3		Enable PEBS on IA32_PMC3. (R/W)
		31:4		Reserved.
		32		Enable Load Latency on IA32_PMC0. (R/W)
		33		Enable Load Latency on IA32_PMC1. (R/W)
		34		Enable Load Latency on IA32_PMC2. (R/W)
		35		Enable Load Latency on IA32_PMC3. (R/W)
		63:36		Reserved.
3F6H	1014	MSR_PEBS_LD_LAT	Thread	See Section 18.3.1.1.2, "Load Latency Performance Monitoring Facility."
		15:0		Minimum threshold latency value of tagged load operation that will be counted. (R/W)
		63:36		Reserved.

Table 2-14. MSRs in Processors Based on Intel® Microarchitecture Code Name Nehalem (Contd.)

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
3F8H	1016	MSR_PKG_C3_RESIDENCY	Package	Note: C-state values are processor specific C-state code names, unrelated to MWAIT extension C-state parameters or ACPI C-States.
		63:0		Package C3 Residency Counter. (R/O) Value since last reset that this package is in processor-specific C3 states. Count at the same frequency as the TSC.
3F9H	1017	MSR_PKG_C6_RESIDENCY	Package	Note: C-state values are processor specific C-state code names, unrelated to MWAIT extension C-state parameters or ACPI C-States.
		63:0		Package C6 Residency Counter. (R/O) Value since last reset that this package is in processor-specific C6 states. Count at the same frequency as the TSC.
3FAH	1018	MSR_PKG_C7_RESIDENCY	Package	Note: C-state values are processor specific C-state code names, unrelated to MWAIT extension C-state parameters or ACPI C-States.
		63:0		Package C7 Residency Counter. (R/O) Value since last reset that this package is in processor-specific C7 states. Count at the same frequency as the TSC.
3FCH	1020	MSR_CORE_C3_RESIDENCY	Core	Note: C-state values are processor specific C-state code names, unrelated to MWAIT extension C-state parameters or ACPI C-States.
		63:0		CORE C3 Residency Counter. (R/O) Value since last reset that this core is in processor-specific C3 states. Count at the same frequency as the TSC.
3FDH	1021	MSR_CORE_C6_RESIDENCY	Core	Note: C-state values are processor specific C-state code names, unrelated to MWAIT extension C-state parameters or ACPI C-States.
		63:0		CORE C6 Residency Counter. (R/O) Value since last reset that this core is in processor-specific C6 states. Count at the same frequency as the TSC.
400H	1024	IA32_MC0_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs."
401H	1025	IA32_MC0_STATUS	Package	See Section 15.3.2.2, "IA32_MCi_STATUS MSRS."
402H	1026	IA32_MC0_ADDR	Package	See Section 15.3.2.3, "IA32_MCi_ADDR MSRs." The IA32_MC0_ADDR register is either not implemented or contains no address if the ADDRV flag in the IA32_MC0_STATUS register is clear. When not implemented in the processor, all reads and writes to this MSR will cause a general-protection exception.
403H	1027	IA32_MC0_MISC	Package	See Section 15.3.2.4, "IA32_MCi_MISC MSRs."
404H	1028	IA32_MC1_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs."
405H	1029	IA32_MC1_STATUS	Package	See Section 15.3.2.2, "IA32_MCi_STATUS MSRS."

Table 2-14. MSRs in Processors Based on Intel® Microarchitecture Code Name Nehalem (Contd.)

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
406H	1030	IA32_MC1_ADDR	Package	See Section 15.3.2.3, "IA32_MCi_ADDR MSRs." The IA32_MC1_ADDR register is either not implemented or contains no address if the ADDRV flag in the IA32_MC1_STATUS register is clear. When not implemented in the processor, all reads and writes to this MSR will cause a general-protection exception.
407H	1031	IA32_MC1_MISC	Package	See Section 15.3.2.4, "IA32_MCi_MISC MSRs."
408H	1032	IA32_MC2_CTL	Core	See Section 15.3.2.1, "IA32_MCi_CTL MSRs."
409H	1033	IA32_MC2_STATUS	Core	See Section 15.3.2.2, "IA32_MCi_STATUS MSRS."
40AH	1034	IA32_MC2_ADDR	Core	See Section 15.3.2.3, "IA32_MCi_ADDR MSRs." The IA32_MC2_ADDR register is either not implemented or contains no address if the ADDRV flag in the IA32_MC2_STATUS register is clear. When not implemented in the processor, all reads and writes to this MSR will cause a general-protection exception.
40BH	1035	IA32_MC2_MISC	Core	See Section 15.3.2.4, "IA32_MCi_MISC MSRs."
40CH	1036	IA32_MC3_CTL	Core	See Section 15.3.2.1, "IA32_MCi_CTL MSRs."
40DH	1037	IA32_MC3_STATUS	Core	See Section 15.3.2.2, "IA32_MCi_STATUS MSRS."
40EH	1038	IA32_MC3_ADDR	Core	See Section 15.3.2.3, "IA32_MCi_ADDR MSRs." The MSR_MC4_ADDR register is either not implemented or contains no address if the ADDRV flag in the MSR_MC4_STATUS register is clear. When not implemented in the processor, all reads and writes to this MSR will cause a general-protection exception.
40FH	1039	IA32_MC3_MISC	Core	See Section 15.3.2.4, "IA32_MCi_MISC MSRs."
410H	1040	IA32_MC4_CTL	Core	See Section 15.3.2.1, "IA32_MCi_CTL MSRs."
411H	1041	IA32_MC4_STATUS	Core	See Section 15.3.2.2, "IA32_MCi_STATUS MSRS."
412H	1042	IA32_MC4_ADDR	Core	See Section 15.3.2.3, "IA32_MCi_ADDR MSRs." The MSR_MC3_ADDR register is either not implemented or contains no address if the ADDRV flag in the MSR_MC3_STATUS register is clear. When not implemented in the processor, all reads and writes to this MSR will cause a general-protection exception.
413H	1043	IA32_MC4_MISC	Core	See Section 15.3.2.4, "IA32_MCi_MISC MSRs."
414H	1044	IA32_MC5_CTL	Core	See Section 15.3.2.1, "IA32_MCi_CTL MSRs."
415H	1045	IA32_MC5_STATUS	Core	See Section 15.3.2.2, "IA32_MCi_STATUS MSRS."
416H	1046	IA32_MC5_ADDR	Core	See Section 15.3.2.3, "IA32_MCi_ADDR MSRs."
417H	1047	IA32_MC5_MISC	Core	See Section 15.3.2.4, "IA32_MCi_MISC MSRs."
418H	1048	IA32_MC6_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs."
419H	1049	IA32_MC6_STATUS	Package	See Section 15.3.2.2, "IA32_MCi_STATUS MSRS," and Chapter 16.
41AH	1050	IA32_MC6_ADDR	Package	See Section 15.3.2.3, "IA32_MCi_ADDR MSRs."

Table 2-14. MSRs in Processors Based on Intel® Microarchitecture Code Name Nehalem (Contd.)

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
41BH	1051	IA32_MC6_MISC	Package	See Section 15.3.2.4, "IA32_MCi_MISC MSRs."
41CH	1052	IA32_MC7_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs."
41DH	1053	IA32_MC7_STATUS	Package	See Section 15.3.2.2, "IA32_MCi_STATUS MSRS," and Chapter 16.
41EH	1054	IA32_MC7_ADDR	Package	See Section 15.3.2.3, "IA32_MCi_ADDR MSRs."
41FH	1055	IA32_MC7_MISC	Package	See Section 15.3.2.4, "IA32_MCi_MISC MSRs."
420H	1056	IA32_MC8_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs."
421H	1057	IA32_MC8_STATUS	Package	See Section 15.3.2.2, "IA32_MCi_STATUS MSRS," and Chapter 16.
422H	1058	IA32_MC8_ADDR	Package	See Section 15.3.2.3, "IA32_MCi_ADDR MSRs."
423H	1059	IA32_MC8_MISC	Package	See Section 15.3.2.4, "IA32_MCi_MISC MSRs."
480H	1152	IA32_VMX_BASIC	Thread	**Reporting Register of Basic VMX Capabilities (R/O)** See Table 2-2. See Appendix A.1, "Basic VMX Information."
481H	1153	IA32_VMX_PINBASED_CTLS	Thread	**Capability Reporting Register of Pin-based VM-execution Controls (R/O)** See Table 2-2. See Appendix A.3, "VM-Execution Controls."
482H	1154	IA32_VMX_PROCBASED_CTLS	Thread	**Capability Reporting Register of Primary Processor-based VM-execution Controls (R/O)** See Appendix A.3, "VM-Execution Controls."
483H	1155	IA32_VMX_EXIT_CTLS	Thread	**Capability Reporting Register of VM-exit Controls (R/O)** See Table 2-2. See Appendix A.4, "VM-Exit Controls."
484H	1156	IA32_VMX_ENTRY_CTLS	Thread	**Capability Reporting Register of VM-entry Controls (R/O)** See Table 2-2. See Appendix A.5, "VM-Entry Controls."
485H	1157	IA32_VMX_MISC	Thread	**Reporting Register of Miscellaneous VMX Capabilities (R/O)** See Table 2-2. See Appendix A.6, "Miscellaneous Data."
486H	1158	IA32_VMX_CR0_FIXED0	Thread	**Capability Reporting Register of CR0 Bits Fixed to 0 (R/O)** See Table 2-2. See Appendix A.7, "VMX-Fixed Bits in CR0."
487H	1159	IA32_VMX_CR0_FIXED1	Thread	**Capability Reporting Register of CR0 Bits Fixed to 1 (R/O)** See Table 2-2. See Appendix A.7, "VMX-Fixed Bits in CR0."
488H	1160	IA32_VMX_CR4_FIXED0	Thread	**Capability Reporting Register of CR4 Bits Fixed to 0 (R/O)** See Table 2-2. See Appendix A.8, "VMX-Fixed Bits in CR4."

Table 2-14. MSRs in Processors Based on Intel® Microarchitecture Code Name Nehalem (Contd.)

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
489H	1161	IA32_VMX_CR4_FIXED1	Thread	**Capability Reporting Register of CR4 Bits Fixed to 1 (R/O)** See Table 2-2. See Appendix A.8, "VMX-Fixed Bits in CR4."
48AH	1162	IA32_VMX_VMCS_ENUM	Thread	**Capability Reporting Register of VMCS Field Enumeration (R/O).** See Table 2-2. See Appendix A.9, "VMCS Enumeration."
48BH	1163	IA32_VMX_PROCBASED_CTLS2	Thread	**Capability Reporting Register of Secondary Processor-based VM-execution Controls (R/O)** See Appendix A.3, "VM-Execution Controls."
600H	1536	IA32_DS_AREA	Thread	**DS Save Area (R/W)** See Table 2-2. See Section 18.6.3.4, "Debug Store (DS) Mechanism."
680H	1664	MSR_LASTBRANCH_0_FROM_IP	Thread	**Last Branch Record 0 From IP (R/W)** One of sixteen pairs of last branch record registers on the last branch record stack. The From_IP part of the stack contains pointers to the **source instruction**. See also: ▪ Last Branch Record Stack TOS at 1C9H ▪ Section 17.9.1 and record format in Section 17.4.8.1
681H	1665	MSR_LASTBRANCH_1_FROM_IP	Thread	**Last Branch Record 1 From IP (R/W)** See description of MSR_LASTBRANCH_0_FROM_IP.
682H	1666	MSR_LASTBRANCH_2_FROM_IP	Thread	**Last Branch Record 2 From IP (R/W)** See description of MSR_LASTBRANCH_0_FROM_IP.
683H	1667	MSR_LASTBRANCH_3_FROM_IP	Thread	**Last Branch Record 3 From IP (R/W)** See description of MSR_LASTBRANCH_0_FROM_IP.
684H	1668	MSR_LASTBRANCH_4_FROM_IP	Thread	**Last Branch Record 4 From IP (R/W)** See description of MSR_LASTBRANCH_0_FROM_IP.
685H	1669	MSR_LASTBRANCH_5_FROM_IP	Thread	**Last Branch Record 5 From IP (R/W)** See description of MSR_LASTBRANCH_0_FROM_IP.
686H	1670	MSR_LASTBRANCH_6_FROM_IP	Thread	**Last Branch Record 6 From IP (R/W)** See description of MSR_LASTBRANCH_0_FROM_IP.
687H	1671	MSR_LASTBRANCH_7_FROM_IP	Thread	**Last Branch Record 7 From IP (R/W)** See description of MSR_LASTBRANCH_0_FROM_IP.
688H	1672	MSR_LASTBRANCH_8_FROM_IP	Thread	**Last Branch Record 8 From IP (R/W)** See description of MSR_LASTBRANCH_0_FROM_IP.
689H	1673	MSR_LASTBRANCH_9_FROM_IP	Thread	**Last Branch Record 9 From IP (R/W)** See description of MSR_LASTBRANCH_0_FROM_IP.
68AH	1674	MSR_LASTBRANCH_10_FROM_IP	Thread	**Last Branch Record 10 From IP (R/W)** See description of MSR_LASTBRANCH_0_FROM_IP.

Table 2-14. MSRs in Processors Based on Intel® Microarchitecture Code Name Nehalem (Contd.)

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
68BH	1675	MSR_ LASTBRANCH_11_FROM_IP	Thread	**Last Branch Record 11 From IP (R/W)** See description of MSR_LASTBRANCH_0_FROM_IP.
68CH	1676	MSR_ LASTBRANCH_12_FROM_IP	Thread	**Last Branch Record 12 From IP (R/W)** See description of MSR_LASTBRANCH_0_FROM_IP.
68DH	1677	MSR_ LASTBRANCH_13_FROM_IP	Thread	**Last Branch Record 13 From IP (R/W)** See description of MSR_LASTBRANCH_0_FROM_IP.
68EH	1678	MSR_ LASTBRANCH_14_FROM_IP	Thread	**Last Branch Record 14 From IP (R/W)** See description of MSR_LASTBRANCH_0_FROM_IP.
68FH	1679	MSR_ LASTBRANCH_15_FROM_IP	Thread	**Last Branch Record 15 From IP (R/W)** See description of MSR_LASTBRANCH_0_FROM_IP.
6C0H	1728	MSR_ LASTBRANCH_0_TO_IP	Thread	**Last Branch Record 0 To IP (R/W)** One of sixteen pairs of last branch record registers on the last branch record stack. This part of the stack contains pointers to the destination instruction.
6C1H	1729	MSR_ LASTBRANCH_1_TO_IP	Thread	**Last Branch Record 1 To IP (R/W)** See description of MSR_LASTBRANCH_0_TO_IP.
6C2H	1730	MSR_ LASTBRANCH_2_TO_IP	Thread	**Last Branch Record 2 To IP (R/W)** See description of MSR_LASTBRANCH_0_TO_IP.
6C3H	1731	MSR_ LASTBRANCH_3_TO_IP	Thread	**Last Branch Record 3 To IP (R/W)** See description of MSR_LASTBRANCH_0_TO_IP.
6C4H	1732	MSR_ LASTBRANCH_4_TO_IP	Thread	**Last Branch Record 4 To IP (R/W)** See description of MSR_LASTBRANCH_0_TO_IP.
6C5H	1733	MSR_ LASTBRANCH_5_TO_IP	Thread	**Last Branch Record 5 To IP (R/W)** See description of MSR_LASTBRANCH_0_TO_IP.
6C6H	1734	MSR_ LASTBRANCH_6_TO_IP	Thread	**Last Branch Record 6 To IP (R/W)** See description of MSR_LASTBRANCH_0_TO_IP.
6C7H	1735	MSR_ LASTBRANCH_7_TO_IP	Thread	**Last Branch Record 7 To IP (R/W)** See description of MSR_LASTBRANCH_0_TO_IP.
6C8H	1736	MSR_ LASTBRANCH_8_TO_IP	Thread	**Last Branch Record 8 To IP (R/W)** See description of MSR_LASTBRANCH_0_TO_IP.
6C9H	1737	MSR_ LASTBRANCH_9_TO_IP	Thread	**Last Branch Record 9 To IP (R/W)** See description of MSR_LASTBRANCH_0_TO_IP.
6CAH	1738	MSR_ LASTBRANCH_10_TO_IP	Thread	**Last Branch Record 10 To IP (R/W)** See description of MSR_LASTBRANCH_0_TO_IP.
6CBH	1739	MSR_ LASTBRANCH_11_TO_IP	Thread	**Last Branch Record 11 To IP (R/W)** See description of MSR_LASTBRANCH_0_TO_IP.
6CCH	1740	MSR_ LASTBRANCH_12_TO_IP	Thread	**Last Branch Record 12 To IP (R/W)** See description of MSR_LASTBRANCH_0_TO_IP.

Table 2-14. MSRs in Processors Based on Intel® Microarchitecture Code Name Nehalem (Contd.)

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
6CDH	1741	MSR_ LASTBRANCH_13_TO_IP	Thread	**Last Branch Record 13 To IP (R/W)** See description of MSR_LASTBRANCH_0_TO_IP.
6CEH	1742	MSR_ LASTBRANCH_14_TO_IP	Thread	**Last Branch Record 14 To IP (R/W)** See description of MSR_LASTBRANCH_0_TO_IP.
6CFH	1743	MSR_ LASTBRANCH_15_TO_IP	Thread	**Last Branch Record 15 To IP (R/W)** See description of MSR_LASTBRANCH_0_TO_IP.
802H	2050	IA32_X2APIC_APICID	Thread	x2APIC ID register (R/O) See x2APIC Specification.
803H	2051	IA32_X2APIC_VERSION	Thread	x2APIC Version register (R/O)
808H	2056	IA32_X2APIC_TPR	Thread	x2APIC Task Priority register (R/W)
80AH	2058	IA32_X2APIC_PPR	Thread	x2APIC Processor Priority register (R/O)
80BH	2059	IA32_X2APIC_EOI	Thread	x2APIC EOI register (W/O)
80DH	2061	IA32_X2APIC_LDR	Thread	x2APIC Logical Destination register (R/O)
80FH	2063	IA32_X2APIC_SIVR	Thread	x2APIC Spurious Interrupt Vector register (R/W)
810H	2064	IA32_X2APIC_ISR0	Thread	x2APIC In-Service register bits [31:0] (R/O)
811H	2065	IA32_X2APIC_ISR1	Thread	x2APIC In-Service register bits [63:32] (R/O)
812H	2066	IA32_X2APIC_ISR2	Thread	x2APIC In-Service register bits [95:64] (R/O)
813H	2067	IA32_X2APIC_ISR3	Thread	x2APIC In-Service register bits [127:96] (R/O)
814H	2068	IA32_X2APIC_ISR4	Thread	x2APIC In-Service register bits [159:128] (R/O)
815H	2069	IA32_X2APIC_ISR5	Thread	x2APIC In-Service register bits [191:160] (R/O)
816H	2070	IA32_X2APIC_ISR6	Thread	x2APIC In-Service register bits [223:192] (R/O)
817H	2071	IA32_X2APIC_ISR7	Thread	x2APIC In-Service register bits [255:224] (R/O)
818H	2072	IA32_X2APIC_TMR0	Thread	x2APIC Trigger Mode register bits [31:0] (R/O)
819H	2073	IA32_X2APIC_TMR1	Thread	x2APIC Trigger Mode register bits [63:32] (R/O)
81AH	2074	IA32_X2APIC_TMR2	Thread	x2APIC Trigger Mode register bits [95:64] (R/O)
81BH	2075	IA32_X2APIC_TMR3	Thread	x2APIC Trigger Mode register bits [127:96] (R/O)
81CH	2076	IA32_X2APIC_TMR4	Thread	x2APIC Trigger Mode register bits [159:128] (R/O)
81DH	2077	IA32_X2APIC_TMR5	Thread	x2APIC Trigger Mode register bits [191:160] (R/O)
81EH	2078	IA32_X2APIC_TMR6	Thread	x2APIC Trigger Mode register bits [223:192] (R/O)
81FH	2079	IA32_X2APIC_TMR7	Thread	x2APIC Trigger Mode register bits [255:224] (R/O)
820H	2080	IA32_X2APIC_IRR0	Thread	x2APIC Interrupt Request register bits [31:0] (R/O)
821H	2081	IA32_X2APIC_IRR1	Thread	x2APIC Interrupt Request register bits [63:32] (R/O)
822H	2082	IA32_X2APIC_IRR2	Thread	x2APIC Interrupt Request register bits [95:64] (R/O)
823H	2083	IA32_X2APIC_IRR3	Thread	x2APIC Interrupt Request register bits [127:96] (R/O)
824H	2084	IA32_X2APIC_IRR4	Thread	x2APIC Interrupt Request register bits [159:128] (R/O)
825H	2085	IA32_X2APIC_IRR5	Thread	x2APIC Interrupt Request register bits [191:160] (R/O)
826H	2086	IA32_X2APIC_IRR6	Thread	x2APIC Interrupt Request register bits [223:192] (R/O)

Table 2-14. MSRs in Processors Based on Intel® Microarchitecture Code Name Nehalem (Contd.)

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
827H	2087	IA32_X2APIC_IRR7	Thread	x2APIC Interrupt Request register bits [255:224] (R/O)
828H	2088	IA32_X2APIC_ESR	Thread	x2APIC Error Status register (R/W)
82FH	2095	IA32_X2APIC_LVT_CMCI	Thread	x2APIC LVT Corrected Machine Check Interrupt register (R/W)
830H	2096	IA32_X2APIC_ICR	Thread	x2APIC Interrupt Command register (R/W)
832H	2098	IA32_X2APIC_LVT_TIMER	Thread	x2APIC LVT Timer Interrupt register (R/W)
833H	2099	IA32_X2APIC_LVT_THERMAL	Thread	x2APIC LVT Thermal Sensor Interrupt register (R/W)
834H	2100	IA32_X2APIC_LVT_PMI	Thread	x2APIC LVT Performance Monitor register (R/W)
835H	2101	IA32_X2APIC_LVT_LINT0	Thread	x2APIC LVT LINT0 register (R/W)
836H	2102	IA32_X2APIC_LVT_LINT1	Thread	x2APIC LVT LINT1 register (R/W)
837H	2103	IA32_X2APIC_LVT_ERROR	Thread	x2APIC LVT Error register (R/W)
838H	2104	IA32_X2APIC_INIT_COUNT	Thread	x2APIC Initial Count register (R/W)
839H	2105	IA32_X2APIC_CUR_COUNT	Thread	x2APIC Current Count register (R/O)
83EH	2110	IA32_X2APIC_DIV_CONF	Thread	x2APIC Divide Configuration register (R/W)
83FH	2111	IA32_X2APIC_SELF_IPI	Thread	x2APIC Self IPI register (W/O)
C000_0080H		IA32_EFER	Thread	**Extended Feature Enables** See Table 2-2.
C000_0081H		IA32_STAR	Thread	**System Call Target Address (R/W)** See Table 2-2.
C000_0082H		IA32_LSTAR	Thread	**IA-32e Mode System Call Target Address (R/W)** See Table 2-2.
C000_0084H		IA32_FMASK	Thread	**System Call Flag Mask (R/W)** See Table 2-2.
C000_0100H		IA32_FS_BASE	Thread	**Map of BASE Address of FS (R/W)** See Table 2-2.
C000_0101H		IA32_GS_BASE	Thread	**Map of BASE Address of GS (R/W)** See Table 2-2.
C000_0102H		IA32_KERNEL_GS_BASE	Thread	**Swap Target of BASE Address of GS (R/W)** See Table 2-2.
C000_0103H		IA32_TSC_AUX	Thread	**AUXILIARY TSC Signature. (R/W)** See Table 2-2 and Section 17.17.2, "IA32_TSC_AUX Register and RDTSCP Support."

2.7.1 Additional MSRs in the Intel® Xeon® Processor 5500 and 3400 Series

Intel Xeon Processor 5500 and 3400 series support additional model-specific registers listed in Table 2-15. These MSRs also apply to Intel Core i7 and i5 processor family CPUID signature with DisplayFamily_DisplayModel of 06_1AH, 06_1EH and 06_1FH, see Table 2-1.

Table 2-15. Additional MSRs in Intel® Xeon® Processor 5500 and 3400 Series

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
1ADH	429	MSR_TURBO_RATIO_LIMIT	Package	Actual maximum turbo frequency is multiplied by 133.33MHz. (not available to model 06_2EH)
		7:0		**Maximum Turbo Ratio Limit 1C (R/O)** Maximum Turbo mode ratio limit with 1 core active.
		15:8		**Maximum Turbo Ratio Limit 2C (R/O)** Maximum Turbo mode ratio limit with 2 cores active.
		23:16		**Maximum Turbo Ratio Limit 3C (R/O)** Maximum Turbo mode ratio limit with 3 cores active.
		31:24		**Maximum Turbo Ratio Limit 4C (R/O)** Maximum Turbo mode ratio limit with 4 cores active.
		63:32		Reserved.
301H	769	MSR_GQ_SNOOP_MESF	Package	
		0		**From M to S (R/W)**
		1		**From E to S (R/W)**
		2		**From S to S (R/W)**
		3		**From F to S (R/W)**
		4		**From M to I (R/W)**
		5		**From E to I (R/W)**
		6		**From S to I (R/W)**
		7		**From F to I (R/W)**
		63:8		Reserved.
391H	913	MSR_UNCORE_PERF_GLOBAL_CTRL	Package	See Section 18.3.1.2.1, "Uncore Performance Monitoring Management Facility."
392H	914	MSR_UNCORE_PERF_GLOBAL_STATUS	Package	See Section 18.3.1.2.1, "Uncore Performance Monitoring Management Facility."
393H	915	MSR_UNCORE_PERF_GLOBAL_OVF_CTRL	Package	See Section 18.3.1.2.1, "Uncore Performance Monitoring Management Facility."
394H	916	MSR_UNCORE_FIXED_CTR0	Package	See Section 18.3.1.2.1, "Uncore Performance Monitoring Management Facility."
395H	917	MSR_UNCORE_FIXED_CTR_CTRL	Package	See Section 18.3.1.2.1, "Uncore Performance Monitoring Management Facility."
396H	918	MSR_UNCORE_ADDR_OPCODE_MATCH	Package	See Section 18.3.1.2.3, "Uncore Address/Opcode Match MSR."
3B0H	960	MSR_UNCORE_PMC0	Package	See Section 18.3.1.2.2, "Uncore Performance Event Configuration Facility."
3B1H	961	MSR_UNCORE_PMC1	Package	See Section 18.3.1.2.2, "Uncore Performance Event Configuration Facility."
3B2H	962	MSR_UNCORE_PMC2	Package	See Section 18.3.1.2.2, "Uncore Performance Event Configuration Facility."

Table 2-15. Additional MSRs in Intel® Xeon® Processor 5500 and 3400 Series (Contd.)

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
3B3H	963	MSR_UNCORE_PMC3	Package	See Section 18.3.1.2.2, "Uncore Performance Event Configuration Facility."
3B4H	964	MSR_UNCORE_PMC4	Package	See Section 18.3.1.2.2, "Uncore Performance Event Configuration Facility."
3B5H	965	MSR_UNCORE_PMC5	Package	See Section 18.3.1.2.2, "Uncore Performance Event Configuration Facility."
3B6H	966	MSR_UNCORE_PMC6	Package	See Section 18.3.1.2.2, "Uncore Performance Event Configuration Facility."
3B7H	967	MSR_UNCORE_PMC7	Package	See Section 18.3.1.2.2, "Uncore Performance Event Configuration Facility."
3C0H	944	MSR_UNCORE_ PERFEVTSEL0	Package	See Section 18.3.1.2.2, "Uncore Performance Event Configuration Facility."
3C1H	945	MSR_UNCORE_ PERFEVTSEL1	Package	See Section 18.3.1.2.2, "Uncore Performance Event Configuration Facility."
3C2H	946	MSR_UNCORE_ PERFEVTSEL2	Package	See Section 18.3.1.2.2, "Uncore Performance Event Configuration Facility."
3C3H	947	MSR_UNCORE_ PERFEVTSEL3	Package	See Section 18.3.1.2.2, "Uncore Performance Event Configuration Facility."
3C4H	948	MSR_UNCORE_ PERFEVTSEL4	Package	See Section 18.3.1.2.2, "Uncore Performance Event Configuration Facility."
3C5H	949	MSR_UNCORE_ PERFEVTSEL5	Package	See Section 18.3.1.2.2, "Uncore Performance Event Configuration Facility."
3C6H	950	MSR_UNCORE_ PERFEVTSEL6	Package	See Section 18.3.1.2.2, "Uncore Performance Event Configuration Facility."
3C7H	951	MSR_UNCORE_ PERFEVTSEL7	Package	See Section 18.3.1.2.2, "Uncore Performance Event Configuration Facility."

2.7.2 Additional MSRs in the Intel® Xeon® Processor 7500 Series

Intel Xeon Processor 7500 series support MSRs listed in Table 2-14 (except MSR address 1ADH) and additional model-specific registers listed in Table 2-16. These processors have a CPUID signature with DisplayFamily_DisplayModel of 06_2EH.

Table 2-16. Additional MSRs in Intel® Xeon® Processor 7500 Series

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
1ADH	429	MSR_TURBO_RATIO_LIMIT	Package	**Reserved** Attempt to read/write will cause #UD.
289H	649	IA32_MC9_CTL2	Package	See Table 2-2.
28AH	650	IA32_MC10_CTL2	Package	See Table 2-2.
28BH	651	IA32_MC11_CTL2	Package	See Table 2-2.
28CH	652	IA32_MC12_CTL2	Package	See Table 2-2.

Table 2-16. Additional MSRs in Intel® Xeon® Processor 7500 Series (Contd.)

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
28DH	653	IA32_MC13_CTL2	Package	See Table 2-2.
28EH	654	IA32_MC14_CTL2	Package	See Table 2-2.
28FH	655	IA32_MC15_CTL2	Package	See Table 2-2.
290H	656	IA32_MC16_CTL2	Package	See Table 2-2.
291H	657	IA32_MC17_CTL2	Package	See Table 2-2.
292H	658	IA32_MC18_CTL2	Package	See Table 2-2.
293H	659	IA32_MC19_CTL2	Package	See Table 2-2.
294H	660	IA32_MC20_CTL2	Package	See Table 2-2.
295H	661	IA32_MC21_CTL2	Package	See Table 2-2.
394H	816	MSR_W_PMON_FIXED_CTR	Package	Uncore W-box perfmon fixed counter
395H	817	MSR_W_PMON_FIXED_ CTR_CTL	Package	Uncore U-box perfmon fixed counter control MSR
424H	1060	IA32_MC9_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs."
425H	1061	IA32_MC9_STATUS	Package	See Section 15.3.2.2, "IA32_MCi_STATUS MSRS," and Chapter 16.
426H	1062	IA32_MC9_ADDR	Package	See Section 15.3.2.3, "IA32_MCi_ADDR MSRs."
427H	1063	IA32_MC9_MISC	Package	See Section 15.3.2.4, "IA32_MCi_MISC MSRs."
428H	1064	IA32_MC10_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs."
429H	1065	IA32_MC10_STATUS	Package	See Section 15.3.2.2, "IA32_MCi_STATUS MSRS," and Chapter 16.
42AH	1066	IA32_MC10_ADDR	Package	See Section 15.3.2.3, "IA32_MCi_ADDR MSRs."
42BH	1067	IA32_MC10_MISC	Package	See Section 15.3.2.4, "IA32_MCi_MISC MSRs."
42CH	1068	IA32_MC11_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs."
42DH	1069	IA32_MC11_STATUS	Package	See Section 15.3.2.2, "IA32_MCi_STATUS MSRS," and Chapter 16.
42EH	1070	IA32_MC11_ADDR	Package	See Section 15.3.2.3, "IA32_MCi_ADDR MSRs."
42FH	1071	IA32_MC11_MISC	Package	See Section 15.3.2.4, "IA32_MCi_MISC MSRs."
430H	1072	IA32_MC12_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs."
431H	1073	IA32_MC12_STATUS	Package	See Section 15.3.2.2, "IA32_MCi_STATUS MSRS," and Chapter 16.
432H	1074	IA32_MC12_ADDR	Package	See Section 15.3.2.3, "IA32_MCi_ADDR MSRs."
433H	1075	IA32_MC12_MISC	Package	See Section 15.3.2.4, "IA32_MCi_MISC MSRs."
434H	1076	IA32_MC13_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs."
435H	1077	IA32_MC13_STATUS	Package	See Section 15.3.2.2, "IA32_MCi_STATUS MSRS," and Chapter 16.
436H	1078	IA32_MC13_ADDR	Package	See Section 15.3.2.3, "IA32_MCi_ADDR MSRs."
437H	1079	IA32_MC13_MISC	Package	See Section 15.3.2.4, "IA32_MCi_MISC MSRs."
438H	1080	IA32_MC14_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs."
439H	1081	IA32_MC14_STATUS	Package	See Section 15.3.2.2, "IA32_MCi_STATUS MSRS," and Chapter 16.
43AH	1082	IA32_MC14_ADDR	Package	See Section 15.3.2.3, "IA32_MCi_ADDR MSRs."
43BH	1083	IA32_MC14_MISC	Package	See Section 15.3.2.4, "IA32_MCi_MISC MSRs."
43CH	1084	IA32_MC15_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs."

Table 2-16. Additional MSRs in Intel® Xeon® Processor 7500 Series (Contd.)

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
43DH	1085	IA32_MC15_STATUS	Package	See Section 15.3.2.2, "IA32_MCi_STATUS MSRS," and Chapter 16.
43EH	1086	IA32_MC15_ADDR	Package	See Section 15.3.2.3, "IA32_MCi_ADDR MSRs."
43FH	1087	IA32_MC15_MISC	Package	See Section 15.3.2.4, "IA32_MCi_MISC MSRs."
440H	1088	IA32_MC16_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs."
441H	1089	IA32_MC16_STATUS	Package	See Section 15.3.2.2, "IA32_MCi_STATUS MSRS," and Chapter 16.
442H	1090	IA32_MC16_ADDR	Package	See Section 15.3.2.3, "IA32_MCi_ADDR MSRs."
443H	1091	IA32_MC16_MISC	Package	See Section 15.3.2.4, "IA32_MCi_MISC MSRs."
444H	1092	IA32_MC17_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs."
445H	1093	IA32_MC17_STATUS	Package	See Section 15.3.2.2, "IA32_MCi_STATUS MSRS," and Chapter 16.
446H	1094	IA32_MC17_ADDR	Package	See Section 15.3.2.3, "IA32_MCi_ADDR MSRs."
447H	1095	IA32_MC17_MISC	Package	See Section 15.3.2.4, "IA32_MCi_MISC MSRs."
448H	1096	IA32_MC18_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs."
449H	1097	IA32_MC18_STATUS	Package	See Section 15.3.2.2, "IA32_MCi_STATUS MSRS," and Chapter 16.
44AH	1098	IA32_MC18_ADDR	Package	See Section 15.3.2.3, "IA32_MCi_ADDR MSRs."
44BH	1099	IA32_MC18_MISC	Package	See Section 15.3.2.4, "IA32_MCi_MISC MSRs."
44CH	1100	IA32_MC19_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs."
44DH	1101	IA32_MC19_STATUS	Package	See Section 15.3.2.2, "IA32_MCi_STATUS MSRS," and Chapter 16.
44EH	1102	IA32_MC19_ADDR	Package	See Section 15.3.2.3, "IA32_MCi_ADDR MSRs."
44FH	1103	IA32_MC19_MISC	Package	See Section 15.3.2.4, "IA32_MCi_MISC MSRs."
450H	1104	IA32_MC20_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs."
451H	1105	IA32_MC20_STATUS	Package	See Section 15.3.2.2, "IA32_MCi_STATUS MSRS," and Chapter 16.
452H	1106	IA32_MC20_ADDR	Package	See Section 15.3.2.3, "IA32_MCi_ADDR MSRs."
453H	1107	IA32_MC20_MISC	Package	See Section 15.3.2.4, "IA32_MCi_MISC MSRs."
454H	1108	IA32_MC21_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs."
455H	1109	IA32_MC21_STATUS	Package	See Section 15.3.2.2, "IA32_MCi_STATUS MSRS," and Chapter 16.
456H	1110	IA32_MC21_ADDR	Package	See Section 15.3.2.3, "IA32_MCi_ADDR MSRs."
457H	1111	IA32_MC21_MISC	Package	See Section 15.3.2.4, "IA32_MCi_MISC MSRs."
C00H	3072	MSR_U_PMON_GLOBAL_ CTRL	Package	Uncore U-box perfmon global control MSR.
C01H	3073	MSR_U_PMON_GLOBAL_ STATUS	Package	Uncore U-box perfmon global status MSR.
C02H	3074	MSR_U_PMON_GLOBAL_ OVF_CTRL	Package	Uncore U-box perfmon global overflow control MSR.
C10H	3088	MSR_U_PMON_EVNT_SEL	Package	Uncore U-box perfmon event select MSR.
C11H	3089	MSR_U_PMON_CTR	Package	Uncore U-box perfmon counter MSR.
C20H	3104	MSR_B0_PMON_BOX_CTRL	Package	Uncore B-box 0 perfmon local box control MSR.

Table 2-16. Additional MSRs in Intel® Xeon® Processor 7500 Series (Contd.)

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
C21H	3105	MSR_B0_PMON_BOX_ STATUS	Package	Uncore B-box 0 perfmon local box status MSR.
C22H	3106	MSR_B0_PMON_BOX_OVF_ CTRL	Package	Uncore B-box 0 perfmon local box overflow control MSR.
C30H	3120	MSR_B0_PMON_EVNT_ SEL0	Package	Uncore B-box 0 perfmon event select MSR.
C31H	3121	MSR_B0_PMON_CTR0	Package	Uncore B-box 0 perfmon counter MSR.
C32H	3122	MSR_B0_PMON_EVNT_ SEL1	Package	Uncore B-box 0 perfmon event select MSR.
C33H	3123	MSR_B0_PMON_CTR1	Package	Uncore B-box 0 perfmon counter MSR.
C34H	3124	MSR_B0_PMON_EVNT_ SEL2	Package	Uncore B-box 0 perfmon event select MSR.
C35H	3125	MSR_B0_PMON_CTR2	Package	Uncore B-box 0 perfmon counter MSR.
C36H	3126	MSR_B0_PMON_EVNT_ SEL3	Package	Uncore B-box 0 perfmon event select MSR.
C37H	3127	MSR_B0_PMON_CTR3	Package	Uncore B-box 0 perfmon counter MSR.
C40H	3136	MSR_S0_PMON_BOX_CTRL	Package	Uncore S-box 0 perfmon local box control MSR.
C41H	3137	MSR_S0_PMON_BOX_ STATUS	Package	Uncore S-box 0 perfmon local box status MSR.
C42H	3138	MSR_S0_PMON_BOX_OVF_ CTRL	Package	Uncore S-box 0 perfmon local box overflow control MSR.
C50H	3152	MSR_S0_PMON_EVNT_ SEL0	Package	Uncore S-box 0 perfmon event select MSR.
C51H	3153	MSR_S0_PMON_CTR0	Package	Uncore S-box 0 perfmon counter MSR.
C52H	3154	MSR_S0_PMON_EVNT_ SEL1	Package	Uncore S-box 0 perfmon event select MSR.
C53H	3155	MSR_S0_PMON_CTR1	Package	Uncore S-box 0 perfmon counter MSR.
C54H	3156	MSR_S0_PMON_EVNT_ SEL2	Package	Uncore S-box 0 perfmon event select MSR.
C55H	3157	MSR_S0_PMON_CTR2	Package	Uncore S-box 0 perfmon counter MSR.
C56H	3158	MSR_S0_PMON_EVNT_ SEL3	Package	Uncore S-box 0 perfmon event select MSR.
C57H	3159	MSR_S0_PMON_CTR3	Package	Uncore S-box 0 perfmon counter MSR.
C60H	3168	MSR_B1_PMON_BOX_CTRL	Package	Uncore B-box 1 perfmon local box control MSR.
C61H	3169	MSR_B1_PMON_BOX_ STATUS	Package	Uncore B-box 1 perfmon local box status MSR.
C62H	3170	MSR_B1_PMON_BOX_OVF_ CTRL	Package	Uncore B-box 1 perfmon local box overflow control MSR.

Table 2-16. Additional MSRs in Intel® Xeon® Processor 7500 Series (Contd.)

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
C70H	3184	MSR_B1_PMON_EVNT_SEL0	Package	Uncore B-box 1 perfmon event select MSR.
C71H	3185	MSR_B1_PMON_CTR0	Package	Uncore B-box 1 perfmon counter MSR.
C72H	3186	MSR_B1_PMON_EVNT_SEL1	Package	Uncore B-box 1 perfmon event select MSR.
C73H	3187	MSR_B1_PMON_CTR1	Package	Uncore B-box 1 perfmon counter MSR.
C74H	3188	MSR_B1_PMON_EVNT_SEL2	Package	Uncore B-box 1 perfmon event select MSR.
C75H	3189	MSR_B1_PMON_CTR2	Package	Uncore B-box 1 perfmon counter MSR.
C76H	3190	MSR_B1_PMON_EVNT_SEL3	Package	Uncore B-box 1vperfmon event select MSR.
C77H	3191	MSR_B1_PMON_CTR3	Package	Uncore B-box 1 perfmon counter MSR.
C80H	3120	MSR_W_PMON_BOX_CTRL	Package	Uncore W-box perfmon local box control MSR.
C81H	3121	MSR_W_PMON_BOX_STATUS	Package	Uncore W-box perfmon local box status MSR.
C82H	3122	MSR_W_PMON_BOX_OVF_CTRL	Package	Uncore W-box perfmon local box overflow control MSR.
C90H	3136	MSR_W_PMON_EVNT_SEL0	Package	Uncore W-box perfmon event select MSR.
C91H	3137	MSR_W_PMON_CTR0	Package	Uncore W-box perfmon counter MSR.
C92H	3138	MSR_W_PMON_EVNT_SEL1	Package	Uncore W-box perfmon event select MSR.
C93H	3139	MSR_W_PMON_CTR1	Package	Uncore W-box perfmon counter MSR.
C94H	3140	MSR_W_PMON_EVNT_SEL2	Package	Uncore W-box perfmon event select MSR.
C95H	3141	MSR_W_PMON_CTR2	Package	Uncore W-box perfmon counter MSR.
C96H	3142	MSR_W_PMON_EVNT_SEL3	Package	Uncore W-box perfmon event select MSR.
C97H	3143	MSR_W_PMON_CTR3	Package	Uncore W-box perfmon counter MSR.
CA0H	3232	MSR_M0_PMON_BOX_CTRL	Package	Uncore M-box 0 perfmon local box control MSR.
CA1H	3233	MSR_M0_PMON_BOX_STATUS	Package	Uncore M-box 0 perfmon local box status MSR.
CA2H	3234	MSR_M0_PMON_BOX_OVF_CTRL	Package	Uncore M-box 0 perfmon local box overflow control MSR.
CA4H	3236	MSR_M0_PMON_TIMESTAMP	Package	Uncore M-box 0 perfmon time stamp unit select MSR.
CA5H	3237	MSR_M0_PMON_DSP	Package	Uncore M-box 0 perfmon DSP unit select MSR.
CA6H	3238	MSR_M0_PMON_ISS	Package	Uncore M-box 0 perfmon ISS unit select MSR.

Table 2-16. Additional MSRs in Intel® Xeon® Processor 7500 Series (Contd.)

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
CA7H	3239	MSR_M0_PMON_MAP	Package	Uncore M-box 0 perfmon MAP unit select MSR.
CA8H	3240	MSR_M0_PMON_MSC_THR	Package	Uncore M-box 0 perfmon MIC THR select MSR.
CA9H	3241	MSR_M0_PMON_PGT	Package	Uncore M-box 0 perfmon PGT unit select MSR.
CAAH	3242	MSR_M0_PMON_PLD	Package	Uncore M-box 0 perfmon PLD unit select MSR.
CABH	3243	MSR_M0_PMON_ZDP	Package	Uncore M-box 0 perfmon ZDP unit select MSR.
CB0H	3248	MSR_M0_PMON_EVNT_SEL0	Package	Uncore M-box 0 perfmon event select MSR.
CB1H	3249	MSR_M0_PMON_CTR0	Package	Uncore M-box 0 perfmon counter MSR.
CB2H	3250	MSR_M0_PMON_EVNT_SEL1	Package	Uncore M-box 0 perfmon event select MSR.
CB3H	3251	MSR_M0_PMON_CTR1	Package	Uncore M-box 0 perfmon counter MSR.
CB4H	3252	MSR_M0_PMON_EVNT_SEL2	Package	Uncore M-box 0 perfmon event select MSR.
CB5H	3253	MSR_M0_PMON_CTR2	Package	Uncore M-box 0 perfmon counter MSR.
CB6H	3254	MSR_M0_PMON_EVNT_SEL3	Package	Uncore M-box 0 perfmon event select MSR.
CB7H	3255	MSR_M0_PMON_CTR3	Package	Uncore M-box 0 perfmon counter MSR.
CB8H	3256	MSR_M0_PMON_EVNT_SEL4	Package	Uncore M-box 0 perfmon event select MSR.
CB9H	3257	MSR_M0_PMON_CTR4	Package	Uncore M-box 0 perfmon counter MSR.
CBAH	3258	MSR_M0_PMON_EVNT_SEL5	Package	Uncore M-box 0 perfmon event select MSR.
CBBH	3259	MSR_M0_PMON_CTR5	Package	Uncore M-box 0 perfmon counter MSR.
CC0H	3264	MSR_S1_PMON_BOX_CTRL	Package	Uncore S-box 1 perfmon local box control MSR.
CC1H	3265	MSR_S1_PMON_BOX_STATUS	Package	Uncore S-box 1 perfmon local box status MSR.
CC2H	3266	MSR_S1_PMON_BOX_OVF_CTRL	Package	Uncore S-box 1 perfmon local box overflow control MSR.
CD0H	3280	MSR_S1_PMON_EVNT_SEL0	Package	Uncore S-box 1 perfmon event select MSR.
CD1H	3281	MSR_S1_PMON_CTR0	Package	Uncore S-box 1 perfmon counter MSR.
CD2H	3282	MSR_S1_PMON_EVNT_SEL1	Package	Uncore S-box 1 perfmon event select MSR.
CD3H	3283	MSR_S1_PMON_CTR1	Package	Uncore S-box 1 perfmon counter MSR.
CD4H	3284	MSR_S1_PMON_EVNT_SEL2	Package	Uncore S-box 1 perfmon event select MSR.
CD5H	3285	MSR_S1_PMON_CTR2	Package	Uncore S-box 1 perfmon counter MSR.

Table 2-16. Additional MSRs in Intel® Xeon® Processor 7500 Series (Contd.)

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
CD6H	3286	MSR_S1_PMON_EVNT_SEL3	Package	Uncore S-box 1 perfmon event select MSR.
CD7H	3287	MSR_S1_PMON_CTR3	Package	Uncore S-box 1 perfmon counter MSR.
CE0H	3296	MSR_M1_PMON_BOX_CTRL	Package	Uncore M-box 1 perfmon local box control MSR.
CE1H	3297	MSR_M1_PMON_BOX_STATUS	Package	Uncore M-box 1 perfmon local box status MSR.
CE2H	3298	MSR_M1_PMON_BOX_OVF_CTRL	Package	Uncore M-box 1 perfmon local box overflow control MSR.
CE4H	3300	MSR_M1_PMON_TIMESTAMP	Package	Uncore M-box 1 perfmon time stamp unit select MSR.
CE5H	3301	MSR_M1_PMON_DSP	Package	Uncore M-box 1 perfmon DSP unit select MSR.
CE6H	3302	MSR_M1_PMON_ISS	Package	Uncore M-box 1 perfmon ISS unit select MSR.
CE7H	3303	MSR_M1_PMON_MAP	Package	Uncore M-box 1 perfmon MAP unit select MSR.
CE8H	3304	MSR_M1_PMON_MSC_THR	Package	Uncore M-box 1 perfmon MIC THR select MSR.
CE9H	3305	MSR_M1_PMON_PGT	Package	Uncore M-box 1 perfmon PGT unit select MSR.
CEAH	3306	MSR_M1_PMON_PLD	Package	Uncore M-box 1 perfmon PLD unit select MSR.
CEBH	3307	MSR_M1_PMON_ZDP	Package	Uncore M-box 1 perfmon ZDP unit select MSR.
CF0H	3312	MSR_M1_PMON_EVNT_SEL0	Package	Uncore M-box 1 perfmon event select MSR.
CF1H	3313	MSR_M1_PMON_CTR0	Package	Uncore M-box 1 perfmon counter MSR.
CF2H	3314	MSR_M1_PMON_EVNT_SEL1	Package	Uncore M-box 1 perfmon event select MSR.
CF3H	3315	MSR_M1_PMON_CTR1	Package	Uncore M-box 1 perfmon counter MSR.
CF4H	3316	MSR_M1_PMON_EVNT_SEL2	Package	Uncore M-box 1 perfmon event select MSR.
CF5H	3317	MSR_M1_PMON_CTR2	Package	Uncore M-box 1 perfmon counter MSR.
CF6H	3318	MSR_M1_PMON_EVNT_SEL3	Package	Uncore M-box 1 perfmon event select MSR.
CF7H	3319	MSR_M1_PMON_CTR3	Package	Uncore M-box 1 perfmon counter MSR.
CF8H	3320	MSR_M1_PMON_EVNT_SEL4	Package	Uncore M-box 1 perfmon event select MSR.
CF9H	3321	MSR_M1_PMON_CTR4	Package	Uncore M-box 1 perfmon counter MSR.
CFAH	3322	MSR_M1_PMON_EVNT_SEL5	Package	Uncore M-box 1 perfmon event select MSR.
CFBH	3323	MSR_M1_PMON_CTR5	Package	Uncore M-box 1 perfmon counter MSR.
D00H	3328	MSR_C0_PMON_BOX_CTRL	Package	Uncore C-box 0 perfmon local box control MSR.

Table 2-16. Additional MSRs in Intel® Xeon® Processor 7500 Series (Contd.)

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
D01H	3329	MSR_C0_PMON_BOX_ STATUS	Package	Uncore C-box 0 perfmon local box status MSR.
D02H	3330	MSR_C0_PMON_BOX_OVF_ CTRL	Package	Uncore C-box 0 perfmon local box overflow control MSR.
D10H	3344	MSR_C0_PMON_EVNT_ SEL0	Package	Uncore C-box 0 perfmon event select MSR.
D11H	3345	MSR_C0_PMON_CTR0	Package	Uncore C-box 0 perfmon counter MSR.
D12H	3346	MSR_C0_PMON_EVNT_ SEL1	Package	Uncore C-box 0 perfmon event select MSR.
D13H	3347	MSR_C0_PMON_CTR1	Package	Uncore C-box 0 perfmon counter MSR.
D14H	3348	MSR_C0_PMON_EVNT_ SEL2	Package	Uncore C-box 0 perfmon event select MSR.
D15H	3349	MSR_C0_PMON_CTR2	Package	Uncore C-box 0 perfmon counter MSR.
D16H	3350	MSR_C0_PMON_EVNT_ SEL3	Package	Uncore C-box 0 perfmon event select MSR.
D17H	3351	MSR_C0_PMON_CTR3	Package	Uncore C-box 0 perfmon counter MSR.
D18H	3352	MSR_C0_PMON_EVNT_ SEL4	Package	Uncore C-box 0 perfmon event select MSR.
D19H	3353	MSR_C0_PMON_CTR4	Package	Uncore C-box 0 perfmon counter MSR.
D1AH	3354	MSR_C0_PMON_EVNT_ SEL5	Package	Uncore C-box 0 perfmon event select MSR.
D1BH	3355	MSR_C0_PMON_CTR5	Package	Uncore C-box 0 perfmon counter MSR.
D20H	3360	MSR_C4_PMON_BOX_CTRL	Package	Uncore C-box 4 perfmon local box control MSR.
D21H	3361	MSR_C4_PMON_BOX_ STATUS	Package	Uncore C-box 4 perfmon local box status MSR.
D22H	3362	MSR_C4_PMON_BOX_OVF_ CTRL	Package	Uncore C-box 4 perfmon local box overflow control MSR.
D30H	3376	MSR_C4_PMON_EVNT_ SEL0	Package	Uncore C-box 4 perfmon event select MSR.
D31H	3377	MSR_C4_PMON_CTR0	Package	Uncore C-box 4 perfmon counter MSR.
D32H	3378	MSR_C4_PMON_EVNT_ SEL1	Package	Uncore C-box 4 perfmon event select MSR.
D33H	3379	MSR_C4_PMON_CTR1	Package	Uncore C-box 4 perfmon counter MSR.
D34H	3380	MSR_C4_PMON_EVNT_ SEL2	Package	Uncore C-box 4 perfmon event select MSR.
D35H	3381	MSR_C4_PMON_CTR2	Package	Uncore C-box 4 perfmon counter MSR.
D36H	3382	MSR_C4_PMON_EVNT_ SEL3	Package	Uncore C-box 4 perfmon event select MSR.
D37H	3383	MSR_C4_PMON_CTR3	Package	Uncore C-box 4 perfmon counter MSR.

Table 2-16. Additional MSRs in Intel® Xeon® Processor 7500 Series (Contd.)

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
D38H	3384	MSR_C4_PMON_EVNT_SEL4	Package	Uncore C-box 4 perfmon event select MSR.
D39H	3385	MSR_C4_PMON_CTR4	Package	Uncore C-box 4 perfmon counter MSR.
D3AH	3386	MSR_C4_PMON_EVNT_SEL5	Package	Uncore C-box 4 perfmon event select MSR.
D3BH	3387	MSR_C4_PMON_CTR5	Package	Uncore C-box 4 perfmon counter MSR.
D40H	3392	MSR_C2_PMON_BOX_CTRL	Package	Uncore C-box 2 perfmon local box control MSR.
D41H	3393	MSR_C2_PMON_BOX_STATUS	Package	Uncore C-box 2 perfmon local box status MSR.
D42H	3394	MSR_C2_PMON_BOX_OVF_CTRL	Package	Uncore C-box 2 perfmon local box overflow control MSR.
D50H	3408	MSR_C2_PMON_EVNT_SEL0	Package	Uncore C-box 2 perfmon event select MSR.
D51H	3409	MSR_C2_PMON_CTR0	Package	Uncore C-box 2 perfmon counter MSR.
D52H	3410	MSR_C2_PMON_EVNT_SEL1	Package	Uncore C-box 2 perfmon event select MSR.
D53H	3411	MSR_C2_PMON_CTR1	Package	Uncore C-box 2 perfmon counter MSR.
D54H	3412	MSR_C2_PMON_EVNT_SEL2	Package	Uncore C-box 2 perfmon event select MSR.
D55H	3413	MSR_C2_PMON_CTR2	Package	Uncore C-box 2 perfmon counter MSR.
D56H	3414	MSR_C2_PMON_EVNT_SEL3	Package	Uncore C-box 2 perfmon event select MSR.
D57H	3415	MSR_C2_PMON_CTR3	Package	Uncore C-box 2 perfmon counter MSR.
D58H	3416	MSR_C2_PMON_EVNT_SEL4	Package	Uncore C-box 2 perfmon event select MSR.
D59H	3417	MSR_C2_PMON_CTR4	Package	Uncore C-box 2 perfmon counter MSR.
D5AH	3418	MSR_C2_PMON_EVNT_SEL5	Package	Uncore C-box 2 perfmon event select MSR.
D5BH	3419	MSR_C2_PMON_CTR5	Package	Uncore C-box 2 perfmon counter MSR.
D60H	3424	MSR_C6_PMON_BOX_CTRL	Package	Uncore C-box 6 perfmon local box control MSR.
D61H	3425	MSR_C6_PMON_BOX_STATUS	Package	Uncore C-box 6 perfmon local box status MSR.
D62H	3426	MSR_C6_PMON_BOX_OVF_CTRL	Package	Uncore C-box 6 perfmon local box overflow control MSR.
D70H	3440	MSR_C6_PMON_EVNT_SEL0	Package	Uncore C-box 6 perfmon event select MSR.
D71H	3441	MSR_C6_PMON_CTR0	Package	Uncore C-box 6 perfmon counter MSR.

Table 2-16. Additional MSRs in Intel® Xeon® Processor 7500 Series (Contd.)

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
D72H	3442	MSR_C6_PMON_EVNT_SEL1	Package	Uncore C-box 6 perfmon event select MSR.
D73H	3443	MSR_C6_PMON_CTR1	Package	Uncore C-box 6 perfmon counter MSR.
D74H	3444	MSR_C6_PMON_EVNT_SEL2	Package	Uncore C-box 6 perfmon event select MSR.
D75H	3445	MSR_C6_PMON_CTR2	Package	Uncore C-box 6 perfmon counter MSR.
D76H	3446	MSR_C6_PMON_EVNT_SEL3	Package	Uncore C-box 6 perfmon event select MSR.
D77H	3447	MSR_C6_PMON_CTR3	Package	Uncore C-box 6 perfmon counter MSR.
D78H	3448	MSR_C6_PMON_EVNT_SEL4	Package	Uncore C-box 6 perfmon event select MSR.
D79H	3449	MSR_C6_PMON_CTR4	Package	Uncore C-box 6 perfmon counter MSR.
D7AH	3450	MSR_C6_PMON_EVNT_SEL5	Package	Uncore C-box 6 perfmon event select MSR.
D7BH	3451	MSR_C6_PMON_CTR5	Package	Uncore C-box 6 perfmon counter MSR.
D80H	3456	MSR_C1_PMON_BOX_CTRL	Package	Uncore C-box 1 perfmon local box control MSR.
D81H	3457	MSR_C1_PMON_BOX_STATUS	Package	Uncore C-box 1 perfmon local box status MSR.
D82H	3458	MSR_C1_PMON_BOX_OVF_CTRL	Package	Uncore C-box 1 perfmon local box overflow control MSR.
D90H	3472	MSR_C1_PMON_EVNT_SEL0	Package	Uncore C-box 1 perfmon event select MSR.
D91H	3473	MSR_C1_PMON_CTR0	Package	Uncore C-box 1 perfmon counter MSR.
D92H	3474	MSR_C1_PMON_EVNT_SEL1	Package	Uncore C-box 1 perfmon event select MSR.
D93H	3475	MSR_C1_PMON_CTR1	Package	Uncore C-box 1 perfmon counter MSR.
D94H	3476	MSR_C1_PMON_EVNT_SEL2	Package	Uncore C-box 1 perfmon event select MSR.
D95H	3477	MSR_C1_PMON_CTR2	Package	Uncore C-box 1 perfmon counter MSR.
D96H	3478	MSR_C1_PMON_EVNT_SEL3	Package	Uncore C-box 1 perfmon event select MSR.
D97H	3479	MSR_C1_PMON_CTR3	Package	Uncore C-box 1 perfmon counter MSR.
D98H	3480	MSR_C1_PMON_EVNT_SEL4	Package	Uncore C-box 1 perfmon event select MSR.
D99H	3481	MSR_C1_PMON_CTR4	Package	Uncore C-box 1 perfmon counter MSR.
D9AH	3482	MSR_C1_PMON_EVNT_SEL5	Package	Uncore C-box 1 perfmon event select MSR.
D9BH	3483	MSR_C1_PMON_CTR5	Package	Uncore C-box 1 perfmon counter MSR.

Table 2-16. Additional MSRs in Intel® Xeon® Processor 7500 Series (Contd.)

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
DA0H	3488	MSR_C5_PMON_BOX_CTRL	Package	Uncore C-box 5 perfmon local box control MSR.
DA1H	3489	MSR_C5_PMON_BOX_ STATUS	Package	Uncore C-box 5 perfmon local box status MSR.
DA2H	3490	MSR_C5_PMON_BOX_OVF_ CTRL	Package	Uncore C-box 5 perfmon local box overflow control MSR.
DB0H	3504	MSR_C5_PMON_EVNT_ SEL0	Package	Uncore C-box 5 perfmon event select MSR.
DB1H	3505	MSR_C5_PMON_CTR0	Package	Uncore C-box 5 perfmon counter MSR.
DB2H	3506	MSR_C5_PMON_EVNT_ SEL1	Package	Uncore C-box 5 perfmon event select MSR.
DB3H	3507	MSR_C5_PMON_CTR1	Package	Uncore C-box 5 perfmon counter MSR.
DB4H	3508	MSR_C5_PMON_EVNT_ SEL2	Package	Uncore C-box 5 perfmon event select MSR.
DB5H	3509	MSR_C5_PMON_CTR2	Package	Uncore C-box 5 perfmon counter MSR.
DB6H	3510	MSR_C5_PMON_EVNT_ SEL3	Package	Uncore C-box 5 perfmon event select MSR.
DB7H	3511	MSR_C5_PMON_CTR3	Package	Uncore C-box 5 perfmon counter MSR.
DB8H	3512	MSR_C5_PMON_EVNT_ SEL4	Package	Uncore C-box 5 perfmon event select MSR.
DB9H	3513	MSR_C5_PMON_CTR4	Package	Uncore C-box 5 perfmon counter MSR.
DBAH	3514	MSR_C5_PMON_EVNT_ SEL5	Package	Uncore C-box 5 perfmon event select MSR.
DBBH	3515	MSR_C5_PMON_CTR5	Package	Uncore C-box 5 perfmon counter MSR.
DC0H	3520	MSR_C3_PMON_BOX_CTRL	Package	Uncore C-box 3 perfmon local box control MSR.
DC1H	3521	MSR_C3_PMON_BOX_ STATUS	Package	Uncore C-box 3 perfmon local box status MSR.
DC2H	3522	MSR_C3_PMON_BOX_OVF_ CTRL	Package	Uncore C-box 3 perfmon local box overflow control MSR.
DD0H	3536	MSR_C3_PMON_EVNT_ SEL0	Package	Uncore C-box 3 perfmon event select MSR.
DD1H	3537	MSR_C3_PMON_CTR0	Package	Uncore C-box 3 perfmon counter MSR.
DD2H	3538	MSR_C3_PMON_EVNT_ SEL1	Package	Uncore C-box 3 perfmon event select MSR.
DD3H	3539	MSR_C3_PMON_CTR1	Package	Uncore C-box 3 perfmon counter MSR.
DD4H	3540	MSR_C3_PMON_EVNT_ SEL2	Package	Uncore C-box 3 perfmon event select MSR.
DD5H	3541	MSR_C3_PMON_CTR2	Package	Uncore C-box 3 perfmon counter MSR.

Table 2-16. Additional MSRs in Intel® Xeon® Processor 7500 Series (Contd.)

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
DD6H	3542	MSR_C3_PMON_EVNT_SEL 3	Package	Uncore C-box 3 perfmon event select MSR.
DD7H	3543	MSR_C3_PMON_CTR3	Package	Uncore C-box 3 perfmon counter MSR.
DD8H	3544	MSR_C3_PMON_EVNT_ SEL4	Package	Uncore C-box 3 perfmon event select MSR.
DD9H	3545	MSR_C3_PMON_CTR4	Package	Uncore C-box 3 perfmon counter MSR.
DDAH	3546	MSR_C3_PMON_EVNT_ SEL5	Package	Uncore C-box 3 perfmon event select MSR.
DDBH	3547	MSR_C3_PMON_CTR5	Package	Uncore C-box 3 perfmon counter MSR.
DE0H	3552	MSR_C7_PMON_BOX_CTRL	Package	Uncore C-box 7 perfmon local box control MSR.
DE1H	3553	MSR_C7_PMON_BOX_ STATUS	Package	Uncore C-box 7 perfmon local box status MSR.
DE2H	3554	MSR_C7_PMON_BOX_OVF_ CTRL	Package	Uncore C-box 7 perfmon local box overflow control MSR.
DF0H	3568	MSR_C7_PMON_EVNT_ SEL0	Package	Uncore C-box 7 perfmon event select MSR.
DF1H	3569	MSR_C7_PMON_CTR0	Package	Uncore C-box 7 perfmon counter MSR.
DF2H	3570	MSR_C7_PMON_EVNT_ SEL1	Package	Uncore C-box 7 perfmon event select MSR.
DF3H	3571	MSR_C7_PMON_CTR1	Package	Uncore C-box 7 perfmon counter MSR.
DF4H	3572	MSR_C7_PMON_EVNT_ SEL2	Package	Uncore C-box 7 perfmon event select MSR.
DF5H	3573	MSR_C7_PMON_CTR2	Package	Uncore C-box 7 perfmon counter MSR.
DF6H	3574	MSR_C7_PMON_EVNT_ SEL3	Package	Uncore C-box 7 perfmon event select MSR.
DF7H	3575	MSR_C7_PMON_CTR3	Package	Uncore C-box 7 perfmon counter MSR.
DF8H	3576	MSR_C7_PMON_EVNT_ SEL4	Package	Uncore C-box 7 perfmon event select MSR.
DF9H	3577	MSR_C7_PMON_CTR4	Package	Uncore C-box 7 perfmon counter MSR.
DFAH	3578	MSR_C7_PMON_EVNT_ SEL5	Package	Uncore C-box 7 perfmon event select MSR.
DFBH	3579	MSR_C7_PMON_CTR5	Package	Uncore C-box 7 perfmon counter MSR.
E00H	3584	MSR_R0_PMON_BOX_CTRL	Package	Uncore R-box 0 perfmon local box control MSR.
E01H	3585	MSR_R0_PMON_BOX_ STATUS	Package	Uncore R-box 0 perfmon local box status MSR.
E02H	3586	MSR_R0_PMON_BOX_OVF_ CTRL	Package	Uncore R-box 0 perfmon local box overflow control MSR.
E04H	3588	MSR_R0_PMON_IPERF0_P0	Package	Uncore R-box 0 perfmon IPERF0 unit Port 0 select MSR.

Table 2-16. Additional MSRs in Intel® Xeon® Processor 7500 Series (Contd.)

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
E05H	3589	MSR_R0_PMON_IPERF0_P1	Package	Uncore R-box 0 perfmon IPERF0 unit Port 1 select MSR.
E06H	3590	MSR_R0_PMON_IPERF0_P2	Package	Uncore R-box 0 perfmon IPERF0 unit Port 2 select MSR.
E07H	3591	MSR_R0_PMON_IPERF0_P3	Package	Uncore R-box 0 perfmon IPERF0 unit Port 3 select MSR.
E08H	3592	MSR_R0_PMON_IPERF0_P4	Package	Uncore R-box 0 perfmon IPERF0 unit Port 4 select MSR.
E09H	3593	MSR_R0_PMON_IPERF0_P5	Package	Uncore R-box 0 perfmon IPERF0 unit Port 5 select MSR.
E0AH	3594	MSR_R0_PMON_IPERF0_P6	Package	Uncore R-box 0 perfmon IPERF0 unit Port 6 select MSR.
E0BH	3595	MSR_R0_PMON_IPERF0_P7	Package	Uncore R-box 0 perfmon IPERF0 unit Port 7 select MSR.
E0CH	3596	MSR_R0_PMON_QLX_P0	Package	Uncore R-box 0 perfmon QLX unit Port 0 select MSR.
E0DH	3597	MSR_R0_PMON_QLX_P1	Package	Uncore R-box 0 perfmon QLX unit Port 1 select MSR.
E0EH	3598	MSR_R0_PMON_QLX_P2	Package	Uncore R-box 0 perfmon QLX unit Port 2 select MSR.
E0FH	3599	MSR_R0_PMON_QLX_P3	Package	Uncore R-box 0 perfmon QLX unit Port 3 select MSR.
E10H	3600	MSR_R0_PMON_EVNT_SEL0	Package	Uncore R-box 0 perfmon event select MSR.
E11H	3601	MSR_R0_PMON_CTR0	Package	Uncore R-box 0 perfmon counter MSR.
E12H	3602	MSR_R0_PMON_EVNT_SEL1	Package	Uncore R-box 0 perfmon event select MSR.
E13H	3603	MSR_R0_PMON_CTR1	Package	Uncore R-box 0 perfmon counter MSR.
E14H	3604	MSR_R0_PMON_EVNT_SEL2	Package	Uncore R-box 0 perfmon event select MSR.
E15H	3605	MSR_R0_PMON_CTR2	Package	Uncore R-box 0 perfmon counter MSR.
E16H	3606	MSR_R0_PMON_EVNT_SEL3	Package	Uncore R-box 0 perfmon event select MSR.
E17H	3607	MSR_R0_PMON_CTR3	Package	Uncore R-box 0 perfmon counter MSR.
E18H	3608	MSR_R0_PMON_EVNT_SEL4	Package	Uncore R-box 0 perfmon event select MSR.
E19H	3609	MSR_R0_PMON_CTR4	Package	Uncore R-box 0 perfmon counter MSR.
E1AH	3610	MSR_R0_PMON_EVNT_SEL5	Package	Uncore R-box 0 perfmon event select MSR.
E1BH	3611	MSR_R0_PMON_CTR5	Package	Uncore R-box 0 perfmon counter MSR.
E1CH	3612	MSR_R0_PMON_EVNT_SEL6	Package	Uncore R-box 0 perfmon event select MSR.
E1DH	3613	MSR_R0_PMON_CTR6	Package	Uncore R-box 0 perfmon counter MSR.
E1EH	3614	MSR_R0_PMON_EVNT_SEL7	Package	Uncore R-box 0 perfmon event select MSR.
E1FH	3615	MSR_R0_PMON_CTR7	Package	Uncore R-box 0 perfmon counter MSR.
E20H	3616	MSR_R1_PMON_BOX_CTRL	Package	Uncore R-box 1 perfmon local box control MSR.
E21H	3617	MSR_R1_PMON_BOX_STATUS	Package	Uncore R-box 1 perfmon local box status MSR.

Table 2-16. Additional MSRs in Intel® Xeon® Processor 7500 Series (Contd.)

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
E22H	3618	MSR_R1_PMON_BOX_OVF_CTRL	Package	Uncore R-box 1 perfmon local box overflow control MSR.
E24H	3620	MSR_R1_PMON_IPERF1_P8	Package	Uncore R-box 1 perfmon IPERF1 unit Port 8 select MSR.
E25H	3621	MSR_R1_PMON_IPERF1_P9	Package	Uncore R-box 1 perfmon IPERF1 unit Port 9 select MSR.
E26H	3622	MSR_R1_PMON_IPERF1_P10	Package	Uncore R-box 1 perfmon IPERF1 unit Port 10 select MSR.
E27H	3623	MSR_R1_PMON_IPERF1_P11	Package	Uncore R-box 1 perfmon IPERF1 unit Port 11 select MSR.
E28H	3624	MSR_R1_PMON_IPERF1_P12	Package	Uncore R-box 1 perfmon IPERF1 unit Port 12 select MSR.
E29H	3625	MSR_R1_PMON_IPERF1_P13	Package	Uncore R-box 1 perfmon IPERF1 unit Port 13 select MSR.
E2AH	3626	MSR_R1_PMON_IPERF1_P14	Package	Uncore R-box 1 perfmon IPERF1 unit Port 14 select MSR.
E2BH	3627	MSR_R1_PMON_IPERF1_P15	Package	Uncore R-box 1 perfmon IPERF1 unit Port 15 select MSR.
E2CH	3628	MSR_R1_PMON_QLX_P4	Package	Uncore R-box 1 perfmon QLX unit Port 4 select MSR.
E2DH	3629	MSR_R1_PMON_QLX_P5	Package	Uncore R-box 1 perfmon QLX unit Port 5 select MSR.
E2EH	3630	MSR_R1_PMON_QLX_P6	Package	Uncore R-box 1 perfmon QLX unit Port 6 select MSR.
E2FH	3631	MSR_R1_PMON_QLX_P7	Package	Uncore R-box 1 perfmon QLX unit Port 7 select MSR.
E30H	3632	MSR_R1_PMON_EVNT_SEL8	Package	Uncore R-box 1 perfmon event select MSR.
E31H	3633	MSR_R1_PMON_CTR8	Package	Uncore R-box 1 perfmon counter MSR.
E32H	3634	MSR_R1_PMON_EVNT_SEL9	Package	Uncore R-box 1 perfmon event select MSR.
E33H	3635	MSR_R1_PMON_CTR9	Package	Uncore R-box 1 perfmon counter MSR.
E34H	3636	MSR_R1_PMON_EVNT_SEL10	Package	Uncore R-box 1 perfmon event select MSR.
E35H	3637	MSR_R1_PMON_CTR10	Package	Uncore R-box 1 perfmon counter MSR.
E36H	3638	MSR_R1_PMON_EVNT_SEL11	Package	Uncore R-box 1 perfmon event select MSR.
E37H	3639	MSR_R1_PMON_CTR11	Package	Uncore R-box 1 perfmon counter MSR.
E38H	3640	MSR_R1_PMON_EVNT_SEL12	Package	Uncore R-box 1 perfmon event select MSR.
E39H	3641	MSR_R1_PMON_CTR12	Package	Uncore R-box 1 perfmon counter MSR.
E3AH	3642	MSR_R1_PMON_EVNT_SEL13	Package	Uncore R-box 1 perfmon event select MSR.
E3BH	3643	MSR_R1_PMON_CTR13	Package	Uncore R-box 1perfmon counter MSR.
E3CH	3644	MSR_R1_PMON_EVNT_SEL14	Package	Uncore R-box 1 perfmon event select MSR.

Table 2-16. Additional MSRs in Intel® Xeon® Processor 7500 Series (Contd.)

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
E3DH	3645	MSR_R1_PMON_CTR14	Package	Uncore R-box 1 perfmon counter MSR.
E3EH	3646	MSR_R1_PMON_EVNT_SEL15	Package	Uncore R-box 1 perfmon event select MSR.
E3FH	3647	MSR_R1_PMON_CTR15	Package	Uncore R-box 1 perfmon counter MSR.
E45H	3653	MSR_B0_PMON_MATCH	Package	Uncore B-box 0 perfmon local box match MSR.
E46H	3654	MSR_B0_PMON_MASK	Package	Uncore B-box 0 perfmon local box mask MSR.
E49H	3657	MSR_S0_PMON_MATCH	Package	Uncore S-box 0 perfmon local box match MSR.
E4AH	3658	MSR_S0_PMON_MASK	Package	Uncore S-box 0 perfmon local box mask MSR.
E4DH	3661	MSR_B1_PMON_MATCH	Package	Uncore B-box 1 perfmon local box match MSR.
E4EH	3662	MSR_B1_PMON_MASK	Package	Uncore B-box 1 perfmon local box mask MSR.
E54H	3668	MSR_M0_PMON_MM_CONFIG	Package	Uncore M-box 0 perfmon local box address match/mask config MSR.
E55H	3669	MSR_M0_PMON_ADDR_MATCH	Package	Uncore M-box 0 perfmon local box address match MSR.
E56H	3670	MSR_M0_PMON_ADDR_MASK	Package	Uncore M-box 0 perfmon local box address mask MSR.
E59H	3673	MSR_S1_PMON_MATCH	Package	Uncore S-box 1 perfmon local box match MSR.
E5AH	3674	MSR_S1_PMON_MASK	Package	Uncore S-box 1 perfmon local box mask MSR.
E5CH	3676	MSR_M1_PMON_MM_CONFIG	Package	Uncore M-box 1 perfmon local box address match/mask config MSR.
E5DH	3677	MSR_M1_PMON_ADDR_MATCH	Package	Uncore M-box 1 perfmon local box address match MSR.
E5EH	3678	MSR_M1_PMON_ADDR_MASK	Package	Uncore M-box 1 perfmon local box address mask MSR.
3B5H	965	MSR_UNCORE_PMC5	Package	See Section 18.3.1.2.2, "Uncore Performance Event Configuration Facility."

2.8 MSRS IN THE INTEL® XEON® PROCESSOR 5600 SERIES (BASED ON INTEL® MICROARCHITECTURE CODE NAME WESTMERE)

Intel® Xeon® Processor 5600 Series (based on Intel® microarchitecture code name Westmere) supports the MSR interfaces listed in Table 2-14, Table 2-15, plus additional MSR listed in Table 2-17. These MSRs apply to Intel Core i7, i5 and i3 processor family with CPUID signature DisplayFamily_DisplayModel of 06_25H and 06_2CH, see Table 2-1.

Table 2-17. Additional MSRs Supported by Intel Processors (Based on Intel® Microarchitecture Code Name Westmere)

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
13CH	52	MSR_FEATURE_CONFIG	Core	**AES Configuration (RW-L)** Privileged post-BIOS agent must provide a #GP handler to handle unsuccessful read of this MSR.
		1:0		**AES Configuration (RW-L)** Upon a successful read of this MSR, the configuration of AES instruction set availability is as follows: 11b: AES instructions are not available until next RESET. otherwise, AES instructions are available. Note, AES instruction set is not available if read is unsuccessful. If the configuration is not 01b, AES instruction can be mis-configured if a privileged agent unintentionally writes 11b.
		63:2		Reserved.
1A7H	423	MSR_OFFCORE_RSP_1	Thread	**Offcore Response Event Select Register (R/W)**
1ADH	429	MSR_TURBO_RATIO_LIMIT	Package	**Maximum Ratio Limit of Turbo Mode** **RO** if MSR_PLATFORM_INFO.[28] = 0, **RW** if MSR_PLATFORM_INFO.[28] = 1
		7:0	Package	**Maximum Ratio Limit for 1C** Maximum turbo ratio limit of 1 core active.
		15:8	Package	**Maximum Ratio Limit for 2C** Maximum turbo ratio limit of 2 core active.
		23:16	Package	**Maximum Ratio Limit for 3C** Maximum turbo ratio limit of 3 core active.
		31:24	Package	**Maximum Ratio Limit for 4C** Maximum turbo ratio limit of 4 core active.
		39:32	Package	**Maximum Ratio Limit for 5C** Maximum turbo ratio limit of 5 core active.
		47:40	Package	**Maximum Ratio Limit for 6C** Maximum turbo ratio limit of 6 core active.
		63:48		Reserved.
1B0H	432	IA32_ENERGY_PERF_BIAS	Package	See Table 2-2.

2.9 MSRS IN THE INTEL® XEON® PROCESSOR E7 FAMILY (BASED ON INTEL® MICROARCHITECTURE CODE NAME WESTMERE)

Intel® Xeon® Processor E7 Family (based on Intel® microarchitecture code name Westmere) supports the MSR interfaces listed in Table 2-14 (except MSR address 1ADH), Table 2-15, plus additional MSR listed in Table 2-18. These processors have a CPUID signature with DisplayFamily_DisplayModel of 06_2FH.

Table 2-18. Additional MSRs Supported by Intel® Xeon® Processor E7 Family

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
13CH	52	MSR_FEATURE_CONFIG	Core	**AES Configuration (RW-L)** Privileged post-BIOS agent must provide a #GP handler to handle unsuccessful read of this MSR.
		1:0		**AES Configuration (RW-L)** Upon a successful read of this MSR, the configuration of AES instruction set availability is as follows: 11b: AES instructions are not available until next RESET. otherwise, AES instructions are available. Note, AES instruction set is not available if read is unsuccessful. If the configuration is not 01b, AES instruction can be mis-configured if a privileged agent unintentionally writes 11b.
		63:2		Reserved.
1A7H	423	MSR_OFFCORE_RSP_1	Thread	**Offcore Response Event Select Register (R/W)**
1ADH	429	MSR_TURBO_RATIO_LIMIT	Package	**Reserved** Attempt to read/write will cause #UD.
1B0H	432	IA32_ENERGY_PERF_BIAS	Package	See Table 2-2.
F40H	3904	MSR_C8_PMON_BOX_CTRL	Package	Uncore C-box 8 perfmon local box control MSR.
F41H	3905	MSR_C8_PMON_BOX_ STATUS	Package	Uncore C-box 8 perfmon local box status MSR.
F42H	3906	MSR_C8_PMON_BOX_OVF_ CTRL	Package	Uncore C-box 8 perfmon local box overflow control MSR.
F50H	3920	MSR_C8_PMON_EVNT_ SEL0	Package	Uncore C-box 8 perfmon event select MSR.
F51H	3921	MSR_C8_PMON_CTR0	Package	Uncore C-box 8 perfmon counter MSR.
F52H	3922	MSR_C8_PMON_EVNT_ SEL1	Package	Uncore C-box 8 perfmon event select MSR.
F53H	3923	MSR_C8_PMON_CTR1	Package	Uncore C-box 8 perfmon counter MSR.
F54H	3924	MSR_C8_PMON_EVNT_ SEL2	Package	Uncore C-box 8 perfmon event select MSR.
F55H	3925	MSR_C8_PMON_CTR2	Package	Uncore C-box 8 perfmon counter MSR.
F56H	3926	MSR_C8_PMON_EVNT_ SEL3	Package	Uncore C-box 8 perfmon event select MSR.
F57H	3927	MSR_C8_PMON_CTR3	Package	Uncore C-box 8 perfmon counter MSR.
F58H	3928	MSR_C8_PMON_EVNT_ SEL4	Package	Uncore C-box 8 perfmon event select MSR.
F59H	3929	MSR_C8_PMON_CTR4	Package	Uncore C-box 8 perfmon counter MSR.
F5AH	3930	MSR_C8_PMON_EVNT_ SEL5	Package	Uncore C-box 8 perfmon event select MSR.
F5BH	3931	MSR_C8_PMON_CTR5	Package	Uncore C-box 8 perfmon counter MSR.

Table 2-18. Additional MSRs Supported by Intel® Xeon® Processor E7 Family (Contd.)

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
FC0H	4032	MSR_C9_PMON_BOX_CTRL	Package	Uncore C-box 9 perfmon local box control MSR.
FC1H	4033	MSR_C9_PMON_BOX_ STATUS	Package	Uncore C-box 9 perfmon local box status MSR.
FC2H	4034	MSR_C9_PMON_BOX_OVF_ CTRL	Package	Uncore C-box 9 perfmon local box overflow control MSR.
FD0H	4048	MSR_C9_PMON_EVNT_ SEL0	Package	Uncore C-box 9 perfmon event select MSR.
FD1H	4049	MSR_C9_PMON_CTR0	Package	Uncore C-box 9 perfmon counter MSR.
FD2H	4050	MSR_C9_PMON_EVNT_ SEL1	Package	Uncore C-box 9 perfmon event select MSR.
FD3H	4051	MSR_C9_PMON_CTR1	Package	Uncore C-box 9 perfmon counter MSR.
FD4H	4052	MSR_C9_PMON_EVNT_ SEL2	Package	Uncore C-box 9 perfmon event select MSR.
FD5H	4053	MSR_C9_PMON_CTR2	Package	Uncore C-box 9 perfmon counter MSR.
FD6H	4054	MSR_C9_PMON_EVNT_ SEL3	Package	Uncore C-box 9 perfmon event select MSR.
FD7H	4055	MSR_C9_PMON_CTR3	Package	Uncore C-box 9 perfmon counter MSR.
FD8H	4056	MSR_C9_PMON_EVNT_ SEL4	Package	Uncore C-box 9 perfmon event select MSR.
FD9H	4057	MSR_C9_PMON_CTR4	Package	Uncore C-box 9 perfmon counter MSR.
FDAH	4058	MSR_C9_PMON_EVNT_ SEL5	Package	Uncore C-box 9 perfmon event select MSR.
FDBH	4059	MSR_C9_PMON_CTR5	Package	Uncore C-box 9 perfmon counter MSR.

2.10 MSRS IN INTEL® PROCESSOR FAMILY BASED ON INTEL® MICROARCHITECTURE CODE NAME SANDY BRIDGE

Table 2-19 lists model-specific registers (MSRs) that are common to Intel® processor family based on Intel micro-architecture code name Sandy Bridge. These processors have a CPUID signature with DisplayFamily_DisplayModel of 06_2AH, 06_2DH, see Table 2-1. Additional MSRs specific to 06_2AH are listed in Table 2-20.

Table 2-19. MSRs Supported by Intel® Processors based on Intel® microarchitecture code name Sandy Bridge

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
0H	0	IA32_P5_MC_ADDR	Thread	See Section 2.22, "MSRs in Pentium Processors."
1H	1	IA32_P5_MC_TYPE	Thread	See Section 2.22, "MSRs in Pentium Processors."

Table 2-19. MSRs Supported by Intel® Processors based on Intel® microarchitecture code name Sandy Bridge (Contd.)

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
6H	6	IA32_MONITOR_FILTER_SIZE	Thread	See Section 8.10.5, "Monitor/Mwait Address Range Determination," and Table 2-2.
10H	16	IA32_TIME_STAMP_COUNTER	Thread	See Section 17.17, "Time-Stamp Counter," and see Table 2-2.
17H	23	IA32_PLATFORM_ID	Package	**Platform ID (R)** See Table 2-2.
1BH	27	IA32_APIC_BASE	Thread	See Section 10.4.4, "Local APIC Status and Location," and Table 2-2.
34H	52	MSR_SMI_COUNT	Thread	**SMI Counter (R/O)**
		31:0		**SMI Count (R/O)** Count SMIs.
		63:32		Reserved.
3AH	58	IA32_FEATURE_CONTROL	Thread	**Control Features in Intel 64 Processor (R/W)** See Table 2-2.
		0		**Lock (R/WL)**
		1		**Enable VMX inside SMX operation (R/WL)**
		2		**Enable VMX outside SMX operation (R/WL)**
		14:8		**SENTER local functions enables (R/WL)**
		15		**SENTER global functions enable (R/WL)**
79H	121	IA32_BIOS_UPDT_TRIG	Core	**BIOS Update Trigger Register (W)** See Table 2-2.
8BH	139	IA32_BIOS_SIGN_ID	Thread	**BIOS Update Signature ID (RO)** See Table 2-2.
C1H	193	IA32_PMC0	Thread	**Performance Counter Register** See Table 2-2.
C2H	194	IA32_PMC1	Thread	**Performance Counter Register** See Table 2-2.
C3H	195	IA32_PMC2	Thread	**Performance Counter Register** See Table 2-2.
C4H	196	IA32_PMC3	Thread	**Performance Counter Register** See Table 2-2.
C5H	197	IA32_PMC4	Core	**Performance Counter Register (if core not shared by threads)**
C6H	198	IA32_PMC5	Core	**Performance Counter Register (if core not shared by threads)**
C7H	199	IA32_PMC6	Core	**Performance Counter Register (if core not shared by threads)**
C8H	200	IA32_PMC7	Core	**Performance Counter Register (if core not shared by threads)**
CEH	206	MSR_PLATFORM_INFO	Package	Platform Information; contains power management and other model specific features enumeration. See http://biosbits.org.
		7:0		Reserved.

Table 2-19. MSRs Supported by Intel® Processors based on Intel® microarchitecture code name Sandy Bridge (Contd.)

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
		15:8	Package	**Maximum Non-Turbo Ratio (R/O)** The is the ratio of the frequency that invariant TSC runs at. Frequency = ratio * 100 MHz.
		27:16		Reserved.
		28	Package	**Programmable Ratio Limit for Turbo Mode (R/O)** When set to 1, indicates that Programmable Ratio Limits for Turbo mode is enabled, and when set to 0, indicates Programmable Ratio Limits for Turbo mode is disabled.
		29	Package	**Programmable TDP Limit for Turbo Mode (R/O)** When set to 1, indicates that TDP Limits for Turbo mode are programmable, and when set to 0, indicates TDP Limit for Turbo mode is not programmable.
		39:30		Reserved.
		47:40	Package	**Maximum Efficiency Ratio (R/O)** The is the minimum ratio (maximum efficiency) that the processor can operates, in units of 100MHz.
		63:48		Reserved.
E2H	226	MSR_PKG_CST_CONFIG_ CONTROL	Core	**C-State Configuration Control (R/W)** Note: C-state values are processor specific C-state code names, unrelated to MWAIT extension C-state parameters or ACPI C-States. See http://biosbits.org.
		2:0		**Package C-State Limit (R/W)** Specifies the lowest processor-specific C-state code name (consuming the least power). for the package. The default is set as factory-configured package C-state limit. The following C-state code name encodings are supported: 000b: C0/C1 (no package C-sate support) 001b: C2 010b: C6 no retention 011b: C6 retention 100b: C7 101b: C7s 111: No package C-state limit. Note: This field cannot be used to limit package C-state to C3.
		9:3		Reserved.
		10		**I/O MWAIT Redirection Enable (R/W)** When set, will map IO_read instructions sent to IO register specified by MSR_PMG_IO_CAPTURE_BASE to MWAIT instructions
		14:11		Reserved.
		15		**CFG Lock (R/WO)** When set, lock bits 15:0 of this register until next reset.

Table 2-19. MSRs Supported by Intel® Processors based on Intel® microarchitecture code name Sandy Bridge (Contd.)

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
		24:16		Reserved.
		25		**C3 state auto demotion enable (R/W)** When set, the processor will conditionally demote C6/C7 requests to C3 based on uncore auto-demote information.
		26		**C1 state auto demotion enable (R/W)** When set, the processor will conditionally demote C3/C6/C7 requests to C1 based on uncore auto-demote information.
		27		**Enable C3 undemotion (R/W)** When set, enables undemotion from demoted C3.
		28		**Enable C1 undemotion (R/W)** When set, enables undemotion from demoted C1.
		63:29		Reserved.
E4H	228	MSR_PMG_IO_CAPTURE_BASE	Core	**Power Management IO Redirection in C-state (R/W)** See http://biosbits.org.
		15:0		**LVL_2 Base Address (R/W)** Specifies the base address visible to software for IO redirection. If IO MWAIT Redirection is enabled, reads to this address will be consumed by the power management logic and decoded to MWAIT instructions. When IO port address redirection is enabled, this is the IO port address reported to the OS/software.
		18:16		**C-state Range (R/W)** Specifies the encoding value of the maximum C-State code name to be included when IO read to MWAIT redirection is enabled by MSR_PKG_CST_CONFIG_CONTROL[bit10]: 000b - C3 is the max C-State to include 001b - C6 is the max C-State to include 010b - C7 is the max C-State to include
		63:19		Reserved.
E7H	231	IA32_MPERF	Thread	**Maximum Performance Frequency Clock Count (RW)** See Table 2-2.
E8H	232	IA32_APERF	Thread	**Actual Performance Frequency Clock Count (RW)** See Table 2-2.
FEH	254	IA32_MTRRCAP	Thread	See Table 2-2.
13CH	52	MSR_FEATURE_CONFIG	Core	**AES Configuration (RW-L)** Privileged post-BIOS agent must provide a #GP handler to handle unsuccessful read of this MSR.

Table 2-19. MSRs Supported by Intel® Processors based on Intel® microarchitecture code name Sandy Bridge (Contd.)

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
		1:0		**AES Configuration (RW-L)** Upon a successful read of this MSR, the configuration of AES instruction set availability is as follows: 11b: AES instructions are not available until next RESET. otherwise, AES instructions are available. Note, AES instruction set is not available if read is unsuccessful. If the configuration is not 01b, AES instruction can be mis-configured if a privileged agent unintentionally writes 11b.
		63:2		Reserved.
174H	372	IA32_SYSENTER_CS	Thread	See Table 2-2.
175H	373	IA32_SYSENTER_ESP	Thread	See Table 2-2.
176H	374	IA32_SYSENTER_EIP	Thread	See Table 2-2.
179H	377	IA32_MCG_CAP	Thread	See Table 2-2.
17AH	378	IA32_MCG_STATUS	Thread	Global Machine Check Status
		0		**RIPV** When set, bit indicates that the instruction addressed by the instruction pointer pushed on the stack (when the machine check was generated) can be used to restart the program. If cleared, the program cannot be reliably restarted.
		1		**EIPV** When set, bit indicates that the instruction addressed by the instruction pointer pushed on the stack (when the machine check was generated) is directly associated with the error.
		2		**MCIP** When set, bit indicates that a machine check has been generated. If a second machine check is detected while this bit is still set, the processor enters a shutdown state. Software should write this bit to 0 after processing a machine check exception.
		63:3		Reserved.
186H	390	IA32_ PERFEVTSEL0	Thread	See Table 2-2.
187H	391	IA32_ PERFEVTSEL1	Thread	See Table 2-2.
188H	392	IA32_ PERFEVTSEL2	Thread	See Table 2-2.
189H	393	IA32_ PERFEVTSEL3	Thread	See Table 2-2.
18AH	394	IA32_ PERFEVTSEL4	Core	See Table 2-2; If CPUID.0AH:EAX[15:8] = 8
18BH	395	IA32_ PERFEVTSEL5	Core	See Table 2-2; If CPUID.0AH:EAX[15:8] = 8
18CH	396	IA32_ PERFEVTSEL6	Core	See Table 2-2; If CPUID.0AH:EAX[15:8] = 8

Table 2-19. MSRs Supported by Intel® Processors based on Intel® microarchitecture code name Sandy Bridge (Contd.)

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
18DH	397	IA32_ PERFEVTSEL7	Core	See Table 2-2; If CPUID.0AH:EAX[15:8] = 8
198H	408	IA32_PERF_STATUS	Package	See Table 2-2.
		15:0		Current Performance State Value.
		63:16		Reserved.
198H	408	MSR_PERF_STATUS	Package	Performance Status
		47:32		Core Voltage (R/O) P-state core voltage can be computed by MSR_PERF_STATUS[37:32] * (float) 1/(2^13).
199H	409	IA32_PERF_CTL	Thread	See Table 2-2.
19AH	410	IA32_CLOCK_ MODULATION	Thread	**Clock Modulation (R/W)** See Table 2-2 IA32_CLOCK_MODULATION MSR was originally named IA32_THERM_CONTROL MSR.
		3:0		**On demand Clock Modulation Duty Cycle (R/W)** In 6.25% increment
		4		**On demand Clock Modulation Enable (R/W)**
		63:5		Reserved.
19BH	411	IA32_THERM_INTERRUPT	Core	**Thermal Interrupt Control (R/W)** See Table 2-2.
19CH	412	IA32_THERM_STATUS	Core	**Thermal Monitor Status (R/W)** See Table 2-2.
		0		**Thermal status (RO)** See Table 2-2.
		1		**Thermal status log (R/WC0)** See Table 2-2.
		2		**PROTCHOT # or FORCEPR# status (RO)** See Table 2-2.
		3		**PROTCHOT # or FORCEPR# log (R/WC0)** See Table 2-2.
		4		**Critical Temperature status (RO)** See Table 2-2.
		5		**Critical Temperature status log (R/WC0)** See Table 2-2.
		6		**Thermal threshold #1 status (RO)** See Table 2-2.
		7		**Thermal threshold #1 log (R/WC0)** See Table 2-2.

Table 2-19. MSRs Supported by Intel® Processors based on Intel® microarchitecture code name Sandy Bridge (Contd.)

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
		8		**Thermal threshold #2 status (RO)** See Table 2-2.
		9		**Thermal threshold #2 log (R/WC0)** See Table 2-2.
		10		**Power Limitation status (RO)** See Table 2-2.
		11		**Power Limitation log (R/WC0)** See Table 2-2.
		15:12		Reserved.
		22:16		**Digital Readout (RO)** See Table 2-2.
		26:23		Reserved.
		30:27		**Resolution in degrees Celsius (RO)** See Table 2-2.
		31		**Reading Valid (RO)** See Table 2-2.
		63:32		Reserved.
1A0H	416	IA32_MISC_ENABLE		**Enable Misc. Processor Features (R/W)** Allows a variety of processor functions to be enabled and disabled.
		0	Thread	**Fast-Strings Enable** See Table 2-2
		6:1		Reserved.
		7	Thread	**Performance Monitoring Available (R)** See Table 2-2.
		10:8		Reserved.
		11	Thread	**Branch Trace Storage Unavailable (RO)** See Table 2-2.
		12	Thread	**Processor Event Based Sampling Unavailable (RO)** See Table 2-2.
		15:13		Reserved.
		16	Package	**Enhanced Intel SpeedStep Technology Enable (R/W)** See Table 2-2.
		18	Thread	ENABLE MONITOR FSM. (R/W) See Table 2-2.
		21:19		Reserved.
		22	Thread	**Limit CPUID Maxval (R/W)** See Table 2-2.
		23	Thread	**xTPR Message Disable (R/W)** See Table 2-2.

Table 2-19. MSRs Supported by Intel® Processors based on Intel® microarchitecture code name Sandy Bridge (Contd.)

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
		33:24		Reserved.
		34	Thread	**XD Bit Disable (R/W)** See Table 2-2.
		37:35		Reserved.
		38	Package	**Turbo Mode Disable (R/W)** When set to 1 on processors that support Intel Turbo Boost Technology, the turbo mode feature is disabled and the IDA_Enable feature flag will be clear (CPUID.06H: EAX[1]=0). When set to a 0 on processors that support IDA, CPUID.06H: EAX[1] reports the processor's support of turbo mode is enabled. **Note:** the power-on default value is used by BIOS to detect hardware support of turbo mode. If power-on default value is 1, turbo mode is available in the processor. If power-on default value is 0, turbo mode is not available.
		63:39		Reserved.
1A2H	418	MSR_ TEMPERATURE_TARGET	Unique	Temperature Target
		15:0		Reserved.
		23:16		**Temperature Target (R)** The minimum temperature at which PROCHOT# will be asserted. The value is degree C.
		63:24		Reserved.
1A4H	420	MSR_MISC_FEATURE_ CONTROL		**Miscellaneous Feature Control (R/W)**
		0	Core	**L2 Hardware Prefetcher Disable (R/W)** If 1, disables the L2 hardware prefetcher, which fetches additional lines of code or data into the L2 cache.
		1	Core	**L2 Adjacent Cache Line Prefetcher Disable (R/W)** If 1, disables the adjacent cache line prefetcher, which fetches the cache line that comprises a cache line pair (128 bytes).
		2	Core	**DCU Hardware Prefetcher Disable (R/W)** If 1, disables the L1 data cache prefetcher, which fetches the next cache line into L1 data cache.
		3	Core	**DCU IP Prefetcher Disable (R/W)** If 1, disables the L1 data cache IP prefetcher, which uses sequential load history (based on instruction Pointer of previous loads) to determine whether to prefetch additional lines.
		63:4		Reserved.
1A6H	422	MSR_OFFCORE_RSP_0	Thread	**Offcore Response Event Select Register (R/W)**
1A7H	422	MSR_OFFCORE_RSP_1	Thread	**Offcore Response Event Select Register (R/W)**
1AAH	426	MSR_MISC_PWR_MGMT		Miscellaneous Power Management Control; various model specific features enumeration. See http://biosbits.org.

Table 2-19. MSRs Supported by Intel® Processors based on Intel® microarchitecture code name Sandy Bridge (Contd.)

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
1B0H	432	IA32_ENERGY_PERF_BIAS	Package	See Table 2-2.
1B1H	433	IA32_PACKAGE_THERM_ STATUS	Package	See Table 2-2.
1B2H	434	IA32_PACKAGE_THERM_ INTERRUPT	Package	See Table 2-2.
1C8H	456	MSR_LBR_SELECT	Thread	**Last Branch Record Filtering Select Register (R/W)** See Section 17.9.2, "Filtering of Last Branch Records."
		0		**CPL_EQ_0**
		1		**CPL_NEQ_0**
		2		**JCC**
		3		**NEAR_REL_CALL**
		4		**NEAR_IND_CALL**
		5		**NEAR_RET**
		6		**NEAR_IND_JMP**
		7		**NEAR_REL_JMP**
		8		**FAR_BRANCH**
		63:9		Reserved.
1C9H	457	MSR_LASTBRANCH_TOS	Thread	**Last Branch Record Stack TOS (R/W)** Contains an index (bits 0-3) that points to the MSR containing the most recent branch record. See MSR_LASTBRANCH_0_FROM_IP (at 680H).
1D9H	473	IA32_DEBUGCTL	Thread	**Debug Control (R/W)** See Table 2-2.
		0		**LBR: Last Branch Record**
		1		**BTF**
		5:2		Reserved.
		6		**TR: Branch Trace**
		7		**BTS: Log Branch Trace Message to BTS buffer**
		8		**BTINT**
		9		**BTS_OFF_OS**
		10		**BTS_OFF_USER**
		11		**FREEZE_LBR_ON_PMI**
		12		**FREEZE_PERFMON_ON_PMI**
		13		**ENABLE_UNCORE_PMI**
		14		**FREEZE_WHILE_SMM**
		63:15		Reserved.

Table 2-19. MSRs Supported by Intel® Processors based on Intel® microarchitecture code name Sandy Bridge (Contd.)

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
1DDH	477	MSR_LER_FROM_LIP	Thread	**Last Exception Record From Linear IP (R)** Contains a pointer to the last branch instruction that the processor executed prior to the last exception that was generated or the last interrupt that was handled.
1DEH	478	MSR_LER_TO_LIP	Thread	**Last Exception Record To Linear IP (R)** This area contains a pointer to the target of the last branch instruction that the processor executed prior to the last exception that was generated or the last interrupt that was handled.
1F2H	498	IA32_SMRR_PHYSBASE	Core	See Table 2-2.
1F3H	499	IA32_SMRR_PHYSMASK	Core	See Table 2-2.
1FCH	508	MSR_POWER_CTL	Core	See http://biosbits.org.
200H	512	IA32_MTRR_PHYSBASE0	Thread	See Table 2-2.
201H	513	IA32_MTRR_PHYSMASK0	Thread	See Table 2-2.
202H	514	IA32_MTRR_PHYSBASE1	Thread	See Table 2-2.
203H	515	IA32_MTRR_PHYSMASK1	Thread	See Table 2-2.
204H	516	IA32_MTRR_PHYSBASE2	Thread	See Table 2-2.
205H	517	IA32_MTRR_PHYSMASK2	Thread	See Table 2-2.
206H	518	IA32_MTRR_PHYSBASE3	Thread	See Table 2-2.
207H	519	IA32_MTRR_PHYSMASK3	Thread	See Table 2-2.
208H	520	IA32_MTRR_PHYSBASE4	Thread	See Table 2-2.
209H	521	IA32_MTRR_PHYSMASK4	Thread	See Table 2-2.
20AH	522	IA32_MTRR_PHYSBASE5	Thread	See Table 2-2.
20BH	523	IA32_MTRR_PHYSMASK5	Thread	See Table 2-2.
20CH	524	IA32_MTRR_PHYSBASE6	Thread	See Table 2-2.
20DH	525	IA32_MTRR_PHYSMASK6	Thread	See Table 2-2.
20EH	526	IA32_MTRR_PHYSBASE7	Thread	See Table 2-2.
20FH	527	IA32_MTRR_PHYSMASK7	Thread	See Table 2-2.
210H	528	IA32_MTRR_PHYSBASE8	Thread	See Table 2-2.
211H	529	IA32_MTRR_PHYSMASK8	Thread	See Table 2-2.
212H	530	IA32_MTRR_PHYSBASE9	Thread	See Table 2-2.
213H	531	IA32_MTRR_PHYSMASK9	Thread	See Table 2-2.
250H	592	IA32_MTRR_FIX64K_ 00000	Thread	See Table 2-2.
258H	600	IA32_MTRR_FIX16K_ 80000	Thread	See Table 2-2.
259H	601	IA32_MTRR_FIX16K_ A0000	Thread	See Table 2-2.

Table 2-19. MSRs Supported by Intel® Processors based on Intel® microarchitecture code name Sandy Bridge (Contd.)

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
268H	616	IA32_MTRR_FIX4K_C0000	Thread	See Table 2-2.
269H	617	IA32_MTRR_FIX4K_C8000	Thread	See Table 2-2.
26AH	618	IA32_MTRR_FIX4K_D0000	Thread	See Table 2-2.
26BH	619	IA32_MTRR_FIX4K_D8000	Thread	See Table 2-2.
26CH	620	IA32_MTRR_FIX4K_E0000	Thread	See Table 2-2.
26DH	621	IA32_MTRR_FIX4K_E8000	Thread	See Table 2-2.
26EH	622	IA32_MTRR_FIX4K_F0000	Thread	See Table 2-2.
26FH	623	IA32_MTRR_FIX4K_F8000	Thread	See Table 2-2.
277H	631	IA32_PAT	Thread	See Table 2-2.
280H	640	IA32_MC0_CTL2	Core	See Table 2-2.
281H	641	IA32_MC1_CTL2	Core	See Table 2-2.
282H	642	IA32_MC2_CTL2	Core	See Table 2-2.
283H	643	IA32_MC3_CTL2	Core	See Table 2-2.
284H	644	IA32_MC4_CTL2	Package	Always 0 (CMCI not supported).
2FFH	767	IA32_MTRR_DEF_TYPE	Thread	**Default Memory Types (R/W)** See Table 2-2.
309H	777	IA32_FIXED_CTR0	Thread	**Fixed-Function Performance Counter Register 0 (R/W)** See Table 2-2.
30AH	778	IA32_FIXED_CTR1	Thread	**Fixed-Function Performance Counter Register 1 (R/W)** See Table 2-2.
30BH	779	IA32_FIXED_CTR2	Thread	**Fixed-Function Performance Counter Register 2 (R/W)** See Table 2-2.
345H	837	IA32_PERF_CAPABILITIES	Thread	See Table 2-2. See Section 17.4.1, "IA32_DEBUGCTL MSR."
		5:0		LBR Format. See Table 2-2.
		6		PEBS Record Format.
		7		PEBSSaveArchRegs. See Table 2-2.
		11:8		PEBS_REC_FORMAT. See Table 2-2.
		12		SMM_FREEZE. See Table 2-2.
		63:13		Reserved.
38DH	909	IA32_FIXED_CTR_CTRL	Thread	**Fixed-Function-Counter Control Register (R/W)** See Table 2-2.
38EH	910	IA32_PERF_GLOBAL_ STATUS		See Table 2-2. See Section 18.6.2.2, "Global Counter Control Facilities."
		0	Thread	**Ovf_PMC0**
		1	Thread	**Ovf_PMC1**
		2	Thread	**Ovf_PMC2**

Table 2-19. MSRs Supported by Intel® Processors based on Intel® microarchitecture code name Sandy Bridge (Contd.)

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
		3	Thread	**Ovf_PMC3**
		4	Core	**Ovf_PMC4 (if CPUID.0AH:EAX[15:8] > 4)**
		5	Core	**Ovf_PMC5 (if CPUID.0AH:EAX[15:8] > 5)**
		6	Core	**Ovf_PMC6 (if CPUID.0AH:EAX[15:8] > 6)**
		7	Core	**Ovf_PMC7 (if CPUID.0AH:EAX[15:8] > 7)**
		31:8		Reserved.
		32	Thread	**Ovf_FixedCtr0**
		33	Thread	**Ovf_FixedCtr1**
		34	Thread	**Ovf_FixedCtr2**
		60:35		Reserved.
		61	Thread	**Ovf_Uncore**
		62	Thread	**Ovf_BufDSSAVE**
		63	Thread	**CondChgd**
38FH	911	IA32_PERF_GLOBAL_CTRL	Thread	See Table 2-2. See Section 18.6.2.2, "Global Counter Control Facilities."
		0	Thread	**Set 1 to enable PMC0 to count**
		1	Thread	**Set 1 to enable PMC1 to count**
		2	Thread	**Set 1 to enable PMC2 to count**
		3	Thread	**Set 1 to enable PMC3 to count**
		4	Core	**Set 1 to enable PMC4 to count (if CPUID.0AH:EAX[15:8] > 4)**
		5	Core	**Set 1 to enable PMC5 to count (if CPUID.0AH:EAX[15:8] > 5)**
		6	Core	**Set 1 to enable PMC6 to count (if CPUID.0AH:EAX[15:8] > 6)**
		7	Core	**Set 1 to enable PMC7 to count (if CPUID.0AH:EAX[15:8] > 7)**
		31:8		Reserved.
		32	Thread	**Set 1 to enable FixedCtr0 to count**
		33	Thread	**Set 1 to enable FixedCtr1 to count**
		34	Thread	**Set 1 to enable FixedCtr2 to count**
		63:35		Reserved.
390H	912	IA32_PERF_GLOBAL_OVF_CTRL		See Table 2-2. See Section 18.6.2.2, "Global Counter Control Facilities."
		0	Thread	**Set 1 to clear Ovf_PMC0**
		1	Thread	**Set 1 to clear Ovf_PMC1**
		2	Thread	**Set 1 to clear Ovf_PMC2**
		3	Thread	**Set 1 to clear Ovf_PMC3**
		4	Core	**Set 1 to clear Ovf_PMC4 (if CPUID.0AH:EAX[15:8] > 4)**
		5	Core	**Set 1 to clear Ovf_PMC5 (if CPUID.0AH:EAX[15:8] > 5)**
		6	Core	**Set 1 to clear Ovf_PMC6 (if CPUID.0AH:EAX[15:8] > 6)**

Table 2-19. MSRs Supported by Intel® Processors based on Intel® microarchitecture code name Sandy Bridge (Contd.)

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
		7	Core	**Set 1 to clear Ovf_PMC7 (if CPUID.0AH:EAX[15:8] > 7)**
		31:8		Reserved.
		32	Thread	**Set 1 to clear Ovf_FixedCtr0**
		33	Thread	**Set 1 to clear Ovf_FixedCtr1**
		34	Thread	**Set 1 to clear Ovf_FixedCtr2**
		60:35		Reserved.
		61	Thread	**Set 1 to clear Ovf_Uncore**
		62	Thread	**Set 1 to clear Ovf_BufDSSAVE**
		63	Thread	**Set 1 to clear CondChgd**
3F1H	1009	MSR_PEBS_ENABLE	Thread	See Section 18.3.1.1.1, "Processor Event Based Sampling (PEBS)."
		0		Enable PEBS on IA32_PMC0. (R/W)
		1		Enable PEBS on IA32_PMC1. (R/W)
		2		Enable PEBS on IA32_PMC2. (R/W)
		3		Enable PEBS on IA32_PMC3. (R/W)
		31:4		Reserved.
		32		Enable Load Latency on IA32_PMC0. (R/W)
		33		Enable Load Latency on IA32_PMC1. (R/W)
		34		Enable Load Latency on IA32_PMC2. (R/W)
		35		Enable Load Latency on IA32_PMC3. (R/W)
		62:36		Reserved.
		63		Enable Precise Store. (R/W)
3F6H	1014	MSR_PEBS_LD_LAT	Thread	see See Section 18.3.1.1.2, "Load Latency Performance Monitoring Facility."
		15:0		Minimum threshold latency value of tagged load operation that will be counted. (R/W)
		63:36		Reserved.
3F8H	1016	MSR_PKG_C3_RESIDENCY	Package	Note: C-state values are processor specific C-state code names, unrelated to MWAIT extension C-state parameters or ACPI C-States.
		63:0		Package C3 Residency Counter. (R/O) Value since last reset that this package is in processor-specific C3 states. Count at the same frequency as the TSC.
3F9H	1017	MSR_PKG_C6_RESIDENCY	Package	Note: C-state values are processor specific C-state code names, unrelated to MWAIT extension C-state parameters or ACPI C-States.
		63:0		Package C6 Residency Counter. (R/O) Value since last reset that this package is in processor-specific C6 states. Count at the same frequency as the TSC.

Table 2-19. MSRs Supported by Intel® Processors based on Intel® microarchitecture code name Sandy Bridge (Contd.)

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
3FAH	1018	MSR_PKG_C7_RESIDENCY	Package	Note: C-state values are processor specific C-state code names, unrelated to MWAIT extension C-state parameters or ACPI C-States.
		63:0		Package C7 Residency Counter. (R/O) Value since last reset that this package is in processor-specific C7 states. Count at the same frequency as the TSC.
3FCH	1020	MSR_CORE_C3_RESIDENCY	Core	Note: C-state values are processor specific C-state code names, unrelated to MWAIT extension C-state parameters or ACPI C-States.
		63:0		CORE C3 Residency Counter. (R/O) Value since last reset that this core is in processor-specific C3 states. Count at the same frequency as the TSC.
3FDH	1021	MSR_CORE_C6_RESIDENCY	Core	Note: C-state values are processor specific C-state code names, unrelated to MWAIT extension C-state parameters or ACPI C-States.
		63:0		CORE C6 Residency Counter. (R/O) Value since last reset that this core is in processor-specific C6 states. Count at the same frequency as the TSC.
3FEH	1022	MSR_CORE_C7_RESIDENCY	Core	Note: C-state values are processor specific C-state code names, unrelated to MWAIT extension C-state parameters or ACPI C-States.
		63:0		CORE C7 Residency Counter. (R/O) Value since last reset that this core is in processor-specific C7 states. Count at the same frequency as the TSC.
400H	1024	IA32_MC0_CTL	Core	See Section 15.3.2.1, "IA32_MCi_CTL MSRs."
401H	1025	IA32_MC0_STATUS	Core	See Section 15.3.2.2, "IA32_MCi_STATUS MSRS," and Chapter 16.
402H	1026	IA32_MC0_ADDR	Core	See Section 15.3.2.3, "IA32_MCi_ADDR MSRs."
403H	1027	IA32_MC0_MISC	Core	See Section 15.3.2.4, "IA32_MCi_MISC MSRs."
404H	1028	IA32_MC1_CTL	Core	See Section 15.3.2.1, "IA32_MCi_CTL MSRs."
405H	1029	IA32_MC1_STATUS	Core	See Section 15.3.2.2, "IA32_MCi_STATUS MSRS," and Chapter 16.
406H	1030	IA32_MC1_ADDR	Core	See Section 15.3.2.3, "IA32_MCi_ADDR MSRs."
407H	1031	IA32_MC1_MISC	Core	See Section 15.3.2.4, "IA32_MCi_MISC MSRs."
408H	1032	IA32_MC2_CTL	Core	See Section 15.3.2.1, "IA32_MCi_CTL MSRs."
409H	1033	IA32_MC2_STATUS	Core	See Section 15.3.2.2, "IA32_MCi_STATUS MSRS," and Chapter 16.
40AH	1034	IA32_MC2_ADDR	Core	See Section 15.3.2.3, "IA32_MCi_ADDR MSRs."
40BH	1035	IA32_MC2_MISC	Core	See Section 15.3.2.4, "IA32_MCi_MISC MSRs."
40CH	1036	IA32_MC3_CTL	Core	See Section 15.3.2.1, "IA32_MCi_CTL MSRs."
40DH	1037	IA32_MC3_STATUS	Core	See Section 15.3.2.2, "IA32_MCi_STATUS MSRS," and Chapter 16.
40EH	1038	IA32_MC3_ADDR	Core	See Section 15.3.2.3, "IA32_MCi_ADDR MSRs."
40FH	1039	IA32_MC3_MISC	Core	See Section 15.3.2.4, "IA32_MCi_MISC MSRs."

Table 2-19. MSRs Supported by Intel® Processors based on Intel® microarchitecture code name Sandy Bridge (Contd.)

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
410H	1040	IA32_MC4_CTL	Core	See Section 15.3.2.1, "IA32_MCi_CTL MSRs."
		0		**PCU Hardware Error (R/W)** When set, enables signaling of PCU hardware detected errors.
		1		**PCU Controller Error (R/W)** When set, enables signaling of PCU controller detected errors
		2		**PCU Firmware Error (R/W)** When set, enables signaling of PCU firmware detected errors
		63:2		Reserved.
411H	1041	IA32_MC4_STATUS	Core	See Section 15.3.2.2, "IA32_MCi_STATUS MSRS," and Chapter 16.
480H	1152	IA32_VMX_BASIC	Thread	**Reporting Register of Basic VMX Capabilities (R/O)** See Table 2-2. See Appendix A.1, "Basic VMX Information."
481H	1153	IA32_VMX_PINBASED_CTLS	Thread	**Capability Reporting Register of Pin-based VM-execution Controls (R/O)** See Table 2-2. See Appendix A.3, "VM-Execution Controls."
482H	1154	IA32_VMX_PROCBASED_CTLS	Thread	**Capability Reporting Register of Primary Processor-based VM-execution Controls (R/O)** See Appendix A.3, "VM-Execution Controls."
483H	1155	IA32_VMX_EXIT_CTLS	Thread	**Capability Reporting Register of VM-exit Controls (R/O)** See Table 2-2. See Appendix A.4, "VM-Exit Controls."
484H	1156	IA32_VMX_ENTRY_CTLS	Thread	**Capability Reporting Register of VM-entry Controls (R/O)** See Table 2-2. See Appendix A.5, "VM-Entry Controls."
485H	1157	IA32_VMX_MISC	Thread	**Reporting Register of Miscellaneous VMX Capabilities (R/O)** See Table 2-2. See Appendix A.6, "Miscellaneous Data."
486H	1158	IA32_VMX_CR0_FIXED0	Thread	**Capability Reporting Register of CR0 Bits Fixed to 0 (R/O)** See Table 2-2. See Appendix A.7, "VMX-Fixed Bits in CR0."
487H	1159	IA32_VMX_CR0_FIXED1	Thread	**Capability Reporting Register of CR0 Bits Fixed to 1 (R/O)** See Table 2-2. See Appendix A.7, "VMX-Fixed Bits in CR0."
488H	1160	IA32_VMX_CR4_FIXED0	Thread	**Capability Reporting Register of CR4 Bits Fixed to 0 (R/O)** See Table 2-2. See Appendix A.8, "VMX-Fixed Bits in CR4."

Table 2-19. MSRs Supported by Intel® Processors based on Intel® microarchitecture code name Sandy Bridge (Contd.)

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
489H	1161	IA32_VMX_CR4_FIXED1	Thread	**Capability Reporting Register of CR4 Bits Fixed to 1 (R/O)** See Table 2-2. See Appendix A.8, "VMX-Fixed Bits in CR4."
48AH	1162	IA32_VMX_VMCS_ENUM	Thread	**Capability Reporting Register of VMCS Field Enumeration (R/O)** See Table 2-2. See Appendix A.9, "VMCS Enumeration."
48BH	1163	IA32_VMX_PROCBASED_CTLS2	Thread	**Capability Reporting Register of Secondary Processor-based VM-execution Controls (R/O)** See Appendix A.3, "VM-Execution Controls."
48CH	1164	IA32_VMX_EPT_VPID_ENUM	Thread	**Capability Reporting Register of EPT and VPID (R/O)** See Table 2-2
48DH	1165	IA32_VMX_TRUE_PINBASED_CTLS	Thread	**Capability Reporting Register of Pin-based VM-execution Flex Controls (R/O)** See Table 2-2
48EH	1166	IA32_VMX_TRUE_PROCBASED_CTLS	Thread	**Capability Reporting Register of Primary Processor-based VM-execution Flex Controls (R/O)** See Table 2-2
48FH	1167	IA32_VMX_TRUE_EXIT_CTLS	Thread	**Capability Reporting Register of VM-exit Flex Controls (R/O)** See Table 2-2
490H	1168	IA32_VMX_TRUE_ENTRY_CTLS	Thread	**Capability Reporting Register of VM-entry Flex Controls (R/O)** See Table 2-2
4C1H	1217	IA32_A_PMC0	Thread	See Table 2-2.
4C2H	1218	IA32_A_PMC1	Thread	See Table 2-2.
4C3H	1219	IA32_A_PMC2	Thread	See Table 2-2.
4C4H	1220	IA32_A_PMC3	Thread	See Table 2-2.
4C5H	1221	IA32_A_PMC4	Core	See Table 2-2.
4C6H	1222	IA32_A_PMC5	Core	See Table 2-2.
4C7H	1223	IA32_A_PMC6	Core	See Table 2-2.
4C8H	1224	IA32_A_PMC7	Core	See Table 2-2.
600H	1536	IA32_DS_AREA	Thread	**DS Save Area (R/W)** See Table 2-2. See Section 18.6.3.4, "Debug Store (DS) Mechanism."
606H	1542	MSR_RAPL_POWER_UNIT	Package	**Unit Multipliers used in RAPL Interfaces (R/O)** See Section 14.9.1, "RAPL Interfaces."
60AH	1546	MSR_PKGC3_IRTL	Package	**Package C3 Interrupt Response Limit (R/W)** Note: C-state values are processor specific C-state code names, unrelated to MWAIT extension C-state parameters or ACPI C-States.

Table 2-19. MSRs Supported by Intel® Processors based on Intel® microarchitecture code name Sandy Bridge (Contd.)

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
		9:0		**Interrupt response time limit (R/W)** Specifies the limit that should be used to decide if the package should be put into a package C3 state.
		12:10		**Time Unit (R/W)** Specifies the encoding value of time unit of the interrupt response time limit. The following time unit encodings are supported: 000b: 1 ns 001b: 32 ns 010b: 1024 ns 011b: 32768 ns 100b: 1048576 ns 101b: 33554432 ns
		14:13		Reserved.
		15		**Valid (R/W)** Indicates whether the values in bits 12:0 are valid and can be used by the processor for package C-sate management.
		63:16		Reserved.
60BH	1547	MSR_PKGC6_IRTL	Package	**Package C6 Interrupt Response Limit (R/W)** This MSR defines the budget allocated for the package to exit from C6 to a C0 state, where interrupt request can be delivered to the core and serviced. Additional core-exit latency amy be applicable depending on the actual C-state the core is in. Note: C-state values are processor specific C-state code names, unrelated to MWAIT extension C-state parameters or ACPI C-States.
		9:0		**Interrupt response time limit (R/W)** Specifies the limit that should be used to decide if the package should be put into a package C6 state.
		12:10		**Time Unit (R/W)** Specifies the encoding value of time unit of the interrupt response time limit. The following time unit encodings are supported: 000b: 1 ns 001b: 32 ns 010b: 1024 ns 011b: 32768 ns 100b: 1048576 ns 101b: 33554432 ns
		14:13		Reserved.
		15		**Valid (R/W)** Indicates whether the values in bits 12:0 are valid and can be used by the processor for package C-sate management.
		63:16		Reserved.

Table 2-19. MSRs Supported by Intel® Processors based on Intel® microarchitecture code name Sandy Bridge (Contd.)

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
60DH	1549	MSR_PKG_C2_RESIDENCY	Package	Note: C-state values are processor specific C-state code names, unrelated to MWAIT extension C-state parameters or ACPI C-States.
		63:0		**Package C2 Residency Counter. (R/O)** Value since last reset that this package is in processor-specific C2 states. Count at the same frequency as the TSC.
610H	1552	MSR_PKG_POWER_LIMIT	Package	**PKG RAPL Power Limit Control (R/W)** See Section 14.9.3, "Package RAPL Domain."
611H	1553	MSR_PKG_ENERGY_STATUS	Package	**PKG Energy Status (R/O)** See Section 14.9.3, "Package RAPL Domain."
614H	1556	MSR_PKG_POWER_INFO	Package	**PKG RAPL Parameters (R/W)** See Section 14.9.3, "Package RAPL Domain."
638H	1592	MSR_PP0_POWER_LIMIT	Package	**PP0 RAPL Power Limit Control (R/W)** See Section 14.9.4, "PP0/PP1 RAPL Domains."
680H	1664	MSR_ LASTBRANCH_0_FROM_IP	Thread	**Last Branch Record 0 From IP (R/W)** One of sixteen pairs of last branch record registers on the last branch record stack. This part of the stack contains pointers to the **source instruction**. See also: ▪ Last Branch Record Stack TOS at 1C9H ▪ Section 17.9.1 and record format in Section 17.4.8.1
681H	1665	MSR_ LASTBRANCH_1_FROM_IP	Thread	**Last Branch Record 1 From IP (R/W)** See description of MSR_LASTBRANCH_0_FROM_IP.
682H	1666	MSR_ LASTBRANCH_2_FROM_IP	Thread	**Last Branch Record 2 From IP (R/W)** See description of MSR_LASTBRANCH_0_FROM_IP.
683H	1667	MSR_ LASTBRANCH_3_FROM_IP	Thread	**Last Branch Record 3 From IP (R/W)** See description of MSR_LASTBRANCH_0_FROM_IP.
684H	1668	MSR_ LASTBRANCH_4_FROM_IP	Thread	**Last Branch Record 4 From IP (R/W)** See description of MSR_LASTBRANCH_0_FROM_IP.
685H	1669	MSR_ LASTBRANCH_5_FROM_IP	Thread	**Last Branch Record 5 From IP (R/W)** See description of MSR_LASTBRANCH_0_FROM_IP.
686H	1670	MSR_ LASTBRANCH_6_FROM_IP	Thread	**Last Branch Record 6 From IP (R/W)** See description of MSR_LASTBRANCH_0_FROM_IP.
687H	1671	MSR_ LASTBRANCH_7_FROM_IP	Thread	**Last Branch Record 7 From IP (R/W)** See description of MSR_LASTBRANCH_0_FROM_IP.
688H	1672	MSR_ LASTBRANCH_8_FROM_IP	Thread	**Last Branch Record 8 From IP (R/W)** See description of MSR_LASTBRANCH_0_FROM_IP.
689H	1673	MSR_ LASTBRANCH_9_FROM_IP	Thread	**Last Branch Record 9 From IP (R/W)** See description of MSR_LASTBRANCH_0_FROM_IP.
68AH	1674	MSR_ LASTBRANCH_10_FROM_IP	Thread	**Last Branch Record 10 From IP (R/W)** See description of MSR_LASTBRANCH_0_FROM_IP.

Table 2-19. MSRs Supported by Intel® Processors based on Intel® microarchitecture code name Sandy Bridge (Contd.)

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
68BH	1675	MSR_ LASTBRANCH_11_FROM_IP	Thread	**Last Branch Record 11 From IP (R/W)** See description of MSR_LASTBRANCH_0_FROM_IP.
68CH	1676	MSR_ LASTBRANCH_12_FROM_IP	Thread	**Last Branch Record 12 From IP (R/W)** See description of MSR_LASTBRANCH_0_FROM_IP.
68DH	1677	MSR_ LASTBRANCH_13_FROM_IP	Thread	**Last Branch Record 13 From IP (R/W)** See description of MSR_LASTBRANCH_0_FROM_IP.
68EH	1678	MSR_ LASTBRANCH_14_FROM_IP	Thread	**Last Branch Record 14 From IP (R/W)** See description of MSR_LASTBRANCH_0_FROM_IP.
68FH	1679	MSR_ LASTBRANCH_15_FROM_IP	Thread	**Last Branch Record 15 From IP (R/W)** See description of MSR_LASTBRANCH_0_FROM_IP.
6C0H	1728	MSR_ LASTBRANCH_0_TO_IP	Thread	**Last Branch Record 0 To IP (R/W)** One of sixteen pairs of last branch record registers on the last branch record stack. This part of the stack contains pointers to the destination instruction.
6C1H	1729	MSR_ LASTBRANCH_1_TO_IP	Thread	**Last Branch Record 1 To IP (R/W)** See description of MSR_LASTBRANCH_0_TO_IP.
6C2H	1730	MSR_ LASTBRANCH_2_TO_IP	Thread	**Last Branch Record 2 To IP (R/W)** See description of MSR_LASTBRANCH_0_TO_IP.
6C3H	1731	MSR_ LASTBRANCH_3_TO_IP	Thread	**Last Branch Record 3 To IP (R/W)** See description of MSR_LASTBRANCH_0_TO_IP.
6C4H	1732	MSR_ LASTBRANCH_4_TO_IP	Thread	**Last Branch Record 4 To IP (R/W)** See description of MSR_LASTBRANCH_0_TO_IP.
6C5H	1733	MSR_ LASTBRANCH_5_TO_IP	Thread	**Last Branch Record 5 To IP (R/W)** See description of MSR_LASTBRANCH_0_TO_IP.
6C6H	1734	MSR_ LASTBRANCH_6_TO_IP	Thread	**Last Branch Record 6 To IP (R/W)** See description of MSR_LASTBRANCH_0_TO_IP.
6C7H	1735	MSR_ LASTBRANCH_7_TO_IP	Thread	**Last Branch Record 7 To IP (R/W)** See description of MSR_LASTBRANCH_0_TO_IP.
6C8H	1736	MSR_ LASTBRANCH_8_TO_IP	Thread	**Last Branch Record 8 To IP (R/W)** See description of MSR_LASTBRANCH_0_TO_IP.
6C9H	1737	MSR_ LASTBRANCH_9_TO_IP	Thread	**Last Branch Record 9 To IP (R/W)** See description of MSR_LASTBRANCH_0_TO_IP.
6CAH	1738	MSR_ LASTBRANCH_10_TO_IP	Thread	**Last Branch Record 10 To IP (R/W)** See description of MSR_LASTBRANCH_0_TO_IP.
6CBH	1739	MSR_ LASTBRANCH_11_TO_IP	Thread	**Last Branch Record 11 To IP (R/W)** See description of MSR_LASTBRANCH_0_TO_IP.
6CCH	1740	MSR_ LASTBRANCH_12_TO_IP	Thread	**Last Branch Record 12 To IP (R/W)** See description of MSR_LASTBRANCH_0_TO_IP.

Table 2-19. MSRs Supported by Intel® Processors based on Intel® microarchitecture code name Sandy Bridge (Contd.)

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
6CDH	1741	MSR_ LASTBRANCH_13_TO_IP	Thread	**Last Branch Record 13 To IP (R/W)** See description of MSR_LASTBRANCH_0_TO_IP.
6CEH	1742	MSR_ LASTBRANCH_14_TO_IP	Thread	**Last Branch Record 14 To IP (R/W)** See description of MSR_LASTBRANCH_0_TO_IP.
6CFH	1743	MSR_ LASTBRANCH_15_TO_IP	Thread	**Last Branch Record 15 To IP (R/W)** See description of MSR_LASTBRANCH_0_TO_IP.
6E0H	1760	IA32_TSC_DEADLINE	Thread	See Table 2-2.
802H-83FH		X2APIC MSRs	Thread	See Table 2-2.
C000_0080H		IA32_EFER	Thread	**Extended Feature Enables** See Table 2-2.
C000_0081H		IA32_STAR	Thread	**System Call Target Address (R/W)** See Table 2-2.
C000_0082H		IA32_LSTAR	Thread	**IA-32e Mode System Call Target Address (R/W)** See Table 2-2.
C000_0084H		IA32_FMASK	Thread	**System Call Flag Mask (R/W)** See Table 2-2.
C000_0100H		IA32_FS_BASE	Thread	**Map of BASE Address of FS (R/W)** See Table 2-2.
C000_0101H		IA32_GS_BASE	Thread	**Map of BASE Address of GS (R/W)** See Table 2-2.
C000_0102H		IA32_KERNEL_GS_BASE	Thread	**Swap Target of BASE Address of GS (R/W)** See Table 2-2.
C000_0103H		IA32_TSC_AUX	Thread	**AUXILIARY TSC Signature (R/W)** See Table 2-2 and Section 17.17.2, "IA32_TSC_AUX Register and RDTSCP Support."

2.10.1 MSRs In 2nd Generation Intel® Core™ Processor Family (Based on Intel® Microarchitecture Code Name Sandy Bridge)

Table 2-20 and Table 2-21 list model-specific registers (MSRs) that are specific to the 2nd generation Intel® Core™ processor family (based on Intel microarchitecture code name Sandy Bridge). These processors have a CPUID signature with DisplayFamily_DisplayModel of 06_2AH; see Table 2-1.

Table 2-20. MSRs Supported by 2nd Generation Intel® Core™ Processors (Intel® microarchitecture code name Sandy Bridge)

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
1ADH	429	MSR_TURBO_RATIO_LIMIT	Package	**Maximum Ratio Limit of Turbo Mode** RO if MSR_PLATFORM_INFO.[28] = 0, RW if MSR_PLATFORM_INFO.[28] = 1
		7:0	Package	**Maximum Ratio Limit for 1C** Maximum turbo ratio limit of 1 core active.
		15:8	Package	**Maximum Ratio Limit for 2C** Maximum turbo ratio limit of 2 core active.
		23:16	Package	**Maximum Ratio Limit for 3C** Maximum turbo ratio limit of 3 core active.
		31:24	Package	**Maximum Ratio Limit for 4C** Maximum turbo ratio limit of 4 core active.
		63:32		Reserved.
60CH	1548	MSR_PKGC7_IRTL	Package	**Package C7 Interrupt Response Limit (R/W)** This MSR defines the budget allocated for the package to exit from C7 to a C0 state, where interrupt request can be delivered to the core and serviced. Additional core-exit latency amy be applicable depending on the actual C-state the core is in. Note: C-state values are processor specific C-state code names, unrelated to MWAIT extension C-state parameters or ACPI C-States.
		9:0		**Interrupt response time limit (R/W)** Specifies the limit that should be used to decide if the package should be put into a package C7 state.
		12:10		**Time Unit (R/W)** Specifies the encoding value of time unit of the interrupt response time limit. The following time unit encodings are supported: 000b: 1 ns 001b: 32 ns 010b: 1024 ns 011b: 32768 ns 100b: 1048576 ns 101b: 33554432 ns
		14:13		Reserved.
		15		**Valid (R/W)** Indicates whether the values in bits 12:0 are valid and can be used by the processor for package C-sate management.
		63:16		Reserved.
639H	1593	MSR_PP0_ENERGY_STATUS	Package	**PP0 Energy Status (R/O)** See Section 14.9.4, "PP0/PP1 RAPL Domains."

Table 2-20. MSRs Supported by 2nd Generation Intel® Core™ Processors (Intel® microarchitecture code name Sandy Bridge) (Contd.)

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
63AH	1594	MSR_PP0_POLICY	Package	**PP0 Balance Policy (R/W)** See Section 14.9.4, "PP0/PP1 RAPL Domains."
640H	1600	MSR_PP1_POWER_LIMIT	Package	**PP1 RAPL Power Limit Control (R/W)** See Section 14.9.4, "PP0/PP1 RAPL Domains."
641H	1601	MSR_PP1_ENERGY_STATUS	Package	**PP1 Energy Status (R/O)** See Section 14.9.4, "PP0/PP1 RAPL Domains."
642H	1602	MSR_PP1_POLICY	Package	**PP1 Balance Policy (R/W)** See Section 14.9.4, "PP0/PP1 RAPL Domains."
See Table 2-19, Table 2-20, and Table 2-21 for MSR definitions applicable to processors with CPUID signature 06_2AH.				

Table 2-21 lists the MSRs of uncore PMU for Intel processors with CPUID signature 06_2AH.

Table 2-21. Uncore PMU MSRs Supported by 2nd Generation Intel® Core™ Processors

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
391H	913	MSR_UNC_PERF_GLOBAL_CTRL	Package	Uncore PMU global control
		0		Slice 0 select
		1		Slice 1 select
		2		Slice 2 select
		3		Slice 3 select
		4		Slice 4 select
		18:5		Reserved.
		29		Enable all uncore counters
		30		Enable wake on PMI
		31		Enable Freezing counter when overflow
		63:32		Reserved.
392H	914	MSR_UNC_PERF_GLOBAL_STATUS	Package	Uncore PMU main status
		0		Fixed counter overflowed
		1		An ARB counter overflowed
		2		Reserved
		3		A CBox counter overflowed (on any slice)
		63:4		Reserved.
394H	916	MSR_UNC_PERF_FIXED_CTRL	Package	Uncore fixed counter control (R/W)
		19:0		Reserved
		20		Enable overflow propagation

Table 2-21. Uncore PMU MSRs Supported by 2nd Generation Intel® Core™ Processors

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
		21		Reserved
		22		Enable counting
		63:23		Reserved.
395H	917	MSR_UNC_PERF_FIXED_ CTR	Package	Uncore fixed counter
		47:0		Current count
		63:48		Reserved.
396H	918	MSR_UNC_CBO_CONFIG	Package	Uncore C-Box configuration information (R/O)
		3:0		Report the number of C-Box units with performance counters, including processor cores and processor graphics"
		63:4		Reserved.
3B0H	946	MSR_UNC_ARB_PERFCTR0	Package	Uncore Arb unit, performance counter 0
3B1H	947	MSR_UNC_ARB_PERFCTR1	Package	Uncore Arb unit, performance counter 1
3B2H	944	MSR_UNC_ARB_ PERFEVTSEL0	Package	Uncore Arb unit, counter 0 event select MSR
3B3H	945	MSR_UNC_ARB_ PERFEVTSEL1	Package	Uncore Arb unit, counter 1 event select MSR
700H	1792	MSR_UNC_CBO_0_ PERFEVTSEL0	Package	Uncore C-Box 0, counter 0 event select MSR
701H	1793	MSR_UNC_CBO_0_ PERFEVTSEL1	Package	Uncore C-Box 0, counter 1 event select MSR
702H	1794	MSR_UNC_CBO_0_ PERFEVTSEL2	Package	Uncore C-Box 0, counter 2 event select MSR.
703H	1795	MSR_UNC_CBO_0_ PERFEVTSEL3	Package	Uncore C-Box 0, counter 3 event select MSR.
705H	1797	MSR_UNC_CBO_0_UNIT_ STATUS	Package	Uncore C-Box 0, unit status for counter 0-3
706H	1798	MSR_UNC_CBO_0_PERFCTR0	Package	Uncore C-Box 0, performance counter 0
707H	1799	MSR_UNC_CBO_0_PERFCTR1	Package	Uncore C-Box 0, performance counter 1
708H	1800	MSR_UNC_CBO_0_PERFCTR2	Package	Uncore C-Box 0, performance counter 2.
709H	1801	MSR_UNC_CBO_0_PERFCTR3	Package	Uncore C-Box 0, performance counter 3.
710H	1808	MSR_UNC_CBO_1_ PERFEVTSEL0	Package	Uncore C-Box 1, counter 0 event select MSR
711H	1809	MSR_UNC_CBO_1_ PERFEVTSEL1	Package	Uncore C-Box 1, counter 1 event select MSR
712H	1810	MSR_UNC_CBO_1_ PERFEVTSEL2	Package	Uncore C-Box 1, counter 2 event select MSR.
713H	1811	MSR_UNC_CBO_1_ PERFEVTSEL3	Package	Uncore C-Box 1, counter 3 event select MSR.

Table 2-21. Uncore PMU MSRs Supported by 2nd Generation Intel® Core™ Processors

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
715H	1813	MSR_UNC_CBO_1_UNIT_ STATUS	Package	Uncore C-Box 1, unit status for counter 0-3
716H	1814	MSR_UNC_CBO_1_PERFCTR0	Package	Uncore C-Box 1, performance counter 0
717H	1815	MSR_UNC_CBO_1_PERFCTR1	Package	Uncore C-Box 1, performance counter 1
718H	1816	MSR_UNC_CBO_1_PERFCTR2	Package	Uncore C-Box 1, performance counter 2.
719H	1817	MSR_UNC_CBO_1_PERFCTR3	Package	Uncore C-Box 1, performance counter 3.
720H	1824	MSR_UNC_CBO_2_ PERFEVTSEL0	Package	Uncore C-Box 2, counter 0 event select MSR
721H	1825	MSR_UNC_CBO_2_ PERFEVTSEL1	Package	Uncore C-Box 2, counter 1 event select MSR
722H	1826	MSR_UNC_CBO_2_ PERFEVTSEL2	Package	Uncore C-Box 2, counter 2 event select MSR.
723H	1827	MSR_UNC_CBO_2_ PERFEVTSEL3	Package	Uncore C-Box 2, counter 3 event select MSR.
725H	1829	MSR_UNC_CBO_2_UNIT_ STATUS	Package	Uncore C-Box 2, unit status for counter 0-3
726H	1830	MSR_UNC_CBO_2_PERFCTR0	Package	Uncore C-Box 2, performance counter 0
727H	1831	MSR_UNC_CBO_2_PERFCTR1	Package	Uncore C-Box 2, performance counter 1
728H	1832	MSR_UNC_CBO_3_PERFCTR2	Package	Uncore C-Box 3, performance counter 2.
729H	1833	MSR_UNC_CBO_3_PERFCTR3	Package	Uncore C-Box 3, performance counter 3.
730H	1840	MSR_UNC_CBO_3_ PERFEVTSEL0	Package	Uncore C-Box 3, counter 0 event select MSR
731H	1841	MSR_UNC_CBO_3_ PERFEVTSEL1	Package	Uncore C-Box 3, counter 1 event select MSR.
732H	1842	MSR_UNC_CBO_3_ PERFEVTSEL2	Package	Uncore C-Box 3, counter 2 event select MSR.
733H	1843	MSR_UNC_CBO_3_ PERFEVTSEL3	Package	Uncore C-Box 3, counter 3 event select MSR.
735H	1845	MSR_UNC_CBO_3_UNIT_ STATUS	Package	Uncore C-Box 3, unit status for counter 0-3
736H	1846	MSR_UNC_CBO_3_PERFCTR0	Package	Uncore C-Box 3, performance counter 0.
737H	1847	MSR_UNC_CBO_3_PERFCTR1	Package	Uncore C-Box 3, performance counter 1.
738H	1848	MSR_UNC_CBO_3_PERFCTR2	Package	Uncore C-Box 3, performance counter 2.
739H	1849	MSR_UNC_CBO_3_PERFCTR3	Package	Uncore C-Box 3, performance counter 3.
740H	1856	MSR_UNC_CBO_4_ PERFEVTSEL0	Package	Uncore C-Box 4, counter 0 event select MSR
741H	1857	MSR_UNC_CBO_4_ PERFEVTSEL1	Package	Uncore C-Box 4, counter 1 event select MSR.
742H	1858	MSR_UNC_CBO_4_ PERFEVTSEL2	Package	Uncore C-Box 4, counter 2 event select MSR.

Table 2-21. Uncore PMU MSRs Supported by 2nd Generation Intel® Core™ Processors

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
743H	1859	MSR_UNC_CBO_4_ PERFEVTSEL3	Package	Uncore C-Box 4, counter 3 event select MSR.
745H	1861	MSR_UNC_CBO_4_UNIT_ STATUS	Package	Uncore C-Box 4, unit status for counter 0-3
746H	1862	MSR_UNC_CBO_4_PERFCTR0	Package	Uncore C-Box 4, performance counter 0.
747H	1863	MSR_UNC_CBO_4_PERFCTR1	Package	Uncore C-Box 4, performance counter 1.
748H	1864	MSR_UNC_CBO_4_PERFCTR2	Package	Uncore C-Box 4, performance counter 2.
749H	1865	MSR_UNC_CBO_4_PERFCTR3	Package	Uncore C-Box 4, performance counter 3.

2.10.2 MSRs In Intel® Xeon® Processor E5 Family (Based on Intel® Microarchitecture Code Name Sandy Bridge)

Table 2-22 lists additional model-specific registers (MSRs) that are specific to the Intel® Xeon® Processor E5 Family (based on Intel® microarchitecture code name Sandy Bridge). These processors have a CPUID signature with DisplayFamily_DisplayModel of 06_2DH, and also supports MSRs listed in Table 2-19 and Table 2-23.

Table 2-22. Selected MSRs Supported by Intel® Xeon® Processors E5 Family (based on Sandy Bridge microarchitecture)

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
17FH	383	MSR_ERROR_CONTROL	Package	**MC Bank Error Configuration (R/W)**
		0		Reserved
		1		**MemError Log Enable (R/W)** When set, enables IMC status bank to log additional info in bits 36:32.
		63:2		Reserved.
1ADH	429	MSR_TURBO_RATIO_LIMIT	Package	**Maximum Ratio Limit of Turbo Mode** RO if MSR_PLATFORM_INFO.[28] = 0, RW if MSR_PLATFORM_INFO.[28] = 1
		7:0	Package	**Maximum Ratio Limit for 1C** Maximum turbo ratio limit of 1 core active.
		15:8	Package	**Maximum Ratio Limit for 2C** Maximum turbo ratio limit of 2 core active.
		23:16	Package	**Maximum Ratio Limit for 3C** Maximum turbo ratio limit of 3 core active.
		31:24	Package	**Maximum Ratio Limit for 4C** Maximum turbo ratio limit of 4 core active.
		39:32	Package	**Maximum Ratio Limit for 5C** Maximum turbo ratio limit of 5 core active.

Table 2-22. Selected MSRs Supported by Intel® Xeon® Processors E5 Family (based on Sandy Bridge microarchitecture) (Contd.)

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
		47:40	Package	**Maximum Ratio Limit for 6C** Maximum turbo ratio limit of 6 core active.
		55:48	Package	**Maximum Ratio Limit for 7C** Maximum turbo ratio limit of 7 core active.
		63:56	Package	**Maximum Ratio Limit for 8C** Maximum turbo ratio limit of 8 core active.
285H	645	IA32_MC5_CTL2	Package	See Table 2-2.
286H	646	IA32_MC6_CTL2	Package	See Table 2-2.
287H	647	IA32_MC7_CTL2	Package	See Table 2-2.
288H	648	IA32_MC8_CTL2	Package	See Table 2-2.
289H	649	IA32_MC9_CTL2	Package	See Table 2-2.
28AH	650	IA32_MC10_CTL2	Package	See Table 2-2.
28BH	651	IA32_MC11_CTL2	Package	See Table 2-2.
28CH	652	IA32_MC12_CTL2	Package	See Table 2-2.
28DH	653	IA32_MC13_CTL2	Package	See Table 2-2.
28EH	654	IA32_MC14_CTL2	Package	See Table 2-2.
28FH	655	IA32_MC15_CTL2	Package	See Table 2-2.
290H	656	IA32_MC16_CTL2	Package	See Table 2-2.
291H	657	IA32_MC17_CTL2	Package	See Table 2-2.
292H	658	IA32_MC18_CTL2	Package	See Table 2-2.
293H	659	IA32_MC19_CTL2	Package	See Table 2-2.
39CH	924	MSR_PEBS_NUM_ALT	Package	ENABLE_PEBS_NUM_ALT (RW)
		0		**ENABLE_PEBS_NUM_ALT (RW)** Write 1 to enable alternate PEBS counting logic for specific events requiring additional configuration, see Table 19-17
		63:1		Reserved (must be zero).
414H	1044	IA32_MC5_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs."
415H	1045	IA32_MC5_STATUS	Package	See Section 15.3.2.2, "IA32_MCi_STATUS MSRS," and Chapter 16.
416H	1046	IA32_MC5_ADDR	Package	See Section 15.3.2.3, "IA32_MCi_ADDR MSRs."
417H	1047	IA32_MC5_MISC	Package	See Section 15.3.2.4, "IA32_MCi_MISC MSRs."
418H	1048	IA32_MC6_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs."
419H	1049	IA32_MC6_STATUS	Package	See Section 15.3.2.2, "IA32_MCi_STATUS MSRS," and Chapter 16.
41AH	1050	IA32_MC6_ADDR	Package	See Section 15.3.2.3, "IA32_MCi_ADDR MSRs."
41BH	1051	IA32_MC6_MISC	Package	See Section 15.3.2.4, "IA32_MCi_MISC MSRs."
41CH	1052	IA32_MC7_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs."
41DH	1053	IA32_MC7_STATUS	Package	See Section 15.3.2.2, "IA32_MCi_STATUS MSRS," and Chapter 16.
41EH	1054	IA32_MC7_ADDR	Package	See Section 15.3.2.3, "IA32_MCi_ADDR MSRs."

Table 2-22. Selected MSRs Supported by Intel® Xeon® Processors E5 Family (based on Sandy Bridge microarchitecture) (Contd.)

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
41FH	1055	IA32_MC7_MISC	Package	See Section 15.3.2.4, "IA32_MCi_MISC MSRs."
420H	1056	IA32_MC8_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs."
421H	1057	IA32_MC8_STATUS	Package	See Section 15.3.2.2, "IA32_MCi_STATUS MSRS," and Chapter 16.
422H	1058	IA32_MC8_ADDR	Package	See Section 15.3.2.3, "IA32_MCi_ADDR MSRs."
423H	1059	IA32_MC8_MISC	Package	See Section 15.3.2.4, "IA32_MCi_MISC MSRs."
424H	1060	IA32_MC9_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs."
425H	1061	IA32_MC9_STATUS	Package	See Section 15.3.2.2, "IA32_MCi_STATUS MSRS," and Chapter 16.
426H	1062	IA32_MC9_ADDR	Package	See Section 15.3.2.3, "IA32_MCi_ADDR MSRs."
427H	1063	IA32_MC9_MISC	Package	See Section 15.3.2.4, "IA32_MCi_MISC MSRs."
428H	1064	IA32_MC10_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs."
429H	1065	IA32_MC10_STATUS	Package	See Section 15.3.2.2, "IA32_MCi_STATUS MSRS," and Chapter 16.
42AH	1066	IA32_MC10_ADDR	Package	See Section 15.3.2.3, "IA32_MCi_ADDR MSRs."
42BH	1067	IA32_MC10_MISC	Package	See Section 15.3.2.4, "IA32_MCi_MISC MSRs."
42CH	1068	IA32_MC11_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs."
42DH	1069	IA32_MC11_STATUS	Package	See Section 15.3.2.2, "IA32_MCi_STATUS MSRS," and Chapter 16.
42EH	1070	IA32_MC11_ADDR	Package	See Section 15.3.2.3, "IA32_MCi_ADDR MSRs."
42FH	1071	IA32_MC11_MISC	Package	See Section 15.3.2.4, "IA32_MCi_MISC MSRs."
430H	1072	IA32_MC12_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs."
431H	1073	IA32_MC12_STATUS	Package	See Section 15.3.2.2, "IA32_MCi_STATUS MSRS," and Chapter 16.
432H	1074	IA32_MC12_ADDR	Package	See Section 15.3.2.3, "IA32_MCi_ADDR MSRs."
433H	1075	IA32_MC12_MISC	Package	See Section 15.3.2.4, "IA32_MCi_MISC MSRs."
434H	1076	IA32_MC13_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs."
435H	1077	IA32_MC13_STATUS	Package	See Section 15.3.2.2, "IA32_MCi_STATUS MSRS," and Chapter 16.
436H	1078	IA32_MC13_ADDR	Package	See Section 15.3.2.3, "IA32_MCi_ADDR MSRs."
437H	1079	IA32_MC13_MISC	Package	See Section 15.3.2.4, "IA32_MCi_MISC MSRs."
438H	1080	IA32_MC14_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs."
439H	1081	IA32_MC14_STATUS	Package	See Section 15.3.2.2, "IA32_MCi_STATUS MSRS," and Chapter 16.
43AH	1082	IA32_MC14_ADDR	Package	See Section 15.3.2.3, "IA32_MCi_ADDR MSRs."
43BH	1083	IA32_MC14_MISC	Package	See Section 15.3.2.4, "IA32_MCi_MISC MSRs."
43CH	1084	IA32_MC15_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs."
43DH	1085	IA32_MC15_STATUS	Package	See Section 15.3.2.2, "IA32_MCi_STATUS MSRS," and Chapter 16.
43EH	1086	IA32_MC15_ADDR	Package	See Section 15.3.2.3, "IA32_MCi_ADDR MSRs."
43FH	1087	IA32_MC15_MISC	Package	See Section 15.3.2.4, "IA32_MCi_MISC MSRs."
440H	1088	IA32_MC16_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs."
441H	1089	IA32_MC16_STATUS	Package	See Section 15.3.2.2, "IA32_MCi_STATUS MSRS," and Chapter 16.
442H	1090	IA32_MC16_ADDR	Package	See Section 15.3.2.3, "IA32_MCi_ADDR MSRs."

Table 2-22. Selected MSRs Supported by Intel® Xeon® Processors E5 Family (based on Sandy Bridge microarchitecture) (Contd.)

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
443H	1091	IA32_MC16_MISC	Package	See Section 15.3.2.4, "IA32_MCi_MISC MSRs."
444H	1092	IA32_MC17_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs."
445H	1093	IA32_MC17_STATUS	Package	See Section 15.3.2.2, "IA32_MCi_STATUS MSRS," and Chapter 16.
446H	1094	IA32_MC17_ADDR	Package	See Section 15.3.2.3, "IA32_MCi_ADDR MSRs."
447H	1095	IA32_MC17_MISC	Package	See Section 15.3.2.4, "IA32_MCi_MISC MSRs."
448H	1096	IA32_MC18_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs."
449H	1097	IA32_MC18_STATUS	Package	See Section 15.3.2.2, "IA32_MCi_STATUS MSRS," and Chapter 16.
44AH	1098	IA32_MC18_ADDR	Package	See Section 15.3.2.3, "IA32_MCi_ADDR MSRs."
44BH	1099	IA32_MC18_MISC	Package	See Section 15.3.2.4, "IA32_MCi_MISC MSRs."
44CH	1100	IA32_MC19_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs."
44DH	1101	IA32_MC19_STATUS	Package	See Section 15.3.2.2, "IA32_MCi_STATUS MSRS," and Chapter 16.
44EH	1102	IA32_MC19_ADDR	Package	See Section 15.3.2.3, "IA32_MCi_ADDR MSRs."
44FH	1103	IA32_MC19_MISC	Package	See Section 15.3.2.4, "IA32_MCi_MISC MSRs."
613H	1555	MSR_PKG_PERF_STATUS	Package	**Package RAPL Perf Status (R/O)**
618H	1560	MSR_DRAM_POWER_LIMIT	Package	**DRAM RAPL Power Limit Control (R/W)** See Section 14.9.5, "DRAM RAPL Domain."
619H	1561	MSR_DRAM_ENERGY_ STATUS	Package	**DRAM Energy Status (R/O)** See Section 14.9.5, "DRAM RAPL Domain."
61BH	1563	MSR_DRAM_PERF_STATUS	Package	**DRAM Performance Throttling Status (R/O)** See Section 14.9.5, "DRAM RAPL Domain."
61CH	1564	MSR_DRAM_POWER_INFO	Package	**DRAM RAPL Parameters (R/W)** See Section 14.9.5, "DRAM RAPL Domain."
639H	1593	MSR_PP0_ENERGY_STATU S	Package	**PP0 Energy Status (R/O)** See Section 14.9.4, "PP0/PP1 RAPL Domains."
See Table 2-19, Table 2-22, and Table 2-23 for MSR definitions applicable to processors with CPUID signature 06_2DH.				

2.10.3 Additional Uncore PMU MSRs in the Intel® Xeon® Processor E5 Family

Intel Xeon Processor E5 family is based on the Sandy Bridge microarchitecture. The MSR-based uncore PMU interfaces are listed in Table 2-23. For complete detail of the uncore PMU, refer to Intel Xeon Processor E5 Product Family Uncore Performance Monitoring Guide. These processors have a CPUID signature with DisplayFamily_DisplayModel of 06_2DH

Table 2-23. Uncore PMU MSRs in Intel® Xeon® Processor E5 Family

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
C08H		MSR_U_PMON_UCLK_FIXED_CTL	Package	Uncore U-box UCLK fixed counter control
C09H		MSR_U_PMON_UCLK_FIXED_CTR	Package	Uncore U-box UCLK fixed counter

Table 2-23. Uncore PMU MSRs in Intel® Xeon® Processor E5 Family (Contd.)

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
C10H		MSR_U_PMON_EVNTSEL0	Package	Uncore U-box perfmon event select for U-box counter 0.
C11H		MSR_U_PMON_EVNTSEL1	Package	Uncore U-box perfmon event select for U-box counter 1.
C16H		MSR_U_PMON_CTR0	Package	Uncore U-box perfmon counter 0
C17H		MSR_U_PMON_CTR1	Package	Uncore U-box perfmon counter 1
C24H		MSR_PCU_PMON_BOX_CTL	Package	Uncore PCU perfmon for PCU-box-wide control
C30H		MSR_PCU_PMON_EVNTSEL0	Package	Uncore PCU perfmon event select for PCU counter 0.
C31H		MSR_PCU_PMON_EVNTSEL1	Package	Uncore PCU perfmon event select for PCU counter 1.
C32H		MSR_PCU_PMON_EVNTSEL2	Package	Uncore PCU perfmon event select for PCU counter 2.
C33H		MSR_PCU_PMON_EVNTSEL3	Package	Uncore PCU perfmon event select for PCU counter 3.
C34H		MSR_PCU_PMON_BOX_FILTER	Package	Uncore PCU perfmon box-wide filter.
C36H		MSR_PCU_PMON_CTR0	Package	Uncore PCU perfmon counter 0.
C37H		MSR_PCU_PMON_CTR1	Package	Uncore PCU perfmon counter 1.
C38H		MSR_PCU_PMON_CTR2	Package	Uncore PCU perfmon counter 2.
C39H		MSR_PCU_PMON_CTR3	Package	Uncore PCU perfmon counter 3.
D04H		MSR_C0_PMON_BOX_CTL	Package	Uncore C-box 0 perfmon local box wide control.
D10H		MSR_C0_PMON_EVNTSEL0	Package	Uncore C-box 0 perfmon event select for C-box 0 counter 0.
D11H		MSR_C0_PMON_EVNTSEL1	Package	Uncore C-box 0 perfmon event select for C-box 0 counter 1.
D12H		MSR_C0_PMON_EVNTSEL2	Package	Uncore C-box 0 perfmon event select for C-box 0 counter 2.
D13H		MSR_C0_PMON_EVNTSEL3	Package	Uncore C-box 0 perfmon event select for C-box 0 counter 3.
D14H		MSR_C0_PMON_BOX_FILTER	Package	Uncore C-box 0 perfmon box wide filter.
D16H		MSR_C0_PMON_CTR0	Package	Uncore C-box 0 perfmon counter 0.
D17H		MSR_C0_PMON_CTR1	Package	Uncore C-box 0 perfmon counter 1.
D18H		MSR_C0_PMON_CTR2	Package	Uncore C-box 0 perfmon counter 2.
D19H		MSR_C0_PMON_CTR3	Package	Uncore C-box 0 perfmon counter 3.
D24H		MSR_C1_PMON_BOX_CTL	Package	Uncore C-box 1 perfmon local box wide control.
D30H		MSR_C1_PMON_EVNTSEL0	Package	Uncore C-box 1 perfmon event select for C-box 1 counter 0.
D31H		MSR_C1_PMON_EVNTSEL1	Package	Uncore C-box 1 perfmon event select for C-box 1 counter 1.
D32H		MSR_C1_PMON_EVNTSEL2	Package	Uncore C-box 1 perfmon event select for C-box 1 counter 2.
D33H		MSR_C1_PMON_EVNTSEL3	Package	Uncore C-box 1 perfmon event select for C-box 1 counter 3.
D34H		MSR_C1_PMON_BOX_FILTER	Package	Uncore C-box 1 perfmon box wide filter.
D36H		MSR_C1_PMON_CTR0	Package	Uncore C-box 1 perfmon counter 0.
D37H		MSR_C1_PMON_CTR1	Package	Uncore C-box 1 perfmon counter 1.
D38H		MSR_C1_PMON_CTR2	Package	Uncore C-box 1 perfmon counter 2.
D39H		MSR_C1_PMON_CTR3	Package	Uncore C-box 1 perfmon counter 3.
D44H		MSR_C2_PMON_BOX_CTL	Package	Uncore C-box 2 perfmon local box wide control.
D50H		MSR_C2_PMON_EVNTSEL0	Package	Uncore C-box 2 perfmon event select for C-box 2 counter 0.
D51H		MSR_C2_PMON_EVNTSEL1	Package	Uncore C-box 2 perfmon event select for C-box 2 counter 1.

Table 2-23. Uncore PMU MSRs in Intel® Xeon® Processor E5 Family (Contd.)

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
D52H		MSR_C2_PMON_EVNTSEL2	Package	Uncore C-box 2 perfmon event select for C-box 2 counter 2.
D53H		MSR_C2_PMON_EVNTSEL3	Package	Uncore C-box 2 perfmon event select for C-box 2 counter 3.
D54H		MSR_C2_PMON_BOX_FILTER	Package	Uncore C-box 2 perfmon box wide filter.
D56H		MSR_C2_PMON_CTR0	Package	Uncore C-box 2 perfmon counter 0.
D57H		MSR_C2_PMON_CTR1	Package	Uncore C-box 2 perfmon counter 1.
D58H		MSR_C2_PMON_CTR2	Package	Uncore C-box 2 perfmon counter 2.
D59H		MSR_C2_PMON_CTR3	Package	Uncore C-box 2 perfmon counter 3.
D64H		MSR_C3_PMON_BOX_CTL	Package	Uncore C-box 3 perfmon local box wide control.
D70H		MSR_C3_PMON_EVNTSEL0	Package	Uncore C-box 3 perfmon event select for C-box 3 counter 0.
D71H		MSR_C3_PMON_EVNTSEL1	Package	Uncore C-box 3 perfmon event select for C-box 3 counter 1.
D72H		MSR_C3_PMON_EVNTSEL2	Package	Uncore C-box 3 perfmon event select for C-box 3 counter 2.
D73H		MSR_C3_PMON_EVNTSEL3	Package	Uncore C-box 3 perfmon event select for C-box 3 counter 3.
D74H		MSR_C3_PMON_BOX_FILTER	Package	Uncore C-box 3 perfmon box wide filter.
D76H		MSR_C3_PMON_CTR0	Package	Uncore C-box 3 perfmon counter 0.
D77H		MSR_C3_PMON_CTR1	Package	Uncore C-box 3 perfmon counter 1.
D78H		MSR_C3_PMON_CTR2	Package	Uncore C-box 3 perfmon counter 2.
D79H		MSR_C3_PMON_CTR3	Package	Uncore C-box 3 perfmon counter 3.
D84H		MSR_C4_PMON_BOX_CTL	Package	Uncore C-box 4 perfmon local box wide control.
D90H		MSR_C4_PMON_EVNTSEL0	Package	Uncore C-box 4 perfmon event select for C-box 4 counter 0.
D91H		MSR_C4_PMON_EVNTSEL1	Package	Uncore C-box 4 perfmon event select for C-box 4 counter 1.
D92H		MSR_C4_PMON_EVNTSEL2	Package	Uncore C-box 4 perfmon event select for C-box 4 counter 2.
D93H		MSR_C4_PMON_EVNTSEL3	Package	Uncore C-box 4 perfmon event select for C-box 4 counter 3.
D94H		MSR_C4_PMON_BOX_FILTER	Package	Uncore C-box 4 perfmon box wide filter.
D96H		MSR_C4_PMON_CTR0	Package	Uncore C-box 4 perfmon counter 0.
D97H		MSR_C4_PMON_CTR1	Package	Uncore C-box 4 perfmon counter 1.
D98H		MSR_C4_PMON_CTR2	Package	Uncore C-box 4 perfmon counter 2.
D99H		MSR_C4_PMON_CTR3	Package	Uncore C-box 4 perfmon counter 3.
DA4H		MSR_C5_PMON_BOX_CTL	Package	Uncore C-box 5 perfmon local box wide control.
DB0H		MSR_C5_PMON_EVNTSEL0	Package	Uncore C-box 5 perfmon event select for C-box 5 counter 0.
DB1H		MSR_C5_PMON_EVNTSEL1	Package	Uncore C-box 5 perfmon event select for C-box 5 counter 1.
DB2H		MSR_C5_PMON_EVNTSEL2	Package	Uncore C-box 5 perfmon event select for C-box 5 counter 2.
DB3H		MSR_C5_PMON_EVNTSEL3	Package	Uncore C-box 5 perfmon event select for C-box 5 counter 3.
DB4H		MSR_C5_PMON_BOX_FILTER	Package	Uncore C-box 5 perfmon box wide filter.
DB6H		MSR_C5_PMON_CTR0	Package	Uncore C-box 5 perfmon counter 0.
DB7H		MSR_C5_PMON_CTR1	Package	Uncore C-box 5 perfmon counter 1.
DB8H		MSR_C5_PMON_CTR2	Package	Uncore C-box 5 perfmon counter 2.
DB9H		MSR_C5_PMON_CTR3	Package	Uncore C-box 5 perfmon counter 3.

Table 2-23. Uncore PMU MSRs in Intel® Xeon® Processor E5 Family (Contd.)

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
DC4H		MSR_C6_PMON_BOX_CTL	Package	Uncore C-box 6 perfmon local box wide control.
DD0H		MSR_C6_PMON_EVNTSEL0	Package	Uncore C-box 6 perfmon event select for C-box 6 counter 0.
DD1H		MSR_C6_PMON_EVNTSEL1	Package	Uncore C-box 6 perfmon event select for C-box 6 counter 1.
DD2H		MSR_C6_PMON_EVNTSEL2	Package	Uncore C-box 6 perfmon event select for C-box 6 counter 2.
DD3H		MSR_C6_PMON_EVNTSEL3	Package	Uncore C-box 6 perfmon event select for C-box 6 counter 3.
DD4H		MSR_C6_PMON_BOX_FILTER	Package	Uncore C-box 6 perfmon box wide filter.
DD6H		MSR_C6_PMON_CTR0	Package	Uncore C-box 6 perfmon counter 0.
DD7H		MSR_C6_PMON_CTR1	Package	Uncore C-box 6 perfmon counter 1.
DD8H		MSR_C6_PMON_CTR2	Package	Uncore C-box 6 perfmon counter 2.
DD9H		MSR_C6_PMON_CTR3	Package	Uncore C-box 6 perfmon counter 3.
DE4H		MSR_C7_PMON_BOX_CTL	Package	Uncore C-box 7 perfmon local box wide control.
DF0H		MSR_C7_PMON_EVNTSEL0	Package	Uncore C-box 7 perfmon event select for C-box 7 counter 0.
DF1H		MSR_C7_PMON_EVNTSEL1	Package	Uncore C-box 7 perfmon event select for C-box 7 counter 1.
DF2H		MSR_C7_PMON_EVNTSEL2	Package	Uncore C-box 7 perfmon event select for C-box 7 counter 2.
DF3H		MSR_C7_PMON_EVNTSEL3	Package	Uncore C-box 7 perfmon event select for C-box 7 counter 3.
DF4H		MSR_C7_PMON_BOX_FILTER	Package	Uncore C-box 7 perfmon box wide filter.
DF6H		MSR_C7_PMON_CTR0	Package	Uncore C-box 7 perfmon counter 0.
DF7H		MSR_C7_PMON_CTR1	Package	Uncore C-box 7 perfmon counter 1.
DF8H		MSR_C7_PMON_CTR2	Package	Uncore C-box 7 perfmon counter 2.
DF9H		MSR_C7_PMON_CTR3	Package	Uncore C-box 7 perfmon counter 3.

2.11 MSRS IN THE 3RD GENERATION INTEL® CORE™ PROCESSOR FAMILY (BASED ON INTEL® MICROARCHITECTURE CODE NAME IVY BRIDGE)

The 3rd generation Intel® Core™ processor family and the Intel® Xeon® processor E3-1200v2 product family (based on Intel microarchitecture code name Ivy Bridge) support the MSR interfaces listed in Table 2-19, Table 2-20, Table 2-21, and Table 2-24. These processors have a CPUID signature with DisplayFamily_DisplayModel of 06_3AH.

Table 2-24. Additional MSRs Supported by 3rd Generation Intel® Core™ Processors (based on Intel® microarchitecture code name Ivy Bridge)

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
CEH	206	MSR_PLATFORM_INFO	Package	Platform Information; contains power management and other model specific features enumeration. See http://biosbits.org.
		7:0		Reserved.

Table 2-24. Additional MSRs Supported by 3rd Generation Intel® Core™ Processors (based on Intel® microarchitecture code name Ivy Bridge) (Contd.)

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
		15:8	Package	**Maximum Non-Turbo Ratio (R/O)** The is the ratio of the frequency that invariant TSC runs at. Frequency = ratio * 100 MHz.
		27:16		Reserved.
		28	Package	**Programmable Ratio Limit for Turbo Mode (R/O)** When set to 1, indicates that Programmable Ratio Limits for Turbo mode is enabled, and when set to 0, indicates Programmable Ratio Limits for Turbo mode is disabled.
		29	Package	**Programmable TDP Limit for Turbo Mode (R/O)** When set to 1, indicates that TDP Limits for Turbo mode are programmable, and when set to 0, indicates TDP Limit for Turbo mode is not programmable.
		31:30		Reserved.
		32	Package	**Low Power Mode Support (LPM) (R/O)** When set to 1, indicates that LPM is supported, and when set to 0, indicates LPM is not supported.
		34:33	Package	**Number of ConfigTDP Levels (R/O)** 00: Only Base TDP level available. 01: One additional TDP level available. 02: Two additional TDP level available. 11: Reserved
		39:35		Reserved.
		47:40	Package	**Maximum Efficiency Ratio (R/O)** The is the minimum ratio (maximum efficiency) that the processor can operates, in units of 100MHz.
		55:48	Package	**Minimum Operating Ratio (R/O)** Contains the minimum supported operating ratio in units of 100 MHz.
		63:56		Reserved.
E2H	226	MSR_PKG_CST_CONFIG_ CONTROL	Core	**C-State Configuration Control (R/W)** Note: C-state values are processor specific C-state code names, unrelated to MWAIT extension C-state parameters or ACPI C-States. See http://biosbits.org.

Table 2-24. Additional MSRs Supported by 3rd Generation Intel® Core™ Processors (based on Intel® microarchitecture code name Ivy Bridge) (Contd.)

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
		2:0		**Package C-State Limit (R/W)** Specifies the lowest processor-specific C-state code name (consuming the least power). for the package. The default is set as factory-configured package C-state limit. The following C-state code name encodings are supported: 000b: C0/C1 (no package C-sate support) 001b: C2 010b: C6 no retention 011b: C6 retention 100b: C7 101b: C7s 111: No package C-state limit. Note: This field cannot be used to limit package C-state to C3.
		9:3		Reserved.
		10		**I/O MWAIT Redirection Enable (R/W)** When set, will map IO_read instructions sent to IO register specified by MSR_PMG_IO_CAPTURE_BASE to MWAIT instructions
		14:11		Reserved.
		15		**CFG Lock (R/WO)** When set, lock bits 15:0 of this register until next reset.
		24:16		Reserved.
		25		**C3 state auto demotion enable (R/W)** When set, the processor will conditionally demote C6/C7 requests to C3 based on uncore auto-demote information.
		26		**C1 state auto demotion enable (R/W)** When set, the processor will conditionally demote C3/C6/C7 requests to C1 based on uncore auto-demote information.
		27		**Enable C3 undemotion (R/W)** When set, enables undemotion from demoted C3.
		28		**Enable C1 undemotion (R/W)** When set, enables undemotion from demoted C1.
		63:29		Reserved.
639H	1593	MSR_PP0_ENERGY_STATUS	Package	**PP0 Energy Status (R/O)** See Section 14.9.4, "PP0/PP1 RAPL Domains."
648H	1608	MSR_CONFIG_TDP_ NOMINAL	Package	**Base TDP Ratio (R/O)**
		7:0		**Config_TDP_Base** Base TDP level ratio to be used for this specific processor (in units of 100 MHz).
		63:8		Reserved.

Table 2-24. Additional MSRs Supported by 3rd Generation Intel® Core™ Processors (based on Intel® microarchitecture code name Ivy Bridge) (Contd.)

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
649H	1609	MSR_CONFIG_TDP_LEVEL1	Package	ConfigTDP Level 1 ratio and power level (R/O)
		14:0		PKG_TDP_LVL1. Power setting for ConfigTDP Level 1.
		15		Reserved
		23:16		Config_TDP_LVL1_Ratio. ConfigTDP level 1 ratio to be used for this specific processor.
		31:24		Reserved
		46:32		PKG_MAX_PWR_LVL1. Max Power setting allowed for ConfigTDP Level 1.
		47		Reserved
		62:48		PKG_MIN_PWR_LVL1. MIN Power setting allowed for ConfigTDP Level 1.
		63		Reserved.
64AH	1610	MSR_CONFIG_TDP_LEVEL2	Package	ConfigTDP Level 2 ratio and power level (R/O)
		14:0		PKG_TDP_LVL2. Power setting for ConfigTDP Level 2.
		15		Reserved
		23:16		Config_TDP_LVL2_Ratio. ConfigTDP level 2 ratio to be used for this specific processor.
		31:24		Reserved
		46:32		PKG_MAX_PWR_LVL2. Max Power setting allowed for ConfigTDP Level 2.
		47		Reserved
		62:48		PKG_MIN_PWR_LVL2. MIN Power setting allowed for ConfigTDP Level 2.
		63		Reserved.
64BH	1611	MSR_CONFIG_TDP_ CONTROL	Package	**ConfigTDP Control (R/W)**
		1:0		**TDP_LEVEL (RW/L)** System BIOS can program this field.
		30:2		Reserved.
		31		**Config_TDP_Lock (RW/L)** When this bit is set, the content of this register is locked until a reset.
		63:32		Reserved.
64CH	1612	MSR_TURBO_ACTIVATION_ RATIO	Package	**ConfigTDP Control (R/W)**
		7:0		**MAX_NON_TURBO_RATIO (RW/L)** System BIOS can program this field.
		30:8		Reserved.

Table 2-24. Additional MSRs Supported by 3rd Generation Intel® Core™ Processors (based on Intel® microarchitecture code name Ivy Bridge) (Contd.)

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
		31		**TURBO_ACTIVATION_RATIO_Lock (RW/L)** When this bit is set, the content of this register is locked until a reset.
		63:32		Reserved.
See Table 2-19, Table 2-20 and Table 2-21 for other MSR definitions applicable to processors with CPUID signature 06_3AH				

2.11.1 MSRs In Intel® Xeon® Processor E5 v2 Product Family (Based on Ivy Bridge-E Microarchitecture)

Table 2-25 lists model-specific registers (MSRs) that are specific to the Intel® Xeon® Processor E5 v2 Product Family (based on Ivy Bridge-E microarchitecture). These processors have a CPUID signature with DisplayFamily_DisplayModel of 06_3EH, see Table 2-1. These processors supports the MSR interfaces listed in Table 2-19, and Table 2-25.

Table 2-25. MSRs Supported by Intel® Xeon® Processors E5 v2 Product Family (based on Ivy Bridge-E microarchitecture)

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
4EH	78	MSR_PPIN_CTL	Package	Protected Processor Inventory Number Enable Control (R/W)
		0		LockOut (R/WO) Set 1to prevent further writes to MSR_PPIN_CTL. Writing 1 to MSR_PPINCTL[bit 0] is permitted only if MSR_PPIN_CTL[bit 1] is clear, Default is 0. BIOS should provide an opt-in menu to enable the user to turn on MSR_PPIN_CTL[bit 1] for privileged inventory initialization agent to access MSR_PPIN. After reading MSR_PPIN, the privileged inventory initialization agent should write '01b' to MSR_PPIN_CTL to disable further access to MSR_PPIN and prevent unauthorized modification to MSR_PPIN_CTL.
		1		Enable_PPIN (R/W) If 1, enables MSR_PPIN to be accessible using RDMSR. Once set, attempt to write 1 to MSR_PPIN_CTL[bit 0] will cause #GP. If 0, an attempt to read MSR_PPIN will cause #GP. Default is 0.
		63:2		Reserved.
4FH	79	MSR_PPIN	Package	Protected Processor Inventory Number (R/O)

Table 2-25. MSRs Supported by Intel® Xeon® Processors E5 v2 Product Family (based on Ivy Bridge-E microarchitecture) (Contd.)

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
		63:0		**Protected Processor Inventory Number (R/O)** A unique value within a given CPUID family/model/stepping signature that a privileged inventory initialization agent can access to identify each physical processor, when access to MSR_PPIN is enabled. Access to MSR_PPIN is permitted only if MSR_PPIN_CTL[bits 1:0] = '10b'
CEH	206	MSR_PLATFORM_INFO	Package	Platform Information; contains power management and other model specific features enumeration. See http://biosbits.org.
		7:0		Reserved.
		15:8	Package	**Maximum Non-Turbo Ratio (R/O)** The is the ratio of the frequency that invariant TSC runs at. Frequency = ratio * 100 MHz.
		22:16		Reserved.
		23	Package	**PPIN_CAP (R/O)** When set to 1, indicates that Protected Processor Inventory Number (PPIN) capability can be enabled for privileged system inventory agent to read PPIN from MSR_PPIN. When set to 0, PPIN capability is not supported. An attempt to access MSR_PPIN_CTL or MSR_PPIN will cause #GP.
		27:24		Reserved.
		28	Package	**Programmable Ratio Limit for Turbo Mode (R/O)** When set to 1, indicates that Programmable Ratio Limits for Turbo mode is enabled, and when set to 0, indicates Programmable Ratio Limits for Turbo mode is disabled.
		29	Package	**Programmable TDP Limit for Turbo Mode (R/O)** When set to 1, indicates that TDP Limits for Turbo mode are programmable, and when set to 0, indicates TDP Limit for Turbo mode is not programmable.
		30	Package	**Programmable TJ OFFSET (R/O)** When set to 1, indicates that MSR_TEMPERATURE_TARGET.[27:24] is valid and writable to specify an temperature offset.
		39:31		Reserved.
		47:40	Package	**Maximum Efficiency Ratio (R/O)** The is the minimum ratio (maximum efficiency) that the processor can operates, in units of 100MHz.
		63:48		Reserved.
E2H	226	MSR_PKG_CST_CONFIG_ CONTROL	Core	**C-State Configuration Control (R/W)** Note: C-state values are processor specific C-state code names, unrelated to MWAIT extension C-state parameters or ACPI C-States. See http://biosbits.org.

Table 2-25. MSRs Supported by Intel® Xeon® Processors E5 v2 Product Family (based on Ivy Bridge-E microarchitecture) (Contd.)

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
		2:0		**Package C-State Limit (R/W)** Specifies the lowest processor-specific C-state code name (consuming the least power). for the package. The default is set as factory-configured package C-state limit. The following C-state code name encodings are supported: 000b: C0/C1 (no package C-sate support) 001b: C2 010b: C6 no retention 011b: C6 retention 100b: C7 101b: C7s 111: No package C-state limit. Note: This field cannot be used to limit package C-state to C3.
		9:3		Reserved.
		10		**I/O MWAIT Redirection Enable (R/W)** When set, will map IO_read instructions sent to IO register specified by MSR_PMG_IO_CAPTURE_BASE to MWAIT instructions
		14:11		Reserved.
		15		**CFG Lock (R/WO)** When set, lock bits 15:0 of this register until next reset.
		63:16		Reserved.
179H	377	IA32_MCG_CAP	Thread	**Global Machine Check Capability (R/O)**
		7:0		**Count**
		8		**MCG_CTL_P**
		9		**MCG_EXT_P**
		10		**MCP_CMCI_P**
		11		**MCG_TES_P**
		15:12		Reserved.
		23:16		**MCG_EXT_CNT**
		24		**MCG_SER_P**
		25		Reserved.
		26		**MCG_ELOG_P**
		63:27		Reserved.
17FH	383	MSR_ERROR_CONTROL	Package	**MC Bank Error Configuration (R/W)**
		0		Reserved
		1		**MemError Log Enable (R/W)** When set, enables IMC status bank to log additional info in bits 36:32.
		63:2		Reserved.

Table 2-25. MSRs Supported by Intel® Xeon® Processors E5 v2 Product Family (based on Ivy Bridge-E microarchitecture) (Contd.)

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
1A2H	418	MSR_ TEMPERATURE_TARGET	Package	Temperature Target
		15:0		Reserved.
		23:16		**Temperature Target (RO)** The minimum temperature at which PROCHOT# will be asserted. The value is degree C.
		27:24		**TCC Activation Offset (R/W)** Specifies a temperature offset in degrees C from the temperature target (bits 23:16). PROCHOT# will assert at the offset target temperature. Write is permitted only MSR_PLATFORM_INFO.[30] is set.
		63:28		Reserved.
1AEH	430	MSR_TURBO_RATIO_LIMIT 1	Package	**Maximum Ratio Limit of Turbo Mode** RO if MSR_PLATFORM_INFO.[28] = 0, RW if MSR_PLATFORM_INFO.[28] = 1
		7:0	Package	**Maximum Ratio Limit for 9C** Maximum turbo ratio limit of 9 core active.
		15:8	Package	**Maximum Ratio Limit for 10C** Maximum turbo ratio limit of 10core active.
		23:16	Package	**Maximum Ratio Limit for 11C** Maximum turbo ratio limit of 11 core active.
		31:24	Package	**Maximum Ratio Limit for 12C** Maximum turbo ratio limit of 12 core active.
		63:32		Reserved
285H	645	IA32_MC5_CTL2	Package	See Table 2-2.
286H	646	IA32_MC6_CTL2	Package	See Table 2-2.
287H	647	IA32_MC7_CTL2	Package	See Table 2-2.
288H	648	IA32_MC8_CTL2	Package	See Table 2-2.
289H	649	IA32_MC9_CTL2	Package	See Table 2-2.
28AH	650	IA32_MC10_CTL2	Package	See Table 2-2.
28BH	651	IA32_MC11_CTL2	Package	See Table 2-2.
28CH	652	IA32_MC12_CTL2	Package	See Table 2-2.
28DH	653	IA32_MC13_CTL2	Package	See Table 2-2.
28EH	654	IA32_MC14_CTL2	Package	See Table 2-2.
28FH	655	IA32_MC15_CTL2	Package	See Table 2-2.
290H	656	IA32_MC16_CTL2	Package	See Table 2-2.
291H	657	IA32_MC17_CTL2	Package	See Table 2-2.
292H	658	IA32_MC18_CTL2	Package	See Table 2-2.
293H	659	IA32_MC19_CTL2	Package	See Table 2-2.

Table 2-25. MSRs Supported by Intel® Xeon® Processors E5 v2 Product Family (based on Ivy Bridge-E microarchitecture) (Contd.)

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
294H	660	IA32_MC20_CTL2	Package	See Table 2-2.
295H	661	IA32_MC21_CTL2	Package	See Table 2-2.
296H	662	IA32_MC22_CTL2	Package	See Table 2-2.
297H	663	IA32_MC23_CTL2	Package	See Table 2-2.
298H	664	IA32_MC24_CTL2	Package	See Table 2-2.
299H	665	IA32_MC25_CTL2	Package	See Table 2-2.
29AH	666	IA32_MC26_CTL2	Package	See Table 2-2.
29BH	667	IA32_MC27_CTL2	Package	See Table 2-2.
29CH	668	IA32_MC28_CTL2	Package	See Table 2-2.
414H	1044	IA32_MC5_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs." through Section 15.3.2.4, "IA32_MCi_MISC MSRs.". Bank MC5 reports MC error from the Intel QPI module.
415H	1045	IA32_MC5_STATUS	Package	
416H	1046	IA32_MC5_ADDR	Package	
417H	1047	IA32_MC5_MISC	Package	
418H	1048	IA32_MC6_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs." through Section 15.3.2.4, "IA32_MCi_MISC MSRs.". Bank MC6 reports MC error from the integrated I/O module.
419H	1049	IA32_MC6_STATUS	Package	
41AH	1050	IA32_MC6_ADDR	Package	
41BH	1051	IA32_MC6_MISC	Package	
41CH	1052	IA32_MC7_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs." through Section 15.3.2.4, "IA32_MCi_MISC MSRs.". Banks MC7 and MC 8 report MC error from the two home agents.
41DH	1053	IA32_MC7_STATUS	Package	
41EH	1054	IA32_MC7_ADDR	Package	
41FH	1055	IA32_MC7_MISC	Package	
420H	1056	IA32_MC8_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs." through Section 15.3.2.4, "IA32_MCi_MISC MSRs.". Banks MC7 and MC 8 report MC error from the two home agents.
421H	1057	IA32_MC8_STATUS	Package	
422H	1058	IA32_MC8_ADDR	Package	
423H	1059	IA32_MC8_MISC	Package	
424H	1060	IA32_MC9_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs." through Section 15.3.2.4, "IA32_MCi_MISC MSRs.". Banks MC9 through MC 16 report MC error from each channel of the integrated memory controllers.
425H	1061	IA32_MC9_STATUS	Package	
426H	1062	IA32_MC9_ADDR	Package	
427H	1063	IA32_MC9_MISC	Package	
428H	1064	IA32_MC10_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs." through Section 15.3.2.4, "IA32_MCi_MISC MSRs.". Banks MC9 through MC 16 report MC error from each channel of the integrated memory controllers.
429H	1065	IA32_MC10_STATUS	Package	
42AH	1066	IA32_MC10_ADDR	Package	
42BH	1067	IA32_MC10_MISC	Package	
42CH	1068	IA32_MC11_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs."

Table 2-25. MSRs Supported by Intel® Xeon® Processors E5 v2 Product Family (based on Ivy Bridge-E microarchitecture) (Contd.)

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
42DH	1069	IA32_MC11_STATUS	Package	Bank MC11 reports MC error from a specific channel of the integrated memory controller.
42EH	1070	IA32_MC11_ADDR	Package	
42FH	1071	IA32_MC11_MISC	Package	
430H	1072	IA32_MC12_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs." through Section 15.3.2.4, "IA32_MCi_MISC MSRs.". Banks MC9 through MC 16 report MC error from each channel of the integrated memory controllers.
431H	1073	IA32_MC12_STATUS	Package	
432H	1074	IA32_MC12_ADDR	Package	
433H	1075	IA32_MC12_MISC	Package	
434H	1076	IA32_MC13_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs." through Section 15.3.2.4, "IA32_MCi_MISC MSRs.". Banks MC9 through MC 16 report MC error from each channel of the integrated memory controllers.
435H	1077	IA32_MC13_STATUS	Package	
436H	1078	IA32_MC13_ADDR	Package	
437H	1079	IA32_MC13_MISC	Package	
438H	1080	IA32_MC14_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs." through Section 15.3.2.4, "IA32_MCi_MISC MSRs.". Banks MC9 through MC 16 report MC error from each channel of the integrated memory controllers.
439H	1081	IA32_MC14_STATUS	Package	
43AH	1082	IA32_MC14_ADDR	Package	
43BH	1083	IA32_MC14_MISC	Package	
43CH	1084	IA32_MC15_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs." through Section 15.3.2.4, "IA32_MCi_MISC MSRs.". Banks MC9 through MC 16 report MC error from each channel of the integrated memory controllers.
43DH	1085	IA32_MC15_STATUS	Package	
43EH	1086	IA32_MC15_ADDR	Package	
43FH	1087	IA32_MC15_MISC	Package	
440H	1088	IA32_MC16_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs." through Section 15.3.2.4, "IA32_MCi_MISC MSRs.". Banks MC9 through MC 16 report MC error from each channel of the integrated memory controllers.
441H	1089	IA32_MC16_STATUS	Package	
442H	1090	IA32_MC16_ADDR	Package	
443H	1091	IA32_MC16_MISC	Package	
444H	1092	IA32_MC17_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs." through Section 15.3.2.4, "IA32_MCi_MISC MSRs.". Bank MC17 reports MC error from a specific CBo (core broadcast) and its corresponding slice of L3.
445H	1093	IA32_MC17_STATUS	Package	
446H	1094	IA32_MC17_ADDR	Package	
447H	1095	IA32_MC17_MISC	Package	
448H	1096	IA32_MC18_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs." through Section 15.3.2.4, "IA32_MCi_MISC MSRs.". Bank MC18 reports MC error from a specific CBo (core broadcast) and its corresponding slice of L3.
449H	1097	IA32_MC18_STATUS	Package	
44AH	1098	IA32_MC18_ADDR	Package	
44BH	1099	IA32_MC18_MISC	Package	
44CH	1100	IA32_MC19_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs." through Section 15.3.2.4, "IA32_MCi_MISC MSRs.". Bank MC19 reports MC error from a specific CBo (core broadcast) and its corresponding slice of L3.
44DH	1101	IA32_MC19_STATUS	Package	
44EH	1102	IA32_MC19_ADDR	Package	
44FH	1103	IA32_MC19_MISC	Package	

Table 2-25. MSRs Supported by Intel® Xeon® Processors E5 v2 Product Family (based on Ivy Bridge-E microarchitecture) (Contd.)

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
450H	1104	IA32_MC20_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs."
451H	1105	IA32_MC20_STATUS	Package	Bank MC20 reports MC error from a specific CBo (core broadcast) and its corresponding slice of L3.
452H	1106	IA32_MC20_ADDR	Package	
453H	1107	IA32_MC20_MISC	Package	
454H	1108	IA32_MC21_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs." through Section 15.3.2.4, "IA32_MCi_MISC MSRs.". Bank MC21 reports MC error from a specific CBo (core broadcast) and its corresponding slice of L3.
455H	1109	IA32_MC21_STATUS	Package	
456H	1110	IA32_MC21_ADDR	Package	
457H	1111	IA32_MC21_MISC	Package	
458H	1112	IA32_MC22_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs." through Section 15.3.2.4, "IA32_MCi_MISC MSRs.". Bank MC22 reports MC error from a specific CBo (core broadcast) and its corresponding slice of L3.
459H	1113	IA32_MC22_STATUS	Package	
45AH	1114	IA32_MC22_ADDR	Package	
45BH	1115	IA32_MC22_MISC	Package	
45CH	1116	IA32_MC23_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs." through Section 15.3.2.4, "IA32_MCi_MISC MSRs.". Bank MC23 reports MC error from a specific CBo (core broadcast) and its corresponding slice of L3.
45DH	1117	IA32_MC23_STATUS	Package	
45EH	1118	IA32_MC23_ADDR	Package	
45FH	1119	IA32_MC23_MISC	Package	
460H	1120	IA32_MC24_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs." through Section 15.3.2.4, "IA32_MCi_MISC MSRs.". Bank MC24 reports MC error from a specific CBo (core broadcast) and its corresponding slice of L3.
461H	1121	IA32_MC24_STATUS	Package	
462H	1122	IA32_MC24_ADDR	Package	
463H	1123	IA32_MC24_MISC	Package	
464H	1124	IA32_MC25_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs." through Section 15.3.2.4, "IA32_MCi_MISC MSRs.". Bank MC25 reports MC error from a specific CBo (core broadcast) and its corresponding slice of L3.
465H	1125	IA32_MC25_STATUS	Package	
466H	1126	IA32_MC25_ADDR	Package	
467H	1127	IA32_MC2MISC	Package	
468H	1128	IA32_MC26_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs." through Section 15.3.2.4, "IA32_MCi_MISC MSRs.". Bank MC26 reports MC error from a specific CBo (core broadcast) and its corresponding slice of L3.
469H	1129	IA32_MC26_STATUS	Package	
46AH	1130	IA32_MC26_ADDR	Package	
46BH	1131	IA32_MC26_MISC	Package	
46CH	1132	IA32_MC27_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs." through Section 15.3.2.4, "IA32_MCi_MISC MSRs.". Bank MC27 reports MC error from a specific CBo (core broadcast) and its corresponding slice of L3.
46DH	1133	IA32_MC27_STATUS	Package	
46EH	1134	IA32_MC27_ADDR	Package	
46FH	1135	IA32_MC27_MISC	Package	

Table 2-25. MSRs Supported by Intel® Xeon® Processors E5 v2 Product Family (based on Ivy Bridge-E microarchitecture) (Contd.)

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
470H	1136	IA32_MC28_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs." through Section 15.3.2.4, "IA32_MCi_MISC MSRs.". Bank MC28 reports MC error from a specific CBo (core broadcast) and its corresponding slice of L3.
471H	1137	IA32_MC28_STATUS	Package	
472H	1138	IA32_MC28_ADDR	Package	
473H	1139	IA32_MC28_MISC	Package	
613H	1555	MSR_PKG_PERF_STATUS	Package	**Package RAPL Perf Status (R/O)**
618H	1560	MSR_DRAM_POWER_LIMIT	Package	**DRAM RAPL Power Limit Control (R/W)** See Section 14.9.5, "DRAM RAPL Domain."
619H	1561	MSR_DRAM_ENERGY_STATUS	Package	**DRAM Energy Status (R/O)** See Section 14.9.5, "DRAM RAPL Domain."
61BH	1563	MSR_DRAM_PERF_STATUS	Package	**DRAM Performance Throttling Status (R/O)** See Section 14.9.5, "DRAM RAPL Domain."
61CH	1564	MSR_DRAM_POWER_INFO	Package	**DRAM RAPL Parameters (R/W)** See Section 14.9.5, "DRAM RAPL Domain."
639H	1593	MSR_PP0_ENERGY_STATUS	Package	**PP0 Energy Status (R/O)** See Section 14.9.4, "PP0/PP1 RAPL Domains."
See Table 2-19, for other MSR definitions applicable to Intel Xeon processor E5 v2 with CPUID signature 06_3EH				

2.11.2 Additional MSRs Supported by Intel® Xeon® Processor E7 v2 Family

Intel® Xeon® processor E7 v2 family (based on Ivy Bridge-E microarchitecture) with CPUID DisplayFamily_DisplayModel signature 06_3EH supports the MSR interfaces listed in Table 2-19, Table 2-25, and Table 2-26.

Table 2-26. Additional MSRs Supported by Intel® Xeon® Processor E7 v2 Family with DisplayFamily_DisplayModel Signature 06_3EH

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
3AH	58	IA32_FEATURE_CONTROL	Thread	**Control Features in Intel 64 Processor (R/W)** See Table 2-2.
		0		**Lock (R/WL)**
		1		**Enable VMX inside SMX operation (R/WL)**
		2		**Enable VMX outside SMX operation (R/WL)**
		14:8		**SENTER local functions enables (R/WL)**
		15		**SENTER global functions enable (R/WL)**
		63:16		Reserved.
179H	377	IA32_MCG_CAP	Thread	**Global Machine Check Capability (R/O)**
		7:0		**Count**
		8		**MCG_CTL_P**

Table 2-26. Additional MSRs Supported by Intel® Xeon® Processor E7 v2 Family with DisplayFamily_DisplayModel Signature 06_3EH

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
		9		**MCG_EXT_P**
		10		**MCP_CMCI_P**
		11		**MCG_TES_P**
		15:12		Reserved.
		23:16		**MCG_EXT_CNT**
		24		**MCG_SER_P**
		63:25		Reserved.
17AH	378	IA32_MCG_STATUS	Thread	Global Machine Check Status (R/WO)
		0		**RIPV**
		1		**EIPV**
		2		**MCIP**
		3		**LMCE signaled**
		63:4		Reserved.
1AEH	430	MSR_TURBO_RATIO_LIMIT1	Package	**Maximum Ratio Limit of Turbo Mode** RO if MSR_PLATFORM_INFO.[28] = 0, RW if MSR_PLATFORM_INFO.[28] = 1
		7:0	Package	**Maximum Ratio Limit for 9C** Maximum turbo ratio limit of 9 core active.
		15:8	Package	**Maximum Ratio Limit for 10C** Maximum turbo ratio limit of 10core active.
		23:16	Package	**Maximum Ratio Limit for 11C** Maximum turbo ratio limit of 11 core active.
		31:24	Package	**Maximum Ratio Limit for 12C** Maximum turbo ratio limit of 12 core active.
		39:32	Package	**Maximum Ratio Limit for 13C** Maximum turbo ratio limit of 13 core active.
		47:40	Package	**Maximum Ratio Limit for 14C** Maximum turbo ratio limit of 14 core active.
		55:48	Package	**Maximum Ratio Limit for 15C** Maximum turbo ratio limit of 15 core active.
		62:56		Reserved
		63	Package	**Semaphore for Turbo Ratio Limit Configuration** If 1, the processor uses override configuration[1] specified in MSR_TURBO_RATIO_LIMIT and MSR_TURBO_RATIO_LIMIT1. If 0, the processor uses factory-set configuration (Default).
29DH	669	IA32_MC29_CTL2	Package	See Table 2-2.
29EH	670	IA32_MC30_CTL2	Package	See Table 2-2.
29FH	671	IA32_MC31_CTL2	Package	See Table 2-2.

Table 2-26. Additional MSRs Supported by Intel® Xeon® Processor E7 v2 Family with DisplayFamily_DisplayModel Signature 06_3EH

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
3F1H	1009	MSR_PEBS_ENABLE	Thread	See Section 18.3.1.1.1, "Processor Event Based Sampling (PEBS)."
		0		Enable PEBS on IA32_PMC0. (R/W)
		1		Enable PEBS on IA32_PMC1. (R/W)
		2		Enable PEBS on IA32_PMC2. (R/W)
		3		Enable PEBS on IA32_PMC3. (R/W)
		31:4		Reserved.
		32		Enable Load Latency on IA32_PMC0. (R/W)
		33		Enable Load Latency on IA32_PMC1. (R/W)
		34		Enable Load Latency on IA32_PMC2. (R/W)
		35		Enable Load Latency on IA32_PMC3. (R/W)
		63:36		Reserved.
41BH	1051	IA32_MC6_MISC	Package	Misc MAC information of Integrated I/O. (R/O) see Section 15.3.2.4
		5:0		Recoverable Address LSB
		8:6		Address Mode
		15:9		Reserved
		31:16		PCI Express Requestor ID
		39:32		PCI Express Segment Number
		63:32		Reserved
474H	1140	IA32_MC29_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs." through Section 15.3.2.4, "IA32_MCi_MISC MSRs.". Bank MC29 reports MC error from a specific CBo (core broadcast) and its corresponding slice of L3.
475H	1141	IA32_MC29_STATUS	Package	
476H	1142	IA32_MC29_ADDR	Package	
477H	1143	IA32_MC29_MISC	Package	
478H	1144	IA32_MC30_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs." through Section 15.3.2.4, "IA32_MCi_MISC MSRs.". Bank MC30 reports MC error from a specific CBo (core broadcast) and its corresponding slice of L3.
479H	1145	IA32_MC30_STATUS	Package	
47AH	1146	IA32_MC30_ADDR	Package	
47BH	1147	IA32_MC30_MISC	Package	
47CH	1148	IA32_MC31_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs." through Section 15.3.2.4, "IA32_MCi_MISC MSRs.". Bank MC31 reports MC error from a specific CBo (core broadcast) and its corresponding slice of L3.
47DH	1149	IA32_MC31_STATUS	Package	
47EH	1150	IA32_MC31_ADDR	Package	
47FH	1147	IA32_MC31_MISC	Package	
See Table 2-19, Table 2-25 for other MSR definitions applicable to Intel Xeon processor E7 v2 with CPUID signature 06_3AH.				

NOTES:

1. An override configuration lower than the factory-set configuration is always supported. An override configuration higher than the factory-set configuration is dependent on features specific to the processor and the platform.

2.11.3 Additional Uncore PMU MSRs in the Intel® Xeon® Processor E5 v2 and E7 v2 Families

Intel Xeon Processor E5 v2 and E7 v2 families are based on the Ivy Bridge-E microarchitecture. The MSR-based uncore PMU interfaces are listed in Table 2-23 and Table 2-27. For complete detail of the uncore PMU, refer to Intel Xeon Processor E5 v2 Product Family Uncore Performance Monitoring Guide. These processors have a CPUID signature with DisplayFamily_DisplayModel of 06_3EH.

Table 2-27. Uncore PMU MSRs in Intel® Xeon® Processor E5 v2 and E7 v2 Families

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
C00H		MSR_PMON_GLOBAL_CTL	Package	Uncore perfmon per-socket global control.
C01H		MSR_PMON_GLOBAL_STATUS	Package	Uncore perfmon per-socket global status.
C06H		MSR_PMON_GLOBAL_CONFIG	Package	Uncore perfmon per-socket global configuration.
C15H		MSR_U_PMON_BOX_STATUS	Package	Uncore U-box perfmon U-box wide status.
C35H		MSR_PCU_PMON_BOX_STATUS	Package	Uncore PCU perfmon box wide status.
D1AH		MSR_C0_PMON_BOX_FILTER1	Package	Uncore C-box 0 perfmon box wide filter1.
D3AH		MSR_C1_PMON_BOX_FILTER1	Package	Uncore C-box 1 perfmon box wide filter1.
D5AH		MSR_C2_PMON_BOX_FILTER1	Package	Uncore C-box 2 perfmon box wide filter1.
D7AH		MSR_C3_PMON_BOX_FILTER1	Package	Uncore C-box 3 perfmon box wide filter1.
D9AH		MSR_C4_PMON_BOX_FILTER1	Package	Uncore C-box 4 perfmon box wide filter1.
DBAH		MSR_C5_PMON_BOX_FILTER1	Package	Uncore C-box 5 perfmon box wide filter1.
DDAH		MSR_C6_PMON_BOX_FILTER1	Package	Uncore C-box 6 perfmon box wide filter1.
DFAH		MSR_C7_PMON_BOX_FILTER1	Package	Uncore C-box 7 perfmon box wide filter1.
E04H		MSR_C8_PMON_BOX_CTL	Package	Uncore C-box 8 perfmon local box wide control.
E10H		MSR_C8_PMON_EVNTSEL0	Package	Uncore C-box 8 perfmon event select for C-box 8 counter 0.
E11H		MSR_C8_PMON_EVNTSEL1	Package	Uncore C-box 8 perfmon event select for C-box 8 counter 1.
E12H		MSR_C8_PMON_EVNTSEL2	Package	Uncore C-box 8 perfmon event select for C-box 8 counter 2.
E13H		MSR_C8_PMON_EVNTSEL3	Package	Uncore C-box 8 perfmon event select for C-box 8 counter 3.
E14H		MSR_C8_PMON_BOX_FILTER	Package	Uncore C-box 8 perfmon box wide filter.
E16H		MSR_C8_PMON_CTR0	Package	Uncore C-box 8 perfmon counter 0.
E17H		MSR_C8_PMON_CTR1	Package	Uncore C-box 8 perfmon counter 1.
E18H		MSR_C8_PMON_CTR2	Package	Uncore C-box 8 perfmon counter 2.
E19H		MSR_C8_PMON_CTR3	Package	Uncore C-box 8 perfmon counter 3.
E1AH		MSR_C8_PMON_BOX_FILTER1	Package	Uncore C-box 8 perfmon box wide filter1.
E24H		MSR_C9_PMON_BOX_CTL	Package	Uncore C-box 9 perfmon local box wide control.
E30H		MSR_C9_PMON_EVNTSEL0	Package	Uncore C-box 9 perfmon event select for C-box 9 counter 0.
E31H		MSR_C9_PMON_EVNTSEL1	Package	Uncore C-box 9 perfmon event select for C-box 9 counter 1.
E32H		MSR_C9_PMON_EVNTSEL2	Package	Uncore C-box 9 perfmon event select for C-box 9 counter 2.
E33H		MSR_C9_PMON_EVNTSEL3	Package	Uncore C-box 9 perfmon event select for C-box 9 counter 3.
E34H		MSR_C9_PMON_BOX_FILTER	Package	Uncore C-box 9 perfmon box wide filter.
E36H		MSR_C9_PMON_CTR0	Package	Uncore C-box 9 perfmon counter 0.
E37H		MSR_C9_PMON_CTR1	Package	Uncore C-box 9 perfmon counter 1.

Table 2-27. Uncore PMU MSRs in Intel® Xeon® Processor E5 v2 and E7 v2 Families (Contd.)

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
E38H		MSR_C9_PMON_CTR2	Package	Uncore C-box 9 perfmon counter 2.
E39H		MSR_C9_PMON_CTR3	Package	Uncore C-box 9 perfmon counter 3.
E3AH		MSR_C9_PMON_BOX_FILTER1	Package	Uncore C-box 9 perfmon box wide filter1.
E44H		MSR_C10_PMON_BOX_CTL	Package	Uncore C-box 10 perfmon local box wide control.
E50H		MSR_C10_PMON_EVNTSEL0	Package	Uncore C-box 10 perfmon event select for C-box 10 counter 0.
E51H		MSR_C10_PMON_EVNTSEL1	Package	Uncore C-box 10 perfmon event select for C-box 10 counter 1.
E52H		MSR_C10_PMON_EVNTSEL2	Package	Uncore C-box 10 perfmon event select for C-box 10 counter 2.
E53H		MSR_C10_PMON_EVNTSEL3	Package	Uncore C-box 10 perfmon event select for C-box 10 counter 3.
E54H		MSR_C10_PMON_BOX_FILTER	Package	Uncore C-box 10 perfmon box wide filter.
E56H		MSR_C10_PMON_CTR0	Package	Uncore C-box 10 perfmon counter 0.
E57H		MSR_C10_PMON_CTR1	Package	Uncore C-box 10 perfmon counter 1.
E58H		MSR_C10_PMON_CTR2	Package	Uncore C-box 10 perfmon counter 2.
E59H		MSR_C10_PMON_CTR3	Package	Uncore C-box 10 perfmon counter 3.
E5AH		MSR_C10_PMON_BOX_FILTER1	Package	Uncore C-box 10 perfmon box wide filter1.
E64H		MSR_C11_PMON_BOX_CTL	Package	Uncore C-box 11 perfmon local box wide control.
E70H		MSR_C11_PMON_EVNTSEL0	Package	Uncore C-box 11 perfmon event select for C-box 11 counter 0.
E71H		MSR_C11_PMON_EVNTSEL1	Package	Uncore C-box 11 perfmon event select for C-box 11 counter 1.
E72H		MSR_C11_PMON_EVNTSEL2	Package	Uncore C-box 11 perfmon event select for C-box 11 counter 2.
E73H		MSR_C11_PMON_EVNTSEL3	Package	Uncore C-box 11 perfmon event select for C-box 11 counter 3.
E74H		MSR_C11_PMON_BOX_FILTER	Package	Uncore C-box 11 perfmon box wide filter.
E76H		MSR_C11_PMON_CTR0	Package	Uncore C-box 11 perfmon counter 0.
E77H		MSR_C11_PMON_CTR1	Package	Uncore C-box 11 perfmon counter 1.
E78H		MSR_C11_PMON_CTR2	Package	Uncore C-box 11 perfmon counter 2.
E79H		MSR_C11_PMON_CTR3	Package	Uncore C-box 11 perfmon counter 3.
E7AH		MSR_C11_PMON_BOX_FILTER1	Package	Uncore C-box 11 perfmon box wide filter1.
E84H		MSR_C12_PMON_BOX_CTL	Package	Uncore C-box 12 perfmon local box wide control.
E90H		MSR_C12_PMON_EVNTSEL0	Package	Uncore C-box 12 perfmon event select for C-box 12 counter 0.
E91H		MSR_C12_PMON_EVNTSEL1	Package	Uncore C-box 12 perfmon event select for C-box 12 counter 1.
E92H		MSR_C12_PMON_EVNTSEL2	Package	Uncore C-box 12 perfmon event select for C-box 12 counter 2.
E93H		MSR_C12_PMON_EVNTSEL3	Package	Uncore C-box 12 perfmon event select for C-box 12 counter 3.
E94H		MSR_C12_PMON_BOX_FILTER	Package	Uncore C-box 12 perfmon box wide filter.
E96H		MSR_C12_PMON_CTR0	Package	Uncore C-box 12 perfmon counter 0.
E97H		MSR_C12_PMON_CTR1	Package	Uncore C-box 12 perfmon counter 1.
E98H		MSR_C12_PMON_CTR2	Package	Uncore C-box 12 perfmon counter 2.
E99H		MSR_C12_PMON_CTR3	Package	Uncore C-box 12 perfmon counter 3.
E9AH		MSR_C12_PMON_BOX_FILTER1	Package	Uncore C-box 12 perfmon box wide filter1.
EA4H		MSR_C13_PMON_BOX_CTL	Package	Uncore C-box 13 perfmon local box wide control.

Table 2-27. Uncore PMU MSRs in Intel® Xeon® Processor E5 v2 and E7 v2 Families (Contd.)

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
EB0H		MSR_C13_PMON_EVNTSEL0	Package	Uncore C-box 13 perfmon event select for C-box 13 counter 0.
EB1H		MSR_C13_PMON_EVNTSEL1	Package	Uncore C-box 13 perfmon event select for C-box 13 counter 1.
EB2H		MSR_C13_PMON_EVNTSEL2	Package	Uncore C-box 13 perfmon event select for C-box 13 counter 2.
EB3H		MSR_C13_PMON_EVNTSEL3	Package	Uncore C-box 13 perfmon event select for C-box 13 counter 3.
EB4H		MSR_C13_PMON_BOX_FILTER	Package	Uncore C-box 13 perfmon box wide filter.
EB6H		MSR_C13_PMON_CTR0	Package	Uncore C-box 13 perfmon counter 0.
EB7H		MSR_C13_PMON_CTR1	Package	Uncore C-box 13 perfmon counter 1.
EB8H		MSR_C13_PMON_CTR2	Package	Uncore C-box 13 perfmon counter 2.
EB9H		MSR_C13_PMON_CTR3	Package	Uncore C-box 13 perfmon counter 3.
EBAH		MSR_C13_PMON_BOX_FILTER1	Package	Uncore C-box 13 perfmon box wide filter1.
EC4H		MSR_C14_PMON_BOX_CTL	Package	Uncore C-box 14 perfmon local box wide control.
ED0H		MSR_C14_PMON_EVNTSEL0	Package	Uncore C-box 14 perfmon event select for C-box 14 counter 0.
ED1H		MSR_C14_PMON_EVNTSEL1	Package	Uncore C-box 14 perfmon event select for C-box 14 counter 1.
ED2H		MSR_C14_PMON_EVNTSEL2	Package	Uncore C-box 14 perfmon event select for C-box 14 counter 2.
ED3H		MSR_C14_PMON_EVNTSEL3	Package	Uncore C-box 14 perfmon event select for C-box 14 counter 3.
ED4H		MSR_C14_PMON_BOX_FILTER	Package	Uncore C-box 14 perfmon box wide filter.
ED6H		MSR_C14_PMON_CTR0	Package	Uncore C-box 14 perfmon counter 0.
ED7H		MSR_C14_PMON_CTR1	Package	Uncore C-box 14 perfmon counter 1.
ED8H		MSR_C14_PMON_CTR2	Package	Uncore C-box 14 perfmon counter 2.
ED9H		MSR_C14_PMON_CTR3	Package	Uncore C-box 14 perfmon counter 3.
EDAH		MSR_C14_PMON_BOX_FILTER1	Package	Uncore C-box 14 perfmon box wide filter1.

2.12 MSRS IN THE 4TH GENERATION INTEL® CORE™ PROCESSORS (BASED ON HASWELL MICROARCHITECTURE)

The 4th generation Intel® Core™ processor family and Intel® Xeon® processor E3-1200v3 product family (based on Haswell microarchitecture), with CPUID DisplayFamily_DisplayModel signature 06_3CH/06_45H/06_46H, support the MSR interfaces listed in Table 2-19, Table 2-20, Table 2-21, and Table 2-28. For an MSR listed in Table 2-19 that also appears in Table 2-28, Table 2-28 supercede Table 2-19.

The MSRs listed in Table 2-28 also apply to processors based on Haswell-E microarchitecture (see Section 2.13).

Table 2-28. Additional MSRs Supported by Processors based on the Haswell or Haswell-E microarchitectures

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
3BH	59	IA32_TSC_ADJUST	THREAD	**Per-Logical-Processor TSC ADJUST (R/W)** See Table 2-2.
CEH	206	MSR_PLATFORM_INFO	Package	Platform Information; contains power management and other model specific features enumeration. See http://biosbits.org.

Table 2-28. Additional MSRs Supported by Processors based on the Haswell or Haswell-E microarchitectures

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
		7:0		Reserved.
		15:8	Package	**Maximum Non-Turbo Ratio (R/O)** The is the ratio of the frequency that invariant TSC runs at. Frequency = ratio * 100 MHz.
		27:16		Reserved.
		28	Package	**Programmable Ratio Limit for Turbo Mode (R/O)** When set to 1, indicates that Programmable Ratio Limits for Turbo mode is enabled, and when set to 0, indicates Programmable Ratio Limits for Turbo mode is disabled.
		29	Package	**Programmable TDP Limit for Turbo Mode (R/O)** When set to 1, indicates that TDP Limits for Turbo mode are programmable, and when set to 0, indicates TDP Limit for Turbo mode is not programmable.
		31:30		Reserved.
		32	Package	**Low Power Mode Support (LPM) (R/O)** When set to 1, indicates that LPM is supported, and when set to 0, indicates LPM is not supported.
		34:33	Package	**Number of ConfigTDP Levels (R/O)** 00: Only Base TDP level available. 01: One additional TDP level available. 02: Two additional TDP level available. 11: Reserved
		39:35		Reserved.
		47:40	Package	**Maximum Efficiency Ratio (R/O)** The is the minimum ratio (maximum efficiency) that the processor can operates, in units of 100MHz.
		55:48	Package	**Minimum Operating Ratio (R/O)** Contains the minimum supported operating ratio in units of 100 MHz.
		63:56		Reserved.
186H	390	IA32_PERFEVTSEL0	THREAD	**Performance Event Select for Counter 0 (R/W)** Supports all fields described inTable 2-2 and the fields below.
		32		IN_TX: see Section 18.3.6.5.1 When IN_TX (bit 32) is set, AnyThread (bit 21) should be cleared to prevent incorrect results
187H	391	IA32_PERFEVTSEL1	THREAD	**Performance Event Select for Counter 1 (R/W)** Supports all fields described inTable 2-2 and the fields below.
		32		IN_TX: see Section 18.3.6.5.1 When IN_TX (bit 32) is set, AnyThread (bit 21) should be cleared to prevent incorrect results
188H	392	IA32_PERFEVTSEL2	THREAD	**Performance Event Select for Counter 2 (R/W)** Supports all fields described inTable 2-2 and the fields below.

Table 2-28. Additional MSRs Supported by Processors based on the Haswell or Haswell-E microarchitectures

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
		32		IN_TX: see Section 18.3.6.5.1 When IN_TX (bit 32) is set, AnyThread (bit 21) should be cleared to prevent incorrect results
		33		IN_TXCP: see Section 18.3.6.5.1 When IN_TXCP=1 & IN_TX=1 and in sampling, spurious PMI may occur and transactions may continuously abort near overflow conditions. Software should favor using IN_TXCP for counting over sampling. If sampling, software should use large "sample-after" value after clearing the counter configured to use IN_TXCP and also always reset the counter even when no overflow condition was reported.
189H	393	IA32_PERFEVTSEL3	THREAD	**Performance Event Select for Counter 3 (R/W)** Supports all fields described inTable 2-2 and the fields below.
		32		IN_TX: see Section 18.3.6.5.1 When IN_TX (bit 32) is set, AnyThread (bit 21) should be cleared to prevent incorrect results
1C8H	456	MSR_LBR_SELECT	Thread	**Last Branch Record Filtering Select Register (R/W)**
		0		**CPL_EQ_0**
		1		**CPL_NEQ_0**
		2		**JCC**
		3		**NEAR_REL_CALL**
		4		**NEAR_IND_CALL**
		5		**NEAR_RET**
		6		**NEAR_IND_JMP**
		7		**NEAR_REL_JMP**
		8		**FAR_BRANCH**
		9		**EN_CALL_STACK**
		63:9		Reserved.
1D9H	473	IA32_DEBUGCTL	Thread	**Debug Control (R/W)** See Table 2-2.
		0		**LBR: Last Branch Record**
		1		**BTF**
		5:2		Reserved.
		6		**TR: Branch Trace**
		7		**BTS: Log Branch Trace Message to BTS buffer**
		8		**BTINT**
		9		**BTS_OFF_OS**
		10		**BTS_OFF_USER**
		11		**FREEZE_LBR_ON_PMI**
		12		**FREEZE_PERFMON_ON_PMI**

Table 2-28. Additional MSRs Supported by Processors based on the Haswell or Haswell-E microarchitectures

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
		13		**ENABLE_UNCORE_PMI**
		14		**FREEZE_WHILE_SMM**
		15		**RTM_DEBUG**
		63:15		Reserved.
491H	1169	IA32_VMX_VMFUNC	THREAD	**Capability Reporting Register of VM-function Controls (R/O)** See Table 2-2
60BH	1548	MSR_PKGC_IRTL1	Package	**Package C6/C7 Interrupt Response Limit 1 (R/W)** This MSR defines the interrupt response time limit used by the processor to manage transition to package C6 or C7 state. The latency programmed in this register is for the shorter-latency sub C-states used by an MWAIT hint to C6 or C7 state. Note: C-state values are processor specific C-state code names, unrelated to MWAIT extension C-state parameters or ACPI C-States.
		9:0		**Interrupt response time limit (R/W)** Specifies the limit that should be used to decide if the package should be put into a package C6 or C7 state.
		12:10		**Time Unit (R/W)** Specifies the encoding value of time unit of the interrupt response time limit. See Table 2-19 for supported time unit encodings.
		14:13		Reserved.
		15		**Valid (R/W)** Indicates whether the values in bits 12:0 are valid and can be used by the processor for package C-sate management.
		63:16		Reserved.
60CH	1548	MSR_PKGC_IRTL2	Package	**Package C6/C7 Interrupt Response Limit 2 (R/W)** This MSR defines the interrupt response time limit used by the processor to manage transition to package C6 or C7 state. The latency programmed in this register is for the longer-latency sub C-states used by an MWAIT hint to C6 or C7 state. Note: C-state values are processor specific C-state code names, unrelated to MWAIT extension C-state parameters or ACPI C-States.
		9:0		**Interrupt response time limit (R/W)** Specifies the limit that should be used to decide if the package should be put into a package C6 or C7 state.
		12:10		**Time Unit (R/W)** Specifies the encoding value of time unit of the interrupt response time limit. See Table 2-19 for supported time unit encodings.
		14:13		Reserved.
		15		**Valid (R/W)** Indicates whether the values in bits 12:0 are valid and can be used by the processor for package C-sate management.
		63:16		Reserved.

Table 2-28. Additional MSRs Supported by Processors based on the Haswell or Haswell-E microarchitectures

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
613H	1555	MSR_PKG_PERF_STATUS	Package	**PKG Perf Status (R/O)** See Section 14.9.3, "Package RAPL Domain."
619H	1561	MSR_DRAM_ENERGY_ STATUS	Package	**DRAM Energy Status (R/O)** See Section 14.9.5, "DRAM RAPL Domain."
61BH	1563	MSR_DRAM_PERF_STATUS	Package	**DRAM Performance Throttling Status (R/O)** See Section 14.9.5, "DRAM RAPL Domain."
648H	1608	MSR_CONFIG_TDP_ NOMINAL	Package	**Base TDP Ratio (R/O)**
		7:0		**Config_TDP_Base** Base TDP level ratio to be used for this specific processor (in units of 100 MHz).
		63:8		Reserved.
649H	1609	MSR_CONFIG_TDP_LEVEL1	Package	ConfigTDP Level 1 ratio and power level (R/O)
		14:0		PKG_TDP_LVL1. Power setting for ConfigTDP Level 1.
		15		Reserved
		23:16		Config_TDP_LVL1_Ratio. ConfigTDP level 1 ratio to be used for this specific processor.
		31:24		Reserved
		46:32		PKG_MAX_PWR_LVL1. Max Power setting allowed for ConfigTDP Level 1.
		62:47		PKG_MIN_PWR_LVL1. MIN Power setting allowed for ConfigTDP Level 1.
		63		Reserved.
64AH	1610	MSR_CONFIG_TDP_LEVEL2	Package	ConfigTDP Level 2 ratio and power level (R/O)
		14:0		PKG_TDP_LVL2. Power setting for ConfigTDP Level 2.
		15		Reserved
		23:16		Config_TDP_LVL2_Ratio. ConfigTDP level 2 ratio to be used for this specific processor.
		31:24		Reserved
		46:32		PKG_MAX_PWR_LVL2. Max Power setting allowed for ConfigTDP Level 2.
		62:47		PKG_MIN_PWR_LVL2. MIN Power setting allowed for ConfigTDP Level 2.
		63		Reserved.
64BH	1611	MSR_CONFIG_TDP_ CONTROL	Package	**ConfigTDP Control (R/W)**
		1:0		**TDP_LEVEL (RW/L)** System BIOS can program this field.
		30:2		Reserved.

Table 2-28. Additional MSRs Supported by Processors based on the Haswell or Haswell-E microarchitectures

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
		31		**Config_TDP_Lock (RW/L)** When this bit is set, the content of this register is locked until a reset.
		63:32		Reserved.
64CH	1612	MSR_TURBO_ACTIVATION_RATIO	Package	**ConfigTDP Control (R/W)**
		7:0		**MAX_NON_TURBO_RATIO (RW/L)** System BIOS can program this field.
		30:8		Reserved.
		31		**TURBO_ACTIVATION_RATIO_Lock (RW/L)** When this bit is set, the content of this register is locked until a reset.
		63:32		Reserved.
C80H	3200	IA32_DEBUG_INTERFACE	Package	**Silicon Debug Feature Control (R/W)** See Table 2-2.

2.12.1 MSRs in 4th Generation Intel® Core™ Processor Family (based on Haswell Microarchitecture)

Table 2-29 lists model-specific registers (MSRs) that are specific to 4th generation Intel® Core™ processor family and Intel® Xeon® processor E3-1200 v3 product family (based on Haswell microarchitecture). These processors have a CPUID signature with DisplayFamily_DisplayModel of 06_3CH/06_45H/06_46H, see Table 2-1.

Table 2-29. MSRs Supported by 4th Generation Intel® Core™ Processors (Haswell microarchitecture)

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
E2H	226	MSR_PKG_CST_CONFIG_CONTROL	Core	**C-State Configuration Control (R/W)** Note: C-state values are processor specific C-state code names, unrelated to MWAIT extension C-state parameters or ACPI C-states. See http://biosbits.org.

Table 2-29. MSRs Supported by 4th Generation Intel® Core™ Processors (Haswell microarchitecture) (Contd.)

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
		3:0		**Package C-State Limit (R/W)** Specifies the lowest processor-specific C-state code name (consuming the least power) for the package. The default is set as factory-configured package C-state limit. The following C-state code name encodings are supported: 0000b: C0/C1 (no package C-state support) 0001b: C2 0010b: C3 0011b: C6 0100b: C7 0101b: C7s Package C states C7 are not available to processor with signature 06_3CH
		9:4		Reserved
		10		**I/O MWAIT Redirection Enable (R/W)**
		14:11		Reserved
		15		**CFG Lock (R/WO)**
		24:16		Reserved
		25		**C3 State Auto Demotion Enable (R/W)**
		26		**C1 State Auto Demotion Enable (R/W)**
		27		**Enable C3 Undemotion (R/W)**
		28		**Enable C1 Undemotion (R/W)**
		63:29		Reserved
17DH	390	MSR_SMM_MCA_CAP	THREAD	**Enhanced SMM Capabilities (SMM-RO)** Reports SMM capability Enhancement. Accessible only while in SMM.
		57:0		**Reserved**
		58		**SMM_Code_Access_Chk (SMM-RO)** If set to 1 indicates that the SMM code access restriction is supported and the MSR_SMM_FEATURE_CONTROL is supported.
		59		**Long_Flow_Indication (SMM-RO)** If set to 1 indicates that the SMM long flow indicator is supported and the MSR_SMM_DELAYED is supported.
		63:60		Reserved
1ADH	429	MSR_TURBO_RATIO_LIMIT	Package	**Maximum Ratio Limit of Turbo Mode** RO if MSR_PLATFORM_INFO.[28] = 0, RW if MSR_PLATFORM_INFO.[28] = 1
		7:0	Package	**Maximum Ratio Limit for 1C** Maximum turbo ratio limit of 1 core active.
		15:8	Package	**Maximum Ratio Limit for 2C** Maximum turbo ratio limit of 2 core active.

Table 2-29. MSRs Supported by 4th Generation Intel® Core™ Processors (Haswell microarchitecture) (Contd.)

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
		23:16	Package	**Maximum Ratio Limit for 3C** Maximum turbo ratio limit of 3 core active.
		31:24	Package	**Maximum Ratio Limit for 4C** Maximum turbo ratio limit of 4 core active.
		63:32		Reserved.
391H	913	MSR_UNC_PERF_GLOBAL_CTRL	Package	Uncore PMU global control
		0		Core 0 select
		1		Core 1 select
		2		Core 2 select
		3		Core 3 select
		18:4		Reserved.
		29		Enable all uncore counters
		30		Enable wake on PMI
		31		Enable Freezing counter when overflow
		63:32		Reserved.
392H	914	MSR_UNC_PERF_GLOBAL_STATUS	Package	Uncore PMU main status
		0		Fixed counter overflowed
		1		An ARB counter overflowed
		2		Reserved
		3		A CBox counter overflowed (on any slice)
		63:4		Reserved.
394H	916	MSR_UNC_PERF_FIXED_CTRL	Package	Uncore fixed counter control (R/W)
		19:0		Reserved
		20		Enable overflow propagation
		21		Reserved
		22		Enable counting
		63:23		Reserved.
395H	917	MSR_UNC_PERF_FIXED_CTR	Package	Uncore fixed counter
		47:0		Current count
		63:48		Reserved.
396H	918	MSR_UNC_CBO_CONFIG	Package	Uncore C-Box configuration information (R/O)
		3:0		Encoded number of C-Box, derive value by "-1"
		63:4		Reserved.
3B0H	946	MSR_UNC_ARB_PERFCTR0	Package	Uncore Arb unit, performance counter 0

Table 2-29. MSRs Supported by 4th Generation Intel® Core™ Processors (Haswell microarchitecture) (Contd.)

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
3B1H	947	MSR_UNC_ARB_PERFCTR1	Package	Uncore Arb unit, performance counter 1
3B2H	944	MSR_UNC_ARB_ PERFEVTSEL0	Package	Uncore Arb unit, counter 0 event select MSR
3B3H	945	MSR_UNC_ARB_ PERFEVTSEL1	Package	Uncore Arb unit, counter 1 event select MSR
391H	913	MSR_UNC_PERF_GLOBAL_ CTRL	Package	Uncore PMU global control
		0		Core 0 select
		1		Core 1 select
		2		Core 2 select
		3		Core 3 select
		18:4		Reserved.
		29		Enable all uncore counters
		30		Enable wake on PMI
		31		Enable Freezing counter when overflow
		63:32		Reserved.
395H	917	MSR_UNC_PERF_FIXED_ CTR	Package	Uncore fixed counter
		47:0		Current count
		63:48		Reserved.
3B3H	945	MSR_UNC_ARB_ PERFEVTSEL1	Package	Uncore Arb unit, counter 1 event select MSR
4E0H	1248	MSR_SMM_FEATURE_CONTR OL	Package	**Enhanced SMM Feature Control (SMM-RW)** Reports SMM capability Enhancement. Accessible only while in SMM.
		0		**Lock (SMM-RWO)** When set to '1' locks this register from further changes
		1		Reserved
		2		**SMM_Code_Chk_En (SMM-RW)** This control bit is available only if MSR_SMM_MCA_CAP[58] == 1. When set to '0' (default) none of the logical processors are prevented from executing SMM code outside the ranges defined by the SMRR. When set to '1' any logical processor in the package that attempts to execute SMM code not within the ranges defined by the SMRR will assert an unrecoverable MCE.
		63:3		Reserved
4E2H	1250	MSR_SMM_DELAYED	Package	**SMM Delayed (SMM-RO)** Reports the interruptible state of all logical processors in the package. Available only while in SMM and MSR_SMM_MCA_CAP[LONG_FLOW_INDICATION] == 1.

Table 2-29. MSRs Supported by 4th Generation Intel® Core™ Processors (Haswell microarchitecture) (Contd.)

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
		N-1:0		**LOG_PROC_STATE (SMM-RO)** Each bit represents a logical processor of its state in a long flow of internal operation which delays servicing an interrupt. The corresponding bit will be set at the start of long events such as: Microcode Update Load, C6, WBINVD, Ratio Change, Throttle. The bit is automatically cleared at the end of each long event. The reset value of this field is 0. Only bit positions below N = CPUID.(EAX=0BH, ECX=PKG_LVL):EBX[15:0] can be updated.
		63:N		Reserved
4E3H	1251	MSR_SMM_BLOCKED	Package	**SMM Blocked (SMM-RO)** Reports the blocked state of all logical processors in the package. Available only while in SMM.
		N-1:0		**LOG_PROC_STATE (SMM-RO)** Each bit represents a logical processor of its blocked state to service an SMI. The corresponding bit will be set if the logical processor is in one of the following states: Wait For SIPI or SENTER Sleep. The reset value of this field is 0FFFH. Only bit positions below N = CPUID.(EAX=0BH, ECX=PKG_LVL):EBX[15:0] can be updated.
		63:N		Reserved
606H	1542	MSR_RAPL_POWER_UNIT	Package	**Unit Multipliers used in RAPL Interfaces (R/O)**
		3:0	Package	**Power Units** See Section 14.9.1, "RAPL Interfaces."
		7:4	Package	**Reserved**
		12:8	Package	**Energy Status Units** Energy related information (in Joules) is based on the multiplier, 1/2^ESU; where ESU is an unsigned integer represented by bits 12:8. Default value is 0EH (or 61 micro-joules)
		15:13	Package	**Reserved**
		19:16	Package	**Time Units** See Section 14.9.1, "RAPL Interfaces."
		63:20		Reserved
639H	1593	MSR_PP0_ENERGY_STATUS	Package	**PP0 Energy Status (R/O)** See Section 14.9.4, "PP0/PP1 RAPL Domains."
640H	1600	MSR_PP1_POWER_LIMIT	Package	**PP1 RAPL Power Limit Control (R/W)** See Section 14.9.4, "PP0/PP1 RAPL Domains."
641H	1601	MSR_PP1_ENERGY_STATUS	Package	**PP1 Energy Status (R/O)** See Section 14.9.4, "PP0/PP1 RAPL Domains."
642H	1602	MSR_PP1_POLICY	Package	**PP1 Balance Policy (R/W)** See Section 14.9.4, "PP0/PP1 RAPL Domains."

Table 2-29. MSRs Supported by 4th Generation Intel® Core™ Processors (Haswell microarchitecture) (Contd.)

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
690H	1680	MSR_CORE_PERF_LIMIT_REASONS	Package	**Indicator of Frequency Clipping in Processor Cores (R/W) (frequency refers to processor core frequency)**
		0		**PROCHOT Status (R0)** When set, processor core frequency is reduced below the operating system request due to assertion of external PROCHOT.
		1		**Thermal Status (R0)** When set, frequency is reduced below the operating system request due to a thermal event.
		3:2		Reserved.
		4		**Graphics Driver Status (R0)** When set, frequency is reduced below the operating system request due to Processor Graphics driver override.
		5		**Autonomous Utilization-Based Frequency Control Status (R0)** When set, frequency is reduced below the operating system request because the processor has detected that utilization is low.
		6		**VR Therm Alert Status (R0)** When set, frequency is reduced below the operating system request due to a thermal alert from the Voltage Regulator.
		7		Reserved.
		8		**Electrical Design Point Status (R0)** When set, frequency is reduced below the operating system request due to electrical design point constraints (e.g. maximum electrical current consumption).
		9		**Core Power Limiting Status (R0)** When set, frequency is reduced below the operating system request due to domain-level power limiting.
		10		**Package-Level Power Limiting PL1 Status (R0)** When set, frequency is reduced below the operating system request due to package-level power limiting PL1.
		11		**Package-Level PL2 Power Limiting Status (R0)** When set, frequency is reduced below the operating system request due to package-level power limiting PL2.
		12		**Max Turbo Limit Status (R0)** When set, frequency is reduced below the operating system request due to multi-core turbo limits.
		13		**Turbo Transition Attenuation Status (R0)** When set, frequency is reduced below the operating system request due to Turbo transition attenuation. This prevents performance degradation due to frequent operating ratio changes.
		15:14		**Reserved**

Table 2-29. MSRs Supported by 4th Generation Intel® Core™ Processors (Haswell microarchitecture) (Contd.)

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
		16		**PROCHOT Log** When set, indicates that the PROCHOT Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.
		17		**Thermal Log** When set, indicates that the Thermal Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.
		19:18		Reserved.
		20		**Graphics Driver Log** When set, indicates that the Graphics Driver Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.
		21		**Autonomous Utilization-Based Frequency Control Log** When set, indicates that the Autonomous Utilization-Based Frequency Control Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.
		22		**VR Therm Alert Log** When set, indicates that the VR Therm Alert Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.
		23		Reserved.
		24		**Electrical Design Point Log** When set, indicates that the EDP Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.
		25		**Core Power Limiting Log** When set, indicates that the Core Power Limiting Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.
		26		**Package-Level PL1 Power Limiting Log** When set, indicates that the Package Level PL1 Power Limiting Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.
		27		**Package-Level PL2 Power Limiting Log** When set, indicates that the Package Level PL2 Power Limiting Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.
		28		**Max Turbo Limit Log** When set, indicates that the Max Turbo Limit Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.

Table 2-29. MSRs Supported by 4th Generation Intel® Core™ Processors (Haswell microarchitecture) (Contd.)

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
		29		**Turbo Transition Attenuation Log** When set, indicates that the Turbo Transition Attenuation Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.
		63:30		Reserved.
6B0H	1712	MSR_GRAPHICS_PERF_LIMIT_REASONS	Package	**Indicator of Frequency Clipping in the Processor Graphics (R/W)** **(frequency refers to processor graphics frequency)**
		0		**PROCHOT Status (R0)** When set, frequency is reduced below the operating system request due to assertion of external PROCHOT.
		1		**Thermal Status (R0)** When set, frequency is reduced below the operating system request due to a thermal event.
		3:2		Reserved.
		4		**Graphics Driver Status (R0)** When set, frequency is reduced below the operating system request due to Processor Graphics driver override.
		5		**Autonomous Utilization-Based Frequency Control Status (R0)** When set, frequency is reduced below the operating system request because the processor has detected that utilization is low
		6		**VR Therm Alert Status (R0)** When set, frequency is reduced below the operating system request due to a thermal alert from the Voltage Regulator.
		7		Reserved.
		8		**Electrical Design Point Status (R0)** When set, frequency is reduced below the operating system request due to electrical design point constraints (e.g. maximum electrical current consumption).
		9		**Graphics Power Limiting Status (R0)** When set, frequency is reduced below the operating system request due to domain-level power limiting.
		10		**Package-Level Power Limiting PL1 Status (R0)** When set, frequency is reduced below the operating system request due to package-level power limiting PL1.
		11		**Package-Level PL2 Power Limiting Status (R0)** When set, frequency is reduced below the operating system request due to package-level power limiting PL2.
		15:12		**Reserved**
		16		**PROCHOT Log** When set, indicates that the PROCHOT Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.

Table 2-29. MSRs Supported by 4th Generation Intel® Core™ Processors (Haswell microarchitecture) (Contd.)

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
		17		**Thermal Log** When set, indicates that the Thermal Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.
		19:18		Reserved.
		20		**Graphics Driver Log** When set, indicates that the Graphics Driver Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.
		21		**Autonomous Utilization-Based Frequency Control Log** When set, indicates that the Autonomous Utilization-Based Frequency Control Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.
		22		**VR Therm Alert Log** When set, indicates that the VR Therm Alert Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.
		23		Reserved.
		24		**Electrical Design Point Log** When set, indicates that the EDP Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.
		25		**Core Power Limiting Log** When set, indicates that the Core Power Limiting Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.
		26		**Package-Level PL1 Power Limiting Log** When set, indicates that the Package Level PL1 Power Limiting Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.
		27		**Package-Level PL2 Power Limiting Log** When set, indicates that the Package Level PL2 Power Limiting Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.
		28		**Max Turbo Limit Log** When set, indicates that the Max Turbo Limit Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.
		29		**Turbo Transition Attenuation Log** When set, indicates that the Turbo Transition Attenuation Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.

Table 2-29. MSRs Supported by 4th Generation Intel® Core™ Processors (Haswell microarchitecture) (Contd.)

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
		63:30		Reserved.
6B1H	1713	MSR_RING_PERF_LIMIT_REASONS	Package	**Indicator of Frequency Clipping in the Ring Interconnect (R/W)** **(frequency refers to ring interconnect in the uncore)**
		0		**PROCHOT Status (R0)** When set, frequency is reduced below the operating system request due to assertion of external PROCHOT.
		1		**Thermal Status (R0)** When set, frequency is reduced below the operating system request due to a thermal event.
		5:2		Reserved.
		6		**VR Therm Alert Status (R0)** When set, frequency is reduced below the operating system request due to a thermal alert from the Voltage Regulator.
		7		Reserved.
		8		**Electrical Design Point Status (R0)** When set, frequency is reduced below the operating system request due to electrical design point constraints (e.g. maximum electrical current consumption).
		9		Reserved.
		10		**Package-Level Power Limiting PL1 Status (R0)** When set, frequency is reduced below the operating system request due to package-level power limiting PL1.
		11		**Package-Level PL2 Power Limiting Status (R0)** When set, frequency is reduced below the operating system request due to package-level power limiting PL2.
		15:12		**Reserved**
		16		**PROCHOT Log** When set, indicates that the PROCHOT Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.
		17		**Thermal Log** When set, indicates that the Thermal Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.
		19:18		Reserved.
		20		**Graphics Driver Log** When set, indicates that the Graphics Driver Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.

Table 2-29. MSRs Supported by 4th Generation Intel® Core™ Processors (Haswell microarchitecture) (Contd.)

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
		21		**Autonomous Utilization-Based Frequency Control Log** When set, indicates that the Autonomous Utilization-Based Frequency Control Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.
		22		**VR Therm Alert Log** When set, indicates that the VR Therm Alert Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.
		23		Reserved.
		24		**Electrical Design Point Log** When set, indicates that the EDP Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.
		25		**Core Power Limiting Log** When set, indicates that the Core Power Limiting Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.
		26		**Package-Level PL1 Power Limiting Log** When set, indicates that the Package Level PL1 Power Limiting Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.
		27		**Package-Level PL2 Power Limiting Log** When set, indicates that the Package Level PL2 Power Limiting Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.
		28		**Max Turbo Limit Log** When set, indicates that the Max Turbo Limit Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.
		29		**Turbo Transition Attenuation Log** When set, indicates that the Turbo Transition Attenuation Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.
		63:30		Reserved.
700H	1792	MSR_UNC_CBO_0_ PERFEVTSEL0	Package	Uncore C-Box 0, counter 0 event select MSR
701H	1793	MSR_UNC_CBO_0_ PERFEVTSEL1	Package	Uncore C-Box 0, counter 1 event select MSR
706H	1798	MSR_UNC_CBO_0_PERFCTR0	Package	Uncore C-Box 0, performance counter 0
707H	1799	MSR_UNC_CBO_0_PERFCTR1	Package	Uncore C-Box 0, performance counter 1

Table 2-29. MSRs Supported by 4th Generation Intel® Core™ Processors (Haswell microarchitecture) (Contd.)

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
710H	1808	MSR_UNC_CBO_1_ PERFEVTSEL0	Package	Uncore C-Box 1, counter 0 event select MSR
711H	1809	MSR_UNC_CBO_1_ PERFEVTSEL1	Package	Uncore C-Box 1, counter 1 event select MSR
716H	1814	MSR_UNC_CBO_1_PERFCTR0	Package	Uncore C-Box 1, performance counter 0
717H	1815	MSR_UNC_CBO_1_PERFCTR1	Package	Uncore C-Box 1, performance counter 1
720H	1824	MSR_UNC_CBO_2_ PERFEVTSEL0	Package	Uncore C-Box 2, counter 0 event select MSR
721H	1824	MSR_UNC_CBO_2_ PERFEVTSEL1	Package	Uncore C-Box 2, counter 1 event select MSR
726H	1830	MSR_UNC_CBO_2_PERFCTR0	Package	Uncore C-Box 2, performance counter 0
727H	1831	MSR_UNC_CBO_2_PERFCTR1	Package	Uncore C-Box 2, performance counter 1
730H	1840	MSR_UNC_CBO_3_ PERFEVTSEL0	Package	Uncore C-Box 3, counter 0 event select MSR
731H	1841	MSR_UNC_CBO_3_ PERFEVTSEL1	Package	Uncore C-Box 3, counter 1 event select MSR.
736H	1846	MSR_UNC_CBO_3_PERFCTR0	Package	Uncore C-Box 3, performance counter 0.
737H	1847	MSR_UNC_CBO_3_PERFCTR1	Package	Uncore C-Box 3, performance counter 1.
See Table 2-19, Table 2-20, Table 2-21, Table 2-24, Table 2-28 for other MSR definitions applicable to processors with CPUID signatures 063CH, 06_46H.				

2.12.2 Additional Residency MSRs Supported in 4th Generation Intel® Core™ Processors

The 4th generation Intel® Core™ processor family (based on Haswell microarchitecture) with CPUID DisplayFamily_DisplayModel signature 06_45H supports the MSR interfaces listed in Table 2-19, Table 2-20, Table 2-28, Table 2-29, and Table 2-30.

Table 2-30. Additional Residency MSRs Supported by 4th Generation Intel® Core™ Processors with DisplayFamily_DisplayModel Signature 06_45H

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
E2H	226	MSR_PKG_CST_CONFIG_ CONTROL	Core	**C-State Configuration Control (R/W)** Note: C-state values are processor specific C-state code names, unrelated to MWAIT extension C-state parameters or ACPI C-states. See http://biosbits.org.

Table 2-30. Additional Residency MSRs Supported by 4th Generation Intel® Core™ Processors with DisplayFamily_DisplayModel Signature 06_45H

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
		3:0		**Package C-State Limit (R/W)** Specifies the lowest processor-specific C-state code name (consuming the least power) for the package. The default is set as factory-configured package C-state limit. The following C-state code name encodings are supported: 0000b: C0/C1 (no package C-state support) 0001b: C2 0010b: C3 0011b: C6 0100b: C7 0101b: C7s 0110b: C8 0111b: C9 1000b: C10
		9:4		Reserved
		10		**I/O MWAIT Redirection Enable (R/W)**
		14:11		Reserved
		15		**CFG Lock (R/WO)**
		24:16		Reserved
		25		**C3 State Auto Demotion Enable (R/W)**
		26		**C1 State Auto Demotion Enable (R/W)**
		27		**Enable C3 Undemotion (R/W)**
		28		**Enable C1 Undemotion (R/W)**
		63:29		Reserved
630H	1584	MSR_PKG_C8_RESIDENCY	Package	Note: C-state values are processor specific C-state code names, unrelated to MWAIT extension C-state parameters or ACPI C-States.
		59:0		Package C8 Residency Counter. (R/O) Value since last reset that this package is in processor-specific C8 states. Count at the same frequency as the TSC.
		63:60		Reserved
631H	1585	MSR_PKG_C9_RESIDENCY	Package	Note: C-state values are processor specific C-state code names, unrelated to MWAIT extension C-state parameters or ACPI C-States.
		59:0		Package C9 Residency Counter. (R/O) Value since last reset that this package is in processor-specific C9 states. Count at the same frequency as the TSC.
		63:60		Reserved
632H	1586	MSR_PKG_C10_RESIDENCY	Package	Note: C-state values are processor specific C-state code names, unrelated to MWAIT extension C-state parameters or ACPI C-States.

Table 2-30. Additional Residency MSRs Supported by 4th Generation Intel® Core™ Processors with DisplayFamily_DisplayModel Signature 06_45H

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
		59:0		Package C10 Residency Counter. (R/O) Value since last reset that this package is in processor-specific C10 states. Count at the same frequency as the TSC.
		63:60		Reserved
See Table 2-19, Table 2-20, Table 2-21, Table 2-28, Table 2-29 for other MSR definitions applicable to processors with CPUID signature 06_45H.				

2.13 MSRS IN INTEL® XEON® PROCESSOR E5 V3 AND E7 V3 PRODUCT FAMILY

Intel® Xeon® processor E5 v3 family and Intel® Xeon® processor E7 v3 family are based on Haswell-E microarchitecture (CPUID DisplayFamily_DisplayModel = 06_3F). These processors supports the MSR interfaces listed in Table 2-19, Table 2-28, and Table 2-31.

Table 2-31. Additional MSRs Supported by Intel® Xeon® Processor E5 v3 Family

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
35H	53	MSR_CORE_THREAD_COUNT	Package	**Configured State of Enabled Processor Core Count and Logical Processor Count (RO)** ▪ After a Power-On RESET, enumerates factory configuration of the number of processor cores and logical processors in the physical package. ▪ Following the sequence of (i) BIOS modified a Configuration Mask which selects a subset of processor cores to be active post RESET and (ii) a RESET event after the modification, enumerates the current configuration of enabled processor core count and logical processor count in the physical package.
		15:0		**Core_COUNT (RO)** The number of processor cores that are currently enabled (by either factory configuration or BIOS configuration) in the physical package.
		31:16		**THREAD_COUNT (RO)** The number of logical processors that are currently enabled (by either factory configuration or BIOS configuration) in the physical package.
		63:32		Reserved
53H	83	MSR_THREAD_ID_INFO	Thread	**A Hardware Assigned ID for the Logical Processor (RO)**
		7:0		**Logical_Processor_ID (RO)** An implementation-specific numerical. value physically assigned to each logical processor. This ID is not related to Initial APIC ID or x2APIC ID, it is unique within a physical package.
		63:8		Reserved

Table 2-31. Additional MSRs Supported by Intel® Xeon® Processor E5 v3 Family

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
E2H	226	MSR_PKG_CST_CONFIG_ CONTROL	Core	**C-State Configuration Control (R/W)** Note: C-state values are processor specific C-state code names, unrelated to MWAIT extension C-state parameters or ACPI C-states. See http://biosbits.org.
		2:0		**Package C-State Limit (R/W)** Specifies the lowest processor-specific C-state code name (consuming the least power) for the package. The default is set as factory-configured package C-state limit. The following C-state code name encodings are supported: 000b: C0/C1 (no package C-state support) 001b: C2 010b: C6 (non-retention) 011b: C6 (retention) 111b: No Package C state limits. All C states supported by the processor are available.
		9:3		Reserved
		10		**I/O MWAIT Redirection Enable (R/W)**
		14:11		Reserved
		15		**CFG Lock (R/WO)**
		24:16		Reserved
		25		**C3 State Auto Demotion Enable (R/W)**
		26		**C1 State Auto Demotion Enable (R/W)**
		27		**Enable C3 Undemotion (R/W)**
		28		**Enable C1 Undemotion (R/W)**
		29		**Package C State Demotion Enable (R/W)**
		30		**Package C State UnDemotion Enable (R/W)**
		63:31		Reserved
179H	377	IA32_MCG_CAP	Thread	**Global Machine Check Capability (R/O)**
		7:0		**Count**
		8		**MCG_CTL_P**
		9		**MCG_EXT_P**
		10		**MCP_CMCI_P**
		11		**MCG_TES_P**
		15:12		Reserved.
		23:16		**MCG_EXT_CNT**
		24		**MCG_SER_P**
		25		**MCG_EM_P**
		26		**MCG_ELOG_P**
		63:27		Reserved.

Table 2-31. Additional MSRs Supported by Intel® Xeon® Processor E5 v3 Family

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
17DH	390	MSR_SMM_MCA_CAP	THREAD	**Enhanced SMM Capabilities (SMM-RO)** Reports SMM capability Enhancement. Accessible only while in SMM.
		57:0		**Reserved**
		58		**SMM_Code_Access_Chk (SMM-RO)** If set to 1 indicates that the SMM code access restriction is supported and a host-space interface available to SMM handler.
		59		**Long_Flow_Indication (SMM-RO)** If set to 1 indicates that the SMM long flow indicator is supported and a host-space interface available to SMM handler.
		63:60		Reserved
17FH	383	MSR_ERROR_CONTROL	Package	**MC Bank Error Configuration (R/W)**
		0		Reserved
		1		**MemError Log Enable (R/W)** When set, enables IMC status bank to log additional info in bits 36:32.
		63:2		Reserved.
1ADH	429	MSR_TURBO_RATIO_LIMIT	Package	**Maximum Ratio Limit of Turbo Mode** RO if MSR_PLATFORM_INFO.[28] = 0, RW if MSR_PLATFORM_INFO.[28] = 1
		7:0	Package	**Maximum Ratio Limit for 1C** Maximum turbo ratio limit of 1 core active.
		15:8	Package	**Maximum Ratio Limit for 2C** Maximum turbo ratio limit of 2 core active.
		23:16	Package	**Maximum Ratio Limit for 3C** Maximum turbo ratio limit of 3 core active.
		31:24	Package	**Maximum Ratio Limit for 4C** Maximum turbo ratio limit of 4 core active.
		39:32	Package	**Maximum Ratio Limit for 5C** Maximum turbo ratio limit of 5 core active.
		47:40	Package	**Maximum Ratio Limit for 6C** Maximum turbo ratio limit of 6 core active.
		55:48	Package	**Maximum Ratio Limit for 7C** Maximum turbo ratio limit of 7 core active.
		63:56	Package	**Maximum Ratio Limit for 8C** Maximum turbo ratio limit of 8 core active.
1AEH	430	MSR_TURBO_RATIO_LIMIT1	Package	**Maximum Ratio Limit of Turbo Mode** RO if MSR_PLATFORM_INFO.[28] = 0, RW if MSR_PLATFORM_INFO.[28] = 1

Table 2-31. Additional MSRs Supported by Intel® Xeon® Processor E5 v3 Family

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
		7:0	Package	**Maximum Ratio Limit for 9C** Maximum turbo ratio limit of 9 core active.
		15:8	Package	**Maximum Ratio Limit for 10C** Maximum turbo ratio limit of 10 core active.
		23:16	Package	**Maximum Ratio Limit for 11C** Maximum turbo ratio limit of 11 core active.
		31:24	Package	**Maximum Ratio Limit for 12C** Maximum turbo ratio limit of 12 core active.
		39:32	Package	**Maximum Ratio Limit for 13C** Maximum turbo ratio limit of 13 core active.
		47:40	Package	**Maximum Ratio Limit for 14C** Maximum turbo ratio limit of 14 core active.
		55:48	Package	**Maximum Ratio Limit for 15C** Maximum turbo ratio limit of 15 core active.
		63:56	Package	**Maximum Ratio Limit for16C** Maximum turbo ratio limit of 16 core active.
1AFH	431	MSR_TURBO_RATIO_LIMIT2	Package	**Maximum Ratio Limit of Turbo Mode** RO if MSR_PLATFORM_INFO.[28] = 0, RW if MSR_PLATFORM_INFO.[28] = 1
		7:0	Package	**Maximum Ratio Limit for 17C** Maximum turbo ratio limit of 17 core active.
		15:8	Package	**Maximum Ratio Limit for 18C** Maximum turbo ratio limit of 18 core active.
		62:16	Package	Reserved
		63	Package	**Semaphore for Turbo Ratio Limit Configuration** If 1, the processor uses override configuration[1] specified in MSR_TURBO_RATIO_LIMIT, MSR_TURBO_RATIO_LIMIT1 and MSR_TURBO_RATIO_LIMIT2. If 0, the processor uses factory-set configuration (Default).
414H	1044	IA32_MC5_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs." through Section 15.3.2.4, "IA32_MCi_MISC MSRs.". Bank MC5 reports MC error from the Intel QPI 0 module.
415H	1045	IA32_MC5_STATUS	Package	
416H	1046	IA32_MC5_ADDR	Package	
417H	1047	IA32_MC5_MISC	Package	
418H	1048	IA32_MC6_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs." through Section 15.3.2.4, "IA32_MCi_MISC MSRs.". Bank MC6 reports MC error from the integrated I/O module.
419H	1049	IA32_MC6_STATUS	Package	
41AH	1050	IA32_MC6_ADDR	Package	
41BH	1051	IA32_MC6_MISC	Package	

Table 2-31. Additional MSRs Supported by Intel® Xeon® Processor E5 v3 Family

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
41CH	1052	IA32_MC7_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs." through Section 15.3.2.4, "IA32_MCi_MISC MSRs.". Bank MC7 reports MC error from the home agent HA 0.
41DH	1053	IA32_MC7_STATUS	Package	
41EH	1054	IA32_MC7_ADDR	Package	
41FH	1055	IA32_MC7_MISC	Package	
420H	1056	IA32_MC8_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs." through Section 15.3.2.4, "IA32_MCi_MISC MSRs.". Bank MC8 reports MC error from the home agent HA 1.
421H	1057	IA32_MC8_STATUS	Package	
422H	1058	IA32_MC8_ADDR	Package	
423H	1059	IA32_MC8_MISC	Package	
424H	1060	IA32_MC9_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs." through Section 15.3.2.4, "IA32_MCi_MISC MSRs.". Banks MC9 through MC 16 report MC error from each channel of the integrated memory controllers.
425H	1061	IA32_MC9_STATUS	Package	
426H	1062	IA32_MC9_ADDR	Package	
427H	1063	IA32_MC9_MISC	Package	
428H	1064	IA32_MC10_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs." through Section 15.3.2.4, "IA32_MCi_MISC MSRs.". Banks MC9 through MC 16 report MC error from each channel of the integrated memory controllers.
429H	1065	IA32_MC10_STATUS	Package	
42AH	1066	IA32_MC10_ADDR	Package	
42BH	1067	IA32_MC10_MISC	Package	
42CH	1068	IA32_MC11_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs." through Section 15.3.2.4, "IA32_MCi_MISC MSRs.". Banks MC9 through MC 16 report MC error from each channel of the integrated memory controllers.
42DH	1069	IA32_MC11_STATUS	Package	
42EH	1070	IA32_MC11_ADDR	Package	
42FH	1071	IA32_MC11_MISC	Package	
430H	1072	IA32_MC12_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs." through Section 15.3.2.4, "IA32_MCi_MISC MSRs.". Banks MC9 through MC 16 report MC error from each channel of the integrated memory controllers.
431H	1073	IA32_MC12_STATUS	Package	
432H	1074	IA32_MC12_ADDR	Package	
433H	1075	IA32_MC12_MISC	Package	
434H	1076	IA32_MC13_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs." through Section 15.3.2.4, "IA32_MCi_MISC MSRs.". Banks MC9 through MC 16 report MC error from each channel of the integrated memory controllers.
435H	1077	IA32_MC13_STATUS	Package	
436H	1078	IA32_MC13_ADDR	Package	
437H	1079	IA32_MC13_MISC	Package	
438H	1080	IA32_MC14_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs." through Section 15.3.2.4, "IA32_MCi_MISC MSRs.". Banks MC9 through MC 16 report MC error from each channel of the integrated memory controllers.
439H	1081	IA32_MC14_STATUS	Package	
43AH	1082	IA32_MC14_ADDR	Package	
43BH	1083	IA32_MC14_MISC	Package	
43CH	1084	IA32_MC15_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs." through Section 15.3.2.4, "IA32_MCi_MISC MSRs.". Banks MC9 through MC 16 report MC error from each channel of the integrated memory controllers.
43DH	1085	IA32_MC15_STATUS	Package	
43EH	1086	IA32_MC15_ADDR	Package	
43FH	1087	IA32_MC15_MISC	Package	

Table 2-31. Additional MSRs Supported by Intel® Xeon® Processor E5 v3 Family

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
440H	1088	IA32_MC16_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs." through Section 15.3.2.4, "IA32_MCi_MISC MSRs.". Banks MC9 through MC 16 report MC error from each channel of the integrated memory controllers.
441H	1089	IA32_MC16_STATUS	Package	
442H	1090	IA32_MC16_ADDR	Package	
443H	1091	IA32_MC16_MISC	Package	
444H	1092	IA32_MC17_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs." through Section 15.3.2.4, "IA32_MCi_MISC MSRs.". Bank MC17 reports MC error from the following pair of CBo/L3 Slices (if the pair is present): CBo0, CBo3, CBo6, CBo9, CBo12, CBo15.
445H	1093	IA32_MC17_STATUS	Package	
446H	1094	IA32_MC17_ADDR	Package	
447H	1095	IA32_MC17_MISC	Package	
448H	1096	IA32_MC18_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs." through Section 15.3.2.4, "IA32_MCi_MISC MSRs.". Bank MC18 reports MC error from the following pair of CBo/L3 Slices (if the pair is present): CBo1, CBo4, CBo7, CBo10, CBo13, CBo16.
449H	1097	IA32_MC18_STATUS	Package	
44AH	1098	IA32_MC18_ADDR	Package	
44BH	1099	IA32_MC18_MISC	Package	
44CH	1100	IA32_MC19_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs." through Section 15.3.2.4, "IA32_MCi_MISC MSRs.". Bank MC19 reports MC error from the following pair of CBo/L3 Slices (if the pair is present): CBo2, CBo5, CBo8, CBo11, CBo14, CBo17.
44DH	1101	IA32_MC19_STATUS	Package	
44EH	1102	IA32_MC19_ADDR	Package	
44FH	1103	IA32_MC19_MISC	Package	
450H	1104	IA32_MC20_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs." through Section 15.3.2.4, "IA32_MCi_MISC MSRs.". Bank MC20 reports MC error from the Intel QPI 1 module.
451H	1105	IA32_MC20_STATUS	Package	
452H	1106	IA32_MC20_ADDR	Package	
453H	1107	IA32_MC20_MISC	Package	
454H	1108	IA32_MC21_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs." through Section 15.3.2.4, "IA32_MCi_MISC MSRs.". Bank MC21 reports MC error from the Intel QPI 2 module.
455H	1109	IA32_MC21_STATUS	Package	
456H	1110	IA32_MC21_ADDR	Package	
457H	1111	IA32_MC21_MISC	Package	
606H	1542	MSR_RAPL_POWER_UNIT	Package	**Unit Multipliers used in RAPL Interfaces (R/O)**
		3:0	Package	**Power Units** See Section 14.9.1, "RAPL Interfaces."
		7:4	Package	**Reserved**
		12:8	Package	**Energy Status Units** Energy related information (in Joules) is based on the multiplier, 1/2^ESU; where ESU is an unsigned integer represented by bits 12:8. Default value is 0EH (or 61 micro-joules)
		15:13	Package	**Reserved**
		19:16	Package	**Time Units** See Section 14.9.1, "RAPL Interfaces."
		63:20		Reserved
618H	1560	MSR_DRAM_POWER_LIMIT	Package	**DRAM RAPL Power Limit Control (R/W)** See Section 14.9.5, "DRAM RAPL Domain."

Table 2-31. Additional MSRs Supported by Intel® Xeon® Processor E5 v3 Family

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
619H	1561	MSR_DRAM_ENERGY_ STATUS	Package	**DRAM Energy Status (R/O)** Energy Consumed by DRAM devices.
		31:0		Energy in 15.3 micro-joules. Requires BIOS configuration to enable DRAM RAPL mode 0 (Direct VR).
		63:32		Reserved
61BH	1563	MSR_DRAM_PERF_STATUS	Package	**DRAM Performance Throttling Status (R/O)** See Section 14.9.5, "DRAM RAPL Domain."
61CH	1564	MSR_DRAM_POWER_INFO	Package	**DRAM RAPL Parameters (R/W)** See Section 14.9.5, "DRAM RAPL Domain."
61EH	1566	MSR_PCIE_PLL_RATIO	Package	**Configuration of PCIE PLL Relative to BCLK(R/W)**
		1:0	Package	**PCIE Ratio (R/W)** 00b: Use 5:5 mapping for100MHz operation (default) 01b: Use 5:4 mapping for125MHz operation 10b: Use 5:3 mapping for166MHz operation 11b: Use 5:2 mapping for250MHz operation
		2	Package	**LPLL Select (R/W)** if 1, use configured setting of PCIE Ratio
		3	Package	**LONG RESET (R/W)** if 1, wait additional time-out before re-locking Gen2/Gen3 PLLs.
		63:4		Reserved
620H	1568	MSR UNCORE_RATIO_LIMIT	Package	Uncore Ratio Limit (R/W) Out of reset, the min_ratio and max_ratio fields represent the widest possible range of uncore frequencies. Writing to these fields allows software to control the minimum and the maximum frequency that hardware will select.
		63:15		Reserved.
		14:8		MIN_RATIO Writing to this field controls the minimum possible ratio of the LLC/Ring.
		7		Reserved.
		6:0		MAX_RATIO This field is used to limit the max ratio of the LLC/Ring.
639H	1593	MSR_PP0_ENERGY_STATUS	Package	**Reserved (R/O)** Reads return 0
690H	1680	MSR_CORE_PERF_LIMIT_RE ASONS	Package	**Indicator of Frequency Clipping in Processor Cores (R/W)** **(frequency refers to processor core frequency)**
		0		**PROCHOT Status (R0)** When set, processor core frequency is reduced below the operating system request due to assertion of external PROCHOT.

Table 2-31. Additional MSRs Supported by Intel® Xeon® Processor E5 v3 Family

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
		1		**Thermal Status (R0)** When set, frequency is reduced below the operating system request due to a thermal event.
		2		**Power Budget Management Status (R0)** When set, frequency is reduced below the operating system request due to PBM limit
		3		**Platform Configuration Services Status (R0)** When set, frequency is reduced below the operating system request due to PCS limit
		4		Reserved.
		5		**Autonomous Utilization-Based Frequency Control Status (R0)** When set, frequency is reduced below the operating system request because the processor has detected that utilization is low
		6		**VR Therm Alert Status (R0)** When set, frequency is reduced below the operating system request due to a thermal alert from the Voltage Regulator.
		7		Reserved.
		8		**Electrical Design Point Status (R0)** When set, frequency is reduced below the operating system request due to electrical design point constraints (e.g. maximum electrical current consumption).
		9		Reserved.
		10		**Multi-Core Turbo Status (R0)** When set, frequency is reduced below the operating system request due to Multi-Core Turbo limits
		12:11		Reserved.
		13		**Core Frequency P1 Status (R0)** When set, frequency is reduced below max non-turbo P1
		14		**Core Max n-core Turbo Frequency Limiting Status (R0)** When set, frequency is reduced below max n-core turbo frequency
		15		**Core Frequency Limiting Status (R0)** When set, frequency is reduced below the operating system request.
		16		**PROCHOT Log** When set, indicates that the PROCHOT Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.
		17		**Thermal Log** When set, indicates that the Thermal Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.

Table 2-31. Additional MSRs Supported by Intel® Xeon® Processor E5 v3 Family

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
		18		**Power Budget Management Log** When set, indicates that the PBM Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.
		19		**Platform Configuration Services Log** When set, indicates that the PCS Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.
		20		Reserved.
		21		**Autonomous Utilization-Based Frequency Control Log** When set, indicates that the AUBFC Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.
		22		**VR Therm Alert Log** When set, indicates that the VR Therm Alert Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.
		23		Reserved.
		24		**Electrical Design Point Log** When set, indicates that the EDP Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.
		25		Reserved.
		26		**Multi-Core Turbo Log** When set, indicates that the Multi-Core Turbo Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.
		28:27		Reserved.
		29		**Core Frequency P1 Log** When set, indicates that the Core Frequency P1 Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.
		30		**Core Max n-core Turbo Frequency Limiting Log** When set, indicates that the Core Max n-core Turbo Frequency Limiting Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.
		31		**Core Frequency Limiting Log** When set, indicates that the Core Frequency Limiting Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.
		63:32		Reserved.

Table 2-31. Additional MSRs Supported by Intel® Xeon® Processor E5 v3 Family

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
C8DH	3213	IA32_QM_EVTSEL	THREAD	**Monitoring Event Select Register (R/W)** if CPUID.(EAX=07H, ECX=0):EBX.RDT-M[bit 12] = 1
		7:0		**EventID (RW)** **Event encoding:** **0x0: no monitoring** **0x1: L3 occupancy monitoring** **all other encoding reserved.**
		31:8		Reserved.
		41:32		**RMID (RW)**
		63:42		Reserved.
C8EH	3214	IA32_QM_CTR	THREAD	**Monitoring Counter Register (R/O).** if CPUID.(EAX=07H, ECX=0):EBX.RDT-M[bit 12] = 1
		61:0		**Resource Monitored Data**
		62		**Unavailable**: If 1, indicates data for this RMID is not available or not monitored for this resource or RMID.
		63		**Error:** If 1, indicates and unsupported RMID or event type was written to IA32_PQR_QM_EVTSEL.
C8FH	3215	IA32_PQR_ASSOC	THREAD	**Resource Association Register (R/W).**
		9:0		**RMID**
		63: 10		**Reserved**
See Table 2-19, Table 2-28 for other MSR definitions applicable to processors with CPUID signature 06_3FH.				

NOTES:

1. An override configuration lower than the factory-set configuration is always supported. An override configuration higher than the factory-set configuration is dependent on features specific to the processor and the platform.

2.13.1 Additional Uncore PMU MSRs in the Intel® Xeon® Processor E5 v3 Family

Intel Xeon Processor E5 v3 and E7 v3 family are based on the Haswell-E microarchitecture. The MSR-based uncore PMU interfaces are listed in Table 2-32. For complete detail of the uncore PMU, refer to Intel Xeon Processor E5 v3 Product Family Uncore Performance Monitoring Guide. These processors have a CPUID signature with DisplayFamily_DisplayModel of 06_3FH.

Table 2-32. Uncore PMU MSRs in Intel® Xeon® Processor E5 v3 Family

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
700H		MSR_PMON_GLOBAL_CTL	Package	Uncore perfmon per-socket global control.
701H		MSR_PMON_GLOBAL_STATUS	Package	Uncore perfmon per-socket global status.
702H		MSR_PMON_GLOBAL_CONFIG	Package	Uncore perfmon per-socket global configuration.
703H		MSR_U_PMON_UCLK_FIXED_CTL	Package	Uncore U-box UCLK fixed counter control
704H		MSR_U_PMON_UCLK_FIXED_CTR	Package	Uncore U-box UCLK fixed counter

Table 2-32. Uncore PMU MSRs in Intel® Xeon® Processor E5 v3 Family (Contd.)

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
705H		MSR_U_PMON_EVNTSEL0	Package	Uncore U-box perfmon event select for U-box counter 0.
706H		MSR_U_PMON_EVNTSEL1	Package	Uncore U-box perfmon event select for U-box counter 1.
708H		MSR_U_PMON_BOX_STATUS	Package	Uncore U-box perfmon U-box wide status.
709H		MSR_U_PMON_CTR0	Package	Uncore U-box perfmon counter 0
70AH		MSR_U_PMON_CTR1	Package	Uncore U-box perfmon counter 1
710H		MSR_PCU_PMON_BOX_CTL	Package	Uncore PCU perfmon for PCU-box-wide control
711H		MSR_PCU_PMON_EVNTSEL0	Package	Uncore PCU perfmon event select for PCU counter 0.
712H		MSR_PCU_PMON_EVNTSEL1	Package	Uncore PCU perfmon event select for PCU counter 1.
713H		MSR_PCU_PMON_EVNTSEL2	Package	Uncore PCU perfmon event select for PCU counter 2.
714H		MSR_PCU_PMON_EVNTSEL3	Package	Uncore PCU perfmon event select for PCU counter 3.
715H		MSR_PCU_PMON_BOX_FILTER	Package	Uncore PCU perfmon box-wide filter.
716H		MSR_PCU_PMON_BOX_STATUS	Package	Uncore PCU perfmon box wide status.
717H		MSR_PCU_PMON_CTR0	Package	Uncore PCU perfmon counter 0.
718H		MSR_PCU_PMON_CTR1	Package	Uncore PCU perfmon counter 1.
719H		MSR_PCU_PMON_CTR2	Package	Uncore PCU perfmon counter 2.
71AH		MSR_PCU_PMON_CTR3	Package	Uncore PCU perfmon counter 3.
720H		MSR_S0_PMON_BOX_CTL	Package	Uncore SBo 0 perfmon for SBo 0 box-wide control
721H		MSR_S0_PMON_EVNTSEL0	Package	Uncore SBo 0 perfmon event select for SBo 0 counter 0.
722H		MSR_S0_PMON_EVNTSEL1	Package	Uncore SBo 0 perfmon event select for SBo 0 counter 1.
723H		MSR_S0_PMON_EVNTSEL2	Package	Uncore SBo 0 perfmon event select for SBo 0 counter 2.
724H		MSR_S0_PMON_EVNTSEL3	Package	Uncore SBo 0 perfmon event select for SBo 0 counter 3.
725H		MSR_S0_PMON_BOX_FILTER	Package	Uncore SBo 0 perfmon box-wide filter.
726H		MSR_S0_PMON_CTR0	Package	Uncore SBo 0 perfmon counter 0.
727H		MSR_S0_PMON_CTR1	Package	Uncore SBo 0 perfmon counter 1.
728H		MSR_S0_PMON_CTR2	Package	Uncore SBo 0 perfmon counter 2.
729H		MSR_S0_PMON_CTR3	Package	Uncore SBo 0 perfmon counter 3.
72AH		MSR_S1_PMON_BOX_CTL	Package	Uncore SBo 1 perfmon for SBo 1 box-wide control
72BH		MSR_S1_PMON_EVNTSEL0	Package	Uncore SBo 1 perfmon event select for SBo 1 counter 0.
72CH		MSR_S1_PMON_EVNTSEL1	Package	Uncore SBo 1 perfmon event select for SBo 1 counter 1.
72DH		MSR_S1_PMON_EVNTSEL2	Package	Uncore SBo 1 perfmon event select for SBo 1 counter 2.
72EH		MSR_S1_PMON_EVNTSEL3	Package	Uncore SBo 1 perfmon event select for SBo 1 counter 3.
72FH		MSR_S1_PMON_BOX_FILTER	Package	Uncore SBo 1 perfmon box-wide filter.
730H		MSR_S1_PMON_CTR0	Package	Uncore SBo 1 perfmon counter 0.
731H		MSR_S1_PMON_CTR1	Package	Uncore SBo 1 perfmon counter 1.
732H		MSR_S1_PMON_CTR2	Package	Uncore SBo 1 perfmon counter 2.
733H		MSR_S1_PMON_CTR3	Package	Uncore SBo 1 perfmon counter 3.
734H		MSR_S2_PMON_BOX_CTL	Package	Uncore SBo 2 perfmon for SBo 2 box-wide control

Table 2-32. Uncore PMU MSRs in Intel® Xeon® Processor E5 v3 Family (Contd.)

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
735H		MSR_S2_PMON_EVNTSEL0	Package	Uncore SBo 2 perfmon event select for SBo 2 counter 0.
736H		MSR_S2_PMON_EVNTSEL1	Package	Uncore SBo 2 perfmon event select for SBo 2 counter 1.
737H		MSR_S2_PMON_EVNTSEL2	Package	Uncore SBo 2 perfmon event select for SBo 2 counter 2.
738H		MSR_S2_PMON_EVNTSEL3	Package	Uncore SBo 2 perfmon event select for SBo 2 counter 3.
739H		MSR_S2_PMON_BOX_FILTER	Package	Uncore SBo 2 perfmon box-wide filter.
73AH		MSR_S2_PMON_CTR0	Package	Uncore SBo 2 perfmon counter 0.
73BH		MSR_S2_PMON_CTR1	Package	Uncore SBo 2 perfmon counter 1.
73CH		MSR_S2_PMON_CTR2	Package	Uncore SBo 2 perfmon counter 2.
73DH		MSR_S2_PMON_CTR3	Package	Uncore SBo 2 perfmon counter 3.
73EH		MSR_S3_PMON_BOX_CTL	Package	Uncore SBo 3 perfmon for SBo 3 box-wide control
73FH		MSR_S3_PMON_EVNTSEL0	Package	Uncore SBo 3 perfmon event select for SBo 3 counter 0.
740H		MSR_S3_PMON_EVNTSEL1	Package	Uncore SBo 3 perfmon event select for SBo 3 counter 1.
741H		MSR_S3_PMON_EVNTSEL2	Package	Uncore SBo 3 perfmon event select for SBo 3 counter 2.
742H		MSR_S3_PMON_EVNTSEL3	Package	Uncore SBo 3 perfmon event select for SBo 3 counter 3.
743H		MSR_S3_PMON_BOX_FILTER	Package	Uncore SBo 3 perfmon box-wide filter.
744H		MSR_S3_PMON_CTR0	Package	Uncore SBo 3 perfmon counter 0.
745H		MSR_S3_PMON_CTR1	Package	Uncore SBo 3 perfmon counter 1.
746H		MSR_S3_PMON_CTR2	Package	Uncore SBo 3 perfmon counter 2.
747H		MSR_S3_PMON_CTR3	Package	Uncore SBo 3 perfmon counter 3.
E00H		MSR_C0_PMON_BOX_CTL	Package	Uncore C-box 0 perfmon for box-wide control
E01H		MSR_C0_PMON_EVNTSEL0	Package	Uncore C-box 0 perfmon event select for C-box 0 counter 0.
E02H		MSR_C0_PMON_EVNTSEL1	Package	Uncore C-box 0 perfmon event select for C-box 0 counter 1.
E03H		MSR_C0_PMON_EVNTSEL2	Package	Uncore C-box 0 perfmon event select for C-box 0 counter 2.
E04H		MSR_C0_PMON_EVNTSEL3	Package	Uncore C-box 0 perfmon event select for C-box 0 counter 3.
E05H		MSR_C0_PMON_BOX_FILTER0	Package	Uncore C-box 0 perfmon box wide filter 0.
E06H		MSR_C0_PMON_BOX_FILTER1	Package	Uncore C-box 0 perfmon box wide filter 1.
E07H		MSR_C0_PMON_BOX_STATUS	Package	Uncore C-box 0 perfmon box wide status.
E08H		MSR_C0_PMON_CTR0	Package	Uncore C-box 0 perfmon counter 0.
E09H		MSR_C0_PMON_CTR1	Package	Uncore C-box 0 perfmon counter 1.
E0AH		MSR_C0_PMON_CTR2	Package	Uncore C-box 0 perfmon counter 2.
E0BH		MSR_C0_PMON_CTR3	Package	Uncore C-box 0 perfmon counter 3.
E10H		MSR_C1_PMON_BOX_CTL	Package	Uncore C-box 1 perfmon for box-wide control
E11H		MSR_C1_PMON_EVNTSEL0	Package	Uncore C-box 1 perfmon event select for C-box 1 counter 0.
E12H		MSR_C1_PMON_EVNTSEL1	Package	Uncore C-box 1 perfmon event select for C-box 1 counter 1.
E13H		MSR_C1_PMON_EVNTSEL2	Package	Uncore C-box 1 perfmon event select for C-box 1 counter 2.
E14H		MSR_C1_PMON_EVNTSEL3	Package	Uncore C-box 1 perfmon event select for C-box 1 counter 3.
E15H		MSR_C1_PMON_BOX_FILTER0	Package	Uncore C-box 1 perfmon box wide filter 0.

Table 2-32. Uncore PMU MSRs in Intel® Xeon® Processor E5 v3 Family (Contd.)

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
E16H		MSR_C1_PMON_BOX_FILTER1	Package	Uncore C-box 1 perfmon box wide filter1.
E17H		MSR_C1_PMON_BOX_STATUS	Package	Uncore C-box 1 perfmon box wide status.
E18H		MSR_C1_PMON_CTR0	Package	Uncore C-box 1 perfmon counter 0.
E19H		MSR_C1_PMON_CTR1	Package	Uncore C-box 1 perfmon counter 1.
E1AH		MSR_C1_PMON_CTR2	Package	Uncore C-box 1 perfmon counter 2.
E1BH		MSR_C1_PMON_CTR3	Package	Uncore C-box 1 perfmon counter 3.
E20H		MSR_C2_PMON_BOX_CTL	Package	Uncore C-box 2 perfmon for box-wide control
E21H		MSR_C2_PMON_EVNTSEL0	Package	Uncore C-box 2 perfmon event select for C-box 2 counter 0.
E22H		MSR_C2_PMON_EVNTSEL1	Package	Uncore C-box 2 perfmon event select for C-box 2 counter 1.
E23H		MSR_C2_PMON_EVNTSEL2	Package	Uncore C-box 2 perfmon event select for C-box 2 counter 2.
E24H		MSR_C2_PMON_EVNTSEL3	Package	Uncore C-box 2 perfmon event select for C-box 2 counter 3.
E25H		MSR_C2_PMON_BOX_FILTER0	Package	Uncore C-box 2 perfmon box wide filter 0.
E26H		MSR_C2_PMON_BOX_FILTER1	Package	Uncore C-box 2 perfmon box wide filter1.
E27H		MSR_C2_PMON_BOX_STATUS	Package	Uncore C-box 2 perfmon box wide status.
E28H		MSR_C2_PMON_CTR0	Package	Uncore C-box 2 perfmon counter 0.
E29H		MSR_C2_PMON_CTR1	Package	Uncore C-box 2 perfmon counter 1.
E2AH		MSR_C2_PMON_CTR2	Package	Uncore C-box 2 perfmon counter 2.
E2BH		MSR_C2_PMON_CTR3	Package	Uncore C-box 2 perfmon counter 3.
E30H		MSR_C3_PMON_BOX_CTL	Package	Uncore C-box 3 perfmon for box-wide control
E31H		MSR_C3_PMON_EVNTSEL0	Package	Uncore C-box 3 perfmon event select for C-box 3 counter 0.
E32H		MSR_C3_PMON_EVNTSEL1	Package	Uncore C-box 3 perfmon event select for C-box 3 counter 1.
E33H		MSR_C3_PMON_EVNTSEL2	Package	Uncore C-box 3 perfmon event select for C-box 3 counter 2.
E34H		MSR_C3_PMON_EVNTSEL3	Package	Uncore C-box 3 perfmon event select for C-box 3 counter 3.
E35H		MSR_C3_PMON_BOX_FILTER0	Package	Uncore C-box 3 perfmon box wide filter 0.
E36H		MSR_C3_PMON_BOX_FILTER1	Package	Uncore C-box 3 perfmon box wide filter1.
E37H		MSR_C3_PMON_BOX_STATUS	Package	Uncore C-box 3 perfmon box wide status.
E38H		MSR_C3_PMON_CTR0	Package	Uncore C-box 3 perfmon counter 0.
E39H		MSR_C3_PMON_CTR1	Package	Uncore C-box 3 perfmon counter 1.
E3AH		MSR_C3_PMON_CTR2	Package	Uncore C-box 3 perfmon counter 2.
E3BH		MSR_C3_PMON_CTR3	Package	Uncore C-box 3 perfmon counter 3.
E40H		MSR_C4_PMON_BOX_CTL	Package	Uncore C-box 4 perfmon for box-wide control
E41H		MSR_C4_PMON_EVNTSEL0	Package	Uncore C-box 4 perfmon event select for C-box 4 counter 0.
E42H		MSR_C4_PMON_EVNTSEL1	Package	Uncore C-box 4 perfmon event select for C-box 4 counter 1.
E43H		MSR_C4_PMON_EVNTSEL2	Package	Uncore C-box 4 perfmon event select for C-box 4 counter 2.
E44H		MSR_C4_PMON_EVNTSEL3	Package	Uncore C-box 4 perfmon event select for C-box 4 counter 3.
E45H		MSR_C4_PMON_BOX_FILTER0	Package	Uncore C-box 4 perfmon box wide filter 0.
E46H		MSR_C4_PMON_BOX_FILTER1	Package	Uncore C-box 4 perfmon box wide filter1.

Table 2-32. Uncore PMU MSRs in Intel® Xeon® Processor E5 v3 Family (Contd.)

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
E47H		MSR_C4_PMON_BOX_STATUS	Package	Uncore C-box 4 perfmon box wide status.
E48H		MSR_C4_PMON_CTR0	Package	Uncore C-box 4 perfmon counter 0.
E49H		MSR_C4_PMON_CTR1	Package	Uncore C-box 4 perfmon counter 1.
E4AH		MSR_C4_PMON_CTR2	Package	Uncore C-box 4 perfmon counter 2.
E4BH		MSR_C4_PMON_CTR3	Package	Uncore C-box 4 perfmon counter 3.
E50H		MSR_C5_PMON_BOX_CTL	Package	Uncore C-box 5 perfmon for box-wide control
E51H		MSR_C5_PMON_EVNTSEL0	Package	Uncore C-box 5 perfmon event select for C-box 5 counter 0.
E52H		MSR_C5_PMON_EVNTSEL1	Package	Uncore C-box 5 perfmon event select for C-box 5 counter 1.
E53H		MSR_C5_PMON_EVNTSEL2	Package	Uncore C-box 5 perfmon event select for C-box 5 counter 2.
E54H		MSR_C5_PMON_EVNTSEL3	Package	Uncore C-box 5 perfmon event select for C-box 5 counter 3.
E55H		MSR_C5_PMON_BOX_FILTER0	Package	Uncore C-box 5 perfmon box wide filter 0.
E56H		MSR_C5_PMON_BOX_FILTER1	Package	Uncore C-box 5 perfmon box wide filter1.
E57H		MSR_C5_PMON_BOX_STATUS	Package	Uncore C-box 5 perfmon box wide status.
E58H		MSR_C5_PMON_CTR0	Package	Uncore C-box 5 perfmon counter 0.
E59H		MSR_C5_PMON_CTR1	Package	Uncore C-box 5 perfmon counter 1.
E5AH		MSR_C5_PMON_CTR2	Package	Uncore C-box 5 perfmon counter 2.
E5BH		MSR_C5_PMON_CTR3	Package	Uncore C-box 5 perfmon counter 3.
E60H		MSR_C6_PMON_BOX_CTL	Package	Uncore C-box 6 perfmon for box-wide control
E61H		MSR_C6_PMON_EVNTSEL0	Package	Uncore C-box 6 perfmon event select for C-box 6 counter 0.
E62H		MSR_C6_PMON_EVNTSEL1	Package	Uncore C-box 6 perfmon event select for C-box 6 counter 1.
E63H		MSR_C6_PMON_EVNTSEL2	Package	Uncore C-box 6 perfmon event select for C-box 6 counter 2.
E64H		MSR_C6_PMON_EVNTSEL3	Package	Uncore C-box 6 perfmon event select for C-box 6 counter 3.
E65H		MSR_C6_PMON_BOX_FILTER0	Package	Uncore C-box 6 perfmon box wide filter 0.
E66H		MSR_C6_PMON_BOX_FILTER1	Package	Uncore C-box 6 perfmon box wide filter1.
E67H		MSR_C6_PMON_BOX_STATUS	Package	Uncore C-box 6 perfmon box wide status.
E68H		MSR_C6_PMON_CTR0	Package	Uncore C-box 6 perfmon counter 0.
E69H		MSR_C6_PMON_CTR1	Package	Uncore C-box 6 perfmon counter 1.
E6AH		MSR_C6_PMON_CTR2	Package	Uncore C-box 6 perfmon counter 2.
E6BH		MSR_C6_PMON_CTR3	Package	Uncore C-box 6 perfmon counter 3.
E70H		MSR_C7_PMON_BOX_CTL	Package	Uncore C-box 7 perfmon for box-wide control.
E71H		MSR_C7_PMON_EVNTSEL0	Package	Uncore C-box 7 perfmon event select for C-box 7 counter 0.
E72H		MSR_C7_PMON_EVNTSEL1	Package	Uncore C-box 7 perfmon event select for C-box 7 counter 1.
E73H		MSR_C7_PMON_EVNTSEL2	Package	Uncore C-box 7 perfmon event select for C-box 7 counter 2.
E74H		MSR_C7_PMON_EVNTSEL3	Package	Uncore C-box 7 perfmon event select for C-box 7 counter 3.
E75H		MSR_C7_PMON_BOX_FILTER0	Package	Uncore C-box 7 perfmon box wide filter 0.
E76H		MSR_C7_PMON_BOX_FILTER1	Package	Uncore C-box 7 perfmon box wide filter1.
E77H		MSR_C7_PMON_BOX_STATUS	Package	Uncore C-box 7 perfmon box wide status.

Table 2-32. Uncore PMU MSRs in Intel® Xeon® Processor E5 v3 Family (Contd.)

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
E78H		MSR_C7_PMON_CTR0	Package	Uncore C-box 7 perfmon counter 0.
E79H		MSR_C7_PMON_CTR1	Package	Uncore C-box 7 perfmon counter 1.
E7AH		MSR_C7_PMON_CTR2	Package	Uncore C-box 7 perfmon counter 2.
E7BH		MSR_C7_PMON_CTR3	Package	Uncore C-box 7 perfmon counter 3.
E80H		MSR_C8_PMON_BOX_CTL	Package	Uncore C-box 8 perfmon local box wide control.
E81H		MSR_C8_PMON_EVNTSEL0	Package	Uncore C-box 8 perfmon event select for C-box 8 counter 0.
E82H		MSR_C8_PMON_EVNTSEL1	Package	Uncore C-box 8 perfmon event select for C-box 8 counter 1.
E83H		MSR_C8_PMON_EVNTSEL2	Package	Uncore C-box 8 perfmon event select for C-box 8 counter 2.
E84H		MSR_C8_PMON_EVNTSEL3	Package	Uncore C-box 8 perfmon event select for C-box 8 counter 3.
E85H		MSR_C8_PMON_BOX_FILTER0	Package	Uncore C-box 8 perfmon box wide filter0.
E86H		MSR_C8_PMON_BOX_FILTER1	Package	Uncore C-box 8 perfmon box wide filter1.
E87H		MSR_C8_PMON_BOX_STATUS	Package	Uncore C-box 8 perfmon box wide status.
E88H		MSR_C8_PMON_CTR0	Package	Uncore C-box 8 perfmon counter 0.
E89H		MSR_C8_PMON_CTR1	Package	Uncore C-box 8 perfmon counter 1.
E8AH		MSR_C8_PMON_CTR2	Package	Uncore C-box 8 perfmon counter 2.
E8BH		MSR_C8_PMON_CTR3	Package	Uncore C-box 8 perfmon counter 3.
E90H		MSR_C9_PMON_BOX_CTL	Package	Uncore C-box 9 perfmon local box wide control.
E91H		MSR_C9_PMON_EVNTSEL0	Package	Uncore C-box 9 perfmon event select for C-box 9 counter 0.
E92H		MSR_C9_PMON_EVNTSEL1	Package	Uncore C-box 9 perfmon event select for C-box 9 counter 1.
E93H		MSR_C9_PMON_EVNTSEL2	Package	Uncore C-box 9 perfmon event select for C-box 9 counter 2.
E94H		MSR_C9_PMON_EVNTSEL3	Package	Uncore C-box 9 perfmon event select for C-box 9 counter 3.
E95H		MSR_C9_PMON_BOX_FILTER0	Package	Uncore C-box 9 perfmon box wide filter0.
E96H		MSR_C9_PMON_BOX_FILTER1	Package	Uncore C-box 9 perfmon box wide filter1.
E97H		MSR_C9_PMON_BOX_STATUS	Package	Uncore C-box 9 perfmon box wide status.
E98H		MSR_C9_PMON_CTR0	Package	Uncore C-box 9 perfmon counter 0.
E99H		MSR_C9_PMON_CTR1	Package	Uncore C-box 9 perfmon counter 1.
E9AH		MSR_C9_PMON_CTR2	Package	Uncore C-box 9 perfmon counter 2.
E9BH		MSR_C9_PMON_CTR3	Package	Uncore C-box 9 perfmon counter 3.
EA0H		MSR_C10_PMON_BOX_CTL	Package	Uncore C-box 10 perfmon local box wide control.
EA1H		MSR_C10_PMON_EVNTSEL0	Package	Uncore C-box 10 perfmon event select for C-box 10 counter 0.
EA2H		MSR_C10_PMON_EVNTSEL1	Package	Uncore C-box 10 perfmon event select for C-box 10 counter 1.
EA3H		MSR_C10_PMON_EVNTSEL2	Package	Uncore C-box 10 perfmon event select for C-box 10 counter 2.
EA4H		MSR_C10_PMON_EVNTSEL3	Package	Uncore C-box 10 perfmon event select for C-box 10 counter 3.
EA5H		MSR_C10_PMON_BOX_FILTER0	Package	Uncore C-box 10 perfmon box wide filter0.
EA6H		MSR_C10_PMON_BOX_FILTER1	Package	Uncore C-box 10 perfmon box wide filter1.
EA7H		MSR_C10_PMON_BOX_STATUS	Package	Uncore C-box 10 perfmon box wide status.
EA8H		MSR_C10_PMON_CTR0	Package	Uncore C-box 10 perfmon counter 0.

Table 2-32. Uncore PMU MSRs in Intel® Xeon® Processor E5 v3 Family (Contd.)

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
EA9H		MSR_C10_PMON_CTR1	Package	Uncore C-box 10 perfmon counter 1.
EAAH		MSR_C10_PMON_CTR2	Package	Uncore C-box 10 perfmon counter 2.
EABH		MSR_C10_PMON_CTR3	Package	Uncore C-box 10 perfmon counter 3.
EB0H		MSR_C11_PMON_BOX_CTL	Package	Uncore C-box 11 perfmon local box wide control.
EB1H		MSR_C11_PMON_EVNTSEL0	Package	Uncore C-box 11 perfmon event select for C-box 11 counter 0.
EB2H		MSR_C11_PMON_EVNTSEL1	Package	Uncore C-box 11 perfmon event select for C-box 11 counter 1.
EB3H		MSR_C11_PMON_EVNTSEL2	Package	Uncore C-box 11 perfmon event select for C-box 11 counter 2.
EB4H		MSR_C11_PMON_EVNTSEL3	Package	Uncore C-box 11 perfmon event select for C-box 11 counter 3.
EB5H		MSR_C11_PMON_BOX_FILTER0	Package	Uncore C-box 11 perfmon box wide filter0.
EB6H		MSR_C11_PMON_BOX_FILTER1	Package	Uncore C-box 11 perfmon box wide filter1.
EB7H		MSR_C11_PMON_BOX_STATUS	Package	Uncore C-box 11 perfmon box wide status.
EB8H		MSR_C11_PMON_CTR0	Package	Uncore C-box 11 perfmon counter 0.
EB9H		MSR_C11_PMON_CTR1	Package	Uncore C-box 11 perfmon counter 1.
EBAH		MSR_C11_PMON_CTR2	Package	Uncore C-box 11 perfmon counter 2.
EBBH		MSR_C11_PMON_CTR3	Package	Uncore C-box 11 perfmon counter 3.
EC0H		MSR_C12_PMON_BOX_CTL	Package	Uncore C-box 12 perfmon local box wide control.
EC1H		MSR_C12_PMON_EVNTSEL0	Package	Uncore C-box 12 perfmon event select for C-box 12 counter 0.
EC2H		MSR_C12_PMON_EVNTSEL1	Package	Uncore C-box 12 perfmon event select for C-box 12 counter 1.
EC3H		MSR_C12_PMON_EVNTSEL2	Package	Uncore C-box 12 perfmon event select for C-box 12 counter 2.
EC4H		MSR_C12_PMON_EVNTSEL3	Package	Uncore C-box 12 perfmon event select for C-box 12 counter 3.
EC5H		MSR_C12_PMON_BOX_FILTER0	Package	Uncore C-box 12 perfmon box wide filter0.
EC6H		MSR_C12_PMON_BOX_FILTER1	Package	Uncore C-box 12 perfmon box wide filter1.
EC7H		MSR_C12_PMON_BOX_STATUS	Package	Uncore C-box 12 perfmon box wide status.
EC8H		MSR_C12_PMON_CTR0	Package	Uncore C-box 12 perfmon counter 0.
EC9H		MSR_C12_PMON_CTR1	Package	Uncore C-box 12 perfmon counter 1.
ECAH		MSR_C12_PMON_CTR2	Package	Uncore C-box 12 perfmon counter 2.
ECBH		MSR_C12_PMON_CTR3	Package	Uncore C-box 12 perfmon counter 3.
ED0H		MSR_C13_PMON_BOX_CTL	Package	Uncore C-box 13 perfmon local box wide control.
ED1H		MSR_C13_PMON_EVNTSEL0	Package	Uncore C-box 13 perfmon event select for C-box 13 counter 0.
ED2H		MSR_C13_PMON_EVNTSEL1	Package	Uncore C-box 13 perfmon event select for C-box 13 counter 1.
ED3H		MSR_C13_PMON_EVNTSEL2	Package	Uncore C-box 13 perfmon event select for C-box 13 counter 2.
ED4H		MSR_C13_PMON_EVNTSEL3	Package	Uncore C-box 13 perfmon event select for C-box 13 counter 3.
ED5H		MSR_C13_PMON_BOX_FILTER0	Package	Uncore C-box 13 perfmon box wide filter0.
ED6H		MSR_C13_PMON_BOX_FILTER1	Package	Uncore C-box 13 perfmon box wide filter1.
ED7H		MSR_C13_PMON_BOX_STATUS	Package	Uncore C-box 13 perfmon box wide status.
ED8H		MSR_C13_PMON_CTR0	Package	Uncore C-box 13 perfmon counter 0.
ED9H		MSR_C13_PMON_CTR1	Package	Uncore C-box 13 perfmon counter 1.

Table 2-32. Uncore PMU MSRs in Intel® Xeon® Processor E5 v3 Family (Contd.)

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
EDAH		MSR_C13_PMON_CTR2	Package	Uncore C-box 13 perfmon counter 2.
EDBH		MSR_C13_PMON_CTR3	Package	Uncore C-box 13 perfmon counter 3.
EE0H		MSR_C14_PMON_BOX_CTL	Package	Uncore C-box 14 perfmon local box wide control.
EE1H		MSR_C14_PMON_EVNTSEL0	Package	Uncore C-box 14 perfmon event select for C-box 14 counter 0.
EE2H		MSR_C14_PMON_EVNTSEL1	Package	Uncore C-box 14 perfmon event select for C-box 14 counter 1.
EE3H		MSR_C14_PMON_EVNTSEL2	Package	Uncore C-box 14 perfmon event select for C-box 14 counter 2.
EE4H		MSR_C14_PMON_EVNTSEL3	Package	Uncore C-box 14 perfmon event select for C-box 14 counter 3.
EE5H		MSR_C14_PMON_BOX_FILTER	Package	Uncore C-box 14 perfmon box wide filter0.
EE6H		MSR_C14_PMON_BOX_FILTER1	Package	Uncore C-box 14 perfmon box wide filter1.
EE7H		MSR_C14_PMON_BOX_STATUS	Package	Uncore C-box 14 perfmon box wide status.
EE8H		MSR_C14_PMON_CTR0	Package	Uncore C-box 14 perfmon counter 0.
EE9H		MSR_C14_PMON_CTR1	Package	Uncore C-box 14 perfmon counter 1.
EEAH		MSR_C14_PMON_CTR2	Package	Uncore C-box 14 perfmon counter 2.
EEBH		MSR_C14_PMON_CTR3	Package	Uncore C-box 14 perfmon counter 3.
EF0H		MSR_C15_PMON_BOX_CTL	Package	Uncore C-box 15 perfmon local box wide control.
EF1H		MSR_C15_PMON_EVNTSEL0	Package	Uncore C-box 15 perfmon event select for C-box 15 counter 0.
EF2H		MSR_C15_PMON_EVNTSEL1	Package	Uncore C-box 15 perfmon event select for C-box 15 counter 1.
EF3H		MSR_C15_PMON_EVNTSEL2	Package	Uncore C-box 15 perfmon event select for C-box 15 counter 2.
EF4H		MSR_C15_PMON_EVNTSEL3	Package	Uncore C-box 15 perfmon event select for C-box 15 counter 3.
EF5H		MSR_C15_PMON_BOX_FILTER0	Package	Uncore C-box 15 perfmon box wide filter0.
EF6H		MSR_C15_PMON_BOX_FILTER1	Package	Uncore C-box 15 perfmon box wide filter1.
EF7H		MSR_C15_PMON_BOX_STATUS	Package	Uncore C-box 15 perfmon box wide status.
EF8H		MSR_C15_PMON_CTR0	Package	Uncore C-box 15 perfmon counter 0.
EF9H		MSR_C15_PMON_CTR1	Package	Uncore C-box 15 perfmon counter 1.
EFAH		MSR_C15_PMON_CTR2	Package	Uncore C-box 15 perfmon counter 2.
EFBH		MSR_C15_PMON_CTR3	Package	Uncore C-box 15 perfmon counter 3.
F00H		MSR_C16_PMON_BOX_CTL	Package	Uncore C-box 16 perfmon for box-wide control
F01H		MSR_C16_PMON_EVNTSEL0	Package	Uncore C-box 16 perfmon event select for C-box 16 counter 0.
F02H		MSR_C16_PMON_EVNTSEL1	Package	Uncore C-box 16 perfmon event select for C-box 16 counter 1.
F03H		MSR_C16_PMON_EVNTSEL2	Package	Uncore C-box 16 perfmon event select for C-box 16 counter 2.
F04H		MSR_C16_PMON_EVNTSEL3	Package	Uncore C-box 16 perfmon event select for C-box 16 counter 3.
F05H		MSR_C16_PMON_BOX_FILTER0	Package	Uncore C-box 16 perfmon box wide filter 0.
F06H		MSR_C16_PMON_BOX_FILTER1	Package	Uncore C-box 16 perfmon box wide filter 1.
F07H		MSR_C16_PMON_BOX_STATUS	Package	Uncore C-box 16 perfmon box wide status.
F08H		MSR_C16_PMON_CTR0	Package	Uncore C-box 16 perfmon counter 0.
F09H		MSR_C16_PMON_CTR1	Package	Uncore C-box 16 perfmon counter 1.
F0AH		MSR_C16_PMON_CTR2	Package	Uncore C-box 16 perfmon counter 2.

Table 2-32. Uncore PMU MSRs in Intel® Xeon® Processor E5 v3 Family (Contd.)

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
E0BH		MSR_C16_PMON_CTR3	Package	Uncore C-box 16 perfmon counter 3.
F10H		MSR_C17_PMON_BOX_CTL	Package	Uncore C-box 17 perfmon for box-wide control
F11H		MSR_C17_PMON_EVNTSEL0	Package	Uncore C-box 17 perfmon event select for C-box 17 counter 0.
F12H		MSR_C17_PMON_EVNTSEL1	Package	Uncore C-box 17 perfmon event select for C-box 17 counter 1.
F13H		MSR_C17_PMON_EVNTSEL2	Package	Uncore C-box 17 perfmon event select for C-box 17 counter 2.
F14H		MSR_C17_PMON_EVNTSEL3	Package	Uncore C-box 17 perfmon event select for C-box 17 counter 3.
F15H		MSR_C17_PMON_BOX_FILTER0	Package	Uncore C-box 17 perfmon box wide filter 0.
F16H		MSR_C17_PMON_BOX_FILTER1	Package	Uncore C-box 17 perfmon box wide filter1.
F17H		MSR_C17_PMON_BOX_STATUS	Package	Uncore C-box 17 perfmon box wide status.
F18H		MSR_C17_PMON_CTR0	Package	Uncore C-box 17 perfmon counter 0.
F19H		MSR_C17_PMON_CTR1	Package	Uncore C-box 17 perfmon counter 1.
F1AH		MSR_C17_PMON_CTR2	Package	Uncore C-box 17 perfmon counter 2.
F1BH		MSR_C17_PMON_CTR3	Package	Uncore C-box 17 perfmon counter 3.

2.14 MSRS IN INTEL® CORE™ M PROCESSORS AND 5TH GENERATION INTEL CORE PROCESSORS

The Intel® Core™ M-5xxx processors and 5th generation Intel® Core™ Processors, and Intel® Xeon® Processor E3-1200 v4 family are based on the Broadwell microarchitecture. The Intel® Core™ M-5xxx processors and 5th generation Intel® Core™ Processors have CPUID DisplayFamily_DisplayModel signature 06_3DH. Intel® Xeon® Processor E3-1200 v4 family and the 5th generation Intel® Core™ Processors have CPUID DisplayFamily_DisplayModel signature 06_47H. Processors with signatures 06_3DH and 06_47H support the MSR interfaces listed in Table 2-19, Table 2-20, Table 2-21, Table 2-24, Table 2-28, Table 2-29, Table 2-33, and Table 2-34. For an MSR listed in Table 2-34 that also appears in the model-specific tables of prior generations, Table 2-34 supercede prior generation tables.

Table 2-33 lists MSRs that are common to processors based on the Broadwell microarchitectures (including CPUID signatures 06_3DH, 06_47H, 06_4FH, and 06_56H).

Table 2-33. Additional MSRs Common to Processors Based the Broadwell Microarchitectures

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
38EH	910	IA32_PERF_GLOBAL_ STATUS	Thread	See Table 2-2. See Section 18.6.2.2, "Global Counter Control Facilities."
		0		**Ovf_PMC0**
		1		**Ovf_PMC1**
		2		**Ovf_PMC2**
		3		**Ovf_PMC3**
		31:4		Reserved.
		32		**Ovf_FixedCtr0**

Table 2-33. Additional MSRs Common to Processors Based the Broadwell Microarchitectures

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
		33		**Ovf_FixedCtr1**
		34		**Ovf_FixedCtr2**
		54:35		Reserved.
		55		**Trace_ToPA_PMI. See** Section 35.2.6.2, "Table of Physical Addresses (ToPA)."
		60:56		Reserved.
		61		**Ovf_Uncore**
		62		**Ovf_BufDSSAVE**
		63		**CondChgd**
390H	912	IA32_PERF_GLOBAL_OVF_CTRL	Thread	See Table 2-2. See Section 18.6.2.2, "Global Counter Control Facilities."
		0		**Set 1 to clear Ovf_PMC0**
		1		**Set 1 to clear Ovf_PMC1**
		2		**Set 1 to clear Ovf_PMC2**
		3		**Set 1 to clear Ovf_PMC3**
		31:4		Reserved.
		32		**Set 1 to clear Ovf_FixedCtr0**
		33		**Set 1 to clear Ovf_FixedCtr1**
		34		**Set 1 to clear Ovf_FixedCtr2**
		54:35		Reserved.
		55		**Set 1 to clear Trace_ToPA_PMI. See** Section 35.2.6.2, "Table of Physical Addresses (ToPA)."
		60:56		Reserved.
		61		**Set 1 to clear Ovf_Uncore**
		62		**Set 1 to clear Ovf_BufDSSAVE**
		63		**Set 1 to clear CondChgd**
560H	1376	IA32_RTIT_OUTPUT_BASE	THREAD	**Trace Output Base Register (R/W)**
		6:0		Reserved.
		MAXPHYADDR[1]-1:7		**Base physical address.**
		63:MAXPHYADDR		Reserved.
561H	1377	IA32_RTIT_OUTPUT_MASK_PTRS	THREAD	**Trace Output Mask Pointers Register (R/W)**
		6:0		Reserved.
		31:7		**MaskOrTableOffset**
		63:32		**Output Offset.**
570H	1392	IA32_RTIT_CTL	Thread	**Trace Control Register (R/W)**
		0		**TraceEn**
		1		Reserved, MBZ.

Table 2-33. Additional MSRs Common to Processors Based the Broadwell Microarchitectures

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
		2		**OS**
		3		**User**
		6:4		Reserved, MBZ
		7		**CR3 filter**
		8		**ToPA; writing 0 will #GP if also setting TraceEn**
		9		Reserved, MBZ
		10		**TSCEn**
		11		**DisRETC**
		12		Reserved, MBZ
		13		**Reserved; writing 0 will #GP if also setting TraceEn**
		63:14		Reserved, MBZ.
571H	1393	IA32_RTIT_STATUS	Thread	**Tracing Status Register (R/W)**
		0		Reserved, writes ignored.
		1		**ContexEn**, writes ignored.
		2		**TriggerEn**, writes ignored.
		3		Reserved
		4		**Error (R/W)**
		5		**Stopped**
		63:6		Reserved, MBZ.
572H	1394	IA32_RTIT_CR3_MATCH	THREAD	**Trace Filter CR3 Match Register (R/W)**
		4:0		Reserved
		63:5		CR3[63:5] value to match
620H		MSR UNCORE_RATIO_LIMIT	Package	Uncore Ratio Limit (R/W) Out of reset, the min_ratio and max_ratio fields represent the widest possible range of uncore frequencies. Writing to these fields allows software to control the minimum and the maximum frequency that hardware will select.
		63:15		Reserved.
		14:8		MIN_RATIO Writing to this field controls the minimum possible ratio of the LLC/Ring.
		7		Reserved.
		6:0		MAX_RATIO This field is used to limit the max ratio of the LLC/Ring.

NOTES:

1. MAXPHYADDR is reported by CPUID.80000008H:EAX[7:0].

Table 2-34 lists MSRs that are specific to Intel Core M processors and 5th Generation Intel Core Processors.

Table 2-34. Additional MSRs Supported by Intel® Core™ M Processors and 5th Generation Intel® Core™ Processors

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
E2H	226	MSR_PKG_CST_CONFIG_ CONTROL	Core	**C-State Configuration Control (R/W)** Note: C-state values are processor specific C-state code names, unrelated to MWAIT extension C-state parameters or ACPI C-states. See http://biosbits.org.
		3:0		**Package C-State Limit (R/W)** Specifies the lowest processor-specific C-state code name (consuming the least power) for the package. The default is set as factory-configured package C-state limit. The following C-state code name encodings are supported: 0000b: C0/C1 (no package C-state support) 0001b: C2 0010b: C3 0011b: C6 0100b: C7 0101b: C7s 0110b: C8 0111b: C9 1000b: C10
		9:4		Reserved
		10		**I/O MWAIT Redirection Enable (R/W)**
		14:11		Reserved
		15		**CFG Lock (R/WO)**
		24:16		Reserved
		25		**C3 State Auto Demotion Enable (R/W)**
		26		**C1 State Auto Demotion Enable (R/W)**
		27		**Enable C3 Undemotion (R/W)**
		28		**Enable C1 Undemotion (R/W)**
		29		**Enable Package C-State Auto-demotion (R/W)**
		30		**Enable Package C-State Undemotion (R/W)**
		63:31		Reserved
1ADH	429	MSR_TURBO_RATIO_LIMIT	Package	**Maximum Ratio Limit of Turbo Mode** RO if MSR_PLATFORM_INFO.[28] = 0, RW if MSR_PLATFORM_INFO.[28] = 1
		7:0	Package	**Maximum Ratio Limit for 1C** Maximum turbo ratio limit of 1 core active.
		15:8	Package	**Maximum Ratio Limit for 2C** Maximum turbo ratio limit of 2 core active.
		23:16	Package	**Maximum Ratio Limit for 3C** Maximum turbo ratio limit of 3 core active.

Table 2-34. Additional MSRs Supported by Intel® Core™ M Processors and 5th Generation Intel® Core™ Processors

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
		31:24	Package	**Maximum Ratio Limit for 4C** Maximum turbo ratio limit of 4 core active.
		39:32	Package	**Maximum Ratio Limit for 5C** Maximum turbo ratio limit of 5core active.
		47:40	Package	**Maximum Ratio Limit for 6C** Maximum turbo ratio limit of 6core active.
		63:48		Reserved.
639H	1593	MSR_PP0_ENERGY_STATUS	Package	**PP0 Energy Status (R/O)** See Section 14.9.4, "PP0/PP1 RAPL Domains."
See Table 2-19, Table 2-20, Table 2-21, Table 2-24, Table 2-28, Table 2-29, Table 2-33 for other MSR definitions applicable to processors with CPUID signature 06_3DH.				

2.15 MSRS IN INTEL® XEON® PROCESSORS E5 V4 FAMILY

The MSRs listed in Table 2-35 are available and common to Intel® Xeon® Processor D product Family (CPUID DisplayFamily_DisplayModel = 06_56H) and to Intel Xeon processors E5 v4, E7 v4 families (CPUID DisplayFamily_DisplayModel = 06_4FH). They are based on the Broadwell microarchitecture.

See Section 2.15.1 for lists of tables of MSRs that are supported by Intel® Xeon® Processor D Family.

Table 2-35. Additional MSRs Common to Intel® Xeon® Processor D and Intel Xeon Processors E5 v4 Family Based on the Broadwell Microarchitecture

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
4EH	78	MSR_PPIN_CTL	Package	Protected Processor Inventory Number Enable Control (R/W)
		0		LockOut (R/WO) See Table 2-25.
		1		Enable_PPIN (R/W) See Table 2-25.
		63:2		Reserved.
4FH	79	MSR_PPIN	Package	Protected Processor Inventory Number (R/O)
		63:0		**Protected Processor Inventory Number (R/O)** See Table 2-25.
CEH	206	MSR_PLATFORM_INFO	Package	Platform Information; contains power management and other model specific features enumeration. See http://biosbits.org.
		7:0		Reserved.
		15:8	Package	**Maximum Non-Turbo Ratio (R/O)** See Table 2-25.
		22:16		Reserved.

Table 2-35. Additional MSRs Common to Intel® Xeon® Processor D and Intel Xeon Processors E5 v4 Family Based on the Broadwell Microarchitecture

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
		23	Package	**PPIN_CAP (R/O)** See Table 2-25.
		27:24		Reserved.
		28	Package	**Programmable Ratio Limit for Turbo Mode (R/O)** See Table 2-25.
		29	Package	**Programmable TDP Limit for Turbo Mode (R/O)** See Table 2-25.
		30	Package	**Programmable TJ OFFSET (R/O)** See Table 2-25.
		39:31		Reserved.
		47:40	Package	**Maximum Efficiency Ratio (R/O)** See Table 2-25.
		63:48		Reserved.
E2H	226	MSR_PKG_CST_CONFIG_ CONTROL	Core	**C-State Configuration Control (R/W)** Note: C-state values are processor specific C-state code names, unrelated to MWAIT extension C-state parameters or ACPI C-states. See http://biosbits.org.
		2:0		**Package C-State Limit (R/W)** Specifies the lowest processor-specific C-state code name (consuming the least power) for the package. The default is set as factory-configured package C-state limit. The following C-state code name encodings are supported: 000b: C0/C1 (no package C-state support) 001b: C2 010b: C6 (non-retention) 011b: C6 (retention) 111b: No Package C state limits. All C states supported by the processor are available.
		9:3		Reserved
		10		**I/O MWAIT Redirection Enable (R/W)**
		14:11		Reserved
		15		**CFG Lock (R/WO)**
		16		**Automatic C-State Conversion Enable (R/W)** If 1, the processor will convert HALT or MWAT(C1) to MWAIT(C6)
		24:17		Reserved
		25		**C3 State Auto Demotion Enable (R/W)**
		26		**C1 State Auto Demotion Enable (R/W)**
		27		**Enable C3 Undemotion (R/W)**
		28		**Enable C1 Undemotion (R/W)**

Table 2-35. Additional MSRs Common to Intel® Xeon® Processor D and Intel Xeon Processors E5 v4 Family Based on the Broadwell Microarchitecture

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
		29		**Package C State Demotion Enable (R/W)**
		30		**Package C State UnDemotion Enable (R/W)**
		63:31		Reserved
179H	377	IA32_MCG_CAP	Thread	**Global Machine Check Capability (R/O)**
		7:0		**Count**
		8		**MCG_CTL_P**
		9		**MCG_EXT_P**
		10		**MCP_CMCI_P**
		11		**MCG_TES_P**
		15:12		Reserved.
		23:16		**MCG_EXT_CNT**
		24		**MCG_SER_P**
		25		**MCG_EM_P**
		26		**MCG_ELOG_P**
		63:27		Reserved.
17DH	390	MSR_SMM_MCA_CAP	THREAD	**Enhanced SMM Capabilities (SMM-RO)** Reports SMM capability Enhancement. Accessible only while in SMM.
		57:0		**Reserved**
		58		**SMM_Code_Access_Chk (SMM-RO)** If set to 1 indicates that the SMM code access restriction is supported and a host-space interface available to SMM handler.
		59		**Long_Flow_Indication (SMM-RO)** If set to 1 indicates that the SMM long flow indicator is supported and a host-space interface available to SMM handler.
		63:60		Reserved
19CH	412	IA32_THERM_STATUS	Core	**Thermal Monitor Status (R/W)** See Table 2-2.
		0		**Thermal status (RO)** See Table 2-2.
		1		**Thermal status log (R/WC0)** See Table 2-2.
		2		**PROTCHOT # or FORCEPR# status (RO)** See Table 2-2.
		3		**PROTCHOT # or FORCEPR# log (R/WC0)** See Table 2-2.
		4		**Critical Temperature status (RO)** See Table 2-2.

Table 2-35. Additional MSRs Common to Intel® Xeon® Processor D and Intel Xeon Processors E5 v4 Family Based on the Broadwell Microarchitecture

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
		5		**Critical Temperature status log (R/WC0)** See Table 2-2.
		6		**Thermal threshold #1 status (RO)** See Table 2-2.
		7		**Thermal threshold #1 log (R/WC0)** See Table 2-2.
		8		**Thermal threshold #2 status (RO)** See Table 2-2.
		9		**Thermal threshold #2 log (R/WC0)** See Table 2-2.
		10		**Power Limitation status (RO)** See Table 2-2.
		11		**Power Limitation log (R/WC0)** See Table 2-2.
		12		**Current Limit status (RO)** See Table 2-2.
		13		**Current Limit log (R/WC0)** See Table 2-2.
		14		**Cross Domain Limit status (RO)** See Table 2-2.
		15		**Cross Domain Limit log (R/WC0)** See Table 2-2.
		22:16		**Digital Readout (RO)** See Table 2-2.
		26:23		Reserved.
		30:27		**Resolution in degrees Celsius (RO)** See Table 2-2.
		31		**Reading Valid (RO)** See Table 2-2.
		63:32		Reserved.
1A2H	418	MSR_ TEMPERATURE_TARGET	Package	Temperature Target
		15:0		Reserved.
		23:16		**Temperature Target (RO)** See Table 2-25.
		27:24		**TCC Activation Offset (R/W)** See Table 2-25.
		63:28		Reserved.

Table 2-35. Additional MSRs Common to Intel® Xeon® Processor D and Intel Xeon Processors E5 v4 Family Based on the Broadwell Microarchitecture

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
1ADH	429	MSR_TURBO_RATIO_LIMIT	Package	**Maximum Ratio Limit of Turbo Mode** RO if MSR_PLATFORM_INFO.[28] = 0, RW if MSR_PLATFORM_INFO.[28] = 1
		7:0	Package	**Maximum Ratio Limit for 1C**
		15:8	Package	**Maximum Ratio Limit for 2C**
		23:16	Package	**Maximum Ratio Limit for 3C**
		31:24	Package	**Maximum Ratio Limit for 4C**
		39:32	Package	**Maximum Ratio Limit for 5C**
		47:40	Package	**Maximum Ratio Limit for 6C**
		55:48	Package	**Maximum Ratio Limit for 7C**
		63:56	Package	**Maximum Ratio Limit for 8C**
1AEH	430	MSR_TURBO_RATIO_LIMIT1	Package	**Maximum Ratio Limit of Turbo Mode** RO if MSR_PLATFORM_INFO.[28] = 0, RW if MSR_PLATFORM_INFO.[28] = 1
		7:0	Package	**Maximum Ratio Limit for 9C**
		15:8	Package	**Maximum Ratio Limit for 10C**
		23:16	Package	**Maximum Ratio Limit for 11C**
		31:24	Package	**Maximum Ratio Limit for 12C**
		39:32	Package	**Maximum Ratio Limit for 13C**
		47:40	Package	**Maximum Ratio Limit for 14C**
		55:48	Package	**Maximum Ratio Limit for 15C**
		63:56	Package	**Maximum Ratio Limit for 16C**
606H	1542	MSR_RAPL_POWER_UNIT	Package	**Unit Multipliers used in RAPL Interfaces (R/O)**
		3:0	Package	**Power Units** See Section 14.9.1, "RAPL Interfaces."
		7:4	Package	**Reserved**
		12:8	Package	**Energy Status Units** Energy related information (in Joules) is based on the multiplier, 1/2^ESU; where ESU is an unsigned integer represented by bits 12:8. Default value is 0EH (or 61 micro-joules)
		15:13	Package	**Reserved**
		19:16	Package	**Time Units** See Section 14.9.1, "RAPL Interfaces."
		63:20		Reserved
618H	1560	MSR_DRAM_POWER_LIMIT	Package	**DRAM RAPL Power Limit Control (R/W)** See Section 14.9.5, "DRAM RAPL Domain."
619H	1561	MSR_DRAM_ENERGY_ STATUS	Package	**DRAM Energy Status (R/O)** Energy consumed by DRAM devices

Table 2-35. Additional MSRs Common to Intel® Xeon® Processor D and Intel Xeon Processors E5 v4 Family Based on the Broadwell Microarchitecture

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
		31:0		Energy in 15.3 micro-joules. Requires BIOS configuration to enable DRAM RAPL mode 0 (Direct VR).
		63:32		Reserved
61BH	1563	MSR_DRAM_PERF_STATUS	Package	**DRAM Performance Throttling Status (R/O)** See Section 14.9.5, "DRAM RAPL Domain."
61CH	1564	MSR_DRAM_POWER_INFO	Package	**DRAM RAPL Parameters (R/W)** See Section 14.9.5, "DRAM RAPL Domain."
620H	1568	MSR UNCORE_RATIO_LIMIT	Package	Uncore Ratio Limit (R/W) Out of reset, the min_ratio and max_ratio fields represent the widest possible range of uncore frequencies. Writing to these fields allows software to control the minimum and the maximum frequency that hardware will select.
		63:15		Reserved.
		14:8		MIN_RATIO Writing to this field controls the minimum possible ratio of the LLC/Ring.
		7		Reserved.
		6:0		MAX_RATIO This field is used to limit the max ratio of the LLC/Ring.
639H	1593	MSR_PP0_ENERGY_STATUS	Package	**Reserved (R/O)** Reads return 0
690H	1680	MSR_CORE_PERF_LIMIT_REASONS	Package	**Indicator of Frequency Clipping in Processor Cores (R/W)** **(frequency refers to processor core frequency)**
		0		**PROCHOT Status (R0)** When set, processor core frequency is reduced below the operating system request due to assertion of external PROCHOT.
		1		**Thermal Status (R0)** When set, frequency is reduced below the operating system request due to a thermal event.
		2		**Power Budget Management Status (R0)** When set, frequency is reduced below the operating system request due to PBM limit
		3		**Platform Configuration Services Status (R0)** When set, frequency is reduced below the operating system request due to PCS limit
		4		Reserved.
		5		**Autonomous Utilization-Based Frequency Control Status (R0)** When set, frequency is reduced below the operating system request because the processor has detected that utilization is low

Table 2-35. Additional MSRs Common to Intel® Xeon® Processor D and Intel Xeon Processors E5 v4 Family Based on the Broadwell Microarchitecture

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
		6		**VR Therm Alert Status (R0)** When set, frequency is reduced below the operating system request due to a thermal alert from the Voltage Regulator.
		7		Reserved.
		8		**Electrical Design Point Status (R0)** When set, frequency is reduced below the operating system request due to electrical design point constraints (e.g. maximum electrical current consumption).
		9		Reserved.
		10		**Multi-Core Turbo Status (R0)** When set, frequency is reduced below the operating system request due to Multi-Core Turbo limits
		12:11		Reserved.
		13		**Core Frequency P1 Status (R0)** When set, frequency is reduced below max non-turbo P1
		14		**Core Max n-core Turbo Frequency Limiting Status (R0)** When set, frequency is reduced below max n-core turbo frequency
		15		**Core Frequency Limiting Status (R0)** When set, frequency is reduced below the operating system request.
		16		**PROCHOT Log** When set, indicates that the PROCHOT Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.
		17		**Thermal Log** When set, indicates that the Thermal Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.
		18		**Power Budget Management Log** When set, indicates that the PBM Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.
		19		**Platform Configuration Services Log** When set, indicates that the PCS Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.
		20		Reserved.
		21		**Autonomous Utilization-Based Frequency Control Log** When set, indicates that the AUBFC Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.

Table 2-35. Additional MSRs Common to Intel® Xeon® Processor D and Intel Xeon Processors E5 v4 Family Based on the Broadwell Microarchitecture

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
		22		**VR Therm Alert Log** When set, indicates that the VR Therm Alert Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.
		23		Reserved.
		24		**Electrical Design Point Log** When set, indicates that the EDP Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.
		25		Reserved.
		26		**Multi-Core Turbo Log** When set, indicates that the Multi-Core Turbo Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.
		28:27		Reserved.
		29		**Core Frequency P1 Log** When set, indicates that the Core Frequency P1 Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.
		30		**Core Max n-core Turbo Frequency Limiting Log** When set, indicates that the Core Max n-core Turbo Frequency Limiting Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.
		31		**Core Frequency Limiting Log** When set, indicates that the Core Frequency Limiting Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.
		63:32		Reserved.
770H	1904	IA32_PM_ENABLE	Package	See Section 14.4.2, "Enabling HWP"
771H	1905	IA32_HWP_CAPABILITIES	Thread	See Section 14.4.3, "HWP Performance Range and Dynamic Capabilities"
774H	1908	IA32_HWP_REQUEST	Thread	See Section 14.4.4, "Managing HWP"
		7:0		**Minimum Performance (R/W)**
		15:8		**Maximum Performance (R/W)**
		23:16		**Desired Performance (R/W)**
		63:24		Reserved.
777H	1911	IA32_HWP_STATUS	Thread	See Section 14.4.5, "HWP Feedback"
		1:0		Reserved.
		2		**Excursion to Minimum (RO)**
		63:3		Reserved.

Table 2-35. Additional MSRs Common to Intel® Xeon® Processor D and Intel Xeon Processors E5 v4 Family Based on the Broadwell Microarchitecture

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
C8DH	3213	IA32_QM_EVTSEL	THREAD	**Monitoring Event Select Register (R/W)** if CPUID.(EAX=07H, ECX=0):EBX.RDT-M[bit 12] = 1
		7:0		**EventID (RW)** **Event encoding:** **0x00: no monitoring** **0x01: L3 occupancy monitoring** **0x02: Total memory bandwidth monitoring** **0x03: Local memory bandwidth monitoring** **All other encoding reserved**
		31:8		Reserved.
		41:32		**RMID (RW)**
		63:42		Reserved.
C8FH	3215	IA32_PQR_ASSOC	THREAD	**Resource Association Register (R/W)**
		9:0		**RMID**
		31:10		**Reserved**
		51:32		**COS (R/W).**
		63: 52		**Reserved**
C90H	3216	IA32_L3_QOS_MASK_0	Package	**L3 Class Of Service Mask - COS 0 (R/W)** if CPUID.(EAX=10H, ECX=1):EDX.COS_MAX[15:0] >=0
		0:19		**CBM: Bit vector of available L3 ways for COS 0 enforcement**
		63:20		**Reserved**
C91H	3217	IA32_L3_QOS_MASK_1	Package	**L3 Class Of Service Mask - COS 1 (R/W)** if CPUID.(EAX=10H, ECX=1):EDX.COS_MAX[15:0] >=1
		0:19		**CBM: Bit vector of available L3 ways for COS 1 enforcement**
		63:20		**Reserved**
C92H	3218	IA32_L3_QOS_MASK_2	Package	**L3 Class Of Service Mask - COS 2 (R/W).** if CPUID.(EAX=10H, ECX=1):EDX.COS_MAX[15:0] >=2
		0:19		**CBM: Bit vector of available L3 ways for COS 2 enforcement**
		63:20		**Reserved**
C93H	3219	IA32_L3_QOS_MASK_3	Package	**L3 Class Of Service Mask - COS 3 (R/W).** if CPUID.(EAX=10H, ECX=1):EDX.COS_MAX[15:0] >=3
		0:19		**CBM: Bit vector of available L3 ways for COS 3 enforcement**
		63:20		**Reserved**
C94H	3220	IA32_L3_QOS_MASK_4	Package	**L3 Class Of Service Mask - COS 4 (R/W).** if CPUID.(EAX=10H, ECX=1):EDX.COS_MAX[15:0] >=4
		0:19		**CBM: Bit vector of available L3 ways for COS 4 enforcement**
		63:20		**Reserved**

Table 2-35. Additional MSRs Common to Intel® Xeon® Processor D and Intel Xeon Processors E5 v4 Family Based on the Broadwell Microarchitecture

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
C95H	3221	IA32_L3_QOS_MASK_5	Package	**L3 Class Of Service Mask - COS 5 (R/W).** if CPUID.(EAX=10H, ECX=1):EDX.COS_MAX[15:0] >=5
		0:19		**CBM: Bit vector of available L3 ways for COS 5 enforcement**
		63:20		**Reserved**
C96H	3222	IA32_L3_QOS_MASK_6	Package	**L3 Class Of Service Mask - COS 6 (R/W).** if CPUID.(EAX=10H, ECX=1):EDX.COS_MAX[15:0] >=6
		0:19		**CBM: Bit vector of available L3 ways for COS 6 enforcement**
		63:20		**Reserved**
C97H	3223	IA32_L3_QOS_MASK_7	Package	**L3 Class Of Service Mask - COS 7 (R/W).** if CPUID.(EAX=10H, ECX=1):EDX.COS_MAX[15:0] >=7
		0:19		**CBM: Bit vector of available L3 ways for COS 7 enforcement**
		63:20		**Reserved**
C98H	3224	IA32_L3_QOS_MASK_8	Package	**L3 Class Of Service Mask - COS 8 (R/W).** if CPUID.(EAX=10H, ECX=1):EDX.COS_MAX[15:0] >=8
		0:19		**CBM: Bit vector of available L3 ways for COS 8 enforcement**
		63:20		**Reserved**
C99H	3225	IA32_L3_QOS_MASK_9	Package	**L3 Class Of Service Mask - COS 9 (R/W).** if CPUID.(EAX=10H, ECX=1):EDX.COS_MAX[15:0] >=9
		0:19		**CBM: Bit vector of available L3 ways for COS 9 enforcement**
		63:20		**Reserved**
C9AH	3226	IA32_L3_QOS_MASK_10	Package	**L3 Class Of Service Mask - COS 10 (R/W).** if CPUID.(EAX=10H, ECX=1):EDX.COS_MAX[15:0] >=10
		0:19		**CBM: Bit vector of available L3 ways for COS 10 enforcement**
		63:20		**Reserved**
C9BH	3227	IA32_L3_QOS_MASK_11	Package	**L3 Class Of Service Mask - COS 11 (R/W).** if CPUID.(EAX=10H, ECX=1):EDX.COS_MAX[15:0] >=11
		0:19		**CBM: Bit vector of available L3 ways for COS 11 enforcement**
		63:20		**Reserved**
C9CH	3228	IA32_L3_QOS_MASK_12	Package	**L3 Class Of Service Mask - COS 12 (R/W).** if CPUID.(EAX=10H, ECX=1):EDX.COS_MAX[15:0] >=12
		0:19		**CBM: Bit vector of available L3 ways for COS 12 enforcement**
		63:20		**Reserved**
C9DH	3229	IA32_L3_QOS_MASK_13	Package	**L3 Class Of Service Mask - COS 13 (R/W).** if CPUID.(EAX=10H, ECX=1):EDX.COS_MAX[15:0] >=13

Table 2-35. Additional MSRs Common to Intel® Xeon® Processor D and Intel Xeon Processors E5 v4 Family Based on the Broadwell Microarchitecture

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
		0:19		**CBM: Bit vector of available L3 ways for COS 13 enforcement**
		63:20		**Reserved**
C9EH	3230	IA32_L3_QOS_MASK_14	Package	**L3 Class Of Service Mask - COS 14 (R/W).** if CPUID.(EAX=10H, ECX=1):EDX.COS_MAX[15:0] >=14
		0:19		**CBM: Bit vector of available L3 ways for COS 14 enforcement**
		63:20		**Reserved**
C9FH	3231	IA32_L3_QOS_MASK_15	Package	**L3 Class Of Service Mask - COS 15 (R/W).** if CPUID.(EAX=10H, ECX=1):EDX.COS_MAX[15:0] >=15
		0:19		**CBM: Bit vector of available L3 ways for COS 15 enforcement**
		63:20		**Reserved**

2.15.1 Additional MSRs Supported in the Intel® Xeon® Processor D Product Family

The MSRs listed in Table 2-36 are available to Intel® Xeon® Processor D Product Family (CPUID DisplayFamily_DisplayModel = 06_56H). The Intel® Xeon® processor D product family is based on the Broadwell microarchitecture and supports the MSR interfaces listed in Table 2-19, Table 2-28, Table 2-33, Table 2-35, and Table 2-36.

Table 2-36. Additional MSRs Supported by Intel® Xeon® Processor D with DisplayFamily_DisplayModel 06_56H

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
1ACH	428	MSR_TURBO_RATIO_LIMIT3	Package	**Config Ratio Limit of Turbo Mode** RO if MSR_PLATFORM_INFO.[28] = 0, RW if MSR_PLATFORM_INFO.[28] = 1
		62:0	Package	Reserved
		63	Package	**Semaphore for Turbo Ratio Limit Configuration** If 1, the processor uses override configuration[1] specified in MSR_TURBO_RATIO_LIMIT, MSR_TURBO_RATIO_LIMIT1. If 0, the processor uses factory-set configuration (Default).
286H	646	IA32_MC6_CTL2	Package	See Table 2-2.
287H	647	IA32_MC7_CTL2	Package	See Table 2-2.
289H	649	IA32_MC9_CTL2	Package	See Table 2-2.
28AH	650	IA32_MC10_CTL2	Package	See Table 2-2.
291H	657	IA32_MC17_CTL2	Package	See Table 2-2.
292H	658	IA32_MC18_CTL2	Package	See Table 2-2.
293H	659	IA32_MC19_CTL2	Package	See Table 2-2.

Table 2-36. Additional MSRs Supported by Intel® Xeon® Processor D with DisplayFamily_DisplayModel 06_56H

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
418H	1048	IA32_MC6_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs." through Section 15.3.2.4, "IA32_MCi_MISC MSRs.". Bank MC6 reports MC error from the integrated I/O module.
419H	1049	IA32_MC6_STATUS	Package	
41AH	1050	IA32_MC6_ADDR	Package	
41BH	1051	IA32_MC6_MISC	Package	
41CH	1052	IA32_MC7_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs." through Section 15.3.2.4, "IA32_MCi_MISC MSRs.". Bank MC7 reports MC error from the home agent HA 0.
41DH	1053	IA32_MC7_STATUS	Package	
41EH	1054	IA32_MC7_ADDR	Package	
41FH	1055	IA32_MC7_MISC	Package	
424H	1060	IA32_MC9_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs." through Section 15.3.2.4, "IA32_MCi_MISC MSRs.". Banks MC9 through MC 10 report MC error from each channel of the integrated memory controllers.
425H	1061	IA32_MC9_STATUS	Package	
426H	1062	IA32_MC9_ADDR	Package	
427H	1063	IA32_MC9_MISC	Package	
428H	1064	IA32_MC10_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs." through Section 15.3.2.4, "IA32_MCi_MISC MSRs.". Banks MC9 through MC 10 report MC error from each channel of the integrated memory controllers.
429H	1065	IA32_MC10_STATUS	Package	
42AH	1066	IA32_MC10_ADDR	Package	
42BH	1067	IA32_MC10_MISC	Package	
444H	1092	IA32_MC17_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs." through Section 15.3.2.4, "IA32_MCi_MISC MSRs.". Bank MC17 reports MC error from the following pair of CBo/L3 Slices (if the pair is present): CBo0, CBo3, CBo6, CBo9, CBo12, CBo15.
445H	1093	IA32_MC17_STATUS	Package	
446H	1094	IA32_MC17_ADDR	Package	
447H	1095	IA32_MC17_MISC	Package	
448H	1096	IA32_MC18_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs." through Section 15.3.2.4, "IA32_MCi_MISC MSRs.". Bank MC18 reports MC error from the following pair of CBo/L3 Slices (if the pair is present): CBo1, CBo4, CBo7, CBo10, CBo13, CBo16.
449H	1097	IA32_MC18_STATUS	Package	
44AH	1098	IA32_MC18_ADDR	Package	
44BH	1099	IA32_MC18_MISC	Package	
44CH	1100	IA32_MC19_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs." through Section 15.3.2.4, "IA32_MCi_MISC MSRs.". Bank MC19 reports MC error from the following pair of CBo/L3 Slices (if the pair is present): CBo2, CBo5, CBo8, CBo11, CBo14, CBo17.
44DH	1101	IA32_MC19_STATUS	Package	
44EH	1102	IA32_MC19_ADDR	Package	
44FH	1103	IA32_MC19_MISC	Package	
See Table 2-19, Table 2-28, Table 2-33, and Table 2-35 for other MSR definitions applicable to processors with CPUID signature 06_56H.				

NOTES:

1. An override configuration lower than the factory-set configuration is always supported. An override configuration higher than the factory-set configuration is dependent on features specific to the processor and the platform.

2.15.2 Additional MSRs Supported in Intel® Xeon® Processors E5 v4 and E7 v4 Families

The MSRs listed in Table 2-36 are available to Intel® Xeon® Processor E5 v4 and E7 v4 Families (CPUID DisplayFamily_DisplayModel = 06_4FH). The Intel® Xeon® processor E5 v4 family is based on the Broadwell

microarchitecture and supports the MSR interfaces listed in Table 2-19, Table 2-20, Table 2-28, Table 2-33, Table 2-35, and Table 2-37.

Table 2-37. Additional MSRs Supported by Intel® Xeon® Processors with DisplayFamily_DisplayModel 06_4FH

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
1ACH	428	MSR_TURBO_RATIO_LIMIT3	Package	**Config Ratio Limit of Turbo Mode** RO if MSR_PLATFORM_INFO.[28] = 0, RW if MSR_PLATFORM_INFO.[28] = 1
		62:0	Package	Reserved
		63	Package	**Semaphore for Turbo Ratio Limit Configuration** If 1, the processor uses override configuration[1] specified in MSR_TURBO_RATIO_LIMIT, MSR_TURBO_RATIO_LIMIT1 and MSR_TURBO_RATIO_LIMIT2. If 0, the processor uses factory-set configuration (Default).
285H	645	IA32_MC5_CTL2	Package	See Table 2-2.
286H	646	IA32_MC6_CTL2	Package	See Table 2-2.
287H	647	IA32_MC7_CTL2	Package	See Table 2-2.
288H	648	IA32_MC8_CTL2	Package	See Table 2-2.
289H	649	IA32_MC9_CTL2	Package	See Table 2-2.
28AH	650	IA32_MC10_CTL2	Package	See Table 2-2.
28BH	651	IA32_MC11_CTL2	Package	See Table 2-2.
28CH	652	IA32_MC12_CTL2	Package	See Table 2-2.
28DH	653	IA32_MC13_CTL2	Package	See Table 2-2.
28EH	654	IA32_MC14_CTL2	Package	See Table 2-2.
28FH	655	IA32_MC15_CTL2	Package	See Table 2-2.
290H	656	IA32_MC16_CTL2	Package	See Table 2-2.
291H	657	IA32_MC17_CTL2	Package	See Table 2-2.
292H	658	IA32_MC18_CTL2	Package	See Table 2-2.
293H	659	IA32_MC19_CTL2	Package	See Table 2-2.
294H	660	IA32_MC20_CTL2	Package	See Table 2-2.
295H	661	IA32_MC21_CTL2	Package	See Table 2-2.
414H	1044	IA32_MC5_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs." through Section 15.3.2.4, "IA32_MCi_MISC MSRs.". Bank MC5 reports MC error from the Intel QPI 0 module.
415H	1045	IA32_MC5_STATUS	Package	
416H	1046	IA32_MC5_ADDR	Package	
417H	1047	IA32_MC5_MISC	Package	
418H	1048	IA32_MC6_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs." through Section 15.3.2.4, "IA32_MCi_MISC MSRs.". Bank MC6 reports MC error from the integrated I/O module.
419H	1049	IA32_MC6_STATUS	Package	
41AH	1050	IA32_MC6_ADDR	Package	
41BH	1051	IA32_MC6_MISC	Package	

Table 2-37. Additional MSRs Supported by Intel® Xeon® Processors with DisplayFamily_DisplayModel 06_4FH

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
41CH	1052	IA32_MC7_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs." through Section 15.3.2.4, "IA32_MCi_MISC MSRs.". Bank MC7 reports MC error from the home agent HA 0.
41DH	1053	IA32_MC7_STATUS	Package	
41EH	1054	IA32_MC7_ADDR	Package	
41FH	1055	IA32_MC7_MISC	Package	
420H	1056	IA32_MC8_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs." through Section 15.3.2.4, "IA32_MCi_MISC MSRs.". Bank MC8 reports MC error from the home agent HA 1.
421H	1057	IA32_MC8_STATUS	Package	
422H	1058	IA32_MC8_ADDR	Package	
423H	1059	IA32_MC8_MISC	Package	
424H	1060	IA32_MC9_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs." through Section 15.3.2.4, "IA32_MCi_MISC MSRs.". Banks MC9 through MC 16 report MC error from each channel of the integrated memory controllers.
425H	1061	IA32_MC9_STATUS	Package	
426H	1062	IA32_MC9_ADDR	Package	
427H	1063	IA32_MC9_MISC	Package	
428H	1064	IA32_MC10_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs." through Section 15.3.2.4, "IA32_MCi_MISC MSRs.". Banks MC9 through MC 16 report MC error from each channel of the integrated memory controllers.
429H	1065	IA32_MC10_STATUS	Package	
42AH	1066	IA32_MC10_ADDR	Package	
42BH	1067	IA32_MC10_MISC	Package	
42CH	1068	IA32_MC11_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs." through Section 15.3.2.4, "IA32_MCi_MISC MSRs.". Banks MC9 through MC 16 report MC error from each channel of the integrated memory controllers.
42DH	1069	IA32_MC11_STATUS	Package	
42EH	1070	IA32_MC11_ADDR	Package	
42FH	1071	IA32_MC11_MISC	Package	
430H	1072	IA32_MC12_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs." through Section 15.3.2.4, "IA32_MCi_MISC MSRs.". Banks MC9 through MC 16 report MC error from each channel of the integrated memory controllers.
431H	1073	IA32_MC12_STATUS	Package	
432H	1074	IA32_MC12_ADDR	Package	
433H	1075	IA32_MC12_MISC	Package	
434H	1076	IA32_MC13_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs." through Section 15.3.2.4, "IA32_MCi_MISC MSRs.". Banks MC9 through MC 16 report MC error from each channel of the integrated memory controllers.
435H	1077	IA32_MC13_STATUS	Package	
436H	1078	IA32_MC13_ADDR	Package	
437H	1079	IA32_MC13_MISC	Package	
438H	1080	IA32_MC14_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs." through Section 15.3.2.4, "IA32_MCi_MISC MSRs.". Banks MC9 through MC 16 report MC error from each channel of the integrated memory controllers.
439H	1081	IA32_MC14_STATUS	Package	
43AH	1082	IA32_MC14_ADDR	Package	
43BH	1083	IA32_MC14_MISC	Package	
43CH	1084	IA32_MC15_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs." through Section 15.3.2.4, "IA32_MCi_MISC MSRs.". Banks MC9 through MC 16 report MC error from each channel of the integrated memory controllers.
43DH	1085	IA32_MC15_STATUS	Package	
43EH	1086	IA32_MC15_ADDR	Package	
43FH	1087	IA32_MC15_MISC	Package	

Table 2-37. Additional MSRs Supported by Intel® Xeon® Processors with DisplayFamily_DisplayModel 06_4FH

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
440H	1088	IA32_MC16_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs." through Section 15.3.2.4, "IA32_MCi_MISC MSRs.". Banks MC9 through MC 16 report MC error from each channel of the integrated memory controllers.
441H	1089	IA32_MC16_STATUS	Package	
442H	1090	IA32_MC16_ADDR	Package	
443H	1091	IA32_MC16_MISC	Package	
444H	1092	IA32_MC17_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs." through Section 15.3.2.4, "IA32_MCi_MISC MSRs.". Bank MC17 reports MC error from the following pair of CBo/L3 Slices (if the pair is present): CBo0, CBo3, CBo6, CBo9, CBo12, CBo15.
445H	1093	IA32_MC17_STATUS	Package	
446H	1094	IA32_MC17_ADDR	Package	
447H	1095	IA32_MC17_MISC	Package	
448H	1096	IA32_MC18_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs." through Section 15.3.2.4, "IA32_MCi_MISC MSRs.". Bank MC18 reports MC error from the following pair of CBo/L3 Slices (if the pair is present): CBo1, CBo4, CBo7, CBo10, CBo13, CBo16.
449H	1097	IA32_MC18_STATUS	Package	
44AH	1098	IA32_MC18_ADDR	Package	
44BH	1099	IA32_MC18_MISC	Package	
44CH	1100	IA32_MC19_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs." through Section 15.3.2.4, "IA32_MCi_MISC MSRs.". Bank MC19 reports MC error from the following pair of CBo/L3 Slices (if the pair is present): CBo2, CBo5, CBo8, CBo11, CBo14, CBo17.
44DH	1101	IA32_MC19_STATUS	Package	
44EH	1102	IA32_MC19_ADDR	Package	
44FH	1103	IA32_MC19_MISC	Package	
450H	1104	IA32_MC20_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs." through Section 15.3.2.4, "IA32_MCi_MISC MSRs.". Bank MC20 reports MC error from the Intel QPI 1 module.
451H	1105	IA32_MC20_STATUS	Package	
452H	1106	IA32_MC20_ADDR	Package	
453H	1107	IA32_MC20_MISC	Package	
454H	1108	IA32_MC21_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs." through Section 15.3.2.4, "IA32_MCi_MISC MSRs.". Bank MC21 reports MC error from the Intel QPI 2 module.
455H	1109	IA32_MC21_STATUS	Package	
456H	1110	IA32_MC21_ADDR	Package	
457H	1111	IA32_MC21_MISC	Package	
C81H	3201	IA32_L3_QOS_CFG	Package	Cache Allocation Technology Configuration (R/W)
		0		CAT Enable. Set 1 to enable Cache Allocation Technology
		63:1		Reserved.
See Table 2-19, Table 2-20, Table 2-28, and Table 2-29 for other MSR definitions applicable to processors with CPUID signature 06_45H.				

NOTES:

1. An override configuration lower than the factory-set configuration is always supported. An override configuration higher than the factory-set configuration is dependent on features specific to the processor and the platform.

2.16 MSRS IN THE 6TH GENERATION INTEL® CORE™ PROCESSORS, INTEL® XEON® PROCESSOR SCALABLE FAMILY, 7TH GENERATION INTEL® CORE™ PROCESSORS, AND FUTURE INTEL® CORE™ PROCESSORS

6th generation Intel® Core™ processors and the Intel® Xeon® Processor Scalable Family are based on the Skylake microarchitecture and have CPUID DisplayFamily_DisplayModel signatures of 06_4EH, 06_5EH, and 06_55H. 7th Generation Intel® Core™ processors are based on the Kaby Lake microarchitecture and have CPUID DisplayFamily_DisplayModel signatures of 06_8EH and 06_9EH. Future Intel® Core™ processors are based on Cannon Lake microarchitecture and have a CPUID DisplayFamily_DisplayModel signature of 06_66H. These processors support the MSR interfaces listed in Table 2-19, Table 2-20, Table 2-24, Table 2-28, Table 2-34, Table 2-38, and Table 2-39. For an MSR listed in Table 2-38 that also appears in the model-specific tables of prior generations, Table 2-38 supercede prior generation tables.

The notation of "Platform" in the Scope column (with respect to MSR_PLATFORM_ENERGY_COUNTER and MSR_PLATFORM_POWER_LIMIT) is limited to the power-delivery domain and the specifics of the power delivery integration may vary by platform vendor's implementation.

Table 2-38. Additional MSRs Supported by 6th Generation Intel® Core™ Processors and the Intel® Xeon® Processor Scalable Family Based on Skylake Microarchitecture, 7th Generation Intel® Core™ Processors Based on Kaby Lake Microarchitecture, and Future Intel® Core™ Processors Based on Cannon Lake Microarchitecture

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
3AH	58	IA32_FEATURE_CONTROL	Thread	**Control Features in Intel 64 Processor (R/W)** See Table 2-2.
		0		**Lock (R/WL)**
		1		**Enable VMX inside SMX operation (R/WL)**
		2		**Enable VMX outside SMX operation (R/WL)**
		14:8		**SENTER local functions enables (R/WL)**
		15		**SENTER global functions enable (R/WL)**
		18		**SGX global functions enable (R/WL)**
		20		**LMCE_ON (R/WL)**
		63:21		Reserved.
FEH	254	IA32_MTRRCAP	Thread	**MTRR Capality (RO, Architectural).** See Table 2-2
19CH	412	IA32_THERM_STATUS	Core	**Thermal Monitor Status (R/W)** See Table 2-2.
		0		**Thermal status (RO)** See Table 2-2.
		1		**Thermal status log (R/WC0)** See Table 2-2.
		2		**PROTCHOT # or FORCEPR# status (RO)** See Table 2-2.
		3		**PROTCHOT # or FORCEPR# log (R/WC0)** See Table 2-2.
		4		**Critical Temperature status (RO)** See Table 2-2.

Table 2-38. Additional MSRs Supported by 6th Generation Intel® Core™ Processors and the Intel® Xeon® Processor Scalable Family Based on Skylake Microarchitecture, 7th Generation Intel® Core™ Processors Based on Kaby Lake Microarchitecture, and Future Intel® Core™ Processors Based on Cannon Lake Microarchitecture

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
		5		**Critical Temperature status log (R/WC0)** See Table 2-2.
		6		**Thermal threshold #1 status (RO)** See Table 2-2.
		7		**Thermal threshold #1 log (R/WC0)** See Table 2-2.
		8		**Thermal threshold #2 status (RO)** See Table 2-2.
		9		**Thermal threshold #2 log (R/WC0)** See Table 2-2.
		10		**Power Limitation status (RO)** See Table 2-2.
		11		**Power Limitation log (R/WC0)** See Table 2-2.
		12		**Current Limit status (RO)** See Table 2-2.
		13		**Current Limit log (R/WC0)** See Table 2-2.
		14		**Cross Domain Limit status (RO)** See Table 2-2.
		15		**Cross Domain Limit log (R/WC0)** See Table 2-2.
		22:16		**Digital Readout (RO)** See Table 2-2.
		26:23		Reserved.
		30:27		**Resolution in degrees Celsius (RO)** See Table 2-2.
		31		**Reading Valid (RO)** See Table 2-2.
		63:32		Reserved.
1ADH	429	MSR_TURBO_RATIO_LIMIT	Package	**Maximum Ratio Limit of Turbo Mode** RO if MSR_PLATFORM_INFO.[28] = 0, RW if MSR_PLATFORM_INFO.[28] = 1
		7:0	Package	**Maximum Ratio Limit for 1C** Maximum turbo ratio limit of 1 core active.
		15:8	Package	**Maximum Ratio Limit for 2C** Maximum turbo ratio limit of 2 core active.

Table 2-38. Additional MSRs Supported by 6th Generation Intel® Core™ Processors and the Intel® Xeon® Processor Scalable Family Based on Skylake Microarchitecture, 7th Generation Intel® Core™ Processors Based on Kaby Lake Microarchitecture, and Future Intel® Core™ Processors Based on Cannon Lake Microarchitecture

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
		23:16	Package	**Maximum Ratio Limit for 3C** Maximum turbo ratio limit of 3 core active.
		31:24	Package	**Maximum Ratio Limit for 4C** Maximum turbo ratio limit of 4 core active.
		63:32		Reserved.
1C9H	457	MSR_LASTBRANCH_TOS	Thread	**Last Branch Record Stack TOS (R/W)** Contains an index (bits 0-4) that points to the MSR containing the most recent branch record.
1FCH	508	MSR_POWER_CTL	Core	Power Control Register. See http://biosbits.org.
		0		Reserved.
		1	Package	**C1E Enable (R/W)** When set to '1', will enable the CPU to switch to the Minimum Enhanced Intel SpeedStep Technology operating point when all execution cores enter MWAIT (C1).
		18:2		Reserved.
		19		**Disable Race to Halt Optimization (R/W)** Setting this bit disables the Race to Halt optimization and avoid this optimization limitation to execute below the most efficient frequency ratio. Default value is 0 for processors that support Race to Halt optimization. Default value is 1 for processors that do not support Race to Halt optimization.
		20		**Disable Energy Efficiency Optimization (R/W)** Setting this bit disables the P-States energy efficiency optimization. Default value is 0. Disable/enable the energy efficiency optimization in P-State legacy mode (when IA32_PM_ENABLE[HWP_ENABLE] = 0), has an effect only in the turbo range or into PERF_MIN_CTL value if it is not zero set. In HWP mode (IA32_PM_ENABLE[HWP_ENABLE] == 1), has an effect between the OS desired or OS maximize to the OS minimize performance setting.
		63:21		Reserved.
300H	768	MSR_SGXOWNEREPOCH0	Package	**Lower 64 Bit CR_SGXOWNEREPOCH.** Writes do not update CR_SGXOWNEREPOCH if CPUID.(EAX=12H, ECX=0):EAX.SGX1 is 1 on any thread in the package.
		63:0		Lower 64 bits of an 128-bit external entropy value for key derivation of an enclave.
301H	768	MSR_SGXOWNEREPOCH1	Package	**Upper 64 Bit CR_SGXOWNEREPOCH.** Writes do not update CR_SGXOWNEREPOCH if CPUID.(EAX=12H, ECX=0):EAX.SGX1 is 1 on any thread in the package.
		63:0		Upper 64 bits of an 128-bit external entropy value for key derivation of an enclave.

Table 2-38. Additional MSRs Supported by 6th Generation Intel® Core™ Processors and the Intel® Xeon® Processor Scalable Family Based on Skylake Microarchitecture, 7th Generation Intel® Core™ Processors Based on Kaby Lake Microarchitecture, and Future Intel® Core™ Processors Based on Cannon Lake Microarchitecture

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
38EH	910	IA32_PERF_GLOBAL_ STATUS		See Table 2-2. See Section 18.2.4, "Architectural Performance Monitoring Version 4."
		0	Thread	**Ovf_PMC0**
		1	Thread	**Ovf_PMC1**
		2	Thread	**Ovf_PMC2**
		3	Thread	**Ovf_PMC3**
		4	Thread	**Ovf_PMC4 (if CPUID.0AH:EAX[15:8] > 4)**
		5	Thread	**Ovf_PMC5 (if CPUID.0AH:EAX[15:8] > 5)**
		6	Thread	**Ovf_PMC6 (if CPUID.0AH:EAX[15:8] > 6)**
		7	Thread	**Ovf_PMC7 (if CPUID.0AH:EAX[15:8] > 7)**
		31:8		Reserved.
		32	Thread	**Ovf_FixedCtr0**
		33	Thread	**Ovf_FixedCtr1**
		34	Thread	**Ovf_FixedCtr2**
		54:35		Reserved.
		55	Thread	**Trace_ToPA_PMI.**
		57:56		Reserved.
		58	Thread	**LBR_Frz.**
		59	Thread	**CTR_Frz.**
		60	Thread	**ASCI.**
		61	Thread	**Ovf_Uncore**
		62	Thread	**Ovf_BufDSSAVE**
		63	Thread	**CondChgd**
390H	912	IA32_PERF_GLOBAL_STAT US_RESET		See Table 2-2. See Section 18.2.4, "Architectural Performance Monitoring Version 4."
		0	Thread	**Set 1 to clear Ovf_PMC0**
		1	Thread	**Set 1 to clear Ovf_PMC1**
		2	Thread	**Set 1 to clear Ovf_PMC2**
		3	Thread	**Set 1 to clear Ovf_PMC3**
		4	Thread	**Set 1 to clear Ovf_PMC4 (if CPUID.0AH:EAX[15:8] > 4)**
		5	Thread	**Set 1 to clear Ovf_PMC5 (if CPUID.0AH:EAX[15:8] > 5)**
		6	Thread	**Set 1 to clear Ovf_PMC6 (if CPUID.0AH:EAX[15:8] > 6)**
		7	Thread	**Set 1 to clear Ovf_PMC7 (if CPUID.0AH:EAX[15:8] > 7)**
		31:8		Reserved.
		32	Thread	**Set 1 to clear Ovf_FixedCtr0**
		33	Thread	**Set 1 to clear Ovf_FixedCtr1**

Table 2-38. Additional MSRs Supported by 6th Generation Intel® Core™ Processors and the Intel® Xeon® Processor Scalable Family Based on Skylake Microarchitecture, 7th Generation Intel® Core™ Processors Based on Kaby Lake Microarchitecture, and Future Intel® Core™ Processors Based on Cannon Lake Microarchitecture

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
		34	Thread	**Set 1 to clear Ovf_FixedCtr2**
		54:35		Reserved.
		55	Thread	**Set 1 to clear Trace_ToPA_PMI.**
		57:56		Reserved.
		58	Thread	**Set 1 to clear LBR_Frz.**
		59	Thread	**Set 1 to clear CTR_Frz.**
		60	Thread	**Set 1 to clear ASCI.**
		61	Thread	**Set 1 to clear Ovf_Uncore**
		62	Thread	**Set 1 to clear Ovf_BufDSSAVE**
		63	Thread	**Set 1 to clear CondChgd**
391H	913	IA32_PERF_GLOBAL_STATUS_SET		See Table 2-2. See Section 18.2.4, "Architectural Performance Monitoring Version 4."
		0	Thread	**Set 1 to cause Ovf_PMC0 = 1**
		1	Thread	**Set 1 to cause Ovf_PMC1 = 1**
		2	Thread	**Set 1 to cause Ovf_PMC2 = 1**
		3	Thread	**Set 1 to cause Ovf_PMC3 = 1**
		4	Thread	**Set 1 to cause Ovf_PMC4=1 (if CPUID.0AH:EAX[15:8] > 4)**
		5	Thread	**Set 1 to cause Ovf_PMC5=1 (if CPUID.0AH:EAX[15:8] > 5)**
		6	Thread	**Set 1 to cause Ovf_PMC6=1 (if CPUID.0AH:EAX[15:8] > 6)**
		7	Thread	**Set 1 to cause Ovf_PMC7=1 (if CPUID.0AH:EAX[15:8] > 7)**
		31:8		Reserved.
		32	Thread	**Set 1 to cause Ovf_FixedCtr0 = 1**
		33	Thread	**Set 1 to cause Ovf_FixedCtr1 = 1**
		34	Thread	**Set 1 to cause Ovf_FixedCtr2 = 1**
		54:35		Reserved.
		55	Thread	**Set 1 to cause Trace_ToPA_PMI = 1**
		57:56		Reserved.
		58	Thread	**Set 1 to cause LBR_Frz = 1**
		59	Thread	**Set 1 to cause CTR_Frz = 1**
		60	Thread	**Set 1 to cause ASCI = 1**
		61	Thread	**Set 1 to cause Ovf_Uncore**
		62	Thread	**Set 1 to cause Ovf_BufDSSAVE**
		63		Reserved.
392H	913	IA32_PERF_GLOBAL_INUSE		See Table 2-2.
3F7H	1015	MSR_PEBS_FRONTEND	Thread	**FrontEnd Precise Event Condition Select (R/W)**
		2:0		**Event Code Select**

Table 2-38. Additional MSRs Supported by 6th Generation Intel® Core™ Processors and the Intel® Xeon® Processor Scalable Family Based on Skylake Microarchitecture, 7th Generation Intel® Core™ Processors Based on Kaby Lake Microarchitecture, and Future Intel® Core™ Processors Based on Cannon Lake Microarchitecture

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
		3		Reserved.
		4		**Event Code Select High**
		7:5		Reserved.
		19:8		**IDQ_Bubble_Length Specifier**
		22:20		**IDQ_Bubble_Width Specifier**
		63:23		**Reserved**
500H	1280	IA32_SGX_SVN_STATUS	Thread	**Status and SVN Threshold of SGX Support for ACM (RO).**
		0		**Lock.** See Section 41.11.3, "Interactions with Authenticated Code Modules (ACMs)"
		15:1		Reserved.
		23:16		**SGX_SVN_SINIT.** See Section 41.11.3, "Interactions with Authenticated Code Modules (ACMs)"
		63:24		Reserved.
560H	1376	IA32_RTIT_OUTPUT_BASE	Thread	**Trace Output Base Register (R/W).** See Table 2-2.
561H	1377	IA32_RTIT_OUTPUT_MASK_PTRS	Thread	**Trace Output Mask Pointers Register (R/W).** See Table 2-2.
570H	1392	IA32_RTIT_CTL	Thread	**Trace Control Register (R/W)**
		0		**TraceEn**
		1		**CYCEn**
		2		**OS**
		3		**User**
		6:4		Reserved, MBZ
		7		**CR3 filter**
		8		**ToPA; writing 0 will #GP if also setting TraceEn**
		9		**MTCEn**
		10		**TSCEn**
		11		**DisRETC**
		12		Reserved, MBZ
		13		**BranchEn**
		17:14		**MTCFreq**
		18		Reserved, MBZ
		22:19		**CYCThresh**
		23		Reserved, MBZ
		27:24		**PSBFreq**
		31:28		Reserved, MBZ
		35:32		**ADDR0_CFG**

Table 2-38. Additional MSRs Supported by 6th Generation Intel® Core™ Processors and the Intel® Xeon® Processor Scalable Family Based on Skylake Microarchitecture, 7th Generation Intel® Core™ Processors Based on Kaby Lake Microarchitecture, and Future Intel® Core™ Processors Based on Cannon Lake Microarchitecture

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
		39:36		**ADDR1_CFG**
		63:40		Reserved, MBZ.
571H	1393	IA32_RTIT_STATUS	Thread	**Tracing Status Register (R/W)**
		0		**FilterEn**, writes ignored.
		1		**ContexEn**, writes ignored.
		2		**TriggerEn**, writes ignored.
		3		Reserved
		4		**Error (R/W)**
		5		**Stopped**
		31:6		Reserved. MBZ
		48:32		**PacketByteCnt**
		63:49		Reserved, MBZ.
572H	1394	IA32_RTIT_CR3_MATCH	Thread	**Trace Filter CR3 Match Register (R/W)**
		4:0		Reserved
		63:5		CR3[63:5] value to match
580H	1408	IA32_RTIT_ADDR0_A	Thread	**Region 0 Start Address (R/W)**
		63:0		See Table 2-2.
581H	1409	IA32_RTIT_ADDR0_B	Thread	**Region 0 End Address (R/W)**
		63:0		See Table 2-2.
582H	1410	IA32_RTIT_ADDR1_A	Thread	**Region 1 Start Address (R/W)**
		63:0		See Table 2-2.
583H	1411	IA32_RTIT_ADDR1_B	Thread	**Region 1 End Address (R/W)**
		63:0		See Table 2-2.
639H	1593	MSR_PP0_ENERGY_STATUS	Package	**PP0 Energy Status (R/O)** See Section 14.9.4, "PP0/PP1 RAPL Domains."
64DH	1613	MSR_PLATFORM_ENERGY_COUNTER	Platform*	Platform Energy Counter. (R/O). This MSR is valid only if both platform vendor hardware implementation and BIOS enablement support it. This MSR will read 0 if not valid.
		31:0		Total energy consumed by all devices in the platform that receive power from integrated power delivery mechanism, Included platform devices are processor cores, SOC, memory, add-on or peripheral devices that get powered directly from the platform power delivery means. The energy units are specified in the MSR_RAPL_POWER_UNIT.Enery_Status_Unit.
		63:32		Reserved.
64EH	1614	MSR_PPERF	Thread	Productive Performance Count. (R/O).
		63:0		Hardware's view of workload scalability. See Section 14.4.5.1

Table 2-38. Additional MSRs Supported by 6th Generation Intel® Core™ Processors and the Intel® Xeon® Processor Scalable Family Based on Skylake Microarchitecture, 7th Generation Intel® Core™ Processors Based on Kaby Lake Microarchitecture, and Future Intel® Core™ Processors Based on Cannon Lake Microarchitecture

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
64FH	1615	MSR_CORE_PERF_LIMIT_REASONS	Package	**Indicator of Frequency Clipping in Processor Cores (R/W) (frequency refers to processor core frequency)**
		0		**PROCHOT Status (R0)** When set, frequency is reduced below the operating system request due to assertion of external PROCHOT.
		1		**Thermal Status (R0)** When set, frequency is reduced below the operating system request due to a thermal event.
		3:2		Reserved.
		4		**Residency State Regulation Status (R0)** When set, frequency is reduced below the operating system request due to residency state regulation limit.
		5		**Running Average Thermal Limit Status (R0)** When set, frequency is reduced below the operating system request due to Running Average Thermal Limit (RATL).
		6		**VR Therm Alert Status (R0)** When set, frequency is reduced below the operating system request due to a thermal alert from a processor Voltage Regulator (VR).
		7		**VR Therm Design Current Status (R0)** When set, frequency is reduced below the operating system request due to VR thermal design current limit.
		8		**Other Status (R0)** When set, frequency is reduced below the operating system request due to electrical or other constraints.
		9		**Reserved**
		10		**Package/Platform-Level Power Limiting PL1 Status (R0)** When set, frequency is reduced below the operating system request due to package/platform-level power limiting PL1.
		11		**Package/Platform-Level PL2 Power Limiting Status (R0)** When set, frequency is reduced below the operating system request due to package/platform-level power limiting PL2/PL3.
		12		**Max Turbo Limit Status (R0)** When set, frequency is reduced below the operating system request due to multi-core turbo limits.
		13		**Turbo Transition Attenuation Status (R0)** When set, frequency is reduced below the operating system request due to Turbo transition attenuation. This prevents performance degradation due to frequent operating ratio changes.
		15:14		**Reserved**

Table 2-38. Additional MSRs Supported by 6th Generation Intel® Core™ Processors and the Intel® Xeon® Processor Scalable Family Based on Skylake Microarchitecture, 7th Generation Intel® Core™ Processors Based on Kaby Lake Microarchitecture, and Future Intel® Core™ Processors Based on Cannon Lake Microarchitecture

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
		16		**PROCHOT Log** When set, indicates that the PROCHOT Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.
		17		**Thermal Log** When set, indicates that the Thermal Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.
		19:18		Reserved.
		20		**Residency State Regulation Log** When set, indicates that the Residency State Regulation Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.
		21		**Running Average Thermal Limit Log** When set, indicates that the RATL Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.
		22		**VR Therm Alert Log** When set, indicates that the VR Therm Alert Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.
		23		**VR Thermal Design Current Log** When set, indicates that the VR TDC Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.
		24		**Other Log** When set, indicates that the Other Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.
		25		**Reserved**
		26		**Package/Platform-Level PL1 Power Limiting Log** When set, indicates that the Package or Platform Level PL1 Power Limiting Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.
		27		**Package/Platform-Level PL2 Power Limiting Log** When set, indicates that the Package or Platform Level PL2/PL3 Power Limiting Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.

Table 2-38. Additional MSRs Supported by 6th Generation Intel® Core™ Processors and the Intel® Xeon® Processor Scalable Family Based on Skylake Microarchitecture, 7th Generation Intel® Core™ Processors Based on Kaby Lake Microarchitecture, and Future Intel® Core™ Processors Based on Cannon Lake Microarchitecture

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
		28		**Max Turbo Limit Log** When set, indicates that the Max Turbo Limit Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.
		29		**Turbo Transition Attenuation Log** When set, indicates that the Turbo Transition Attenuation Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.
		63:30		Reserved.
652H	1618	MSR_PKG_HDC_CONFIG	Package	**HDC Configuration (R/W).**
		2:0		**PKG_Cx_Monitor.** **Configures Package Cx state threshold for MSR_PKG_HDC_DEEP_RESIDENCY**
		63: 3		**Reserved**
653H	1619	MSR_CORE_HDC_ RESIDENCY	Core	Core HDC Idle Residency. (R/O).
		63:0		Core_Cx_Duty_Cycle_Cnt.
655H	1621	MSR_PKG_HDC_SHALLOW_ RESIDENCY	Package	Accumulate the cycles the package was in C2 state and at least one logical processor was in forced idle. (R/O).
		63:0		Pkg_C2_Duty_Cycle_Cnt.
656H	1622	MSR_PKG_HDC_DEEP_ RESIDENCY	Package	Package Cx HDC Idle Residency. (R/O).
		63:0		Pkg_Cx_Duty_Cycle_Cnt.
658H	1624	MSR_WEIGHTED_CORE_C0	Package	Core-count Weighted C0 Residency. (R/O).
		63:0		Increment at the same rate as the TSC. The increment each cycle is weighted by the number of processor cores in the package that reside in C0. If N cores are simultaneously in C0, then each cycle the counter increments by N.
659H	1625	MSR_ANY_CORE_C0	Package	Any Core C0 Residency. (R/O)
		63:0		Increment at the same rate as the TSC. The increment each cycle is one if any processor core in the package is in C0.
65AH	1626	MSR_ANY_GFXE_C0	Package	Any Graphics Engine C0 Residency. (R/O)
		63:0		Increment at the same rate as the TSC. The increment each cycle is one if any processor graphic device's compute engines are in C0.
65BH	1627	MSR_CORE_GFXE_OVERLAP_C0	Package	Core and Graphics Engine Overlapped C0 Residency. (R/O)
		63:0		Increment at the same rate as the TSC. The increment each cycle is one if at least one compute engine of the processor graphics is in C0 and at least one processor core in the package is also in C0.

Table 2-38. Additional MSRs Supported by 6th Generation Intel® Core™ Processors and the Intel® Xeon® Processor Scalable Family Based on Skylake Microarchitecture, 7th Generation Intel® Core™ Processors Based on Kaby Lake Microarchitecture, and Future Intel® Core™ Processors Based on Cannon Lake Microarchitecture

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
65CH	1628	MSR_PLATFORM_POWER_LIMIT	Platform*	**Platform Power Limit Control (R/W-L)** Allows platform BIOS to limit power consumption of the platform devices to the specified values. The Long Duration power consumption is specified via Platform_Power_Limit_1 and Platform_Power_Limit_1_Time. The Short Duration power consumption limit is specified via the Platform_Power_Limit_2 with duration chosen by the processor. The processor implements an exponential-weighted algorithm in the placement of the time windows.
		14:0		Platform Power Limit #1. Average Power limit value which the platform must not exceed over a time window as specified by Power_Limit_1_TIME field. The default value is the Thermal Design Power (TDP) and varies with product skus. The unit is specified in MSR_RAPLPOWER_UNIT.
		15		Enable Platform Power Limit #1. When set, enables the processor to apply control policy such that the platform power does not exceed Platform Power limit #1 over the time window specified by Power Limit #1 Time Window.
		16		Platform Clamping Limitation #1. When set, allows the processor to go below the OS requested P states in order to maintain the power below specified Platform Power Limit #1 value. This bit is writeable only when CPUID (EAX=6):EAX[4] is set.
		23:17		Time Window for Platform Power Limit #1. Specifies the duration of the time window over which Platform Power Limit 1 value should be maintained for sustained long duration. This field is made up of two numbers from the following equation: Time Window = (float) ((1+(X/4))*(2^Y)), where: X = POWER_LIMIT_1_TIME[23:22] Y = POWER_LIMIT_1_TIME[21:17]. The maximum allowed value in this field is defined in MSR_PKG_POWER_INFO[PKG_MAX_WIN]. The default value is 0DH, The unit is specified in MSR_RAPLPOWER_UNIT[Time Unit].
		31:24		Reserved
		46:32		Platform Power Limit #2. Average Power limit value which the platform must not exceed over the Short Duration time window chosen by the processor. The recommended default value is 1.25 times the Long Duration Power Limit (i.e. Platform Power Limit # 1)

Table 2-38. Additional MSRs Supported by 6th Generation Intel® Core™ Processors and the Intel® Xeon® Processor Scalable Family Based on Skylake Microarchitecture, 7th Generation Intel® Core™ Processors Based on Kaby Lake Microarchitecture, and Future Intel® Core™ Processors Based on Cannon Lake Microarchitecture

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
		47		Enable Platform Power Limit #2. When set, enables the processor to apply control policy such that the platform power does not exceed Platform Power limit #2 over the Short Duration time window.
		48		Platform Clamping Limitation #2. When set, allows the processor to go below the OS requested P states in order to maintain the power below specified Platform Power Limit #2 value.
		62:49		Reserved
		63		Lock. Setting this bit will lock all other bits of this MSR until system RESET.
690H	1680	MSR_ LASTBRANCH_16_FROM_IP	Thread	**Last Branch Record 16 From IP (R/W)** One of 32 triplets of last branch record registers on the last branch record stack. This part of the stack contains pointers to the source instruction. See also: ▪ Last Branch Record Stack TOS at 1C9H ▪ Section 17.12
691H	1681	MSR_ LASTBRANCH_17_FROM_IP	Thread	**Last Branch Record 17 From IP (R/W)** See description of MSR_LASTBRANCH_0_FROM_IP.
692H	1682	MSR_ LASTBRANCH_18_FROM_IP	Thread	**Last Branch Record 18 From IP (R/W)** See description of MSR_LASTBRANCH_0_FROM_IP.
693H	1683	MSR_ LASTBRANCH_19_FROM_IP	Thread	**Last Branch Record 19From IP (R/W)** See description of MSR_LASTBRANCH_0_FROM_IP.
694H	1684	MSR_ LASTBRANCH_20_FROM_IP	Thread	**Last Branch Record 20 From IP (R/W)** See description of MSR_LASTBRANCH_0_FROM_IP.
695H	1685	MSR_ LASTBRANCH_21_FROM_IP	Thread	**Last Branch Record 21 From IP (R/W)** See description of MSR_LASTBRANCH_0_FROM_IP.
696H	1686	MSR_ LASTBRANCH_22_FROM_IP	Thread	**Last Branch Record 22 From IP (R/W)** See description of MSR_LASTBRANCH_0_FROM_IP.
697H	1687	MSR_ LASTBRANCH_23_FROM_IP	Thread	**Last Branch Record 23 From IP (R/W)** See description of MSR_LASTBRANCH_0_FROM_IP.
698H	1688	MSR_ LASTBRANCH_24_FROM_IP	Thread	**Last Branch Record 24 From IP (R/W)** See description of MSR_LASTBRANCH_0_FROM_IP.
699H	1689	MSR_ LASTBRANCH_25_FROM_IP	Thread	**Last Branch Record 25 From IP (R/W)** See description of MSR_LASTBRANCH_0_FROM_IP.
69AH	1690	MSR_ LASTBRANCH_26_FROM_IP	Thread	**Last Branch Record 26 From IP (R/W)** See description of MSR_LASTBRANCH_0_FROM_IP.
69BH	1691	MSR_ LASTBRANCH_27_FROM_IP	Thread	**Last Branch Record 27 From IP (R/W)** See description of MSR_LASTBRANCH_0_FROM_IP.

Table 2-38. Additional MSRs Supported by 6th Generation Intel® Core™ Processors and the Intel® Xeon® Processor Scalable Family Based on Skylake Microarchitecture, 7th Generation Intel® Core™ Processors Based on Kaby Lake Microarchitecture, and Future Intel® Core™ Processors Based on Cannon Lake Microarchitecture

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
69CH	1692	MSR_ LASTBRANCH_28_FROM_IP	Thread	**Last Branch Record 28 From IP (R/W)** See description of MSR_LASTBRANCH_0_FROM_IP.
69DH	1693	MSR_ LASTBRANCH_29_FROM_IP	Thread	**Last Branch Record 29 From IP (R/W)** See description of MSR_LASTBRANCH_0_FROM_IP.
69EH	1694	MSR_ LASTBRANCH_30_FROM_IP	Thread	**Last Branch Record 30 From IP (R/W)** See description of MSR_LASTBRANCH_0_FROM_IP.
69FH	1695	MSR_ LASTBRANCH_31_FROM_IP	Thread	**Last Branch Record 31 From IP (R/W)** See description of MSR_LASTBRANCH_0_FROM_IP.
6B0H	1712	MSR_GRAPHICS_PERF_LIMIT_REASONS	Package	**Indicator of Frequency Clipping in the Processor Graphics (R/W) (frequency refers to processor graphics frequency)**
		0		**PROCHOT Status (R0)** When set, frequency is reduced due to assertion of external PROCHOT.
		1		**Thermal Status (R0)** When set, frequency is reduced due to a thermal event.
		4:2		Reserved.
		5		**Running Average Thermal Limit Status (R0)** When set, frequency is reduced due to running average thermal limit.
		6		**VR Therm Alert Status (R0)** When set, frequency is reduced due to a thermal alert from a processor Voltage Regulator.
		7		**VR Thermal Design Current Status (R0)** When set, frequency is reduced due to VR TDC limit.
		8		**Other Status (R0)** When set, frequency is reduced due to electrical or other constraints.
		9		**Reserved**
		10		**Package/Platform-Level Power Limiting PL1 Status (R0)** When set, frequency is reduced due to package/platform-level power limiting PL1.
		11		**Package/Platform-Level PL2 Power Limiting Status (R0)** When set, frequency is reduced due to package/platform-level power limiting PL2/PL3.
		12		**Inefficient Operation Status (R0)** When set, processor graphics frequency is operating below target frequency.
		15:13		**Reserved**

Table 2-38. Additional MSRs Supported by 6th Generation Intel® Core™ Processors and the Intel® Xeon® Processor Scalable Family Based on Skylake Microarchitecture, 7th Generation Intel® Core™ Processors Based on Kaby Lake Microarchitecture, and Future Intel® Core™ Processors Based on Cannon Lake Microarchitecture

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
		16		**PROCHOT Log** When set, indicates that the PROCHOT Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.
		17		**Thermal Log** When set, indicates that the Thermal Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.
		20:18		Reserved.
		21		**Running Average Thermal Limit Log** When set, indicates that the RATL Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.
		22		**VR Therm Alert Log** When set, indicates that the VR Therm Alert Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.
		23		**VR Thermal Design Current Log** When set, indicates that the VR Therm Alert Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.
		24		**Other Log** When set, indicates that the OTHER Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.
		25		**Reserved**
		26		**Package/Platform-Level PL1 Power Limiting Log** When set, indicates that the Package/Platform Level PL1 Power Limiting Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.
		27		**Package/Platform-Level PL2 Power Limiting Log** When set, indicates that the Package/Platform Level PL2 Power Limiting Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.
		28		**Inefficient Operation Log** When set, indicates that the Inefficient Operation Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.
		63:29		Reserved.

Table 2-38. Additional MSRs Supported by 6th Generation Intel® Core™ Processors and the Intel® Xeon® Processor Scalable Family Based on Skylake Microarchitecture, 7th Generation Intel® Core™ Processors Based on Kaby Lake Microarchitecture, and Future Intel® Core™ Processors Based on Cannon Lake Microarchitecture

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
6B1H	1713	MSR_RING_PERF_LIMIT_REASONS	Package	**Indicator of Frequency Clipping in the Ring Interconnect (R/W)** **(frequency refers to ring interconnect in the uncore)**
		0		**PROCHOT Status (R0)** When set, frequency is reduced due to assertion of external PROCHOT.
		1		**Thermal Status (R0)** When set, frequency is reduced due to a thermal event.
		4:2		Reserved.
		5		**Running Average Thermal Limit Status (R0)** When set, frequency is reduced due to running average thermal limit.
		6		**VR Therm Alert Status (R0)** When set, frequency is reduced due to a thermal alert from a processor Voltage Regulator.
		7		**VR Thermal Design Current Status (R0)** When set, frequency is reduced due to VR TDC limit.
		8		**Other Status (R0)** When set, frequency is reduced due to electrical or other constraints.
		9		Reserved.
		10		**Package/Platform-Level Power Limiting PL1 Status (R0)** When set, frequency is reduced due to package/Platform-level power limiting PL1.
		11		**Package/Platform-Level PL2 Power Limiting Status (R0)** When set, frequency is reduced due to package/Platform-level power limiting PL2/PL3.
		15:12		**Reserved**
		16		**PROCHOT Log** When set, indicates that the PROCHOT Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.
		17		**Thermal Log** When set, indicates that the Thermal Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.
		20:18		Reserved.
		21		**Running Average Thermal Limit Log** When set, indicates that the RATL Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.

Table 2-38. Additional MSRs Supported by 6th Generation Intel® Core™ Processors and the Intel® Xeon® Processor Scalable Family Based on Skylake Microarchitecture, 7th Generation Intel® Core™ Processors Based on Kaby Lake Microarchitecture, and Future Intel® Core™ Processors Based on Cannon Lake Microarchitecture

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
		22		**VR Therm Alert Log** When set, indicates that the VR Therm Alert Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.
		23		**VR Thermal Design Current Log** When set, indicates that the VR Therm Alert Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.
		24		**Other Log** When set, indicates that the OTHER Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.
		25		**Reserved**
		26		**Package/Platform-Level PL1 Power Limiting Log** When set, indicates that the Package/Platform Level PL1 Power Limiting Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.
		27		**Package/Platform-Level PL2 Power Limiting Log** When set, indicates that the Package/Platform Level PL2 Power Limiting Status bit has asserted since the log bit was last cleared. This log bit will remain set until cleared by software writing 0.
		63:28		Reserved.
6D0H	1744	MSR_ LASTBRANCH_16_TO_IP	Thread	**Last Branch Record 16 To IP (R/W)** One of 32 triplets of last branch record registers on the last branch record stack. This part of the stack contains pointers to the **destination instruction**. See also: ▪ Last Branch Record Stack TOS at 1C9H ▪ Section 17.12
6D1H	1745	MSR_ LASTBRANCH_17_TO_IP	Thread	**Last Branch Record 17 To IP (R/W)** See description of MSR_LASTBRANCH_0_TO_IP.
6D2H	1746	MSR_ LASTBRANCH_18_TO_IP	Thread	**Last Branch Record 18 To IP (R/W)** See description of MSR_LASTBRANCH_0_TO_IP.
6D3H	1747	MSR_ LASTBRANCH_19_TO_IP	Thread	**Last Branch Record 19To IP (R/W)** See description of MSR_LASTBRANCH_0_TO_IP.
6D4H	1748	MSR_ LASTBRANCH_20_TO_IP	Thread	**Last Branch Record 20 To IP (R/W)** See description of MSR_LASTBRANCH_0_TO_IP.
6D5H	1749	MSR_ LASTBRANCH_21_TO_IP	Thread	**Last Branch Record 21 To IP (R/W)** See description of MSR_LASTBRANCH_0_TO_IP.
6D6H	1750	MSR_ LASTBRANCH_22_TO_IP	Thread	**Last Branch Record 22 To IP (R/W)** See description of MSR_LASTBRANCH_0_TO_IP.

Table 2-38. Additional MSRs Supported by 6th Generation Intel® Core™ Processors and the Intel® Xeon® Processor Scalable Family Based on Skylake Microarchitecture, 7th Generation Intel® Core™ Processors Based on Kaby Lake Microarchitecture, and Future Intel® Core™ Processors Based on Cannon Lake Microarchitecture

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
6D7H	1751	MSR_ LASTBRANCH_23_TO_IP	Thread	**Last Branch Record 23 To IP (R/W)** See description of MSR_LASTBRANCH_0_TO_IP.
6D8H	1752	MSR_ LASTBRANCH_24_TO_IP	Thread	**Last Branch Record 24 To IP (R/W)** See description of MSR_LASTBRANCH_0_TO_IP.
6D9H	1753	MSR_ LASTBRANCH_25_TO_IP	Thread	**Last Branch Record 25 To IP (R/W)** See description of MSR_LASTBRANCH_0_TO_IP.
6DAH	1754	MSR_ LASTBRANCH_26_TO_IP	Thread	**Last Branch Record 26 To IP (R/W)** See description of MSR_LASTBRANCH_0_TO_IP.
6DBH	1755	MSR_ LASTBRANCH_27_TO_IP	Thread	**Last Branch Record 27 To IP (R/W)** See description of MSR_LASTBRANCH_0_TO_IP.
6DCH	1756	MSR_ LASTBRANCH_28_TO_IP	Thread	**Last Branch Record 28 To IP (R/W)** See description of MSR_LASTBRANCH_0_TO_IP.
6DDH	1757	MSR_ LASTBRANCH_29_TO_IP	Thread	**Last Branch Record 29 To IP (R/W)** See description of MSR_LASTBRANCH_0_TO_IP.
6DEH	1758	MSR_ LASTBRANCH_30_TO_IP	Thread	**Last Branch Record 30 To IP (R/W)** See description of MSR_LASTBRANCH_0_TO_IP.
6DFH	1759	MSR_ LASTBRANCH_31_TO_IP	Thread	**Last Branch Record 31 To IP (R/W)** See description of MSR_LASTBRANCH_0_TO_IP.
770H	1904	IA32_PM_ENABLE	Package	See Section 14.4.2, "Enabling HWP"
771H	1905	IA32_HWP_CAPABILITIES	Thread	See Section 14.4.3, "HWP Performance Range and Dynamic Capabilities"
772H	1906	IA32_HWP_REQUEST_PKG	Package	See Section 14.4.4, "Managing HWP"
773H	1907	IA32_HWP_INTERRUPT	Thread	See Section 14.4.6, "HWP Notifications"
774H	1908	IA32_HWP_REQUEST	Thread	See Section 14.4.4, "Managing HWP"
		7:0		**Minimum Performance (R/W).**
		15:8		**Maximum Performance (R/W).**
		23:16		**Desired Performance (R/W).**
		31:24		**Energy/Performance Preference (R/W).**
		41:32		**Activity Window (R/W).**
		42		**Package Control (R/W).**
		63:43		Reserved.
777H	1911	IA32_HWP_STATUS	Thread	See Section 14.4.5, "HWP Feedback"
D90H	3472	IA32_BNDCFGS	Thread	See Table 2-2.
DA0H	3488	IA32_XSS	Thread	See Table 2-2.
DB0H	3504	IA32_PKG_HDC_CTL	Package	See Section 14.5.2, "Package level Enabling HDC"
DB1H	3505	IA32_PM_CTL1	Thread	See Section 14.5.3, "Logical-Processor Level HDC Control"
DB2H	3506	IA32_THREAD_STALL	Thread	See Section 14.5.4.1, "IA32_THREAD_STALL"

Table 2-38. Additional MSRs Supported by 6th Generation Intel® Core™ Processors and the Intel® Xeon® Processor Scalable Family Based on Skylake Microarchitecture, 7th Generation Intel® Core™ Processors Based on Kaby Lake Microarchitecture, and Future Intel® Core™ Processors Based on Cannon Lake Microarchitecture

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
DC0H	3520	MSR_LBR_INFO_0	Thread	**Last Branch Record 0 Additional Information (R/W)** One of 32 triplet of last branch record registers on the last branch record stack. This part of the stack contains flag, TSX-related and elapsed cycle information. See also: ▪ Last Branch Record Stack TOS at 1C9H ▪ Section 17.9.1, "LBR Stack."
DC1H	3521	MSR_LBR_INFO_1	Thread	**Last Branch Record 1 Additional Information (R/W)** See description of MSR_LBR_INFO_0.
DC2H	3522	MSR_LBR_INFO_2	Thread	**Last Branch Record 2 Additional Information (R/W)** See description of MSR_LBR_INFO_0.
DC3H	3523	MSR_LBR_INFO_3	Thread	**Last Branch Record 3 Additional Information (R/W)** See description of MSR_LBR_INFO_0.
DC4H	3524	MSR_LBR_INFO_4	Thread	**Last Branch Record 4 Additional Information (R/W)** See description of MSR_LBR_INFO_0.
DC5H	3525	MSR_LBR_INFO_5	Thread	**Last Branch Record 5 Additional Information (R/W)** See description of MSR_LBR_INFO_0.
DC6H	3526	MSR_LBR_INFO_6	Thread	**Last Branch Record 6 Additional Information (R/W)** See description of MSR_LBR_INFO_0.
DC7H	3527	MSR_LBR_INFO_7	Thread	**Last Branch Record 7 Additional Information (R/W)** See description of MSR_LBR_INFO_0.
DC8H	3528	MSR_LBR_INFO_8	Thread	**Last Branch Record 8 Additional Information (R/W)** See description of MSR_LBR_INFO_0.
DC9H	3529	MSR_LBR_INFO_9	Thread	**Last Branch Record 9 Additional Information (R/W)** See description of MSR_LBR_INFO_0.
DCAH	3530	MSR_LBR_INFO_10	Thread	**Last Branch Record 10 Additional Information (R/W)** See description of MSR_LBR_INFO_0.
DCBH	3531	MSR_LBR_INFO_11	Thread	**Last Branch Record 11 Additional Information (R/W)** See description of MSR_LBR_INFO_0.
DCCH	3532	MSR_LBR_INFO_12	Thread	**Last Branch Record 12 Additional Information (R/W)** See description of MSR_LBR_INFO_0.
DCDH	3533	MSR_LBR_INFO_13	Thread	**Last Branch Record 13 Additional Information (R/W)** See description of MSR_LBR_INFO_0.
DCEH	3534	MSR_LBR_INFO_14	Thread	**Last Branch Record 14 Additional Information (R/W)** See description of MSR_LBR_INFO_0.
DCFH	3535	MSR_LBR_INFO_15	Thread	**Last Branch Record 15 Additional Information (R/W)** See description of MSR_LBR_INFO_0.
DD0H	3536	MSR_LBR_INFO_16	Thread	**Last Branch Record 16 Additional Information (R/W)** See description of MSR_LBR_INFO_0.

Table 2-38. Additional MSRs Supported by 6th Generation Intel® Core™ Processors and the Intel® Xeon® Processor Scalable Family Based on Skylake Microarchitecture, 7th Generation Intel® Core™ Processors Based on Kaby Lake Microarchitecture, and Future Intel® Core™ Processors Based on Cannon Lake Microarchitecture

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
DD1H	3537	MSR_LBR_INFO_17	Thread	**Last Branch Record 17 Additional Information (R/W)** See description of MSR_LBR_INFO_0.
DD2H	3538	MSR_LBR_INFO_18	Thread	**Last Branch Record 18 Additional Information (R/W)** See description of MSR_LBR_INFO_0.
DD3H	3539	MSR_LBR_INFO_19	Thread	**Last Branch Record 19 Additional Information (R/W)** See description of MSR_LBR_INFO_0.
DD4H	3520	MSR_LBR_INFO_20	Thread	**Last Branch Record 20 Additional Information (R/W)** See description of MSR_LBR_INFO_0.
DD5H	3521	MSR_LBR_INFO_21	Thread	**Last Branch Record 21 Additional Information (R/W)** See description of MSR_LBR_INFO_0.
DD6H	3522	MSR_LBR_INFO_22	Thread	**Last Branch Record 22 Additional Information (R/W)** See description of MSR_LBR_INFO_0.
DD7H	3523	MSR_LBR_INFO_23	Thread	**Last Branch Record 23 Additional Information (R/W)** See description of MSR_LBR_INFO_0.
DD8H	3524	MSR_LBR_INFO_24	Thread	**Last Branch Record 24 Additional Information (R/W)** See description of MSR_LBR_INFO_0.
DD9H	3525	MSR_LBR_INFO_25	Thread	**Last Branch Record 25 Additional Information (R/W)** See description of MSR_LBR_INFO_0.
DDAH	3526	MSR_LBR_INFO_26	Thread	**Last Branch Record 26 Additional Information (R/W)** See description of MSR_LBR_INFO_0.
DDBH	3527	MSR_LBR_INFO_27	Thread	**Last Branch Record 27 Additional Information (R/W)** See description of MSR_LBR_INFO_0.
DDCH	3528	MSR_LBR_INFO_28	Thread	**Last Branch Record 28 Additional Information (R/W)** See description of MSR_LBR_INFO_0.
DDDH	3529	MSR_LBR_INFO_29	Thread	**Last Branch Record 29 Additional Information (R/W)** See description of MSR_LBR_INFO_0.
DDEH	3530	MSR_LBR_INFO_30	Thread	**Last Branch Record 30 Additional Information (R/W)** See description of MSR_LBR_INFO_0.
DDFH	3531	MSR_LBR_INFO_31	Thread	**Last Branch Record 31 Additional Information (R/W)** See description of MSR_LBR_INFO_0.

Table 2-39 lists the MSRs of uncore PMU for Intel processors with CPUID DisplayFamily_DisplayModel signatures of 06_4EH, 06_5EH, 06_8EH, 06_9EH, and 06_66H.

Table 2-39. Uncore PMU MSRs Supported by 6th Generation Intel® Core™ Processors, 7th Generation Intel® Core™ Processors, and Future Intel® Core™ Processors

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
394H	916	MSR_UNC_PERF_FIXED_ CTRL	Package	Uncore fixed counter control (R/W)
		19:0		Reserved
		20		Enable overflow propagation
		21		Reserved
		22		Enable counting
		63:23		Reserved.
395H	917	MSR_UNC_PERF_FIXED_ CTR	Package	Uncore fixed counter
		43:0		Current count
		63:44		Reserved.
396H	918	MSR_UNC_CBO_CONFIG	Package	Uncore C-Box configuration information (R/O)
		3:0		Specifies the number of C-Box units with programmable counters (including processor cores and processor graphics),
		63:4		Reserved.
3B0H	946	MSR_UNC_ARB_PERFCTR0	Package	Uncore Arb unit, performance counter 0
3B1H	947	MSR_UNC_ARB_PERFCTR1	Package	Uncore Arb unit, performance counter 1
3B2H	944	MSR_UNC_ARB_ PERFEVTSEL0	Package	Uncore Arb unit, counter 0 event select MSR
3B3H	945	MSR_UNC_ARB_ PERFEVTSEL1	Package	Uncore Arb unit, counter 1 event select MSR
700H	1792	MSR_UNC_CBO_0_ PERFEVTSEL0	Package	Uncore C-Box 0, counter 0 event select MSR
701H	1793	MSR_UNC_CBO_0_ PERFEVTSEL1	Package	Uncore C-Box 0, counter 1 event select MSR
706H	1798	MSR_UNC_CBO_0_PERFCTR0	Package	Uncore C-Box 0, performance counter 0
707H	1799	MSR_UNC_CBO_0_PERFCTR1	Package	Uncore C-Box 0, performance counter 1
710H	1808	MSR_UNC_CBO_1_ PERFEVTSEL0	Package	Uncore C-Box 1, counter 0 event select MSR
711H	1809	MSR_UNC_CBO_1_ PERFEVTSEL1	Package	Uncore C-Box 1, counter 1 event select MSR
716H	1814	MSR_UNC_CBO_1_PERFCTR0	Package	Uncore C-Box 1, performance counter 0
717H	1815	MSR_UNC_CBO_1_PERFCTR1	Package	Uncore C-Box 1, performance counter 1
720H	1824	MSR_UNC_CBO_2_ PERFEVTSEL0	Package	Uncore C-Box 2, counter 0 event select MSR
721H	1825	MSR_UNC_CBO_2_ PERFEVTSEL1	Package	Uncore C-Box 2, counter 1 event select MSR

Table 2-39. Uncore PMU MSRs Supported by 6th Generation Intel® Core™ Processors, 7th Generation Intel® Core™ Processors, and Future Intel® Core™ Processors

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
726H	1830	MSR_UNC_CBO_2_PERFCTR0	Package	Uncore C-Box 2, performance counter 0
727H	1831	MSR_UNC_CBO_2_PERFCTR1	Package	Uncore C-Box 2, performance counter 1
730H	1840	MSR_UNC_CBO_3_ PERFEVTSEL0	Package	Uncore C-Box 3, counter 0 event select MSR
731H	1841	MSR_UNC_CBO_3_ PERFEVTSEL1	Package	Uncore C-Box 3, counter 1 event select MSR.
736H	1846	MSR_UNC_CBO_3_PERFCTR0	Package	Uncore C-Box 3, performance counter 0.
737H	1847	MSR_UNC_CBO_3_PERFCTR1	Package	Uncore C-Box 3, performance counter 1.
E01H	3585	MSR_UNC_PERF_GLOBAL_ CTRL	Package	Uncore PMU global control
		0		Slice 0 select
		1		Slice 1 select
		2		Slice 2 select
		3		Slice 3 select
		4		Slice 4select
		18:5		Reserved.
		29		Enable all uncore counters
		30		Enable wake on PMI
		31		Enable Freezing counter when overflow
		63:32		Reserved.
E02H	3586	MSR_UNC_PERF_GLOBAL_ STATUS	Package	Uncore PMU main status
		0		Fixed counter overflowed
		1		An ARB counter overflowed
		2		Reserved
		3		A CBox counter overflowed (on any slice)
		63:4		Reserved.

2.16.1 MSRs Specific to Future Intel® Core™ Processors

Table 2-40 lists additional MSRs for Future Intel Core processors with a CPUID DisplayFamily_DisplayModel signature of 06_66H. For an MSR listed in Table 2-40 that also appears in the model-specific tables of prior generations, Table 2-40 supersede prior generation tables.

Table 2-40. Additional MSRs Supported by Future Intel® Core™ Processors Based on Cannon Lake Microarchitecture

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
3AH	58	IA32_FEATURE_CONTROL	Thread	**Control Features in Intel 64 Processor (R/W)** See Table 2-2.
		0		**Lock (R/WL)**
		1		**Enable VMX inside SMX operation (R/WL)**
		2		**Enable VMX outside SMX operation (R/WL)**
		14:8		**SENTER local functions enables (R/WL)**
		15		**SENTER global functions enable (R/WL)**
		17		**SGX Launch Control Enable (R/WL)** This bit must be set to enable runtime reconfiguration of SGX Launch Control via IA32_SGXLEPUBKEYHASHn MSR. Available only if CPUID.(EAX=07H, ECX=0H): ECX[30] = 1.
		18		**SGX global functions enable (R/WL)**
		20		**LMCE_ON (R/WL)**
		63:21		Reserved.
350H	848	MSR_BR_DETECT_CTRL		Branch Monitoring Global Control (R/W)
		0		EnMonitoring Global enable for branch monitoring.
		1		EnExcept Enable branch monitoring event signaling on threshold trip. The branch monitoring event handler is signaled via the existing PMI signaling mechanism as programmed from the corresponding local APIC LVT entry.
		2		EnLBRFrz Enable LBR freeze on threshold trip. This will result in causing the LBR frozen bit 58 to be set in IA32_PERF_GLOBAL_STATUS when a triggering condition occurs and this bit is enabled.
		3		DisableInGuest When set to '1', branch monitoring, event triggering and LBR freeze actions are disabled when operating at VMX non-root operation.
		7:4		Reserved.
		17:8		WindowSize Window size defined by WindowCntSel. Values 0 – 1023 are supported.
		23:18		Reserved.

Table 2-40. Additional MSRs Supported by Future Intel® Core™ Processors Based on Cannon Lake Microarchitecture

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
		25:24		WindowCntSel Window event count select: '00 = Instructions retired. '01 = Branch instructions retired '10 = return instructions retired. '11 = Indirect branch instructions retired.
		26		CntAndMode When set to '1', overall branch monitoring event triggering condition is true only if both enabled counters' threshold conditions are true. When '0', the threshold tripping condition is true if either enabled counters' threshold is true.
		63:27		Reserved.
351H	849	MSR_BR_DETECT_STATUS		Branch Monitoring Global Status (R/W)
		0		Branch Monitoring Event Signaled When set to '1', Branch Monitoring event signaling is blocked until this bit is cleared by software.
		1		LBRsValid This status bit is set to '1' if the LBR state is considered valid for sampling by branch monitoring software.
		7:2		Reserved.
		8		CntrHit0 Branch monitoring counter #0 threshold hit. This status bit is sticky and once set requires clearing by software. Counter operation continues independent of the state of the bit.
		9		CntrHit1 Branch monitoring counter #1 threshold hit. This status bit is sticky and once set requires clearing by software. Counter operation continues independent of the state of the bit.
		15:10		Reserved. Reserved for additional branch monitoring counters threshold hit status.
		25:16		CountWindow The current value of window counter. The count value is frozen on a valid branch monitoring triggering condition. This is an 10-bit unsigned value.
		31:26		Reserved. Reserved for future extension of CountWindow.

Table 2-40. Additional MSRs Supported by Future Intel® Core™ Processors Based on Cannon Lake Microarchitecture

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
		39:32		Count0 The current value of counter 0 updated after each occurrence of the event being counted. The count value is frozen on a valid branch monitoring triggering condition (in which case CntrHit0 will also be set). This is an 8-bit signed value (2's complement). Heuristic events which only increment will saturate and freeze at maximum value 0xFF (256). RET-CALL event counter saturate at maximum value 0x7F (+127) and minimum value 0x80 (-128).
		47:40		Count1 The current value of counter 1 updated after each occurrence of the event being counted. The count value is frozen on a valid branch monitoring triggering condition (in which case CntrHit1 will also be set). This is an 8-bit signed value (2's complement). Heuristic events which only increment will saturate and freeze at maximum value 0xFF (256). RET-CALL event counter saturate at maximum value 0x7F (+127) and minimum value 0x80 (-128).
		63:48		Reserved.
354H - 355H	852 - 853	MSR_BR_DETECT_COUNTER_CONFIG_i		Branch Monitoring Detect Counter Configuration (R/W)
		0		CntrEn Enable counter.
		7:1		CntrEvSel Event select (other values #GP) '0000000 = RETs. '0000001 = RET-CALL bias. '0000010 = RET mispredicts. '0000011 = Branch (all) mispredicts. '0000100 = Indirect branch mispredicts. '0000101 = Far branch instructions.
		14:8		CntrThreshold Threshold (an unsigned value of 0 to 127 supported). The value 0 of counter threshold will result in event signaled after every instruction.
		15		MispredEventCnt Mispredict events counting behavior: '0 = Mispredict events are counted in a window. '1 = Mispredict events are counted based on a consecutive occurrence. CntrThreshold is treated as # of consecutive mispredicts. This control bit only applies to events specified by CntrEvSel that involve a prediction (0000010, 0000011, 0000100). Setting this bit for other events is ignored.
		63:16		Reserved.

Table 2-40. Additional MSRs Supported by Future Intel® Core™ Processors Based on Cannon Lake Microarchitecture

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
3F8H	1016	MSR_PKG_C3_RESIDENCY	Package	Package C3 Residency Counter.
		63:0		Always returns 0.
660H	1632	MSR_CORE_C1_RESIDENCY	Core	Core C1 Residency Counter (RO)
		63:0		Value since last reset for the Core C1 residency. Counter rate is the Max Non-Turbo frequency (same as TSC). This counter count in case that both of the core's thread are in idle state and at least one of the core's thread residency in C1 state or in one of its sub state. The counter is updated only after core C state exit. Note: Always reads 0 if core C1 is unsupported. A value of zero indicates that this processor does not support core C1 or never entered core C1 level state.

2.16.2 MSRs Specific to Intel® Xeon® Processor Scalable Family

Intel® Xeon® Processor Scalable Family (CPUID DisplayFamily_DisplayModel = 06_55H) support the MSRs listed in Table 2-41.

Table 2-41. MSRs Supported by Intel® Xeon® Processor Scalable Family with DisplayFamily_DisplayModel 06_55H

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
4EH	78	MSR_PPIN_CTL	Package	Protected Processor Inventory Number Enable Control (R/W)
		0		LockOut (R/WO) See Table 2-25.
		1		Enable_PPIN (R/W) See Table 2-25.
		63:2		Reserved.
4FH	79	MSR_PPIN	Package	Protected Processor Inventory Number (R/O)
		63:0		**Protected Processor Inventory Number (R/O)** See Table 2-25.
CEH	206	MSR_PLATFORM_INFO	Package	Platform Information; contains power management and other model specific features enumeration. See http://biosbits.org.
		7:0		Reserved.
		15:8	Package	**Maximum Non-Turbo Ratio (R/O)** See Table 2-25.
		22:16		Reserved.
		23	Package	**PPIN_CAP (R/O)** See Table 2-25.
		27:24		Reserved.
		28	Package	**Programmable Ratio Limit for Turbo Mode (R/O)** See Table 2-25.

Table 2-41. MSRs Supported by Intel® Xeon® Processor Scalable Family with DisplayFamily_DisplayModel 06_55H

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
		29	Package	**Programmable TDP Limit for Turbo Mode (R/O)** See Table 2-25.
		30	Package	**Programmable TJ OFFSET (R/O)** See Table 2-25.
		39:31		Reserved.
		47:40	Package	**Maximum Efficiency Ratio (R/O)** See Table 2-25.
		63:48		Reserved.
E2H	226	MSR_PKG_CST_CONFIG_ CONTROL	Core	**C-State Configuration Control (R/W)** Note: C-state values are processor specific C-state code names, unrelated to MWAIT extension C-state parameters or ACPI C-states. See http://biosbits.org.
		2:0		**Package C-State Limit (R/W)** Specifies the lowest processor-specific C-state code name (consuming the least power) for the package. The default is set as factory-configured package C-state limit. The following C-state code name encodings are supported: 000b: C0/C1 (no package C-state support) 001b: C2 010b: C6 (non-retention) 011b: C6 (retention) 111b: No Package C state limits. All C states supported by the processor are available.
		9:3		Reserved
		10		**I/O MWAIT Redirection Enable (R/W)**
		14:11		Reserved
		15		**CFG Lock (R/WO)**
		16		**Automatic C-State Conversion Enable (R/W)** If 1, the processor will convert HALT or MWAT(C1) to MWAIT(C6)
		24:17		Reserved
		25		**C3 State Auto Demotion Enable (R/W)**
		26		**C1 State Auto Demotion Enable (R/W)**
		27		**Enable C3 Undemotion (R/W)**
		28		**Enable C1 Undemotion (R/W)**
		29		**Package C State Demotion Enable (R/W)**
		30		**Package C State UnDemotion Enable (R/W)**
		63:31		Reserved
179H	377	IA32_MCG_CAP	Thread	**Global Machine Check Capability (R/O)**
		7:0		**Count**
		8		**MCG_CTL_P**

Table 2-41. MSRs Supported by Intel® Xeon® Processor Scalable Family with DisplayFamily_DisplayModel 06_55H

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
		9		**MCG_EXT_P**
		10		**MCP_CMCI_P**
		11		**MCG_TES_P**
		15:12		Reserved.
		23:16		**MCG_EXT_CNT**
		24		**MCG_SER_P**
		25		**MCG_EM_P**
		26		**MCG_ELOG_P**
		63:27		Reserved.
17DH	390	MSR_SMM_MCA_CAP	THREAD	**Enhanced SMM Capabilities (SMM-RO)** Reports SMM capability Enhancement. Accessible only while in SMM.
		57:0		**Reserved**
		58		**SMM_Code_Access_Chk (SMM-RO)** If set to 1 indicates that the SMM code access restriction is supported and a host-space interface available to SMM handler.
		59		**Long_Flow_Indication (SMM-RO)** If set to 1 indicates that the SMM long flow indicator is supported and a host-space interface available to SMM handler.
		63:60		Reserved
19CH	412	IA32_THERM_STATUS	Core	**Thermal Monitor Status (R/W)** See Table 2-2.
		0		**Thermal status (RO)** See Table 2-2.
		1		**Thermal status log (R/WC0)** See Table 2-2.
		2		**PROTCHOT # or FORCEPR# status (RO)** See Table 2-2.
		3		**PROTCHOT # or FORCEPR# log (R/WC0)** See Table 2-2.
		4		**Critical Temperature status (RO)** See Table 2-2.
		5		**Critical Temperature status log (R/WC0)** See Table 2-2.
		6		**Thermal threshold #1 status (RO)** See Table 2-2.
		7		**Thermal threshold #1 log (R/WC0)** See Table 2-2.

Table 2-41. MSRs Supported by Intel® Xeon® Processor Scalable Family with DisplayFamily_DisplayModel 06_55H

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
		8		**Thermal threshold #2 status (RO)** See Table 2-2.
		9		**Thermal threshold #2 log (R/WC0)** See Table 2-2.
		10		**Power Limitation status (RO)** See Table 2-2.
		11		**Power Limitation log (R/WC0)** See Table 2-2.
		12		**Current Limit status (RO)** See Table 2-2.
		13		**Current Limit log (R/WC0)** See Table 2-2.
		14		**Cross Domain Limit status (RO)** See Table 2-2.
		15		**Cross Domain Limit log (R/WC0)** See Table 2-2.
		22:16		**Digital Readout (RO)** See Table 2-2.
		26:23		Reserved.
		30:27		**Resolution in degrees Celsius (RO)** See Table 2-2.
		31		**Reading Valid (RO)** See Table 2-2.
		63:32		Reserved.
1A2H	418	MSR_ TEMPERATURE_TARGET	Package	Temperature Target
		15:0		Reserved.
		23:16		**Temperature Target (RO)** See Table 2-25.
		27:24		**TCC Activation Offset (R/W)** See Table 2-25.
		63:28		Reserved.
1ADH	429	MSR_TURBO_RATIO_LIMIT	Package	This register defines the ratio limits. RATIO[0:7] must be populated in ascending order. RATIO[i+1] must be less than or equal to RATIO[i]. Entries with RATIO[i] will be ignored. If any of the rules above are broken, the configuration is silently rejected. If the programmed ratio is: ▪ Above the fused ratio for that core count, it will be clipped to the fuse limits (assuming !OC). ▪ Below the min supported ratio, it will be clipped.

Table 2-41. MSRs Supported by Intel® Xeon® Processor Scalable Family with DisplayFamily_DisplayModel 06_55H

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
		7:0		RATIO_0 Defines ratio limits.
		15:8		RATIO_1 Defines ratio limits.
		23:16		RATIO_2 Defines ratio limits.
		31:24		RATIO_3 Defines ratio limits.
		39:32		RATIO_4 Defines ratio limits.
		47:40		RATIO_5 Defines ratio limits.
		55:48		RATIO_6 Defines ratio limits.
		63:56		RATIO_7 Defines ratio limits.
1AEH	430	MSR_TURBO_RATIO_LIMIT_CORES	Package	This register defines the active core ranges for each frequency point. NUMCORE[0:7] must be populated in ascending order. NUMCORE[i+1] must be greater than NUMCORE[i]. Entries with NUMCORE[i] == 0 will be ignored. The last valid entry must have NUMCORE >= the number of cores in the SKU. If any of the rules above are broken, the configuration is silently rejected.
		7:0		NUMCORE_0 Defines the active core ranges for each frequency point.
		15:8		NUMCORE_1 Defines the active core ranges for each frequency point.
		23:16		NUMCORE_2 Defines the active core ranges for each frequency point.
		31:24		NUMCORE_3 Defines the active core ranges for each frequency point.
		39:32		NUMCORE_4 Defines the active core ranges for each frequency point.
		47:40		NUMCORE_5 Defines the active core ranges for each frequency point.
		55:48		NUMCORE_6 Defines the active core ranges for each frequency point.
		63:56		NUMCORE_7 Defines the active core ranges for each frequency point.
280H	640	IA32_MC0_CTL2	Core	See Table 2-2.
281H	641	IA32_MC1_CTL2	Core	See Table 2-2.

Table 2-41. MSRs Supported by Intel® Xeon® Processor Scalable Family with DisplayFamily_DisplayModel 06_55H

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
282H	642	IA32_MC2_CTL2	Core	See Table 2-2.
283H	643	IA32_MC3_CTL2	Core	See Table 2-2.
284H	644	IA32_MC4_CTL2	Package	See Table 2-2.
285H	645	IA32_MC5_CTL2	Package	See Table 2-2.
286H	646	IA32_MC6_CTL2	Package	See Table 2-2.
287H	647	IA32_MC7_CTL2	Package	See Table 2-2.
288H	648	IA32_MC8_CTL2	Package	See Table 2-2.
289H	649	IA32_MC9_CTL2	Package	See Table 2-2.
28AH	650	IA32_MC10_CTL2	Package	See Table 2-2.
28BH	651	IA32_MC11_CTL2	Package	See Table 2-2.
28CH	652	IA32_MC12_CTL2	Package	See Table 2-2.
28DH	653	IA32_MC13_CTL2	Package	See Table 2-2.
28EH	654	IA32_MC14_CTL2	Package	See Table 2-2.
28FH	655	IA32_MC15_CTL2	Package	See Table 2-2.
290H	656	IA32_MC16_CTL2	Package	See Table 2-2.
291H	657	IA32_MC17_CTL2	Package	See Table 2-2.
292H	658	IA32_MC18_CTL2	Package	See Table 2-2.
293H	659	IA32_MC19_CTL2	Package	See Table 2-2.
400H	1024	IA32_MC0_CTL	Core	See Section 15.3.2.1, "IA32_MCi_CTL MSRs." through Section 15.3.2.4, "IA32_MCi_MISC MSRs.". Bank MC0 reports MC error from the IFU module.
401H	1025	IA32_MC0_STATUS	Core	
402H	1026	IA32_MC0_ADDR	Core	
403H	1027	IA32_MC0_MISC	Core	
404H	1028	IA32_MC1_CTL	Core	See Section 15.3.2.1, "IA32_MCi_CTL MSRs." through Section 15.3.2.4, "IA32_MCi_MISC MSRs.". Bank MC1 reports MC error from the DCU module.
405H	1029	IA32_MC1_STATUS	Core	
406H	1030	IA32_MC1_ADDR	Core	
407H	1031	IA32_MC1_MISC	Core	
408H	1032	IA32_MC2_CTL	Core	See Section 15.3.2.1, "IA32_MCi_CTL MSRs." through Section 15.3.2.4, "IA32_MCi_MISC MSRs.". Bank MC2 reports MC error from the DTLB module.
409H	1033	IA32_MC2_STATUS	Core	
40AH	1034	IA32_MC2_ADDR	Core	
40BH	1035	IA32_MC2_MISC	Core	
40CH	1036	IA32_MC3_CTL	Core	See Section 15.3.2.1, "IA32_MCi_CTL MSRs." through Section 15.3.2.4, "IA32_MCi_MISC MSRs.". Bank MC3 reports MC error from the MLC module.
40DH	1037	IA32_MC3_STATUS	Core	
40EH	1038	IA32_MC3_ADDR	Core	
40FH	1039	IA32_MC3_MISC	Core	

Table 2-41. MSRs Supported by Intel® Xeon® Processor Scalable Family with DisplayFamily_DisplayModel 06_55H

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
410H	1040	IA32_MC4_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs." through Section 15.3.2.4, "IA32_MCi_MISC MSRs.". Bank MC4 reports MC error from the PCU module.
411H	1041	IA32_MC4_STATUS	Package	
412H	1042	IA32_MC4_ADDR	Package	
413H	1043	IA32_MC4_MISC	Package	
414H	1044	IA32_MC5_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs." through Section 15.3.2.4, "IA32_MCi_MISC MSRs.". Bank MC5 reports MC error from a link interconnect module.
415H	1045	IA32_MC5_STATUS	Package	
416H	1046	IA32_MC5_ADDR	Package	
417H	1047	IA32_MC5_MISC	Package	
418H	1048	IA32_MC6_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs." through Section 15.3.2.4, "IA32_MCi_MISC MSRs.". Bank MC6 reports MC error from the integrated I/O module.
419H	1049	IA32_MC6_STATUS	Package	
41AH	1050	IA32_MC6_ADDR	Package	
41BH	1051	IA32_MC6_MISC	Package	
41CH	1052	IA32_MC7_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs." through Section 15.3.2.4, "IA32_MCi_MISC MSRs.". Bank MC7 reports MC error from the M2M 0.
41DH	1053	IA32_MC7_STATUS	Package	
41EH	1054	IA32_MC7_ADDR	Package	
41FH	1055	IA32_MC7_MISC	Package	
420H	1056	IA32_MC8_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs." through Section 15.3.2.4, "IA32_MCi_MISC MSRs.". Bank MC8 reports MC error from the M2M 1.
421H	1057	IA32_MC8_STATUS	Package	
422H	1058	IA32_MC8_ADDR	Package	
423H	1059	IA32_MC8_MISC	Package	
424H	1060	IA32_MC9_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs." through Section 15.3.2.4, "IA32_MCi_MISC MSRs.". Banks MC9 - MC11 report MC error from the CHA
425H	1061	IA32_MC9_STATUS	Package	
426H	1062	IA32_MC9_ADDR	Package	
427H	1063	IA32_MC9_MISC	Package	
428H	1064	IA32_MC10_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs." through Section 15.3.2.4, "IA32_MCi_MISC MSRs.". Banks MC9 - MC11 report MC error from the CHA.
429H	1065	IA32_MC10_STATUS	Package	
42AH	1066	IA32_MC10_ADDR	Package	
42BH	1067	IA32_MC10_MISC	Package	
42CH	1068	IA32_MC11_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs." through Section 15.3.2.4, "IA32_MCi_MISC MSRs.". Banks MC9 - MC11 report MC error from the CHA.
42DH	1069	IA32_MC11_STATUS	Package	
42EH	1070	IA32_MC11_ADDR	Package	
42FH	1071	IA32_MC11_MISC	Package	
430H	1072	IA32_MC12_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs." through Section 15.3.2.4, "IA32_MCi_MISC MSRs.". Banks MC12 report MC error from each channel of a link interconnect module.
431H	1073	IA32_MC12_STATUS	Package	
432H	1074	IA32_MC12_ADDR	Package	
433H	1075	IA32_MC12_MISC	Package	

Table 2-41. MSRs Supported by Intel® Xeon® Processor Scalable Family with DisplayFamily_DisplayModel 06_55H

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
434H	1076	IA32_MC13_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs." through Section 15.3.2.4, "IA32_MCi_MISC MSRs.". Banks MC13 through MC 18 report MC error from the integrated memory controllers.
435H	1077	IA32_MC13_STATUS	Package	
436H	1078	IA32_MC13_ADDR	Package	
437H	1079	IA32_MC13_MISC	Package	
438H	1080	IA32_MC14_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs." through Section 15.3.2.4, "IA32_MCi_MISC MSRs.". Banks MC13 through MC 18 report MC error from the integrated memory controllers.
439H	1081	IA32_MC14_STATUS	Package	
43AH	1082	IA32_MC14_ADDR	Package	
43BH	1083	IA32_MC14_MISC	Package	
43CH	1084	IA32_MC15_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs." through Section 15.3.2.4, "IA32_MCi_MISC MSRs.". Banks MC13 through MC 18 report MC error from the integrated memory controllers.
43DH	1085	IA32_MC15_STATUS	Package	
43EH	1086	IA32_MC15_ADDR	Package	
43FH	1087	IA32_MC15_MISC	Package	
440H	1088	IA32_MC16_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs." through Section 15.3.2.4, "IA32_MCi_MISC MSRs.". Banks MC13 through MC 18 report MC error from the integrated memory controllers
441H	1089	IA32_MC16_STATUS	Package	
442H	1090	IA32_MC16_ADDR	Package	
443H	1091	IA32_MC16_MISC	Package	
444H	1092	IA32_MC17_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs." through Section 15.3.2.4, "IA32_MCi_MISC MSRs.". Banks MC13 through MC 18 report MC error from the integrated memory controllers.
445H	1093	IA32_MC17_STATUS	Package	
446H	1094	IA32_MC17_ADDR	Package	
447H	1095	IA32_MC17_MISC	Package	
448H	1096	IA32_MC18_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs." through Section 15.3.2.4, "IA32_MCi_MISC MSRs.". Banks MC13 through MC 18 report MC error from the integrated memory controllers.
449H	1097	IA32_MC18_STATUS	Package	
44AH	1098	IA32_MC18_ADDR	Package	
44BH	1099	IA32_MC18_MISC	Package	
44CH	1100	IA32_MC19_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs." through Section 15.3.2.4, "IA32_MCi_MISC MSRs.". Bank MC19 reports MC error from a link interconnect module.
44DH	1101	IA32_MC19_STATUS	Package	
44EH	1102	IA32_MC19_ADDR	Package	
44FH	1103	IA32_MC19_MISC	Package	
606H	1542	MSR_RAPL_POWER_UNIT	Package	**Unit Multipliers used in RAPL Interfaces (R/O)**
		3:0	Package	**Power Units** See Section 14.9.1, "RAPL Interfaces."
		7:4	Package	**Reserved**
		12:8	Package	**Energy Status Units** Energy related information (in Joules) is based on the multiplier, 1/2^ESU; where ESU is an unsigned integer represented by bits 12:8. Default value is 0EH (or 61 micro-joules)
		15:13	Package	**Reserved**

Table 2-41. MSRs Supported by Intel® Xeon® Processor Scalable Family with DisplayFamily_DisplayModel 06_55H

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
		19:16	Package	**Time Units** See Section 14.9.1, "RAPL Interfaces."
		63:20		Reserved
618H	1560	MSR_DRAM_POWER_LIMIT	Package	**DRAM RAPL Power Limit Control (R/W)** See Section 14.9.5, "DRAM RAPL Domain."
619H	1561	MSR_DRAM_ENERGY_ STATUS	Package	**DRAM Energy Status (R/O)** Energy consumed by DRAM devices
		31:0		Energy in 15.3 micro-joules. Requires BIOS configuration to enable DRAM RAPL mode 0 (Direct VR).
		63:32		Reserved
61BH	1563	MSR_DRAM_PERF_STATUS	Package	**DRAM Performance Throttling Status (R/O)** See Section 14.9.5, "DRAM RAPL Domain."
61CH	1564	MSR_DRAM_POWER_INFO	Package	**DRAM RAPL Parameters (R/W)** See Section 14.9.5, "DRAM RAPL Domain."
620H	1568	MSR UNCORE_RATIO_LIMIT	Package	Uncore Ratio Limit (R/W) Out of reset, the min_ratio and max_ratio fields represent the widest possible range of uncore frequencies. Writing to these fields allows software to control the minimum and the maximum frequency that hardware will select.
		63:15		Reserved.
		14:8		MIN_RATIO Writing to this field controls the minimum possible ratio of the LLC/Ring.
		7		Reserved.
		6:0		MAX_RATIO This field is used to limit the max ratio of the LLC/Ring.
639H	1593	MSR_PP0_ENERGY_STATUS	Package	**Reserved (R/O)** Reads return 0
C8DH	3213	IA32_QM_EVTSEL	THREAD	**Monitoring Event Select Register (R/W)** if CPUID.(EAX=07H, ECX=0):EBX.RDT-M[bit 12] = 1
		7:0		**EventID (RW)** **Event encoding:** **0x00: no monitoring** **0x01: L3 occupancy monitoring** **0x02: Total memory bandwidth monitoring** **0x03: Local memory bandwidth monitoring** **All other encoding reserved**
		31:8		Reserved.
		41:32		**RMID (RW)**
		63:42		Reserved.
C8FH	3215	IA32_PQR_ASSOC	THREAD	**Resource Association Register (R/W)**

Table 2-41. MSRs Supported by Intel® Xeon® Processor Scalable Family with DisplayFamily_DisplayModel 06_55H

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
		9:0		**RMID**
		31:10		**Reserved**
		51:32		**COS (R/W).**
		63: 52		**Reserved**
C90H	3216	IA32_L3_QOS_MASK_0	Package	**L3 Class Of Service Mask - COS 0 (R/W)** if CPUID.(EAX=10H, ECX=1):EDX.COS_MAX[15:0] >=0
		0:19		**CBM: Bit vector of available L3 ways for COS 0 enforcement**
		63:20		**Reserved**
C91H	3217	IA32_L3_QOS_MASK_1	Package	**L3 Class Of Service Mask - COS 1 (R/W)** if CPUID.(EAX=10H, ECX=1):EDX.COS_MAX[15:0] >=1
		0:19		**CBM: Bit vector of available L3 ways for COS 1 enforcement**
		63:20		**Reserved**
C92H	3218	IA32_L3_QOS_MASK_2	Package	**L3 Class Of Service Mask - COS 2 (R/W).** if CPUID.(EAX=10H, ECX=1):EDX.COS_MAX[15:0] >=2
		0:19		**CBM: Bit vector of available L3 ways for COS 2 enforcement**
		63:20		**Reserved**
C93H	3219	IA32_L3_QOS_MASK_3	Package	**L3 Class Of Service Mask - COS 3 (R/W).** if CPUID.(EAX=10H, ECX=1):EDX.COS_MAX[15:0] >=3
		0:19		**CBM: Bit vector of available L3 ways for COS 3 enforcement**
		63:20		**Reserved**
C94H	3220	IA32_L3_QOS_MASK_4	Package	**L3 Class Of Service Mask - COS 4 (R/W).** if CPUID.(EAX=10H, ECX=1):EDX.COS_MAX[15:0] >=4
		0:19		**CBM: Bit vector of available L3 ways for COS 4 enforcement**
		63:20		**Reserved**
C95H	3221	IA32_L3_QOS_MASK_5	Package	**L3 Class Of Service Mask - COS 5 (R/W).** if CPUID.(EAX=10H, ECX=1):EDX.COS_MAX[15:0] >=5
		0:19		**CBM: Bit vector of available L3 ways for COS 5 enforcement**
		63:20		**Reserved**
C96H	3222	IA32_L3_QOS_MASK_6	Package	**L3 Class Of Service Mask - COS 6 (R/W).** if CPUID.(EAX=10H, ECX=1):EDX.COS_MAX[15:0] >=6
		0:19		**CBM: Bit vector of available L3 ways for COS 6 enforcement**
		63:20		**Reserved**
C97H	3223	IA32_L3_QOS_MASK_7	Package	**L3 Class Of Service Mask - COS 7 (R/W).** if CPUID.(EAX=10H, ECX=1):EDX.COS_MAX[15:0] >=7
		0:19		**CBM: Bit vector of available L3 ways for COS 7 enforcement**
		63:20		**Reserved**
C98H	3224	IA32_L3_QOS_MASK_8	Package	**L3 Class Of Service Mask - COS 8 (R/W).** if CPUID.(EAX=10H, ECX=1):EDX.COS_MAX[15:0] >=8

Table 2-41. MSRs Supported by Intel® Xeon® Processor Scalable Family with DisplayFamily_DisplayModel 06_55H

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
		0:19		**CBM: Bit vector of available L3 ways for COS 8 enforcement**
		63:20		**Reserved**
C99H	3225	IA32_L3_QOS_MASK_9	Package	**L3 Class Of Service Mask - COS 9 (R/W).** if CPUID.(EAX=10H, ECX=1):EDX.COS_MAX[15:0] >=9
		0:19		**CBM: Bit vector of available L3 ways for COS 9 enforcement**
		63:20		**Reserved**
C9AH	3226	IA32_L3_QOS_MASK_10	Package	**L3 Class Of Service Mask - COS 10 (R/W).** if CPUID.(EAX=10H, ECX=1):EDX.COS_MAX[15:0] >=10
		0:19		**CBM: Bit vector of available L3 ways for COS 10 enforcement**
		63:20		**Reserved**
C9BH	3227	IA32_L3_QOS_MASK_11	Package	**L3 Class Of Service Mask - COS 11 (R/W).** if CPUID.(EAX=10H, ECX=1):EDX.COS_MAX[15:0] >=11
		0:19		**CBM: Bit vector of available L3 ways for COS 11 enforcement**
		63:20		**Reserved**
C9CH	3228	IA32_L3_QOS_MASK_12	Package	**L3 Class Of Service Mask - COS 12 (R/W).** if CPUID.(EAX=10H, ECX=1):EDX.COS_MAX[15:0] >=12
		0:19		**CBM: Bit vector of available L3 ways for COS 12 enforcement**
		63:20		**Reserved**
C9DH	3229	IA32_L3_QOS_MASK_13	Package	**L3 Class Of Service Mask - COS 13 (R/W).** if CPUID.(EAX=10H, ECX=1):EDX.COS_MAX[15:0] >=13
		0:19		**CBM: Bit vector of available L3 ways for COS 13 enforcement**
		63:20		**Reserved**
C9EH	3230	IA32_L3_QOS_MASK_14	Package	**L3 Class Of Service Mask - COS 14 (R/W).** if CPUID.(EAX=10H, ECX=1):EDX.COS_MAX[15:0] >=14
		0:19		**CBM: Bit vector of available L3 ways for COS 14 enforcement**
		63:20		**Reserved**
C9FH	3231	IA32_L3_QOS_MASK_15	Package	**L3 Class Of Service Mask - COS 15 (R/W).** if CPUID.(EAX=10H, ECX=1):EDX.COS_MAX[15:0] >=15
		0:19		**CBM: Bit vector of available L3 ways for COS 15 enforcement**
		63:20		**Reserved**

2.17 MSRS IN INTEL® XEON PHI™ PROCESSOR 3200/5200/7200 SERIES AND FUTURE INTEL® XEON PHI™ PROCESSORS

Intel® Xeon Phi™ processor 3200, 5200, 7200 series, with CPUID DisplayFamily_DisplayModel signature 06_57H, support the MSR interfaces listed in Table 2-42. These processors are based on the Knights Landing microarchitecture. Future Intel® Xeon Phi™ Processors, with CPUID DisplayFamily_DisplayModel signature 06_85H, support the MSR interfaces listed in Table 2-42 and Table 2-43. Some MSRs are shared between a pair of processor cores, the scope is marked as module.

Table 2-42. Selected MSRs Supported by Intel® Xeon Phi™ Processors with DisplayFamily_DisplayModel Signatures 06_57H and 06_85H

Address		Register Name	Scope	Bit Description
Hex	Dec			
0H	0	IA32_P5_MC_ADDR	Module	See Section 2.22, "MSRs in Pentium Processors."
1H	1	IA32_P5_MC_TYPE	Module	See Section 2.22, "MSRs in Pentium Processors."
6H	6	IA32_MONITOR_FILTER_ SIZE	Thread	See Section 8.10.5, "Monitor/Mwait Address Range Determination." andTable 2-2
10H	16	IA32_TIME_STAMP_ COUNTER	Thread	See Section 17.17, "Time-Stamp Counter," and see Table 2-2.
17H	23	IA32_PLATFORM_ID	Package	**Platform ID (R)** See Table 2-2.
1BH	27	IA32_APIC_BASE	Thread	See Section 10.4.4, "Local APIC Status and Location," and Table 2-2.
34H	52	MSR_SMI_COUNT	Thread	**SMI Counter (R/O)**
		31:0		**SMI Count (R/O)**
		63:32		Reserved.
3AH	58	IA32_FEATURE_CONTROL	Thread	**Control Features in Intel 64Processor (R/W)** See Table 2-2.
		0		**Lock (R/WL)**
		1		**Reserved**
		2		**Enable VMX outside SMX operation (R/WL)**
3BH	59	IA32_TSC_ADJUST	THREAD	**Per-Logical-Processor TSC ADJUST (R/W)** See Table 2-2.
4EH	78	MSR_PPIN_CTL	Package	Protected Processor Inventory Number Enable Control (R/W)
		0		LockOut (R/WO) Set 1to prevent further writes to MSR_PPIN_CTL. Writing 1 to MSR_PPINCTL[bit 0] is permitted only if MSR_PPIN_CTL[bit 1] is clear, Default is 0. BIOS should provide an opt-in menu to enable the user to turn on MSR_PPIN_CTL[bit 1] for privileged inventory initialization agent to access MSR_PPIN. After reading MSR_PPIN, the privileged inventory initialization agent should write '01b' to MSR_PPIN_CTL to disable further access to MSR_PPIN and prevent unauthorized modification to MSR_PPIN_CTL.

Table 2-42. Selected MSRs Supported by Intel® Xeon Phi™ Processors with DisplayFamily_DisplayModel Signatures 06_57H and 06_85H

Address		Register Name	Scope	Bit Description
Hex	Dec			
		1		Enable_PPIN (R/W) If 1, enables MSR_PPIN to be accessible using RDMSR. Once set, attempt to write 1 to MSR_PPIN_CTL[bit 0] will cause #GP. If 0, an attempt to read MSR_PPIN will cause #GP. Default is 0.
		63:2		Reserved.
4FH	79	MSR_PPIN	Package	Protected Processor Inventory Number (R/O)
		63:0		**Protected Processor Inventory Number (R/O)** A unique value within a given CPUID family/model/stepping signature that a privileged inventory initialization agent can access to identify each physical processor, when access to MSR_PPIN is enabled. Access to MSR_PPIN is permitted only if MSR_PPIN_CTL[bits 1:0] = '10b'
79H	121	IA32_BIOS_UPDT_TRIG	Core	**BIOS Update Trigger Register (W)** See Table 2-2.
8BH	139	IA32_BIOS_SIGN_ID	THREAD	**BIOS Update Signature ID (RO)** See Table 2-2.
C1H	193	IA32_PMC0	THREAD	**Performance counter register** See Table 2-2.
C2H	194	IA32_PMC1	THREAD	**Performance Counter Register** See Table 2-2.
CEH	206	MSR_PLATFORM_INFO	Package	Platform Information; contains power management and other model specific features enumeration. See http://biosbits.org.
		7:0		Reserved.
		15:8	Package	**Maximum Non-Turbo Ratio (R/O)** The is the ratio of the frequency that invariant TSC runs at. Frequency = ratio * 100 MHz.
		27:16		Reserved.
		28	Package	**Programmable Ratio Limit for Turbo Mode (R/O)** When set to 1, indicates that Programmable Ratio Limits for Turbo mode is enabled, and when set to 0, indicates Programmable Ratio Limits for Turbo mode is disabled.
		29	Package	**Programmable TDP Limit for Turbo Mode (R/O)** When set to 1, indicates that TDP Limits for Turbo mode are programmable, and when set to 0, indicates TDP Limit for Turbo mode is not programmable.
		39:30		Reserved.
		47:40	Package	**Maximum Efficiency Ratio (R/O)** The is the minimum ratio (maximum efficiency) that the processor can operates, in units of 100MHz.
		63:48		Reserved.

Table 2-42. Selected MSRs Supported by Intel® Xeon Phi™ Processors with DisplayFamily_DisplayModel Signatures 06_57H and 06_85H

Address		Register Name	Scope	Bit Description
Hex	Dec			
E2H	226	MSR_PKG_CST_CONFIG_ CONTROL	Package	**C-State Configuration Control (R/W)**
		2:0		**Package C-State Limit (R/W)** Specifies the lowest C-state for the package. This feature does not limit the processor core C-state. The power-on default value from bit[2:0] of this register reports the deepest package C-state the processor is capable to support when manufactured. It is recommended that BIOS always read the power-on default value reported from this bit field to determine the supported deepest C-state on the processor and leave it as default without changing it. 000b - C0/C1 (No package C-state support) 001b - C2 010b - C6 (non retention)* 011b - C6 (Retention)* 100b - Reserved 101b - Reserved 110b - Reserved 111b - No package C-state limit. All C-States supported by the processor are available. Note: C6 retention mode provides more power saving than C6 non-retention mode. Limiting the package to C6 non retention mode does prevent the MSR_PKG_C6_RESIDENCY counter (MSR 3F9h) from being incremented.
		9:3		Reserved.
		10		**I/O MWAIT Redirection Enable (R/W)** When set, will map IO_read instructions sent to IO registers at MSR_PMG_IO_CAPTURE_BASE[15:0] to MWAIT instructions.
		14:11		Reserved.
		15		**CFG Lock (RO)** When set, locks bits [15:0] of this register for further writes until the next reset occurs.
		25		Reserved.
		26		**C1 State Auto Demotion Enable (R/W)** When set, processor will conditionally demote C3/C6/C7 requests to C1 based on uncore auto-demote information.
		27		Reserved.
		28		**C1 State Auto Undemotion Enable (R/W)** When set, enables Undemotion from Demoted C1.
		29		**PKG C-State Auto Demotion Enable (R/W)** When set, enables Package C state demotion.
		63:30		Reserved.
E4H	228	MSR_PMG_IO_CAPTURE_ BASE	Tile	**Power Management IO Capture Base (R/W)**

Table 2-42. Selected MSRs Supported by Intel® Xeon Phi™ Processors with DisplayFamily_DisplayModel Signatures 06_57H and 06_85H

Address		Register Name	Scope	Bit Description
Hex	Dec			
		15:0		**LVL_2 Base Address (R/W)** Microcode will compare IO-read zone to this base address to determine if an MWAIT(C2/3/4) needs to be issued instead of the IO-read. Should be programmed to the chipset Plevel_2 IO address.
		22:16		**C-State Range (R/W)** The IO-port block size in which IO-redirection will be executed (0-127). Should be programmed based on the number of LVLx registers existing in the chipset.
		63:23		Reserved.
E7H	231	IA32_MPERF	Thread	**Maximum Performance Frequency Clock Count (RW)** See Table 2-2.
E8H	232	IA32_APERF	Thread	**Actual Performance Frequency Clock Count (RW)** See Table 2-2.
FEH	254	IA32_MTRRCAP	Core	**Memory Type Range Register (R)** See Table 2-2.
13CH	52	MSR_FEATURE_CONFIG	Core	**AES Configuration (RW-L)** Privileged post-BIOS agent must provide a #GP handler to handle unsuccessful read of this MSR.
		1:0		**AES Configuration (RW-L)** Upon a successful read of this MSR, the configuration of AES instruction set availability is as follows: 11b: AES instructions are not available until next RESET. otherwise, AES instructions are available. Note, AES instruction set is not available if read is unsuccessful. If the configuration is not 01b, AES instruction can be mis-configured if a privileged agent unintentionally writes 11b.
		63:2		Reserved.
140H	320	MISC_FEATURE_ENABLES	Thread	**MISC_FEATURE_ENABLES**
		0		Reserved.
		1		**User Mode MONITOR and MWAIT (R/W)** If set to 1, the MONITOR and MWAIT instructions do not cause invalid-opcode exceptions when executed with CPL > 0 or in virtual-8086 mode. If MWAIT is executed when CPL > 0 or in virtual-8086 mode, and if EAX indicates a C-state other than C0 or C1, the instruction operates as if EAX indicated the C-state C1.
		63:2		Reserved.
174H	372	IA32_SYSENTER_CS	Thread	See Table 2-2.
175H	373	IA32_SYSENTER_ESP	Thread	See Table 2-2.
176H	374	IA32_SYSENTER_EIP	Thread	See Table 2-2.
179H	377	IA32_MCG_CAP	Thread	See Table 2-2.
17AH	378	IA32_MCG_STATUS	Thread	See Table 2-2.

Table 2-42. Selected MSRs Supported by Intel® Xeon Phi™ Processors with DisplayFamily_DisplayModel Signatures 06_57H and 06_85H

Address		Register Name	Scope	Bit Description
Hex	Dec			
17DH	390	MSR_SMM_MCA_CAP	Thread	**Enhanced SMM Capabilities (SMM-RO)** Reports SMM capability Enhancement. Accessible only while in SMM.
		31:0		**Bank Support (SMM-RO)** One bit per MCA bank. If the bit is set, that bank supports Enhanced MCA (Default all 0; does not support EMCA).
		55:32		Reserved.
		56		**Targeted SMI (SMM-RO)** Set if targeted SMI is supported.
		57		**SMM_CPU_SVRSTR (SMM-RO)** Set if SMM SRAM save/restore feature is supported.
		58		**SMM_CODE_ACCESS_CHK (SMM-RO)** Set if SMM code access check feature is supported.
		59		**Long_Flow_Indication (SMM-RO)** If set to 1 indicates that the SMM long flow indicator is supported and a host-space interface available to SMM handler.
		63:60		Reserved.
186H	390	IA32_PERFEVTSEL0	Thread	Performance Monitoring Event Select Register (R/W) See Table 2-2.
		7:0		**Event Select**
		15:8		**UMask**
		16		**USR**
		17		**OS**
		18		**Edge**
		19		**PC**
		20		**INT**
		21		**AnyThread**
		22		**EN**
		23		**INV**
		31:24		**CMASK**
		63:32		Reserved.
187H	391	IA32_PERFEVTSEL1	Thread	See Table 2-2.
198H	408	IA32_PERF_STATUS	Package	See Table 2-2.
199H	409	IA32_PERF_CTL	Thread	See Table 2-2.
19AH	410	IA32_CLOCK_MODULATION	Thread	**Clock Modulation (R/W)** See Table 2-2.
19BH	411	IA32_THERM_INTERRUPT	Module	**Thermal Interrupt Control (R/W)** See Table 2-2.

Table 2-42. Selected MSRs Supported by Intel® Xeon Phi™ Processors with DisplayFamily_DisplayModel Signatures 06_57H and 06_85H

Address		Register Name	Scope	Bit Description
Hex	Dec			
19CH	412	IA32_THERM_STATUS	Module	**Thermal Monitor Status (R/W)** See Table 2-2.
		0		**Thermal status (RO)**
		1		**Thermal status log (R/WC0)**
		2		**PROTCHOT # or FORCEPR# status (RO)**
		3		**PROTCHOT # or FORCEPR# log (R/WC0)**
		4		**Critical Temperature status (RO)**
		5		**Critical Temperature status log (R/WC0)**
		6		**Thermal threshold #1 status (RO)**
		7		**Thermal threshold #1 log (R/WC0)**
		8		**Thermal threshold #2 status (RO)**
		9		**Thermal threshold #2 log (R/WC0)**
		10		**Power Limitation status (RO)**
		11		**Power Limitation log (R/WC0)**
		15:12		Reserved.
		22:16		**Digital Readout (RO)**
		26:23		Reserved.
		30:27		**Resolution in degrees Celsius (RO)**
		31		**Reading Valid (RO)**
		63:32		Reserved.
1A0H	416	IA32_MISC_ENABLE	Thread	**Enable Misc. Processor Features (R/W)** Allows a variety of processor functions to be enabled and disabled.
		0		**Fast-Strings Enable**
		2:1		Reserved.
		3		**Automatic Thermal Control Circuit Enable (R/W)**
		6:4		Reserved.
		7		**Performance Monitoring Available (R)**
		10:8		Reserved.
		11		**Branch Trace Storage Unavailable (RO)**
		12		**Processor Event Based Sampling Unavailable (RO)**
		15:13		Reserved.
		16		**Enhanced Intel SpeedStep Technology Enable (R/W)**
		18		**ENABLE MONITOR FSM (R/W)**
		21:19		Reserved.
		22		**Limit CPUID Maxval (R/W)**
		23		**xTPR Message Disable (R/W)**
		33:24		Reserved.

Table 2-42. Selected MSRs Supported by Intel® Xeon Phi™ Processors with DisplayFamily_DisplayModel Signatures 06_57H and 06_85H

Address		Register Name	Scope	Bit Description
Hex	Dec			
		34		**XD Bit Disable (R/W)**
		37:35		Reserved.
		38		**Turbo Mode Disable (R/W)**
		63:39		Reserved.
1A2H	418	MSR_ TEMPERATURE_TARGET	Package	Temperature Target
		15:0		Reserved.
		23:16		**Temperature Target (R)**
		29:24		**Target Offset (R/W)**
		63:30		Reserved.
1A4H	420	MSR_MISC_FEATURE_ CONTROL		**Miscellaneous Feature Control (R/W)**
		0	Core	**DCU Hardware Prefetcher Disable (R/W)** If 1, disables the L1 data cache prefetcher.
		1	Core	**L2 Hardware Prefetcher Disable (R/W)** If 1, disables the L2 hardware prefetcher.
		63:2		Reserved.
1A6H	422	MSR_OFFCORE_RSP_0	Shared	**Offcore Response Event Select Register (R/W)**
1A7H	423	MSR_OFFCORE_RSP_1	Shared	**Offcore Response Event Select Register (R/W)**
1ADH	429	MSR_TURBO_RATIO_LIMIT	Package	**Maximum Ratio Limit of Turbo Mode for Groups of Cores (RW)**
		0		**Reserved**
		7:1	Package	**Maximum Number of Cores in Group 0** Number active processor cores which operates under the maximum ratio limit for group 0.
		15:8	Package	**Maximum Ratio Limit for Group 0** Maximum turbo ratio limit when the number of active cores are not more than the group 0 maximum core count.
		20:16	Package	**Number of Incremental Cores Added to Group 1** Group 1, which includes the specified number of additional cores plus the cores in group 0, operates under the group 1 turbo max ratio limit = “group 0 Max ratio limit” - “group ratio delta for group 1”.
		23:21	Package	**Group Ratio Delta for Group 1** An unsigned integer specifying the ratio decrement relative to the Max ratio limit to Group 0.
		28:24	Package	**Number of Incremental Cores Added to Group 2** Group 2, which includes the specified number of additional cores plus all the cores in group 1, operates under the group 2 turbo max ratio limit = “group 1 Max ratio limit” - “group ratio delta for group 2”.

Table 2-42. Selected MSRs Supported by Intel® Xeon Phi™ Processors with DisplayFamily_DisplayModel Signatures 06_57H and 06_85H

Address		Register Name	Scope	Bit Description
Hex	Dec			
		31:29	Package	**Group Ratio Delta for Group 2** An unsigned integer specifying the ratio decrement relative to the Max ratio limit for Group 1.
		36:32	Package	**Number of Incremental Cores Added to Group 3** Group 3, which includes the specified number of additional cores plus all the cores in group 2, operates under the group 3 turbo max ratio limit = "group 2 Max ratio limit" - "group ratio delta for group 3".
		39:37	Package	**Group Ratio Delta for Group 3** An unsigned integer specifying the ratio decrement relative to the Max ratio limit for Group 2.
		44:40	Package	**Number of Incremental Cores Added to Group 4** Group 4, which includes the specified number of additional cores plus all the cores in group 3, operates under the group 4 turbo max ratio limit = "group 3 Max ratio limit" - "group ratio delta for group 4".
		47:45	Package	**Group Ratio Delta for Group 4** An unsigned integer specifying the ratio decrement relative to the Max ratio limit for Group 3.
		52:48	Package	**Number of Incremental Cores Added to Group 5** Group 5, which includes the specified number of additional cores plus all the cores in group 4, operates under the group 5 turbo max ratio limit = "group 4 Max ratio limit" - "group ratio delta for group 5".
		55:53	Package	**Group Ratio Delta for Group 5** An unsigned integer specifying the ratio decrement relative to the Max ratio limit for Group 4.
		60:56	Package	**Number of Incremental Cores Added to Group 6** Group 6, which includes the specified number of additional cores plus all the cores in group 5, operates under the group 6 turbo max ratio limit = "group 5 Max ratio limit" - "group ratio delta for group 6".
		63:61	Package	**Group Ratio Delta for Group 6** An unsigned integer specifying the ratio decrement relative to the Max ratio limit for Group 5.
1B0H	432	IA32_ENERGY_PERF_BIAS	Thread	See Table 2-2.
1B1H	433	IA32_PACKAGE_THERM_ STATUS	Package	See Table 2-2.
1B2H	434	IA32_PACKAGE_THERM_ INTERRUPT	Package	See Table 2-2.
1C8H	456	MSR_LBR_SELECT	Thread	**Last Branch Record Filtering Select Register (R/W)** See Section 17.9.2, "Filtering of Last Branch Records."
		0		**CPL_EQ_0**
		1		**CPL_NEQ_0**

Table 2-42. Selected MSRs Supported by Intel® Xeon Phi™ Processors with DisplayFamily_DisplayModel Signatures 06_57H and 06_85H

Address		Register Name	Scope	Bit Description
Hex	Dec			
		2		**JCC**
		3		**NEAR_REL_CALL**
		4		**NEAR_IND_CALL**
		5		**NEAR_RET**
		6		**NEAR_IND_JMP**
		7		**NEAR_REL_JMP**
		8		**FAR_BRANCH**
		63:9		Reserved.
1C9H	457	MSR_LASTBRANCH_TOS	Thread	**Last Branch Record Stack TOS (R/W)** Contains an index (bits 0-2) that points to the MSR containing the most recent branch record. See MSR_LASTBRANCH_0_FROM_IP.
1D9H	473	IA32_DEBUGCTL	Thread	**Debug Control (R/W)**
		0		**LBR** Setting this bit to 1 enables the processor to record a running trace of the most recent branches taken by the processor in the LBR stack.
		1		**BTF** Setting this bit to 1 enables the processor to treat EFLAGS.TF as single-step on branches instead of single-step on instructions.
		5:2		Reserved.
		6		**TR** Setting this bit to 1 enables branch trace messages to be sent.
		7		**BTS** Setting this bit enables branch trace messages (BTMs) to be logged in a BTS buffer.
		8		**BTINT** When clear, BTMs are logged in a BTS buffer in circular fashion. When this bit is set, an interrupt is generated by the BTS facility when the BTS buffer is full.
		9		**BTS_OFF_OS** When set, BTS or BTM is skipped if CPL = 0.
		10		**BTS_OFF_USR** When set, BTS or BTM is skipped if CPL > 0.
		11		**FREEZE_LBRS_ON_PMI** When set, the LBR stack is frozen on a PMI request.
		12		**FREEZE_PERFMON_ON_PMI** When set, each ENABLE bit of the global counter control MSR are frozen (address 3BFH) on a PMI request.
		13		Reserved.

Table 2-42. Selected MSRs Supported by Intel® Xeon Phi™ Processors with DisplayFamily_DisplayModel Signatures 06_57H and 06_85H

Address		Register Name	Scope	Bit Description
Hex	Dec			
		14		**FREEZE_WHILE_SMM_EN** When set, freezes perfmon and trace messages while in SMM.
		31:15		Reserved.
1DDH	477	MSR_LER_FROM_LIP	Thread	**Last Exception Record From Linear IP (R)**
1DEH	478	MSR_LER_TO_LIP	Thread	**Last Exception Record To Linear IP (R)**
1F2H	498	IA32_SMRR_PHYSBASE	Core	See Table 2-2.
1F3H	499	IA32_SMRR_PHYSMASK	Core	See Table 2-2.
200H	512	IA32_MTRR_PHYSBASE0	Core	See Table 2-2.
201H	513	IA32_MTRR_PHYSMASK0	Core	See Table 2-2.
202H	514	IA32_MTRR_PHYSBASE1	Core	See Table 2-2.
203H	515	IA32_MTRR_PHYSMASK1	Core	See Table 2-2.
204H	516	IA32_MTRR_PHYSBASE2	Core	See Table 2-2.
205H	517	IA32_MTRR_PHYSMASK2	Core	See Table 2-2.
206H	518	IA32_MTRR_PHYSBASE3	Core	See Table 2-2.
207H	519	IA32_MTRR_PHYSMASK3	Core	See Table 2-2.
208H	520	IA32_MTRR_PHYSBASE4	Core	See Table 2-2.
209H	521	IA32_MTRR_PHYSMASK4	Core	See Table 2-2.
20AH	522	IA32_MTRR_PHYSBASE5	Core	See Table 2-2.
20BH	523	IA32_MTRR_PHYSMASK5	Core	See Table 2-2.
20CH	524	IA32_MTRR_PHYSBASE6	Core	See Table 2-2.
20DH	525	IA32_MTRR_PHYSMASK6	Core	See Table 2-2.
20EH	526	IA32_MTRR_PHYSBASE7	Core	See Table 2-2.
20FH	527	IA32_MTRR_PHYSMASK7	Core	See Table 2-2.
250H	592	IA32_MTRR_FIX64K_00000	Core	See Table 2-2.
258H	600	IA32_MTRR_FIX16K_80000	Core	See Table 2-2.
259H	601	IA32_MTRR_FIX16K_A0000	Core	See Table 2-2.
268H	616	IA32_MTRR_FIX4K_C0000	Core	See Table 2-2.
269H	617	IA32_MTRR_FIX4K_C8000	Core	See Table 2-2.
26AH	618	IA32_MTRR_FIX4K_D0000	Core	See Table 2-2.
26BH	619	IA32_MTRR_FIX4K_D8000	Core	See Table 2-2.
26CH	620	IA32_MTRR_FIX4K_E0000	Core	See Table 2-2.
26DH	621	IA32_MTRR_FIX4K_E8000	Core	See Table 2-2.
26EH	622	IA32_MTRR_FIX4K_F0000	Core	See Table 2-2.
26FH	623	IA32_MTRR_FIX4K_F8000	Core	See Table 2-2.
277H	631	IA32_PAT	Core	See Table 2-2.

Table 2-42. Selected MSRs Supported by Intel® Xeon Phi™ Processors with DisplayFamily_DisplayModel Signatures 06_57H and 06_85H

Address		Register Name	Scope	Bit Description
Hex	Dec			
2FFH	767	IA32_MTRR_DEF_TYPE	Core	**Default Memory Types (R/W)** See Table 2-2.
309H	777	IA32_FIXED_CTR0	Thread	**Fixed-Function Performance Counter Register 0 (R/W)** See Table 2-2.
30AH	778	IA32_FIXED_CTR1	Thread	**Fixed-Function Performance Counter Register 1 (R/W)** See Table 2-2.
30BH	779	IA32_FIXED_CTR2	Thread	**Fixed-Function Performance Counter Register 2 (R/W)** See Table 2-2.
345H	837	IA32_PERF_CAPABILITIES	Package	See Table 2-2. See Section 17.4.1, "IA32_DEBUGCTL MSR."
38DH	909	IA32_FIXED_CTR_CTRL	Thread	**Fixed-Function-Counter Control Register (R/W)** See Table 2-2.
38EH	910	IA32_PERF_GLOBAL_STATUS	Thread	See Table 2-2.
38FH	911	IA32_PERF_GLOBAL_CTRL	Thread	See Table 2-2.
390H	912	IA32_PERF_GLOBAL_OVF_CTRL	Thread	See Table 2-2.
3F1H	1009	MSR_PEBS_ENABLE	Thread	See Table 2-2.
3F8H	1016	MSR_PKG_C3_RESIDENCY	Package	Note: C-state values are processor specific C-state code names, unrelated to MWAIT extension C-state parameters or ACPI C-States.
		63:0		Package C3 Residency Counter. (R/O)
3F9H	1017	MSR_PKG_C6_RESIDENCY	Package	
		63:0		Package C6 Residency Counter. (R/O)
3FAH	1018	MSR_PKG_C7_RESIDENCY	Package	
		63:0		Package C7 Residency Counter. (R/O)
3FCH	1020	MSR_MC0_RESIDENCY	Module	Note: C-state values are processor specific C-state code names, unrelated to MWAIT extension C-state parameters or ACPI C-States.
		63:0		Module C0 Residency Counter. (R/O)
3FDH	1021	MSR_MC6_RESIDENCY	Module	
		63:0		Module C6 Residency Counter. (R/O)
3FFH	1023	MSR_CORE_C6_RESIDENCY	Core	Note: C-state values are processor specific C-state code names, unrelated to MWAIT extension C-state parameters or ACPI C-States.
		63:0		CORE C6 Residency Counter. (R/O)
400H	1024	IA32_MC0_CTL	Core	See Section 15.3.2.1, "IA32_MCi_CTL MSRs."
401H	1025	IA32_MC0_STATUS	Core	See Section 15.3.2.2, "IA32_MCi_STATUS MSRS."
402H	1026	IA32_MC0_ADDR	Core	See Section 15.3.2.3, "IA32_MCi_ADDR MSRs."
404H	1028	IA32_MC1_CTL	Core	See Section 15.3.2.1, "IA32_MCi_CTL MSRs."
405H	1029	IA32_MC1_STATUS	Core	See Section 15.3.2.2, "IA32_MCi_STATUS MSRS."
408H	1032	IA32_MC2_CTL	Core	See Section 15.3.2.1, "IA32_MCi_CTL MSRs."
409H	1033	IA32_MC2_STATUS	Core	See Section 15.3.2.2, "IA32_MCi_STATUS MSRS."

Table 2-42. Selected MSRs Supported by Intel® Xeon Phi™ Processors with DisplayFamily_DisplayModel Signatures 06_57H and 06_85H

Address		Register Name	Scope	Bit Description
Hex	Dec			
40AH	1034	IA32_MC2_ADDR	Core	See Section 15.3.2.3, "IA32_MCi_ADDR MSRs."
40CH	1036	IA32_MC3_CTL	Core	See Section 15.3.2.1, "IA32_MCi_CTL MSRs."
40DH	1037	IA32_MC3_STATUS	Core	See Section 15.3.2.2, "IA32_MCi_STATUS MSRS."
40EH	1038	IA32_MC3_ADDR	Core	See Section 15.3.2.3, "IA32_MCi_ADDR MSRs."
410H	1040	IA32_MC4_CTL	Core	See Section 15.3.2.1, "IA32_MCi_CTL MSRs."
411H	1041	IA32_MC4_STATUS	Core	See Section 15.3.2.2, "IA32_MCi_STATUS MSRS."
412H	1042	IA32_MC4_ADDR	Core	See Section 15.3.2.3, "IA32_MCi_ADDR MSRs." The MSR_MC4_ADDR register is either not implemented or contains no address if the ADDRV flag in the MSR_MC4_STATUS register is clear. When not implemented in the processor, all reads and writes to this MSR will cause a general-protection exception.
414H	1044	IA32_MC5_CTL	Package	See Section 15.3.2.1, "IA32_MCi_CTL MSRs."
415H	1045	IA32_MC5_STATUS	Package	See Section 15.3.2.2, "IA32_MCi_STATUS MSRS."
416H	1046	IA32_MC5_ADDR	Package	See Section 15.3.2.3, "IA32_MCi_ADDR MSRs."
4C1H	1217	IA32_A_PMC0	Thread	See Table 2-2.
4C2H	1218	IA32_A_PMC1	Thread	See Table 2-2.
600H	1536	IA32_DS_AREA	Thread	**DS Save Area (R/W)** See Table 2-2.
606H	1542	MSR_RAPL_POWER_UNIT	Package	**Unit Multipliers used in RAPL Interfaces (R/O)**
		3:0	Package	**Power Units** See Section 14.9.1, "RAPL Interfaces."
		7:4	Package	**Reserved**
		12:8	Package	**Energy Status Units** Energy related information (in Joules) is based on the multiplier, 1/2^ESU; where ESU is an unsigned integer represented by bits 12:8. Default value is 0EH (or 61 micro-joules)
		15:13	Package	**Reserved**
		19:16	Package	**Time Units** See Section 14.9.1, "RAPL Interfaces."
		63:20		Reserved
60DH	1549	MSR_PKG_C2_RESIDENCY	Package	Note: C-state values are processor specific C-state code names, unrelated to MWAIT extension C-state parameters or ACPI C-States.
		63:0		Package C2 Residency Counter. (R/O)
610H	1552	MSR_PKG_POWER_LIMIT	Package	**PKG RAPL Power Limit Control (R/W)** See Section 14.9.3, "Package RAPL Domain."
611H	1553	MSR_PKG_ENERGY_STATUS	Package	**PKG Energy Status (R/O)** See Section 14.9.3, "Package RAPL Domain."
613H	1555	MSR_PKG_PERF_STATUS	Package	**PKG Perf Status (R/O)** See Section 14.9.3, "Package RAPL Domain."

Table 2-42. Selected MSRs Supported by Intel® Xeon Phi™ Processors with DisplayFamily_DisplayModel Signatures 06_57H and 06_85H

Address		Register Name	Scope	Bit Description
Hex	Dec			
614H	1556	MSR_PKG_POWER_INFO	Package	**PKG RAPL Parameters (R/W)** See Section 14.9.3, "Package RAPL Domain."
618H	1560	MSR_DRAM_POWER_LIMIT	Package	**DRAM RAPL Power Limit Control (R/W)** See Section 14.9.5, "DRAM RAPL Domain."
619H	1561	MSR_DRAM_ENERGY_ STATUS	Package	**DRAM Energy Status (R/O)** See Section 14.9.5, "DRAM RAPL Domain."
61BH	1563	MSR_DRAM_PERF_STATUS	Package	**DRAM Performance Throttling Status (R/O)** See Section 14.9.5, "DRAM RAPL Domain."
61CH	1564	MSR_DRAM_POWER_INFO	Package	**DRAM RAPL Parameters (R/W)** See Section 14.9.5, "DRAM RAPL Domain."
620H	1568	MSR UNCORE_RATIO_LIMIT	Package	Uncore Ratio Limit (R/W) Out of reset, the min_ratio and max_ratio fields represent the widest possible range of uncore frequencies. Writing to these fields allows software to control the minimum and the maximum frequency that hardware will select.
		63:15		Reserved.
		14:8		MIN_RATIO Writing to this field controls the minimum possible ratio of the LLC/Ring.
		7		Reserved.
		6:0		MAX_RATIO This field is used to limit the max ratio of the LLC/Ring.
638H	1592	MSR_PP0_POWER_LIMIT	Package	**PP0 RAPL Power Limit Control (R/W)** See Section 14.9.4, "PP0/PP1 RAPL Domains."
639H	1593	MSR_PP0_ENERGY_STATUS	Package	**PP0 Energy Status (R/O)** See Section 14.9.4, "PP0/PP1 RAPL Domains."
648H	1608	MSR_CONFIG_TDP_ NOMINAL	Package	**Base TDP Ratio (R/O)** See Table 2-24
649H	1609	MSR_CONFIG_TDP_LEVEL1	Package	ConfigTDP Level 1 ratio and power level (R/O). See Table 2-24
64AH	1610	MSR_CONFIG_TDP_LEVEL2	Package	ConfigTDP Level 2 ratio and power level (R/O). See Table 2-24
64BH	1611	MSR_CONFIG_TDP_ CONTROL	Package	**ConfigTDP Control (R/W)** See Table 2-24
64CH	1612	MSR_TURBO_ACTIVATION_ RATIO	Package	**ConfigTDP Control (R/W)** See Table 2-24
690H	1680	MSR_CORE_PERF_LIMIT_RE ASONS	Package	**Indicator of Frequency Clipping in Processor Cores (R/W)** **(frequency refers to processor core frequency)**
		0		**PROCHOT Status (R0)**
		1		**Thermal Status (R0)**
		5:2		Reserved.
		6		**VR Therm Alert Status (R0)**

Table 2-42. Selected MSRs Supported by Intel® Xeon Phi™ Processors with DisplayFamily_DisplayModel Signatures 06_57H and 06_85H

Address		Register Name	Scope	Bit Description
Hex	Dec			
		7		Reserved.
		8		**Electrical Design Point Status (R0)**
		63:9		Reserved.
6E0H	1760	IA32_TSC_DEADLINE	Core	**TSC Target of Local APIC's TSC Deadline Mode (R/W)** See Table 2-2
802H	2050	IA32_X2APIC_APICID	Thread	x2APIC ID register (R/0) See x2APIC Specification.
803H	2051	IA32_X2APIC_VERSION	Thread	x2APIC Version register (R/0)
808H	2056	IA32_X2APIC_TPR	Thread	x2APIC Task Priority register (R/W)
80AH	2058	IA32_X2APIC_PPR	Thread	x2APIC Processor Priority register (R/0)
80BH	2059	IA32_X2APIC_EOI	Thread	x2APIC EOI register (W/0)
80DH	2061	IA32_X2APIC_LDR	Thread	x2APIC Logical Destination register (R/0)
80FH	2063	IA32_X2APIC_SIVR	Thread	x2APIC Spurious Interrupt Vector register (R/W)
810H	2064	IA32_X2APIC_ISR0	Thread	x2APIC In-Service register bits [31:0] (R/0)
811H	2065	IA32_X2APIC_ISR1	Thread	x2APIC In-Service register bits [63:32] (R/0)
812H	2066	IA32_X2APIC_ISR2	Thread	x2APIC In-Service register bits [95:64] (R/0)
813H	2067	IA32_X2APIC_ISR3	Thread	x2APIC In-Service register bits [127:96] (R/0)
814H	2068	IA32_X2APIC_ISR4	Thread	x2APIC In-Service register bits [159:128] (R/0)
815H	2069	IA32_X2APIC_ISR5	Thread	x2APIC In-Service register bits [191:160] (R/0)
816H	2070	IA32_X2APIC_ISR6	Thread	x2APIC In-Service register bits [223:192] (R/0)
817H	2071	IA32_X2APIC_ISR7	Thread	x2APIC In-Service register bits [255:224] (R/0)
818H	2072	IA32_X2APIC_TMR0	Thread	x2APIC Trigger Mode register bits [31:0] (R/0)
819H	2073	IA32_X2APIC_TMR1	Thread	x2APIC Trigger Mode register bits [63:32] (R/0)
81AH	2074	IA32_X2APIC_TMR2	Thread	x2APIC Trigger Mode register bits [95:64] (R/0)
81BH	2075	IA32_X2APIC_TMR3	Thread	x2APIC Trigger Mode register bits [127:96] (R/0)
81CH	2076	IA32_X2APIC_TMR4	Thread	x2APIC Trigger Mode register bits [159:128] (R/0)
81DH	2077	IA32_X2APIC_TMR5	Thread	x2APIC Trigger Mode register bits [191:160] (R/0)
81EH	2078	IA32_X2APIC_TMR6	Thread	x2APIC Trigger Mode register bits [223:192] (R/0)
81FH	2079	IA32_X2APIC_TMR7	Thread	x2APIC Trigger Mode register bits [255:224] (R/0)
820H	2080	IA32_X2APIC_IRR0	Thread	x2APIC Interrupt Request register bits [31:0] (R/0)
821H	2081	IA32_X2APIC_IRR1	Thread	x2APIC Interrupt Request register bits [63:32] (R/0)
822H	2082	IA32_X2APIC_IRR2	Thread	x2APIC Interrupt Request register bits [95:64] (R/0)
823H	2083	IA32_X2APIC_IRR3	Thread	x2APIC Interrupt Request register bits [127:96] (R/0)
824H	2084	IA32_X2APIC_IRR4	Thread	x2APIC Interrupt Request register bits [159:128] (R/0)
825H	2085	IA32_X2APIC_IRR5	Thread	x2APIC Interrupt Request register bits [191:160] (R/0)
826H	2086	IA32_X2APIC_IRR6	Thread	x2APIC Interrupt Request register bits [223:192] (R/0)
827H	2087	IA32_X2APIC_IRR7	Thread	x2APIC Interrupt Request register bits [255:224] (R/0)
828H	2088	IA32_X2APIC_ESR	Thread	x2APIC Error Status register (R/W)

Table 2-42. Selected MSRs Supported by Intel® Xeon Phi™ Processors with DisplayFamily_DisplayModel Signatures 06_57H and 06_85H

Address		Register Name	Scope	Bit Description
Hex	Dec			
82FH	2095	IA32_X2APIC_LVT_CMCI	Thread	x2APIC LVT Corrected Machine Check Interrupt register (R/W)
830H	2096	IA32_X2APIC_ICR	Thread	x2APIC Interrupt Command register (R/W)
832H	2098	IA32_X2APIC_LVT_TIMER	Thread	x2APIC LVT Timer Interrupt register (R/W)
833H	2099	IA32_X2APIC_LVT_THERMAL	Thread	x2APIC LVT Thermal Sensor Interrupt register (R/W)
834H	2100	IA32_X2APIC_LVT_PMI	Thread	x2APIC LVT Performance Monitor register (R/W)
835H	2101	IA32_X2APIC_LVT_LINT0	Thread	x2APIC LVT LINT0 register (R/W)
836H	2102	IA32_X2APIC_LVT_LINT1	Thread	x2APIC LVT LINT1 register (R/W)
837H	2103	IA32_X2APIC_LVT_ERROR	Thread	x2APIC LVT Error register (R/W)
838H	2104	IA32_X2APIC_INIT_COUNT	Thread	x2APIC Initial Count register (R/W)
839H	2105	IA32_X2APIC_CUR_COUNT	Thread	x2APIC Current Count register (R/O)
83EH	2110	IA32_X2APIC_DIV_CONF	Thread	x2APIC Divide Configuration register (R/W)
83FH	2111	IA32_X2APIC_SELF_IPI	Thread	x2APIC Self IPI register (W/O)
C000_0080H		IA32_EFER	Thread	**Extended Feature Enables** See Table 2-2.
C000_0081H		IA32_STAR	Thread	**System Call Target Address (R/W)** See Table 2-2.
C000_0082H		IA32_LSTAR	Thread	**IA-32e Mode System Call Target Address (R/W)** See Table 2-2.
C000_0084H		IA32_FMASK	Thread	**System Call Flag Mask (R/W)** See Table 2-2.
C000_0100H		IA32_FS_BASE	Thread	**Map of BASE Address of FS (R/W)** See Table 2-2.
C000_0101H		IA32_GS_BASE	Thread	**Map of BASE Address of GS (R/W)** See Table 2-2.
C000_0102H		IA32_KERNEL_GS_BASE	Thread	**Swap Target of BASE Address of GS (R/W)** See Table 2-2.
C000_0103H		IA32_TSC_AUX	Thread	**AUXILIARY TSC Signature. (R/W)** See Table 2-2

Table 2-43 lists model-specific registers that are supported by future Intel® Xeon Phi™ Processors based on the Knights Mill microarchitecture.

Table 2-43. Additional MSRs Supported by Future Intel® Xeon Phi™ Processors with DisplayFamily_DisplayModel Signature 06_85H

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
9BH	155	IA32_SMM_MONITOR_CTL	Core	SMM Monitor Configuration (R/W). This MSR is readable only if VMX is enabled, and writeable only if VMX is enabled and in SMM mode, and is used to configure the VMX MSEG base address. See Table 2-2.
480H	1152	IA32_VMX_BASIC	Core	Reporting Register of Basic VMX Capabilities (R/O) See Table 2-2.
481H	1153	IA32_VMX_PINBASED_CTLS	Core	Capability Reporting Register of Pin-based VM-execution Controls (R/O) See Table 2-2.
482H	1154	IA32_VMX_PROCBASED_CTLS	Core	Capability Reporting Register of Primary Processor-based VM-execution Controls (R/O)
483H	1155	IA32_VMX_EXIT_CTLS	Core	Capability Reporting Register of VM-exit Controls (R/O) See Table 2-2.
484H	1156	IA32_VMX_ENTRY_CTLS	Core	Capability Reporting Register of VM-entry Controls (R/O) See Table 2-2.
485H	1157	IA32_VMX_MISC	Core	Reporting Register of Miscellaneous VMX Capabilities (R/O) See Table 2-2.
486H	1158	IA32_VMX_CR0_FIXED0	Core	Capability Reporting Register of CR0 Bits Fixed to 0 (R/O) See Table 2-2.
487H	1159	IA32_VMX_CR0_FIXED1	Core	Capability Reporting Register of CR0 Bits Fixed to 1 (R/O) See Table 2-2.
488H	1160	IA32_VMX_CR4_FIXED0	Core	Capability Reporting Register of CR4 Bits Fixed to 0 (R/O) See Table 2-2.
489H	1161	IA32_VMX_CR4_FIXED1	Core	Capability Reporting Register of CR4 Bits Fixed to 1 (R/O) See Table 2-2.
48AH	1162	IA32_VMX_VMCS_ENUM	Core	Capability Reporting Register of VMCS Field Enumeration (R/O) See Table 2-2.
48BH	1163	IA32_VMX_PROCBASED_CTLS2	Core	Capability Reporting Register of Secondary Processor-based VM-execution Controls (R/O) See Table 2-2.
48CH	1164	IA32_VMX_EPT_VPID_ENUM	Core	Capability Reporting Register of EPT and VPID (R/O) See Table 2-2.
48DH	1165	IA32_VMX_TRUE_PINBASED_CTLS	Core	Capability Reporting Register of Pin-based VM-execution Flex Controls (R/O) See Table 2-2.
48EH	1166	IA32_VMX_TRUE_PROCBASED_CTLS	Core	Capability Reporting Register of Primary Processor-based VM-execution Flex Controls (R/O) See Table 2-2.
48FH	1167	IA32_VMX_TRUE_EXIT_CTLS	Core	Capability Reporting Register of VM-exit Flex Controls (R/O) See Table 2-2.

Table 2-43. Additional MSRs Supported by Future Intel® Xeon Phi™ Processors with DisplayFamily_DisplayModel Signature 06_85H

Register Address		Register Name	Scope	Bit Description
Hex	Dec			
490H	1168	IA32_VMX_TRUE_ENTRY_CTLS	Core	Capability Reporting Register of VM-entry Flex Controls (R/O) See Table 2-2.
491H	1169	IA32_VMX_FMFUNC	Core	Capability Reporting Register of VM-function Controls (R/O) See Table 2-2.

2.18 MSRS IN THE PENTIUM® 4 AND INTEL® XEON® PROCESSORS

Table 2-44 lists MSRs (architectural and model-specific) that are defined across processor generations based on Intel NetBurst microarchitecture. The processor can be identified by its CPUID signatures of DisplayFamily encoding of 0FH, see Table 2-1.

- MSRs with an "IA32_" prefix are designated as "architectural." This means that the functions of these MSRs and their addresses remain the same for succeeding families of IA-32 processors.
- MSRs with an "MSR_" prefix are model specific with respect to address functionalities. The column "Model Availability" lists the model encoding value(s) within the Pentium 4 and Intel Xeon processor family at the specified register address. The model encoding value of a processor can be queried using CPUID. See "CPUID—CPU Identification" in Chapter 3 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A*.

Table 2-44. MSRs in the Pentium® 4 and Intel® Xeon® Processors

Register Address		Register Name Fields and Flags	Model Availability	Shared/ Unique[1]	Bit Description
Hex	Dec				
0H	0	IA32_P5_MC_ADDR	0, 1, 2, 3, 4, 6	Shared	See Section 2.22, "MSRs in Pentium Processors."
1H	1	IA32_P5_MC_TYPE	0, 1, 2, 3, 4, 6	Shared	See Section 2.22, "MSRs in Pentium Processors."
6H	6	IA32_MONITOR_FILTER_LINE_SIZE	3, 4, 6	Shared	See Section 8.10.5, "Monitor/Mwait Address Range Determination."
10H	16	IA32_TIME_STAMP_COUNTER	0, 1, 2, 3, 4, 6	Unique	**Time Stamp Counter** See Table 2-2.
					On earlier processors, only the lower 32 bits are writable. On any write to the lower 32 bits, the upper 32 bits are cleared. For processor family 0FH, models 3 and 4: all 64 bits are writable.
17H	23	IA32_PLATFORM_ID	0, 1, 2, 3, 4, 6	Shared	**Platform ID (R)** See Table 2-2. The operating system can use this MSR to determine "slot" information for the processor and the proper microcode update to load.
1BH	27	IA32_APIC_BASE	0, 1, 2, 3, 4, 6	Unique	**APIC Location and Status (R/W)** See Table 2-2. See Section 10.4.4, "Local APIC Status and Location."

Table 2-44. MSRs in the Pentium® 4 and Intel® Xeon® Processors (Contd.)

Register Address		Register Name Fields and Flags	Model Avail-ability	Shared/ Unique[1]	Bit Description
Hex	Dec				
2AH	42	MSR_EBC_HARD_POWERON	0, 1, 2, 3, 4, 6	Shared	**Processor Hard Power-On Configuration** **(R/W)** Enables and disables processor features; **(R)** indicates current processor configuration.
		0			**Output Tri-state Enabled (R)** Indicates whether tri-state output is enabled (1) or disabled (0) as set by the strapping of SMI#. The value in this bit is written on the deassertion of RESET#; the bit is set to 1 when the address bus signal is asserted.
		1			**Execute BIST (R)** Indicates whether the execution of the BIST is enabled (1) or disabled (0) as set by the strapping of INIT#. The value in this bit is written on the deassertion of RESET#; the bit is set to 1 when the address bus signal is asserted.
		2			**In Order Queue Depth (R)** Indicates whether the in order queue depth for the system bus is 1 (1) or up to 12 (0) as set by the strapping of A7#. The value in this bit is written on the deassertion of RESET#; the bit is set to 1 when the address bus signal is asserted.
		3			**MCERR# Observation Disabled (R)** Indicates whether MCERR# observation is enabled (0) or disabled (1) as determined by the strapping of A9#. The value in this bit is written on the deassertion of RESET#; the bit is set to 1 when the address bus signal is asserted.
		4			**BINIT# Observation Enabled (R)** Indicates whether BINIT# observation is enabled (0) or disabled (1) as determined by the strapping of A10#. The value in this bit is written on the deassertion of RESET#; the bit is set to 1 when the address bus signal is asserted.
		6:5			**APIC Cluster ID (R)** Contains the logical APIC cluster ID value as set by the strapping of A12# and A11#. The logical cluster ID value is written into the field on the deassertion of RESET#; the field is set to 1 when the address bus signal is asserted.
		7			**Bus Park Disable (R)** Indicates whether bus park is enabled (0) or disabled (1) as set by the strapping of A15#. The value in this bit is written on the deassertion of RESET#; the bit is set to 1 when the address bus signal is asserted.
		11:8			Reserved.

Table 2-44. MSRs in the Pentium® 4 and Intel® Xeon® Processors (Contd.)

Register Address		Register Name Fields and Flags	Model Avail-ability	Shared/ Unique[1]	Bit Description
Hex	Dec				
		13:12			**Agent ID (R)** Contains the logical agent ID value as set by the strapping of BR[3:0]. The logical ID value is written into the field on the deassertion of RESET#; the field is set to 1 when the address bus signal is asserted.
		63:14			Reserved.
2BH	43	MSR_EBC_SOFT_POWERON	0, 1, 2, 3, 4, 6	Shared	**Processor Soft Power-On Configuration (R/W)** Enables and disables processor features.
		0			**RCNT/SCNT On Request Encoding Enable (R/W)** Controls the driving of RCNT/SCNT on the request encoding. Set to enable (1); clear to disabled (0, default).
		1			**Data Error Checking Disable (R/W)** Set to disable system data bus parity checking; clear to enable parity checking.
		2			**Response Error Checking Disable (R/W)** Set to disable (default); clear to enable.
		3			**Address/Request Error Checking Disable (R/W)** Set to disable (default); clear to enable.
		4			**Initiator MCERR# Disable (R/W)** Set to disable MCERR# driving for initiator bus requests (default); clear to enable.
		5			**Internal MCERR# Disable (R/W)** Set to disable MCERR# driving for initiator internal errors (default); clear to enable.
		6			**BINIT# Driver Disable (R/W)** Set to disable BINIT# driver (default); clear to enable driver.
		63:7			Reserved.
2CH	44	MSR_EBC_FREQUENCY_ID	2,3, 4, 6	Shared	**Processor Frequency Configuration** The bit field layout of this MSR varies according to the MODEL value in the CPUID version information. The following bit field layout applies to Pentium 4 and Xeon Processors with MODEL encoding equal or greater than 2. (R) The field Indicates the current processor frequency configuration.
		15:0			Reserved.

Table 2-44. MSRs in the Pentium® 4 and Intel® Xeon® Processors (Contd.)

Register Address		Register Name Fields and Flags	Model Avail-ability	Shared/ Unique[1]	Bit Description
Hex	Dec				
		18:16			**Scalable Bus Speed (R/W)** Indicates the intended scalable bus speed: Encoding Scalable Bus Speed 000B 100 MHz (Model 2) 000B 266 MHz (Model 3 or 4) 001B 133 MHz 010B 200 MHz 011B 166 MHz 100B 333 MHz (Model 6)
					133.33 MHz should be utilized if performing calculation with System Bus Speed when encoding is 001B. 166.67 MHz should be utilized if performing calculation with System Bus Speed when encoding is 011B.
					266.67 MHz should be utilized if performing calculation with System Bus Speed when encoding is 000B and model encoding = 3 or 4. 333.33 MHz should be utilized if performing calculation with System Bus Speed when encoding is 100B and model encoding = 6. All other values are reserved.
		23:19			Reserved.
		31:24			**Core Clock Frequency to System Bus Frequency Ratio (R)** The processor core clock frequency to system bus frequency ratio observed at the de-assertion of the reset pin.
		63:25			Reserved.
2CH	44	MSR_EBC_FREQUENCY_ID	0, 1	Shared	**Processor Frequency Configuration (R)** The bit field layout of this MSR varies according to the MODEL value of the CPUID version information. This bit field layout applies to Pentium 4 and Xeon Processors with MODEL encoding less than 2. Indicates current processor frequency configuration.
		20:0			Reserved.
		23:21			**Scalable Bus Speed (R/W)** Indicates the intended scalable bus speed: Encoding Scalable Bus Speed 000B 100 MHz All others values reserved.
		63:24			Reserved.

Table 2-44. MSRs in the Pentium® 4 and Intel® Xeon® Processors (Contd.)

Register Address		Register Name Fields and Flags	Model Avail-ability	Shared/ Unique[1]	Bit Description
Hex	Dec				
3AH	58	IA32_FEATURE_CONTROL	3, 4, 6	Unique	**Control Features in IA-32 Processor (R/W)** See Table 2-2 (If CPUID.01H:ECX.[bit 5])
79H	121	IA32_BIOS_UPDT_TRIG	0, 1, 2, 3, 4, 6	Shared	**BIOS Update Trigger Register (W)** See Table 2-2.
8BH	139	IA32_BIOS_SIGN_ID	0, 1, 2, 3, 4, 6	Unique	**BIOS Update Signature ID (R/W)** See Table 2-2.
9BH	155	IA32_SMM_MONITOR_CTL	3, 4, 6	Unique	**SMM Monitor Configuration (R/W)** See Table 2-2.
FEH	254	IA32_MTRRCAP	0, 1, 2, 3, 4, 6	Unique	**MTRR Information** See Section 11.11.1, "MTRR Feature Identification.".
174H	372	IA32_SYSENTER_CS	0, 1, 2, 3, 4, 6	Unique	**CS register target for CPL 0 code (R/W)** See Table 2-2. See Section 5.8.7, "Performing Fast Calls to System Procedures with the SYSENTER and SYSEXIT Instructions."
175H	373	IA32_SYSENTER_ESP	0, 1, 2, 3, 4, 6	Unique	**Stack pointer for CPL 0 stack (R/W)** See Table 2-2. See Section 5.8.7, "Performing Fast Calls to System Procedures with the SYSENTER and SYSEXIT Instructions."
176H	374	IA32_SYSENTER_EIP	0, 1, 2, 3, 4, 6	Unique	**CPL 0 code entry point (R/W)** See Table 2-2. See Section 5.8.7, "Performing Fast Calls to System Procedures with the SYSENTER and SYSEXIT Instructions."
179H	377	IA32_MCG_CAP	0, 1, 2, 3, 4, 6	Unique	**Machine Check Capabilities (R)** See Table 2-2. See Section 15.3.1.1, "IA32_MCG_CAP MSR."
17AH	378	IA32_MCG_STATUS	0, 1, 2, 3, 4, 6	Unique	**Machine Check Status. (R)** See Table 2-2. See Section 15.3.1.2, "IA32_MCG_STATUS MSR."
17BH	379	IA32_MCG_CTL			**Machine Check Feature Enable (R/W)** See Table 2-2. See Section 15.3.1.3, "IA32_MCG_CTL MSR."
180H	384	MSR_MCG_RAX	0, 1, 2, 3, 4, 6	Unique	**Machine Check EAX/RAX Save State** See Section 15.3.2.6, "IA32_MCG Extended Machine Check State MSRs."
		63:0			Contains register state at time of machine check error. When in non-64-bit modes at the time of the error, bits 63-32 do not contain valid data.

Table 2-44. MSRs in the Pentium® 4 and Intel® Xeon® Processors (Contd.)

Register Address		Register Name Fields and Flags	Model Avail-ability	Shared/ Unique[1]	Bit Description
Hex	Dec				
181H	385	MSR_MCG_RBX	0, 1, 2, 3, 4, 6	Unique	**Machine Check EBX/RBX Save State** See Section 15.3.2.6, "IA32_MCG Extended Machine Check State MSRs."
		63:0			Contains register state at time of machine check error. When in non-64-bit modes at the time of the error, bits 63-32 do not contain valid data.
182H	386	MSR_MCG_RCX	0, 1, 2, 3, 4, 6	Unique	**Machine Check ECX/RCX Save State** See Section 15.3.2.6, "IA32_MCG Extended Machine Check State MSRs."
		63:0			Contains register state at time of machine check error. When in non-64-bit modes at the time of the error, bits 63-32 do not contain valid data.
183H	387	MSR_MCG_RDX	0, 1, 2, 3, 4, 6	Unique	**Machine Check EDX/RDX Save State** See Section 15.3.2.6, "IA32_MCG Extended Machine Check State MSRs."
		63:0			Contains register state at time of machine check error. When in non-64-bit modes at the time of the error, bits 63-32 do not contain valid data.
184H	388	MSR_MCG_RSI	0, 1, 2, 3, 4, 6	Unique	**Machine Check ESI/RSI Save State** See Section 15.3.2.6, "IA32_MCG Extended Machine Check State MSRs."
		63:0			Contains register state at time of machine check error. When in non-64-bit modes at the time of the error, bits 63-32 do not contain valid data.
185H	389	MSR_MCG_RDI	0, 1, 2, 3, 4, 6	Unique	**Machine Check EDI/RDI Save State** See Section 15.3.2.6, "IA32_MCG Extended Machine Check State MSRs."
		63:0			Contains register state at time of machine check error. When in non-64-bit modes at the time of the error, bits 63-32 do not contain valid data.
186H	390	MSR_MCG_RBP	0, 1, 2, 3, 4, 6	Unique	**Machine Check EBP/RBP Save State** See Section 15.3.2.6, "IA32_MCG Extended Machine Check State MSRs."
		63:0			Contains register state at time of machine check error. When in non-64-bit modes at the time of the error, bits 63-32 do not contain valid data.
187H	391	MSR_MCG_RSP	0, 1, 2, 3, 4, 6	Unique	**Machine Check ESP/RSP Save State** See Section 15.3.2.6, "IA32_MCG Extended Machine Check State MSRs."
		63:0			Contains register state at time of machine check error. When in non-64-bit modes at the time of the error, bits 63-32 do not contain valid data.
188H	392	MSR_MCG_RFLAGS	0, 1, 2, 3, 4, 6	Unique	**Machine Check EFLAGS/RFLAG Save State** See Section 15.3.2.6, "IA32_MCG Extended Machine Check State MSRs."

Table 2-44. MSRs in the Pentium® 4 and Intel® Xeon® Processors (Contd.)

Register Address		Register Name Fields and Flags	Model Avail-ability	Shared/ Unique[1]	Bit Description
Hex	Dec				
		63:0			Contains register state at time of machine check error. When in non-64-bit modes at the time of the error, bits 63-32 do not contain valid data.
189H	393	MSR_MCG_RIP	0, 1, 2, 3, 4, 6	Unique	**Machine Check EIP/RIP Save State** See Section 15.3.2.6, "IA32_MCG Extended Machine Check State MSRs."
		63:0			Contains register state at time of machine check error. When in non-64-bit modes at the time of the error, bits 63-32 do not contain valid data.
18AH	394	MSR_MCG_MISC	0, 1, 2, 3, 4, 6	Unique	**Machine Check Miscellaneous** See Section 15.3.2.6, "IA32_MCG Extended Machine Check State MSRs."
		0			**DS** When set, the bit indicates that a page assist or page fault occurred during DS normal operation. The processors response is to shut down. The bit is used as an aid for debugging DS handling code. It is the responsibility of the user (BIOS or operating system) to clear this bit for normal operation.
		63:1			Reserved.
18BH-18FH	395	MSR_MCG_RESERVED1 - MSR_MCG_RESERVED5			Reserved.
190H	400	MSR_MCG_R8	0, 1, 2, 3, 4, 6	Unique	**Machine Check R8** See Section 15.3.2.6, "IA32_MCG Extended Machine Check State MSRs."
		63:0			Registers R8-15 (and the associated state-save MSRs) exist only in Intel 64 processors. These registers contain valid information only when the processor is operating in 64-bit mode at the time of the error.
191H	401	MSR_MCG_R9	0, 1, 2, 3, 4, 6	Unique	**Machine Check R9D/R9** See Section 15.3.2.6, "IA32_MCG Extended Machine Check State MSRs."
		63:0			Registers R8-15 (and the associated state-save MSRs) exist only in Intel 64 processors. These registers contain valid information only when the processor is operating in 64-bit mode at the time of the error.
192H	402	MSR_MCG_R10	0, 1, 2, 3, 4, 6	Unique	**Machine Check R10** See Section 15.3.2.6, "IA32_MCG Extended Machine Check State MSRs."

Table 2-44. MSRs in the Pentium® 4 and Intel® Xeon® Processors (Contd.)

Register Address		Register Name Fields and Flags	Model Avail-ability	Shared/ Unique[1]	Bit Description
Hex	Dec				
		63:0			Registers R8-15 (and the associated state-save MSRs) exist only in Intel 64 processors. These registers contain valid information only when the processor is operating in 64-bit mode at the time of the error.
193H	403	MSR_MCG_R11	0, 1, 2, 3, 4, 6	Unique	**Machine Check R11** See Section 15.3.2.6, "IA32_MCG Extended Machine Check State MSRs."
		63:0			Registers R8-15 (and the associated state-save MSRs) exist only in Intel 64 processors. These registers contain valid information only when the processor is operating in 64-bit mode at the time of the error.
194H	404	MSR_MCG_R12	0, 1, 2, 3, 4, 6	Unique	**Machine Check R12** See Section 15.3.2.6, "IA32_MCG Extended Machine Check State MSRs."
		63:0			Registers R8-15 (and the associated state-save MSRs) exist only in Intel 64 processors. These registers contain valid information only when the processor is operating in 64-bit mode at the time of the error.
195H	405	MSR_MCG_R13	0, 1, 2, 3, 4, 6	Unique	**Machine Check R13** See Section 15.3.2.6, "IA32_MCG Extended Machine Check State MSRs."
		63:0			Registers R8-15 (and the associated state-save MSRs) exist only in Intel 64 processors. These registers contain valid information only when the processor is operating in 64-bit mode at the time of the error.
196H	406	MSR_MCG_R14	0, 1, 2, 3, 4, 6	Unique	**Machine Check R14** See Section 15.3.2.6, "IA32_MCG Extended Machine Check State MSRs."
		63:0			Registers R8-15 (and the associated state-save MSRs) exist only in Intel 64 processors. These registers contain valid information only when the processor is operating in 64-bit mode at the time of the error.
197H	407	MSR_MCG_R15	0, 1, 2, 3, 4, 6	Unique	**Machine Check R15** See Section 15.3.2.6, "IA32_MCG Extended Machine Check State MSRs."
		63:0			Registers R8-15 (and the associated state-save MSRs) exist only in Intel 64 processors. These registers contain valid information only when the processor is operating in 64-bit mode at the time of the error.
198H	408	IA32_PERF_STATUS	3, 4, 6	Unique	See Table 2-2. See Section 14.1, "Enhanced Intel Speedstep® Technology."

Table 2-44. MSRs in the Pentium® 4 and Intel® Xeon® Processors (Contd.)

Register Address		Register Name Fields and Flags	Model Avail-ability	Shared/ Unique[1]	Bit Description
Hex	Dec				
199H	409	IA32_PERF_CTL	3, 4, 6	Unique	See Table 2-2. See Section 14.1, "Enhanced Intel Speedstep® Technology."
19AH	410	IA32_CLOCK_MODULATION	0, 1, 2, 3, 4, 6	Unique	**Thermal Monitor Control (R/W)** See Table 2-2. See Section 14.7.3, "Software Controlled Clock Modulation."
19BH	411	IA32_THERM_INTERRUPT	0, 1, 2, 3, 4, 6	Unique	**Thermal Interrupt Control (R/W)** See Section 14.7.2, "Thermal Monitor," and see Table 2-2.
19CH	412	IA32_THERM_STATUS	0, 1, 2, 3, 4, 6	Shared	**Thermal Monitor Status (R/W)** See Section 14.7.2, "Thermal Monitor," and see Table 2-2.
19DH	413	MSR_THERM2_CTL			Thermal Monitor 2 Control.
			3,	Shared	For Family F, Model 3 processors: When read, specifies the value of the target TM2 transition last written. When set, it sets the next target value for TM2 transition.
			4, 6	Shared	For Family F, Model 4 and Model 6 processors: When read, specifies the value of the target TM2 transition last written. Writes may cause #GP exceptions.
1A0H	416	IA32_MISC_ENABLE	0, 1, 2, 3, 4, 6	Shared	**Enable Miscellaneous Processor Features (R/W)**
		0			Fast-Strings Enable. See Table 2-2.
		1			Reserved.
		2			**x87 FPU Fopcode Compatibility Mode Enable**
		3			**Thermal Monitor 1 Enable** See Section 14.7.2, "Thermal Monitor," and see Table 2-2.
		4			**Split-Lock Disable** When set, the bit causes an #AC exception to be issued instead of a split-lock cycle. Operating systems that set this bit must align system structures to avoid split-lock scenarios. When the bit is clear (default), normal split-locks are issued to the bus.
					This debug feature is specific to the Pentium 4 processor.
		5			Reserved.

Table 2-44. MSRs in the Pentium® 4 and Intel® Xeon® Processors (Contd.)

Register Address		Register Name Fields and Flags	Model Avail-ability	Shared/ Unique[1]	Bit Description
Hex	Dec				
		6			**Third-Level Cache Disable (R/W)** When set, the third-level cache is disabled; when clear (default) the third-level cache is enabled. This flag is reserved for processors that do not have a third-level cache. Note that the bit controls only the third-level cache; and only if overall caching is enabled through the CD flag of control register CR0, the page-level cache controls, and/or the MTRRs. See Section 11.5.4, "Disabling and Enabling the L3 Cache."
		7			**Performance Monitoring Available (R)** See Table 2-2.
		8			**Suppress Lock Enable** When set, assertion of LOCK on the bus is suppressed during a Split Lock access. When clear (default), LOCK is not suppressed.
		9			**Prefetch Queue Disable** When set, disables the prefetch queue. When clear (default), enables the prefetch queue.
		10			**FERR# Interrupt Reporting Enable (R/W)** When set, interrupt reporting through the FERR# pin is enabled; when clear, this interrupt reporting function is disabled.
					When this flag is set and the processor is in the stop-clock state (STPCLK# is asserted), asserting the FERR# pin signals to the processor that an interrupt (such as, INIT#, BINIT#, INTR, NMI, SMI#, or RESET#) is pending and that the processor should return to normal operation to handle the interrupt.
					This flag does not affect the normal operation of the FERR# pin (to indicate an unmasked floating-point error) when the STPCLK# pin is not asserted.
		11			**Branch Trace Storage Unavailable (BTS_UNAVILABLE) (R)** See Table 2-2. When set, the processor does not support branch trace storage (BTS); when clear, BTS is supported.
		12			**PEBS_UNAVILABLE: Processor Event Based Sampling Unavailable (R)** See Table 2-2. When set, the processor does not support processor event-based sampling (PEBS); when clear, PEBS is supported.

Table 2-44. MSRs in the Pentium® 4 and Intel® Xeon® Processors (Contd.)

Register Address		Register Name Fields and Flags	Model Avail-ability	Shared/ Unique[1]	Bit Description
Hex	Dec				
		13	3		**TM2 Enable (R/W)** When this bit is set (1) and the thermal sensor indicates that the die temperature is at the pre-determined threshold, the Thermal Monitor 2 mechanism is engaged. TM2 will reduce the bus to core ratio and voltage according to the value last written to MSR_THERM2_CTL bits 15:0. When this bit is clear (0, default), the processor does not change the VID signals or the bus to core ratio when the processor enters a thermal managed state. If the TM2 feature flag (ECX[8]) is not set to 1 after executing CPUID with EAX = 1, then this feature is not supported and BIOS must not alter the contents of this bit location. The processor is operating out of spec if both this bit and the TM1 bit are set to disabled states.
		17:14			Reserved.
		18	3, 4, 6		**ENABLE MONITOR FSM (R/W)** See Table 2-2.
		19			**Adjacent Cache Line Prefetch Disable (R/W)** When set to 1, the processor fetches the cache line of the 128-byte sector containing currently required data. When set to 0, the processor fetches both cache lines in the sector.
					Single processor platforms should not set this bit. Server platforms should set or clear this bit based on platform performance observed in validation and testing. BIOS may contain a setup option that controls the setting of this bit.
		21:20			Reserved.
		22	3, 4, 6		**Limit CPUID MAXVAL (R/W)** See Table 2-2. Setting this can cause unexpected behavior to software that depends on the availability of CPUID leaves greater than 3.
		23		Shared	**xTPR Message Disable (R/W)** See Table 2-2.

Table 2-44. MSRs in the Pentium® 4 and Intel® Xeon® Processors (Contd.)

Register Address		Register Name Fields and Flags	Model Avail-ability	Shared/ Unique[1]	Bit Description
Hex	Dec				
		24			**L1 Data Cache Context Mode (R/W)** When set, the L1 data cache is placed in shared mode; when clear (default), the cache is placed in adaptive mode. This bit is only enabled for IA-32 processors that support Intel Hyper-Threading Technology. See Section 11.5.6, "L1 Data Cache Context Mode." When L1 is running in adaptive mode and CR3s are identical, data in L1 is shared across logical processors. Otherwise, L1 is not shared and cache use is competitive. If the Context ID feature flag (ECX[10]) is set to 0 after executing CPUID with EAX = 1, the ability to switch modes is not supported. BIOS must not alter the contents of IA32_MISC_ENABLE[24].
		33:25			Reserved.
		34		Unique	**XD Bit Disable (R/W)** See Table 2-2.
		63:35			Reserved.
1A1H	417	MSR_PLATFORM_BRV	3, 4, 6	Shared	**Platform Feature Requirements (R)**
		17:0			Reserved.
		18			**PLATFORM Requirements** When set to 1, indicates the processor has specific platform requirements. The details of the platform requirements are listed in the respective data sheets of the processor.
		63:19			Reserved.
1D7H	471	MSR_LER_FROM_LIP	0, 1, 2, 3, 4, 6	Unique	**Last Exception Record From Linear IP (R)** Contains a pointer to the last branch instruction that the processor executed prior to the last exception that was generated or the last interrupt that was handled. See Section 17.13.3, "Last Exception Records."
		31:0			**From Linear IP** Linear address of the last branch instruction.
		63:32			Reserved.
1D7H	471	63:0		Unique	**From Linear IP** Linear address of the last branch instruction (If IA-32e mode is active).
1D8H	472	MSR_LER_TO_LIP	0, 1, 2, 3, 4, 6	Unique	**Last Exception Record To Linear IP (R)** This area contains a pointer to the target of the last branch instruction that the processor executed prior to the last exception that was generated or the last interrupt that was handled. See Section 17.13.3, "Last Exception Records."

Table 2-44. MSRs in the Pentium® 4 and Intel® Xeon® Processors (Contd.)

Register Address		Register Name Fields and Flags	Model Avail-ability	Shared/ Unique[1]	Bit Description
Hex	Dec				
		31:0			**From Linear IP** Linear address of the target of the last branch instruction.
		63:32			Reserved.
1D8H	472	63:0		Unique	**From Linear IP** Linear address of the target of the last branch instruction (If IA-32e mode is active).
1D9H	473	MSR_DEBUGCTLA	0, 1, 2, 3, 4, 6	Unique	**Debug Control (R/W)** Controls how several debug features are used. Bit definitions are discussed in the referenced section. See Section 17.13.1, "MSR_DEBUGCTLA MSR."
1DAH	474	MSR_LASTBRANCH _TOS	0, 1, 2, 3, 4, 6	Unique	**Last Branch Record Stack TOS (R/O)** Contains an index (0-3 or 0-15) that points to the top of the last branch record stack (that is, that points the index of the MSR containing the most recent branch record). See Section 17.13.2, "LBR Stack for Processors Based on Intel NetBurst® Microarchitecture"; and addresses 1DBH-1DEH and 680H-68FH.
1DBH	475	MSR_LASTBRANCH_0	0, 1, 2	Unique	**Last Branch Record 0 (R/O)** One of four last branch record registers on the last branch record stack. It contains pointers to the source and destination instruction for one of the last four branches, exceptions, or interrupts that the processor took. MSR_LASTBRANCH_0 through MSR_LASTBRANCH_3 at 1DBH-1DEH are available only on family 0FH, models 0H-02H. They have been replaced by the MSRs at 680H-68FH and 6C0H-6CFH.
					See Section 17.12, "Last Branch, Call Stack, Interrupt, and Exception Recording for Processors based on Skylake Microarchitecture."
1DCH	477	MSR_LASTBRANCH_1	0, 1, 2	Unique	**Last Branch Record 1** See description of the MSR_LASTBRANCH_0 MSR at 1DBH.
1DDH	477	MSR_LASTBRANCH_2	0, 1, 2	Unique	**Last Branch Record 2** See description of the MSR_LASTBRANCH_0 MSR at 1DBH.
1DEH	478	MSR_LASTBRANCH_3	0, 1, 2	Unique	**Last Branch Record 3** See description of the MSR_LASTBRANCH_0 MSR at 1DBH.
200H	512	IA32_MTRR_PHYSBASE0	0, 1, 2, 3, 4, 6	Shared	**Variable Range Base MTRR** See Section 11.11.2.3, "Variable Range MTRRs."

Table 2-44. MSRs in the Pentium® 4 and Intel® Xeon® Processors (Contd.)

Register Address		Register Name Fields and Flags	Model Avail-ability	Shared/ Unique[1]	Bit Description
Hex	Dec				
201H	513	IA32_MTRR_PHYSMASK0	0, 1, 2, 3, 4, 6	Shared	**Variable Range Mask MTRR** See Section 11.11.2.3, "Variable Range MTRRs."
202H	514	IA32_MTRR_PHYSBASE1	0, 1, 2, 3, 4, 6	Shared	**Variable Range Mask MTRR** See Section 11.11.2.3, "Variable Range MTRRs."
203H	515	IA32_MTRR_PHYSMASK1	0, 1, 2, 3, 4, 6	Shared	**Variable Range Mask MTRR** See Section 11.11.2.3, "Variable Range MTRRs."
204H	516	IA32_MTRR_PHYSBASE2	0, 1, 2, 3, 4, 6	Shared	**Variable Range Mask MTRR** See Section 11.11.2.3, "Variable Range MTRRs."
205H	517	IA32_MTRR_PHYSMASK2	0, 1, 2, 3, 4, 6	Shared	**Variable Range Mask MTRR** See Section 11.11.2.3, "Variable Range MTRRs".
206H	518	IA32_MTRR_PHYSBASE3	0, 1, 2, 3, 4, 6	Shared	**Variable Range Mask MTRR** See Section 11.11.2.3, "Variable Range MTRRs."
207H	519	IA32_MTRR_PHYSMASK3	0, 1, 2, 3, 4, 6	Shared	**Variable Range Mask MTRR** See Section 11.11.2.3, "Variable Range MTRRs."
208H	520	IA32_MTRR_PHYSBASE4	0, 1, 2, 3, 4, 6	Shared	**Variable Range Mask MTRR** See Section 11.11.2.3, "Variable Range MTRRs."
209H	521	IA32_MTRR_PHYSMASK4	0, 1, 2, 3, 4, 6	Shared	**Variable Range Mask MTRR** See Section 11.11.2.3, "Variable Range MTRRs."
20AH	522	IA32_MTRR_PHYSBASE5	0, 1, 2, 3, 4, 6	Shared	**Variable Range Mask MTRR** See Section 11.11.2.3, "Variable Range MTRRs."
20BH	523	IA32_MTRR_PHYSMASK5	0, 1, 2, 3, 4, 6	Shared	**Variable Range Mask MTRR** See Section 11.11.2.3, "Variable Range MTRRs."
20CH	524	IA32_MTRR_PHYSBASE6	0, 1, 2, 3, 4, 6	Shared	**Variable Range Mask MTRR** See Section 11.11.2.3, "Variable Range MTRRs."
20DH	525	IA32_MTRR_PHYSMASK6	0, 1, 2, 3, 4, 6	Shared	**Variable Range Mask MTRR** See Section 11.11.2.3, "Variable Range MTRRs."
20EH	526	IA32_MTRR_PHYSBASE7	0, 1, 2, 3, 4, 6	Shared	**Variable Range Mask MTRR** See Section 11.11.2.3, "Variable Range MTRRs."
20FH	527	IA32_MTRR_PHYSMASK7	0, 1, 2, 3, 4, 6	Shared	**Variable Range Mask MTRR** See Section 11.11.2.3, "Variable Range MTRRs."
250H	592	IA32_MTRR_FIX64K_00000	0, 1, 2, 3, 4, 6	Shared	**Fixed Range MTRR** See Section 11.11.2.2, "Fixed Range MTRRs."
258H	600	IA32_MTRR_FIX16K_80000	0, 1, 2, 3, 4, 6	Shared	**Fixed Range MTRR** See Section 11.11.2.2, "Fixed Range MTRRs."
259H	601	IA32_MTRR_FIX16K_A0000	0, 1, 2, 3, 4, 6	Shared	**Fixed Range MTRR** See Section 11.11.2.2, "Fixed Range MTRRs."
268H	616	IA32_MTRR_FIX4K_C0000	0, 1, 2, 3, 4, 6	Shared	**Fixed Range MTRR** See Section 11.11.2.2, "Fixed Range MTRRs."

Table 2-44. MSRs in the Pentium® 4 and Intel® Xeon® Processors (Contd.)

Register Address		Register Name Fields and Flags	Model Avail-ability	Shared/ Unique[1]	Bit Description
Hex	Dec				
269H	617	IA32_MTRR_FIX4K_C8000	0, 1, 2, 3, 4, 6	Shared	**Fixed Range MTRR** See Section 11.11.2.2, "Fixed Range MTRRs".
26AH	618	IA32_MTRR_FIX4K_D0000	0, 1, 2, 3, 4, 6	Shared	**Fixed Range MTRR** See Section 11.11.2.2, "Fixed Range MTRRs".
26BH	619	IA32_MTRR_FIX4K_D8000	0, 1, 2, 3, 4, 6	Shared	**Fixed Range MTRR** See Section 11.11.2.2, "Fixed Range MTRRs."
26CH	620	IA32_MTRR_FIX4K_E0000	0, 1, 2, 3, 4, 6	Shared	**Fixed Range MTRR** See Section 11.11.2.2, "Fixed Range MTRRs."
26DH	621	IA32_MTRR_FIX4K_E8000	0, 1, 2, 3, 4, 6	Shared	**Fixed Range MTRR** See Section 11.11.2.2, "Fixed Range MTRRs."
26EH	622	IA32_MTRR_FIX4K_F0000	0, 1, 2, 3, 4, 6	Shared	**Fixed Range MTRR** See Section 11.11.2.2, "Fixed Range MTRRs."
26FH	623	IA32_MTRR_FIX4K_F8000	0, 1, 2, 3, 4, 6	Shared	**Fixed Range MTRR** See Section 11.11.2.2, "Fixed Range MTRRs."
277H	631	IA32_PAT	0, 1, 2, 3, 4, 6	Unique	**Page Attribute Table** See Section 11.11.2.2, "Fixed Range MTRRs."
2FFH	767	IA32_MTRR_DEF_TYPE	0, 1, 2, 3, 4, 6	Shared	**Default Memory Types (R/W)** See Table 2-2. See Section 11.11.2.1, "IA32_MTRR_DEF_TYPE MSR."
300H	768	MSR_BPU_COUNTER0	0, 1, 2, 3, 4, 6	Shared	See Section 18.6.3.2, "Performance Counters."
301H	769	MSR_BPU_COUNTER1	0, 1, 2, 3, 4, 6	Shared	See Section 18.6.3.2, "Performance Counters."
302H	770	MSR_BPU_COUNTER2	0, 1, 2, 3, 4, 6	Shared	See Section 18.6.3.2, "Performance Counters."
303H	771	MSR_BPU_COUNTER3	0, 1, 2, 3, 4, 6	Shared	See Section 18.6.3.2, "Performance Counters."
304H	772	MSR_MS_COUNTER0	0, 1, 2, 3, 4, 6	Shared	See Section 18.6.3.2, "Performance Counters."
305H	773	MSR_MS_COUNTER1	0, 1, 2, 3, 4, 6	Shared	See Section 18.6.3.2, "Performance Counters."
306H	774	MSR_MS_COUNTER2	0, 1, 2, 3, 4, 6	Shared	See Section 18.6.3.2, "Performance Counters."
307H	775	MSR_MS_COUNTER3	0, 1, 2, 3, 4, 6	Shared	See Section 18.6.3.2, "Performance Counters."
308H	776	MSR_FLAME_COUNTER0	0, 1, 2, 3, 4, 6	Shared	See Section 18.6.3.2, "Performance Counters."
309H	777	MSR_FLAME_COUNTER1	0, 1, 2, 3, 4, 6	Shared	See Section 18.6.3.2, "Performance Counters."
30AH	778	MSR_FLAME_COUNTER2	0, 1, 2, 3, 4, 6	Shared	See Section 18.6.3.2, "Performance Counters."

Table 2-44. MSRs in the Pentium® 4 and Intel® Xeon® Processors (Contd.)

Register Address		Register Name Fields and Flags	Model Avail-ability	Shared/ Unique[1]	Bit Description
Hex	Dec				
30BH	779	MSR_FLAME_COUNTER3	0, 1, 2, 3, 4, 6	Shared	See Section 18.6.3.2, "Performance Counters."
30CH	780	MSR_IQ_COUNTER0	0, 1, 2, 3, 4, 6	Shared	See Section 18.6.3.2, "Performance Counters."
30DH	781	MSR_IQ_COUNTER1	0, 1, 2, 3, 4, 6	Shared	See Section 18.6.3.2, "Performance Counters."
30EH	782	MSR_IQ_COUNTER2	0, 1, 2, 3, 4, 6	Shared	See Section 18.6.3.2, "Performance Counters."
30FH	783	MSR_IQ_COUNTER3	0, 1, 2, 3, 4, 6	Shared	See Section 18.6.3.2, "Performance Counters."
310H	784	MSR_IQ_COUNTER4	0, 1, 2, 3, 4, 6	Shared	See Section 18.6.3.2, "Performance Counters."
311H	785	MSR_IQ_COUNTER5	0, 1, 2, 3, 4, 6	Shared	See Section 18.6.3.2, "Performance Counters."
360H	864	MSR_BPU_CCCR0	0, 1, 2, 3, 4, 6	Shared	See Section 18.6.3.3, "CCCR MSRs."
361H	865	MSR_BPU_CCCR1	0, 1, 2, 3, 4, 6	Shared	See Section 18.6.3.3, "CCCR MSRs."
362H	866	MSR_BPU_CCCR2	0, 1, 2, 3, 4, 6	Shared	See Section 18.6.3.3, "CCCR MSRs."
363H	867	MSR_BPU_CCCR3	0, 1, 2, 3, 4, 6	Shared	See Section 18.6.3.3, "CCCR MSRs."
364H	868	MSR_MS_CCCR0	0, 1, 2, 3, 4, 6	Shared	See Section 18.6.3.3, "CCCR MSRs."
365H	869	MSR_MS_CCCR1	0, 1, 2, 3, 4, 6	Shared	See Section 18.6.3.3, "CCCR MSRs."
366H	870	MSR_MS_CCCR2	0, 1, 2, 3, 4, 6	Shared	See Section 18.6.3.3, "CCCR MSRs."
367H	871	MSR_MS_CCCR3	0, 1, 2, 3, 4, 6	Shared	See Section 18.6.3.3, "CCCR MSRs."
368H	872	MSR_FLAME_CCCR0	0, 1, 2, 3, 4, 6	Shared	See Section 18.6.3.3, "CCCR MSRs."
369H	873	MSR_FLAME_CCCR1	0, 1, 2, 3, 4, 6	Shared	See Section 18.6.3.3, "CCCR MSRs."
36AH	874	MSR_FLAME_CCCR2	0, 1, 2, 3, 4, 6	Shared	See Section 18.6.3.3, "CCCR MSRs."
36BH	875	MSR_FLAME_CCCR3	0, 1, 2, 3, 4, 6	Shared	See Section 18.6.3.3, "CCCR MSRs."
36CH	876	MSR_IQ_CCCR0	0, 1, 2, 3, 4, 6	Shared	See Section 18.6.3.3, "CCCR MSRs."
36DH	877	MSR_IQ_CCCR1	0, 1, 2, 3, 4, 6	Shared	See Section 18.6.3.3, "CCCR MSRs."
36EH	878	MSR_IQ_CCCR2	0, 1, 2, 3, 4, 6	Shared	See Section 18.6.3.3, "CCCR MSRs."

Table 2-44. MSRs in the Pentium® 4 and Intel® Xeon® Processors (Contd.)

Register Address		Register Name Fields and Flags	Model Avail-ability	Shared/ Unique[1]	Bit Description
Hex	Dec				
36FH	879	MSR_IQ_CCCR3	0, 1, 2, 3, 4, 6	Shared	See Section 18.6.3.3, "CCCR MSRs."
370H	880	MSR_IQ_CCCR4	0, 1, 2, 3, 4, 6	Shared	See Section 18.6.3.3, "CCCR MSRs."
371H	881	MSR_IQ_CCCR5	0, 1, 2, 3, 4, 6	Shared	See Section 18.6.3.3, "CCCR MSRs."
3A0H	928	MSR_BSU_ESCR0	0, 1, 2, 3, 4, 6	Shared	See Section 18.6.3.1, "ESCR MSRs."
3A1H	929	MSR_BSU_ESCR1	0, 1, 2, 3, 4, 6	Shared	See Section 18.6.3.1, "ESCR MSRs."
3A2H	930	MSR_FSB_ESCR0	0, 1, 2, 3, 4, 6	Shared	See Section 18.6.3.1, "ESCR MSRs."
3A3H	931	MSR_FSB_ESCR1	0, 1, 2, 3, 4, 6	Shared	See Section 18.6.3.1, "ESCR MSRs."
3A4H	932	MSR_FIRM_ESCR0	0, 1, 2, 3, 4, 6	Shared	See Section 18.6.3.1, "ESCR MSRs."
3A5H	933	MSR_FIRM_ESCR1	0, 1, 2, 3, 4, 6	Shared	See Section 18.6.3.1, "ESCR MSRs."
3A6H	934	MSR_FLAME_ESCR0	0, 1, 2, 3, 4, 6	Shared	See Section 18.6.3.1, "ESCR MSRs."
3A7H	935	MSR_FLAME_ESCR1	0, 1, 2, 3, 4, 6	Shared	See Section 18.6.3.1, "ESCR MSRs."
3A8H	936	MSR_DAC_ESCR0	0, 1, 2, 3, 4, 6	Shared	See Section 18.6.3.1, "ESCR MSRs."
3A9H	937	MSR_DAC_ESCR1	0, 1, 2, 3, 4, 6	Shared	See Section 18.6.3.1, "ESCR MSRs."
3AAH	938	MSR_MOB_ESCR0	0, 1, 2, 3, 4, 6	Shared	See Section 18.6.3.1, "ESCR MSRs."
3ABH	939	MSR_MOB_ESCR1	0, 1, 2, 3, 4, 6	Shared	See Section 18.6.3.1, "ESCR MSRs."
3ACH	940	MSR_PMH_ESCR0	0, 1, 2, 3, 4, 6	Shared	See Section 18.6.3.1, "ESCR MSRs."
3ADH	941	MSR_PMH_ESCR1	0, 1, 2, 3, 4, 6	Shared	See Section 18.6.3.1, "ESCR MSRs."
3AEH	942	MSR_SAAT_ESCR0	0, 1, 2, 3, 4, 6	Shared	See Section 18.6.3.1, "ESCR MSRs."
3AFH	943	MSR_SAAT_ESCR1	0, 1, 2, 3, 4, 6	Shared	See Section 18.6.3.1, "ESCR MSRs."
3B0H	944	MSR_U2L_ESCR0	0, 1, 2, 3, 4, 6	Shared	See Section 18.6.3.1, "ESCR MSRs."
3B1H	945	MSR_U2L_ESCR1	0, 1, 2, 3, 4, 6	Shared	See Section 18.6.3.1, "ESCR MSRs."
3B2H	946	MSR_BPU_ESCR0	0, 1, 2, 3, 4, 6	Shared	See Section 18.6.3.1, "ESCR MSRs."

Table 2-44. MSRs in the Pentium® 4 and Intel® Xeon® Processors (Contd.)

Register Address		Register Name Fields and Flags	Model Avail-ability	Shared/ Unique[1]	Bit Description
Hex	Dec				
3B3H	947	MSR_BPU_ESCR1	0, 1, 2, 3, 4, 6	Shared	See Section 18.6.3.1, "ESCR MSRs."
3B4H	948	MSR_IS_ESCR0	0, 1, 2, 3, 4, 6	Shared	See Section 18.6.3.1, "ESCR MSRs."
3B5H	949	MSR_IS_ESCR1	0, 1, 2, 3, 4, 6	Shared	See Section 18.6.3.1, "ESCR MSRs."
3B6H	950	MSR_ITLB_ESCR0	0, 1, 2, 3, 4, 6	Shared	See Section 18.6.3.1, "ESCR MSRs."
3B7H	951	MSR_ITLB_ESCR1	0, 1, 2, 3, 4, 6	Shared	See Section 18.6.3.1, "ESCR MSRs."
3B8H	952	MSR_CRU_ESCR0	0, 1, 2, 3, 4, 6	Shared	See Section 18.6.3.1, "ESCR MSRs."
3B9H	953	MSR_CRU_ESCR1	0, 1, 2, 3, 4, 6	Shared	See Section 18.6.3.1, "ESCR MSRs."
3BAH	954	MSR_IQ_ESCR0	0, 1, 2	Shared	See Section 18.6.3.1, "ESCR MSRs." This MSR is not available on later processors. It is only available on processor family 0FH, models 01H-02H.
3BBH	955	MSR_IQ_ESCR1	0, 1, 2	Shared	See Section 18.6.3.1, "ESCR MSRs." This MSR is not available on later processors. It is only available on processor family 0FH, models 01H-02H.
3BCH	956	MSR_RAT_ESCR0	0, 1, 2, 3, 4, 6	Shared	See Section 18.6.3.1, "ESCR MSRs."
3BDH	957	MSR_RAT_ESCR1	0, 1, 2, 3, 4, 6	Shared	See Section 18.6.3.1, "ESCR MSRs."
3BEH	958	MSR_SSU_ESCR0	0, 1, 2, 3, 4, 6	Shared	See Section 18.6.3.1, "ESCR MSRs."
3C0H	960	MSR_MS_ESCR0	0, 1, 2, 3, 4, 6	Shared	See Section 18.6.3.1, "ESCR MSRs."
3C1H	961	MSR_MS_ESCR1	0, 1, 2, 3, 4, 6	Shared	See Section 18.6.3.1, "ESCR MSRs."
3C2H	962	MSR_TBPU_ESCR0	0, 1, 2, 3, 4, 6	Shared	See Section 18.6.3.1, "ESCR MSRs."
3C3H	963	MSR_TBPU_ESCR1	0, 1, 2, 3, 4, 6	Shared	See Section 18.6.3.1, "ESCR MSRs."
3C4H	964	MSR_TC_ESCR0	0, 1, 2, 3, 4, 6	Shared	See Section 18.6.3.1, "ESCR MSRs."
3C5H	965	MSR_TC_ESCR1	0, 1, 2, 3, 4, 6	Shared	See Section 18.6.3.1, "ESCR MSRs."
3C8H	968	MSR_IX_ESCR0	0, 1, 2, 3, 4, 6	Shared	See Section 18.6.3.1, "ESCR MSRs."
3C9H	969	MSR_IX_ESCR1	0, 1, 2, 3, 4, 6	Shared	See Section 18.6.3.1, "ESCR MSRs."

Table 2-44. MSRs in the Pentium® 4 and Intel® Xeon® Processors (Contd.)

Register Address		Register Name Fields and Flags	Model Avail-ability	Shared/ Unique[1]	Bit Description
Hex	Dec				
3CAH	970	MSR_ALF_ESCR0	0, 1, 2, 3, 4, 6	Shared	See Section 18.6.3.1, "ESCR MSRs."
3CBH	971	MSR_ALF_ESCR1	0, 1, 2, 3, 4, 6	Shared	See Section 18.6.3.1, "ESCR MSRs."
3CCH	972	MSR_CRU_ESCR2	0, 1, 2, 3, 4, 6	Shared	See Section 18.6.3.1, "ESCR MSRs."
3CDH	973	MSR_CRU_ESCR3	0, 1, 2, 3, 4, 6	Shared	See Section 18.6.3.1, "ESCR MSRs."
3E0H	992	MSR_CRU_ESCR4	0, 1, 2, 3, 4, 6	Shared	See Section 18.6.3.1, "ESCR MSRs."
3E1H	993	MSR_CRU_ESCR5	0, 1, 2, 3, 4, 6	Shared	See Section 18.6.3.1, "ESCR MSRs."
3F0H	1008	MSR_TC_PRECISE_EVENT	0, 1, 2, 3, 4, 6	Shared	See Section 18.6.3.1, "ESCR MSRs."
3F1H	1009	MSR_PEBS_ENABLE	0, 1, 2, 3, 4, 6	Shared	**Processor Event Based Sampling (PEBS) (R/W)** Controls the enabling of processor event sampling and replay tagging.
		12:0			See Table 19-35.
		23:13			Reserved.
		24			**UOP Tag** Enables replay tagging when set.
		25			**ENABLE_PEBS_MY_THR (R/W)** Enables PEBS for the target logical processor when set; disables PEBS when clear (default). See Section 18.6.4.3, "IA32_PEBS_ENABLE MSR," for an explanation of the target logical processor. This bit is called ENABLE_PEBS in IA-32 processors that do not support Intel Hyper-Threading Technology.
		26			**ENABLE_PEBS_OTH_THR (R/W)** Enables PEBS for the target logical processor when set; disables PEBS when clear (default). See Section 18.6.4.3, "IA32_PEBS_ENABLE MSR," for an explanation of the target logical processor. This bit is reserved for IA-32 processors that do not support Intel Hyper-Threading Technology.
		63:27			Reserved.
3F2H	1010	MSR_PEBS_MATRIX_VERT	0, 1, 2, 3, 4, 6	Shared	See Table 19-35.
400H	1024	IA32_MC0_CTL	0, 1, 2, 3, 4, 6	Shared	See Section 15.3.2.1, "IA32_MCi_CTL MSRs."
401H	1025	IA32_MC0_STATUS	0, 1, 2, 3, 4, 6	Shared	See Section 15.3.2.2, "IA32_MCi_STATUS MSRS."

Table 2-44. MSRs in the Pentium® 4 and Intel® Xeon® Processors (Contd.)

Register Address		Register Name Fields and Flags	Model Avail-ability	Shared/ Unique[1]	Bit Description
Hex	Dec				
402H	1026	IA32_MC0_ADDR	0, 1, 2, 3, 4, 6	Shared	See Section 15.3.2.3, "IA32_MCi_ADDR MSRs." The IA32_MC0_ADDR register is either not implemented or contains no address if the ADDRV flag in the IA32_MC0_STATUS register is clear. When not implemented in the processor, all reads and writes to this MSR will cause a general-protection exception.
403H	1027	IA32_MC0_MISC	0, 1, 2, 3, 4, 6	Shared	See Section 15.3.2.4, "IA32_MCi_MISC MSRs." The IA32_MC0_MISC MSR is either not implemented or does not contain additional information if the MISCV flag in the IA32_MC0_STATUS register is clear. When not implemented in the processor, all reads and writes to this MSR will cause a general-protection exception.
404H	1028	IA32_MC1_CTL	0, 1, 2, 3, 4, 6	Shared	See Section 15.3.2.1, "IA32_MCi_CTL MSRs."
405H	1029	IA32_MC1_STATUS	0, 1, 2, 3, 4, 6	Shared	See Section 15.3.2.2, "IA32_MCi_STATUS MSRS."
406H	1030	IA32_MC1_ADDR	0, 1, 2, 3, 4, 6	Shared	See Section 15.3.2.3, "IA32_MCi_ADDR MSRs." The IA32_MC1_ADDR register is either not implemented or contains no address if the ADDRV flag in the IA32_MC1_STATUS register is clear. When not implemented in the processor, all reads and writes to this MSR will cause a general-protection exception.
407H	1031	IA32_MC1_MISC		Shared	See Section 15.3.2.4, "IA32_MCi_MISC MSRs." The IA32_MC1_MISC MSR is either not implemented or does not contain additional information if the MISCV flag in the IA32_MC1_STATUS register is clear. When not implemented in the processor, all reads and writes to this MSR will cause a general-protection exception.
408H	1032	IA32_MC2_CTL	0, 1, 2, 3, 4, 6	Shared	See Section 15.3.2.1, "IA32_MCi_CTL MSRs."
409H	1033	IA32_MC2_STATUS	0, 1, 2, 3, 4, 6	Shared	See Section 15.3.2.2, "IA32_MCi_STATUS MSRS."
40AH	1034	IA32_MC2_ADDR			See Section 15.3.2.3, "IA32_MCi_ADDR MSRs." The IA32_MC2_ADDR register is either not implemented or contains no address if the ADDRV flag in the IA32_MC2_STATUS register is clear. When not implemented in the processor, all reads and writes to this MSR will cause a general-protection exception.

Table 2-44. MSRs in the Pentium® 4 and Intel® Xeon® Processors (Contd.)

Register Address		Register Name Fields and Flags	Model Avail-ability	Shared/ Unique[1]	Bit Description
Hex	Dec				
40BH	1035	IA32_MC2_MISC			See Section 15.3.2.4, "IA32_MCi_MISC MSRs." The IA32_MC2_MISC MSR is either not implemented or does not contain additional information if the MISCV flag in the IA32_MC2_STATUS register is clear. When not implemented in the processor, all reads and writes to this MSR will cause a general-protection exception.
40CH	1036	IA32_MC3_CTL	0, 1, 2, 3, 4, 6	Shared	See Section 15.3.2.1, "IA32_MCi_CTL MSRs."
40DH	1037	IA32_MC3_STATUS	0, 1, 2, 3, 4, 6	Shared	See Section 15.3.2.2, "IA32_MCi_STATUS MSRS."
40EH	1038	IA32_MC3_ADDR	0, 1, 2, 3, 4, 6	Shared	See Section 15.3.2.3, "IA32_MCi_ADDR MSRs." The IA32_MC3_ADDR register is either not implemented or contains no address if the ADDRV flag in the IA32_MC3_STATUS register is clear. When not implemented in the processor, all reads and writes to this MSR will cause a general-protection exception.
40FH	1039	IA32_MC3_MISC	0, 1, 2, 3, 4, 6	Shared	See Section 15.3.2.4, "IA32_MCi_MISC MSRs." The IA32_MC3_MISC MSR is either not implemented or does not contain additional information if the MISCV flag in the IA32_MC3_STATUS register is clear. When not implemented in the processor, all reads and writes to this MSR will cause a general-protection exception.
410H	1040	IA32_MC4_CTL	0, 1, 2, 3, 4, 6	Shared	See Section 15.3.2.1, "IA32_MCi_CTL MSRs."
411H	1041	IA32_MC4_STATUS	0, 1, 2, 3, 4, 6	Shared	See Section 15.3.2.2, "IA32_MCi_STATUS MSRS."
412H	1042	IA32_MC4_ADDR			See Section 15.3.2.3, "IA32_MCi_ADDR MSRs." The IA32_MC2_ADDR register is either not implemented or contains no address if the ADDRV flag in the IA32_MC4_STATUS register is clear. When not implemented in the processor, all reads and writes to this MSR will cause a general-protection exception.
413H	1043	IA32_MC4_MISC			See Section 15.3.2.4, "IA32_MCi_MISC MSRs." The IA32_MC2_MISC MSR is either not implemented or does not contain additional information if the MISCV flag in the IA32_MC4_STATUS register is clear. When not implemented in the processor, all reads and writes to this MSR will cause a general-protection exception.

Table 2-44. MSRs in the Pentium® 4 and Intel® Xeon® Processors (Contd.)

Register Address		Register Name Fields and Flags	Model Avail-ability	Shared/ Unique[1]	Bit Description
Hex	Dec				
480H	1152	IA32_VMX_BASIC	3, 4, 6	Unique	**Reporting Register of Basic VMX Capabilities (R/O)** See Table 2-2. See Appendix A.1, "Basic VMX Information."
481H	1153	IA32_VMX_PINBASED_CTLS	3, 4, 6	Unique	**Capability Reporting Register of Pin-based VM-execution Controls (R/O)** See Table 2-2. See Appendix A.3, "VM-Execution Controls."
482H	1154	IA32_VMX_PROCBASED_CTLS	3, 4, 6	Unique	**Capability Reporting Register of Primary Processor-based VM-execution Controls (R/O)** See Appendix A.3, "VM-Execution Controls," and see Table 2-2.
483H	1155	IA32_VMX_EXIT_CTLS	3, 4, 6	Unique	**Capability Reporting Register of VM-exit Controls (R/O)** See Appendix A.4, "VM-Exit Controls," and see Table 2-2.
484H	1156	IA32_VMX_ENTRY_CTLS	3, 4, 6	Unique	**Capability Reporting Register of VM-entry Controls (R/O)** See Appendix A.5, "VM-Entry Controls," and see Table 2-2.
485H	1157	IA32_VMX_MISC	3, 4, 6	Unique	**Reporting Register of Miscellaneous VMX Capabilities (R/O)** See Appendix A.6, "Miscellaneous Data," and see Table 2-2.
486H	1158	IA32_VMX_CR0_FIXED0	3, 4, 6	Unique	**Capability Reporting Register of CR0 Bits Fixed to 0 (R/O)** See Appendix A.7, "VMX-Fixed Bits in CR0," and see Table 2-2.
487H	1159	IA32_VMX_CR0_FIXED1	3, 4, 6	Unique	**Capability Reporting Register of CR0 Bits Fixed to 1 (R/O)** See Appendix A.7, "VMX-Fixed Bits in CR0," and see Table 2-2.
488H	1160	IA32_VMX_CR4_FIXED0	3, 4, 6	Unique	**Capability Reporting Register of CR4 Bits Fixed to 0 (R/O)** See Appendix A.8, "VMX-Fixed Bits in CR4," and see Table 2-2.
489H	1161	IA32_VMX_CR4_FIXED1	3, 4, 6	Unique	**Capability Reporting Register of CR4 Bits Fixed to 1 (R/O)** See Appendix A.8, "VMX-Fixed Bits in CR4," and see Table 2-2.
48AH	1162	IA32_VMX_VMCS_ENUM	3, 4, 6	Unique	**Capability Reporting Register of VMCS Field Enumeration (R/O)** See Appendix A.9, "VMCS Enumeration," and see Table 2-2.

Table 2-44. MSRs in the Pentium® 4 and Intel® Xeon® Processors (Contd.)

Register Address		Register Name Fields and Flags	Model Avail-ability	Shared/ Unique[1]	Bit Description
Hex	Dec				
48BH	1163	IA32_VMX_PROCBASED_CTLS2	3, 4, 6	Unique	**Capability Reporting Register of Secondary Processor-based VM-execution Controls (R/O)** See Appendix A.3, "VM-Execution Controls," and see Table 2-2.
600H	1536	IA32_DS_AREA	0, 1, 2, 3, 4, 6	Unique	**DS Save Area (R/W)** See Table 2-2. See Section 18.6.3.4, "Debug Store (DS) Mechanism."
680H	1664	MSR_LASTBRANCH_0_FROM_IP	3, 4, 6	Unique	**Last Branch Record 0 (R/W)** One of 16 pairs of last branch record registers on the last branch record stack (680H-68FH). This part of the stack contains pointers to the **source instruction** for one of the last 16 branches, exceptions, or interrupts taken by the processor.
					The MSRs at 680H-68FH, 6C0H-6CfH are not available in processor releases before family 0FH, model 03H. These MSRs replace MSRs previously located at 1DBH-1DEH.which performed the same function for early releases. See Section 17.12, "Last Branch, Call Stack, Interrupt, and Exception Recording for Processors based on Skylake Microarchitecture."
681H	1665	MSR_LASTBRANCH_1_FROM_IP	3, 4, 6	Unique	**Last Branch Record 1** See description of MSR_LASTBRANCH_0 at 680H.
682H	1666	MSR_LASTBRANCH_2_FROM_IP	3, 4, 6	Unique	**Last Branch Record 2** See description of MSR_LASTBRANCH_0 at 680H.
683H	1667	MSR_LASTBRANCH_3_FROM_IP	3, 4, 6	Unique	**Last Branch Record 3** See description of MSR_LASTBRANCH_0 at 680H.
684H	1668	MSR_LASTBRANCH_4_FROM_IP	3, 4, 6	Unique	**Last Branch Record 4** See description of MSR_LASTBRANCH_0 at 680H.
685H	1669	MSR_LASTBRANCH_5_FROM_IP	3, 4, 6	Unique	**Last Branch Record 5** See description of MSR_LASTBRANCH_0 at 680H.
686H	1670	MSR_LASTBRANCH_6_FROM_IP	3, 4, 6	Unique	**Last Branch Record 6** See description of MSR_LASTBRANCH_0 at 680H.
687H	1671	MSR_LASTBRANCH_7_FROM_IP	3, 4, 6	Unique	**Last Branch Record 7** See description of MSR_LASTBRANCH_0 at 680H.
688H	1672	MSR_LASTBRANCH_8_FROM_IP	3, 4, 6	Unique	**Last Branch Record 8** See description of MSR_LASTBRANCH_0 at 680H.
689H	1673	MSR_LASTBRANCH_9_FROM_IP	3, 4, 6	Unique	**Last Branch Record 9** See description of MSR_LASTBRANCH_0 at 680H.
68AH	1674	MSR_LASTBRANCH_10_FROM_IP	3, 4, 6	Unique	**Last Branch Record 10** See description of MSR_LASTBRANCH_0 at 680H.

Table 2-44. MSRs in the Pentium® 4 and Intel® Xeon® Processors (Contd.)

Register Address		Register Name Fields and Flags	Model Avail-ability	Shared/ Unique[1]	Bit Description
Hex	Dec				
68BH	1675	MSR_LASTBRANCH_11_FROM_IP	3, 4, 6	Unique	**Last Branch Record 11** See description of MSR_LASTBRANCH_0 at 680H.
68CH	1676	MSR_LASTBRANCH_12_FROM_IP	3, 4, 6	Unique	**Last Branch Record 12** See description of MSR_LASTBRANCH_0 at 680H.
68DH	1677	MSR_LASTBRANCH_13_FROM_IP	3, 4, 6	Unique	**Last Branch Record 13** See description of MSR_LASTBRANCH_0 at 680H.
68EH	1678	MSR_LASTBRANCH_14_FROM_IP	3, 4, 6	Unique	**Last Branch Record 14** See description of MSR_LASTBRANCH_0 at 680H.
68FH	1679	MSR_LASTBRANCH_15_FROM_IP	3, 4, 6	Unique	**Last Branch Record 15** See description of MSR_LASTBRANCH_0 at 680H.
6C0H	1728	MSR_LASTBRANCH_0_TO_IP	3, 4, 6	Unique	**Last Branch Record 0 (R/W)** One of 16 pairs of last branch record registers on the last branch record stack (6C0H-6CFH). This part of the stack contains pointers to the destination instruction for one of the last 16 branches, exceptions, or interrupts that the processor took. See Section 17.12, "Last Branch, Call Stack, Interrupt, and Exception Recording for Processors based on Skylake Microarchitecture."
6C1H	1729	MSR_LASTBRANCH_1_TO_IP	3, 4, 6	Unique	**Last Branch Record 1** See description of MSR_LASTBRANCH_0 at 6C0H.
6C2H	1730	MSR_LASTBRANCH_2_TO_IP	3, 4, 6	Unique	**Last Branch Record 2** See description of MSR_LASTBRANCH_0 at 6C0H.
6C3H	1731	MSR_LASTBRANCH_3_TO_IP	3, 4, 6	Unique	**Last Branch Record 3** See description of MSR_LASTBRANCH_0 at 6C0H.
6C4H	1732	MSR_LASTBRANCH_4_TO_IP	3, 4, 6	Unique	**Last Branch Record 4** See description of MSR_LASTBRANCH_0 at 6C0H.
6C5H	1733	MSR_LASTBRANCH_5_TO_IP	3, 4, 6	Unique	**Last Branch Record 5** See description of MSR_LASTBRANCH_0 at 6C0H.
6C6H	1734	MSR_LASTBRANCH_6_TO_IP	3, 4, 6	Unique	**Last Branch Record 6** See description of MSR_LASTBRANCH_0 at 6C0H.
6C7H	1735	MSR_LASTBRANCH_7_TO_IP	3, 4, 6	Unique	**Last Branch Record 7** See description of MSR_LASTBRANCH_0 at 6C0H.
6C8H	1736	MSR_LASTBRANCH_8_TO_IP	3, 4, 6	Unique	**Last Branch Record 8** See description of MSR_LASTBRANCH_0 at 6C0H.
6C9H	1737	MSR_LASTBRANCH_9_TO_IP	3, 4, 6	Unique	**Last Branch Record 9** See description of MSR_LASTBRANCH_0 at 6C0H.
6CAH	1738	MSR_LASTBRANCH_10_TO_IP	3, 4, 6	Unique	**Last Branch Record 10** See description of MSR_LASTBRANCH_0 at 6C0H.

Table 2-44. MSRs in the Pentium® 4 and Intel® Xeon® Processors (Contd.)

Register Address		Register Name Fields and Flags	Model Avail-ability	Shared/ Unique[1]	Bit Description
Hex	Dec				
6CBH	1739	MSR_LASTBRANCH_11_TO_IP	3, 4, 6	Unique	**Last Branch Record 11** See description of MSR_LASTBRANCH_0 at 6C0H.
6CCH	1740	MSR_LASTBRANCH_12_TO_IP	3, 4, 6	Unique	**Last Branch Record 12** See description of MSR_LASTBRANCH_0 at 6C0H.
6CDH	1741	MSR_LASTBRANCH_13_TO_IP	3, 4, 6	Unique	**Last Branch Record 13** See description of MSR_LASTBRANCH_0 at 6C0H.
6CEH	1742	MSR_LASTBRANCH_14_TO_IP	3, 4, 6	Unique	**Last Branch Record 14** See description of MSR_LASTBRANCH_0 at 6C0H.
6CFH	1743	MSR_LASTBRANCH_15_TO_IP	3, 4, 6	Unique	**Last Branch Record 15** See description of MSR_LASTBRANCH_0 at 6C0H.
C000_0080H		IA32_EFER	3, 4, 6	Unique	**Extended Feature Enables** See Table 2-2.
C000_0081H		IA32_STAR	3, 4, 6	Unique	**System Call Target Address (R/W)** See Table 2-2.
C000_0082H		IA32_LSTAR	3, 4, 6	Unique	**IA-32e Mode System Call Target Address (R/W)** See Table 2-2.
C000_0084H		IA32_FMASK	3, 4, 6	Unique	**System Call Flag Mask (R/W)** See Table 2-2.
C000_0100H		IA32_FS_BASE	3, 4, 6	Unique	**Map of BASE Address of FS (R/W)** See Table 2-2.
C000_0101H		IA32_GS_BASE	3, 4, 6	Unique	**Map of BASE Address of GS (R/W)** See Table 2-2.
C000_0102H		IA32_KERNEL_GS_BASE	3, 4, 6	Unique	**Swap Target of BASE Address of GS (R/W)** See Table 2-2.

NOTES

1. For HT-enabled processors, there may be more than one logical processors per physical unit. If an MSR is Shared, this means that one MSR is shared between logical processors. If an MSR is unique, this means that each logical processor has its own MSR.

2.18.1 MSRs Unique to Intel® Xeon® Processor MP with L3 Cache

The MSRs listed in Table 2-45 apply to Intel® Xeon® Processor MP with up to 8MB level three cache. These processors can be detected by enumerating the deterministic cache parameter leaf of CPUID instruction (with EAX = 4 as input) to detect the presence of the third level cache, and with CPUID reporting family encoding 0FH, model encoding 3 or 4 (see CPUID instruction for more details).

Table 2-45. MSRs Unique to 64-bit Intel® Xeon® Processor MP with Up to an 8 MB L3 Cache

Register Address		Register Name Fields and Flags	Model Availability	Shared/ Unique	Bit Description
107CCH		MSR_IFSB_BUSQ0	3, 4	Shared	**IFSB BUSQ Event Control and Counter Register (R/W)** See Section 18.6.6, "Performance Monitoring on 64-bit Intel Xeon Processor MP with Up to 8-MByte L3 Cache."
107CDH		MSR_IFSB_BUSQ1	3, 4	Shared	**IFSB BUSQ Event Control and Counter Register (R/W)**
107CEH		MSR_IFSB_SNPQ0	3, 4	Shared	**IFSB SNPQ Event Control and Counter Register (R/W)** See Section 18.6.6, "Performance Monitoring on 64-bit Intel Xeon Processor MP with Up to 8-MByte L3 Cache."
107CFH		MSR_IFSB_SNPQ1	3, 4	Shared	**IFSB SNPQ Event Control and Counter Register (R/W)**
107D0H		MSR_EFSB_DRDY0	3, 4	Shared	**EFSB DRDY Event Control and Counter Register (R/W)** See Section 18.6.6, "Performance Monitoring on 64-bit Intel Xeon Processor MP with Up to 8-MByte L3 Cache."
107D1H		MSR_EFSB_DRDY1	3, 4	Shared	**EFSB DRDY Event Control and Counter Register (R/W)**
107D2H		MSR_IFSB_CTL6	3, 4	Shared	**IFSB Latency Event Control Register (R/W)** See Section 18.6.6, "Performance Monitoring on 64-bit Intel Xeon Processor MP with Up to 8-MByte L3 Cache."
107D3H		MSR_IFSB_CNTR7	3, 4	Shared	**IFSB Latency Event Counter Register (R/W)** See Section 18.6.6, "Performance Monitoring on 64-bit Intel Xeon Processor MP with Up to 8-MByte L3 Cache."

The MSRs listed in Table 2-46 apply to Intel® Xeon® Processor 7100 series. These processors can be detected by enumerating the deterministic cache parameter leaf of CPUID instruction (with EAX = 4 as input) to detect the presence of the third level cache, and with CPUID reporting family encoding 0FH, model encoding 6 (See CPUID instruction for more details.). The performance monitoring MSRs listed in Table 2-46 are shared between logical processors in the same core, but are replicated for each core.

Table 2-46. MSRs Unique to Intel® Xeon® Processor 7100 Series

Register Address		Register Name Fields and Flags	Model Availability	Shared/ Unique	Bit Description
107CCH		MSR_EMON_L3_CTR_CTL0	6	Shared	**GBUSQ Event Control and Counter Register (R/W)** See Section 18.6.6, "Performance Monitoring on 64-bit Intel Xeon Processor MP with Up to 8-MByte L3 Cache."
107CDH		MSR_EMON_L3_CTR_CTL1	6	Shared	**GBUSQ Event Control and Counter Register (R/W)**
107CEH		MSR_EMON_L3_CTR_CTL2	6	Shared	**GSNPQ Event Control and Counter Register (R/W)** See Section 18.6.6, "Performance Monitoring on 64-bit Intel Xeon Processor MP with Up to 8-MByte L3 Cache."
107CFH		MSR_EMON_L3_CTR_CTL3	6	Shared	**GSNPQ Event Control and Counter Register (R/W)**
107D0H		MSR_EMON_L3_CTR_CTL4	6	Shared	**FSB Event Control and Counter Register (R/W)** See Section 18.6.6, "Performance Monitoring on 64-bit Intel Xeon Processor MP with Up to 8-MByte L3 Cache."
107D1H		MSR_EMON_L3_CTR_CTL5	6	Shared	**FSB Event Control and Counter Register (R/W)**
107D2H		MSR_EMON_L3_CTR_CTL6	6	Shared	**FSB Event Control and Counter Register (R/W)**
107D3H		MSR_EMON_L3_CTR_CTL7	6	Shared	**FSB Event Control and Counter Register (R/W)**

2.19 MSRS IN INTEL® CORE™ SOLO AND INTEL® CORE™ DUO PROCESSORS

Model-specific registers (MSRs) for Intel Core Solo, Intel Core Duo processors, and Dual-core Intel Xeon processor LV are listed in Table 2-47. The column "Shared/Unique" applies to Intel Core Duo processor. "Unique" means each processor core has a separate MSR, or a bit field in an MSR governs only a core independently. "Shared" means the MSR or the bit field in an MSR address governs the operation of both processor cores.

Table 2-47. MSRs in Intel® Core™ Solo, Intel® Core™ Duo Processors, and Dual-Core Intel® Xeon® Processor LV

Register Address		Register Name	Shared/ Unique	Bit Description
Hex	Dec			
0H	0	P5_MC_ADDR	Unique	See Section 2.22, "MSRs in Pentium Processors," and see Table 2-2.
1H	1	P5_MC_TYPE	Unique	See Section 2.22, "MSRs in Pentium Processors," and see Table 2-2.
6H	6	IA32_MONITOR_FILTER_SIZE	Unique	See Section 8.10.5, "Monitor/Mwait Address Range Determination," and see Table 2-2.
10H	16	IA32_TIME_STAMP_COUNTER	Unique	See Section 17.17, "Time-Stamp Counter," and see Table 2-2.

Table 2-47. MSRs in Intel® Core™ Solo, Intel® Core™ Duo Processors, and Dual-Core Intel® Xeon® Processor LV (Contd.)

Register Address		Register Name	Shared/ Unique	Bit Description
Hex	Dec			
17H	23	IA32_PLATFORM_ID	Shared	**Platform ID (R)** See Table 2-2. The operating system can use this MSR to determine "slot" information for the processor and the proper microcode update to load.
1BH	27	IA32_APIC_BASE	Unique	See Section 10.4.4, "Local APIC Status and Location," and see Table 2-2.
2AH	42	MSR_EBL_CR_POWERON	Shared	**Processor Hard Power-On Configuration (R/W)** Enables and disables processor features; (R) indicates current processor configuration.
		0		Reserved.
		1		**Data Error Checking Enable (R/W)** 1 = Enabled; 0 = Disabled Note: Not all processor implements R/W.
		2		**Response Error Checking Enable (R/W)** 1 = Enabled; 0 = Disabled Note: Not all processor implements R/W.
		3		**MCERR# Drive Enable (R/W)** 1 = Enabled; 0 = Disabled Note: Not all processor implements R/W.
		4		**Address Parity Enable (R/W)** 1 = Enabled; 0 = Disabled Note: Not all processor implements R/W.
		6: 5		Reserved
		7		**BINIT# Driver Enable (R/W)** 1 = Enabled; 0 = Disabled Note: Not all processor implements R/W.
		8		**Output Tri-state Enabled (R/O)** 1 = Enabled; 0 = Disabled
		9		**Execute BIST (R/O)** 1 = Enabled; 0 = Disabled
		10		**MCERR# Observation Enabled (R/O)** 1 = Enabled; 0 = Disabled
		11		Reserved
		12		**BINIT# Observation Enabled (R/O)** 1 = Enabled; 0 = Disabled
		13		Reserved
		14		**1 MByte Power on Reset Vector (R/O)** 1 = 1 MByte; 0 = 4 GBytes
		15		Reserved
		17:16		**APIC Cluster ID (R/O)**

Table 2-47. MSRs in Intel® Core™ Solo, Intel® Core™ Duo Processors, and Dual-Core Intel® Xeon® Processor LV (Contd.)

Register Address		Register Name	Shared/ Unique	Bit Description
Hex	Dec			
		18		**System Bus Frequency (R/O)** 0 = 100 MHz 1 = Reserved
		19		Reserved.
		21: 20		**Symmetric Arbitration ID (R/O)**
		26:22		**Clock Frequency Ratio (R/O)**
3AH	58	IA32_FEATURE_CONTROL	Unique	**Control Features in IA-32 Processor (R/W)** See Table 2-2.
40H	64	MSR_LASTBRANCH_0	Unique	**Last Branch Record 0 (R/W)** One of 8 last branch record registers on the last branch record stack: bits 31-0 hold the 'from' address and bits 63-32 hold the 'to' address. See also: ▪ Last Branch Record Stack TOS at 1C9H ▪ Section 17.15, "Last Branch, Interrupt, and Exception Recording (Pentium M Processors)."
41H	65	MSR_LASTBRANCH_1	Unique	**Last Branch Record 1 (R/W)** See description of MSR_LASTBRANCH_0.
42H	66	MSR_LASTBRANCH_2	Unique	**Last Branch Record 2 (R/W)** See description of MSR_LASTBRANCH_0.
43H	67	MSR_LASTBRANCH_3	Unique	**Last Branch Record 3 (R/W)** See description of MSR_LASTBRANCH_0.
44H	68	MSR_LASTBRANCH_4	Unique	**Last Branch Record 4 (R/W)** See description of MSR_LASTBRANCH_0.
45H	69	MSR_LASTBRANCH_5	Unique	**Last Branch Record 5 (R/W)** See description of MSR_LASTBRANCH_0.
46H	70	MSR_LASTBRANCH_6	Unique	**Last Branch Record 6 (R/W)** See description of MSR_LASTBRANCH_0.
47H	71	MSR_LASTBRANCH_7	Unique	**Last Branch Record 7 (R/W)** See description of MSR_LASTBRANCH_0.
79H	121	IA32_BIOS_UPDT_TRIG	Unique	**BIOS Update Trigger Register (W)** See Table 2-2.
8BH	139	IA32_BIOS_SIGN_ID	Unique	**BIOS Update Signature ID (RO)** See Table 2-2.
C1H	193	IA32_PMC0	Unique	**Performance counter register** See Table 2-2.
C2H	194	IA32_PMC1	Unique	**Performance counter register** See Table 2-2.
CDH	205	MSR_FSB_FREQ	Shared	**Scaleable Bus Speed (RO)** This field indicates the scaleable bus clock speed:

Table 2-47. MSRs in Intel® Core™ Solo, Intel® Core™ Duo Processors, and Dual-Core Intel® Xeon® Processor LV (Contd.)

Register Address		Register Name	Shared/ Unique	Bit Description
Hex	Dec			
		2:0		▪ 101B: 100 MHz (FSB 400) ▪ 001B: 133 MHz (FSB 533) ▪ 011B: 167 MHz (FSB 667) 133.33 MHz should be utilized if performing calculation with System Bus Speed when encoding is 101B. 166.67 MHz should be utilized if performing calculation with System Bus Speed when encoding is 001B.
		63:3		Reserved.
E7H	231	IA32_MPERF	Unique	**Maximum Performance Frequency Clock Count. (RW)** See Table 2-2.
E8H	232	IA32_APERF	Unique	**Actual Performance Frequency Clock Count. (RW)** See Table 2-2.
FEH	254	IA32_MTRRCAP	Unique	See Table 2-2.
11EH	281	MSR_BBL_CR_CTL3	Shared	Control register 3. Used to configure the L2 Cache.
		0		**L2 Hardware Enabled (RO)** 1 = If the L2 is hardware-enabled 0 = Indicates if the L2 is hardware-disabled
		7:1		Reserved.
		8		**L2 Enabled (R/W)** 1 = L2 cache has been initialized 0 = Disabled (default) Until this bit is set the processor will not respond to the WBINVD instruction or the assertion of the FLUSH# input.
		22:9		Reserved.
		23		**L2 Not Present (RO)** 0 = L2 Present 1 = L2 Not Present
		63:24		Reserved.
174H	372	IA32_SYSENTER_CS	Unique	See Table 2-2.
175H	373	IA32_SYSENTER_ESP	Unique	See Table 2-2.
176H	374	IA32_SYSENTER_EIP	Unique	See Table 2-2.
179H	377	IA32_MCG_CAP	Unique	See Table 2-2.
17AH	378	IA32_MCG_STATUS	Unique	Global Machine Check Status
		0		**RIPV** When set, this bit indicates that the instruction addressed by the instruction pointer pushed on the stack (when the machine check was generated) can be used to restart the program. If this bit is cleared, the program cannot be reliably restarted.

Table 2-47. MSRs in Intel® Core™ Solo, Intel® Core™ Duo Processors, and Dual-Core Intel® Xeon® Processor LV (Contd.)

Register Address		Register Name	Shared/ Unique	Bit Description
Hex	Dec			
		1		**EIPV** When set, this bit indicates that the instruction addressed by the instruction pointer pushed on the stack (when the machine check was generated) is directly associated with the error.
		2		**MCIP** When set, this bit indicates that a machine check has been generated. If a second machine check is detected while this bit is still set, the processor enters a shutdown state. Software should write this bit to 0 after processing a machine check exception.
		63:3		Reserved.
186H	390	IA32_PERFEVTSEL0	Unique	See Table 2-2.
187H	391	IA32_PERFEVTSEL1	Unique	See Table 2-2.
198H	408	IA32_PERF_STATUS	Shared	See Table 2-2.
199H	409	IA32_PERF_CTL	Unique	See Table 2-2.
19AH	410	IA32_CLOCK_ MODULATION	Unique	**Clock Modulation (R/W)** See Table 2-2.
19BH	411	IA32_THERM_ INTERRUPT	Unique	**Thermal Interrupt Control (R/W)** See Table 2-2. See Section 14.7.2, "Thermal Monitor."
19CH	412	IA32_THERM_STATUS	Unique	**Thermal Monitor Status (R/W)** See Table 2-2. See Section 14.7.2, "Thermal Monitor".
19DH	413	MSR_THERM2_CTL	Unique	Thermal Monitor 2 Control
		15:0		Reserved.
		16		**TM_SELECT (R/W)** Mode of automatic thermal monitor: 0 = Thermal Monitor 1 (thermally-initiated on-die modulation of the stop-clock duty cycle) 1 = Thermal Monitor 2 (thermally-initiated frequency transitions) If bit 3 of the IA32_MISC_ENABLE register is cleared, TM_SELECT has no effect. Neither TM1 nor TM2 will be enabled.
		63:16		Reserved.
1A0H	416	IA32_MISC_ENABLE		**Enable Miscellaneous Processor Features (R/W)** Allows a variety of processor functions to be enabled and disabled.
		2:0		Reserved.
		3	Unique	**Automatic Thermal Control Circuit Enable (R/W)** See Table 2-2.
		6:4		Reserved.
		7	Shared	**Performance Monitoring Available (R)** See Table 2-2.

Table 2-47. MSRs in Intel® Core™ Solo, Intel® Core™ Duo Processors, and Dual-Core Intel® Xeon® Processor LV (Contd.)

Register Address		Register Name	Shared/ Unique	Bit Description
Hex	Dec			
		9:8		Reserved.
		10	Shared	**FERR# Multiplexing Enable (R/W)** 1 = FERR# asserted by the processor to indicate a pending break event within the processor 0 = Indicates compatible FERR# signaling behavior This bit must be set to 1 to support XAPIC interrupt model usage.
		11	Shared	**Branch Trace Storage Unavailable (RO)** See Table 2-2.
		12		Reserved.
		13	Shared	**TM2 Enable (R/W)** When this bit is set (1) and the thermal sensor indicates that the die temperature is at the pre-determined threshold, the Thermal Monitor 2 mechanism is engaged. TM2 will reduce the bus to core ratio and voltage according to the value last written to MSR_THERM2_CTL bits 15:0. When this bit is clear (0, default), the processor does not change the VID signals or the bus to core ratio when the processor enters a thermal managed state. If the TM2 feature flag (ECX[8]) is not set to 1 after executing CPUID with EAX = 1, then this feature is not supported and BIOS must not alter the contents of this bit location. The processor is operating out of spec if both this bit and the TM1 bit are set to disabled states.
		15:14		Reserved.
		16	Shared	**Enhanced Intel SpeedStep Technology Enable (R/W)** 1 = Enhanced Intel SpeedStep Technology enabled
		18	Shared	**ENABLE MONITOR FSM (R/W)** See Table 2-2.
		19		**Reserved.**
		22	Shared	**Limit CPUID Maxval (R/W)** See Table 2-2. Setting this bit may cause behavior in software that depends on the availability of CPUID leaves greater than 2.
		33:23		Reserved.
		34	Shared	**XD Bit Disable (R/W)** See Table 2-2.
		63:35		Reserved.
1C9H	457	MSR_LASTBRANCH_TOS	Unique	**Last Branch Record Stack TOS (R/W)** Contains an index (bits 0-3) that points to the MSR containing the most recent branch record. See MSR_LASTBRANCH_0_FROM_IP (at 40H).

Table 2-47. MSRs in Intel® Core™ Solo, Intel® Core™ Duo Processors, and Dual-Core Intel® Xeon® Processor LV (Contd.)

Register Address		Register Name	Shared/ Unique	Bit Description
Hex	Dec			
1D9H	473	IA32_DEBUGCTL	Unique	**Debug Control (R/W)** Controls how several debug features are used. Bit definitions are discussed in Table 2-2.
1DDH	477	MSR_LER_FROM_LIP	Unique	**Last Exception Record From Linear IP (R)** Contains a pointer to the last branch instruction that the processor executed prior to the last exception that was generated or the last interrupt that was handled.
1DEH	478	MSR_LER_TO_LIP	Unique	**Last Exception Record To Linear IP (R)** This area contains a pointer to the target of the last branch instruction that the processor executed prior to the last exception that was generated or the last interrupt that was handled.
200H	512	MTRRphysBase0	Unique	Memory Type Range Registers
201H	513	MTRRphysMask0	Unique	Memory Type Range Registers
202H	514	MTRRphysBase1	Unique	Memory Type Range Registers
203H	515	MTRRphysMask1	Unique	Memory Type Range Registers
204H	516	MTRRphysBase2	Unique	Memory Type Range Registers
205H	517	MTRRphysMask2	Unique	Memory Type Range Registers
206H	518	MTRRphysBase3	Unique	Memory Type Range Registers
207H	519	MTRRphysMask3	Unique	Memory Type Range Registers
208H	520	MTRRphysBase4	Unique	Memory Type Range Registers
209H	521	MTRRphysMask4	Unique	Memory Type Range Registers
20AH	522	MTRRphysBase5	Unique	Memory Type Range Registers
20BH	523	MTRRphysMask5	Unique	Memory Type Range Registers
20CH	524	MTRRphysBase6	Unique	Memory Type Range Registers
20DH	525	MTRRphysMask6	Unique	Memory Type Range Registers
20EH	526	MTRRphysBase7	Unique	Memory Type Range Registers
20FH	527	MTRRphysMask7	Unique	Memory Type Range Registers
250H	592	MTRRfix64K_00000	Unique	Memory Type Range Registers
258H	600	MTRRfix16K_80000	Unique	Memory Type Range Registers
259H	601	MTRRfix16K_A0000	Unique	Memory Type Range Registers
268H	616	MTRRfix4K_C0000	Unique	Memory Type Range Registers
269H	617	MTRRfix4K_C8000	Unique	Memory Type Range Registers
26AH	618	MTRRfix4K_D0000	Unique	Memory Type Range Registers
26BH	619	MTRRfix4K_D8000	Unique	Memory Type Range Registers
26CH	620	MTRRfix4K_E0000	Unique	Memory Type Range Registers
26DH	621	MTRRfix4K_E8000	Unique	Memory Type Range Registers
26EH	622	MTRRfix4K_F0000	Unique	Memory Type Range Registers
26FH	623	MTRRfix4K_F8000	Unique	Memory Type Range Registers

Table 2-47. MSRs in Intel® Core™ Solo, Intel® Core™ Duo Processors, and Dual-Core Intel® Xeon® Processor LV (Contd.)

Register Address		Register Name	Shared/ Unique	Bit Description
Hex	Dec			
2FFH	767	IA32_MTRR_DEF_TYPE	Unique	**Default Memory Types (R/W)** See Table 2-2. See Section 11.11.2.1, "IA32_MTRR_DEF_TYPE MSR."
400H	1024	IA32_MC0_CTL	Unique	See Section 15.3.2.1, "IA32_MCi_CTL MSRs."
401H	1025	IA32_MC0_STATUS	Unique	See Section 15.3.2.2, "IA32_MCi_STATUS MSRS."
402H	1026	IA32_MC0_ADDR	Unique	See Section 15.3.2.3, "IA32_MCi_ADDR MSRs." The IA32_MC0_ADDR register is either not implemented or contains no address if the ADDRV flag in the IA32_MC0_STATUS register is clear. When not implemented in the processor, all reads and writes to this MSR will cause a general-protection exception.
404H	1028	IA32_MC1_CTL	Unique	See Section 15.3.2.1, "IA32_MCi_CTL MSRs."
405H	1029	IA32_MC1_STATUS	Unique	See Section 15.3.2.2, "IA32_MCi_STATUS MSRS."
406H	1030	IA32_MC1_ADDR	Unique	See Section 15.3.2.3, "IA32_MCi_ADDR MSRs." The IA32_MC1_ADDR register is either not implemented or contains no address if the ADDRV flag in the IA32_MC1_STATUS register is clear. When not implemented in the processor, all reads and writes to this MSR will cause a general-protection exception.
408H	1032	IA32_MC2_CTL	Unique	See Section 15.3.2.1, "IA32_MCi_CTL MSRs."
409H	1033	IA32_MC2_STATUS	Unique	See Section 15.3.2.2, "IA32_MCi_STATUS MSRS."
40AH	1034	IA32_MC2_ADDR	Unique	See Section 15.3.2.3, "IA32_MCi_ADDR MSRs." The IA32_MC2_ADDR register is either not implemented or contains no address if the ADDRV flag in the IA32_MC2_STATUS register is clear. When not implemented in the processor, all reads and writes to this MSR will cause a general-protection exception.
40CH	1036	MSR_MC4_CTL	Unique	See Section 15.3.2.1, "IA32_MCi_CTL MSRs."
40DH	1037	MSR_MC4_STATUS	Unique	See Section 15.3.2.2, "IA32_MCi_STATUS MSRS."
40EH	1038	MSR_MC4_ADDR	Unique	See Section 15.3.2.3, "IA32_MCi_ADDR MSRs." The MSR_MC4_ADDR register is either not implemented or contains no address if the ADDRV flag in the MSR_MC4_STATUS register is clear. When not implemented in the processor, all reads and writes to this MSR will cause a general-protection exception.
410H	1040	IA32_MC3_CTL		See Section 15.3.2.1, "IA32_MCi_CTL MSRs."
411H	1041	IA32_MC3_STATUS		See Section 15.3.2.2, "IA32_MCi_STATUS MSRS."
412H	1042	MSR_MC3_ADDR	Unique	See Section 15.3.2.3, "IA32_MCi_ADDR MSRs." The MSR_MC3_ADDR register is either not implemented or contains no address if the ADDRV flag in the MSR_MC3_STATUS register is clear. When not implemented in the processor, all reads and writes to this MSR will cause a general-protection exception.
413H	1043	MSR_MC3_MISC	Unique	Machine Check Error Reporting Register - contains additional information describing the machine-check error if the MISCV flag in the IA32_MCi_STATUS register is set.
414H	1044	MSR_MC5_CTL	Unique	Machine Check Error Reporting Register - controls signaling of #MC for errors produced by a particular hardware unit (or group of hardware units).

Table 2-47. MSRs in Intel® Core™ Solo, Intel® Core™ Duo Processors, and Dual-Core Intel® Xeon® Processor LV (Contd.)

Register Address		Register Name	Shared/ Unique	Bit Description
Hex	Dec			
415H	1045	MSR_MC5_STATUS	Unique	Machine Check Error Reporting Register - contains information related to a machine-check error if its VAL (valid) flag is set. Software is responsible for clearing IA32_MCi_STATUS MSRs by explicitly writing 0s to them; writing 1s to them causes a general-protection exception.
416H	1046	MSR_MC5_ADDR	Unique	Machine Check Error Reporting Register - contains the address of the code or data memory location that produced the machine-check error if the ADDRV flag in the IA32_MCi_STATUS register is set.
417H	1047	MSR_MC5_MISC	Unique	Machine Check Error Reporting Register - contains additional information describing the machine-check error if the MISCV flag in the IA32_MCi_STATUS register is set.
480H	1152	IA32_VMX_BASIC	Unique	**Reporting Register of Basic VMX Capabilities (R/O)** See Table 2-2. See Appendix A.1, "Basic VMX Information" (If CPUID.01H:ECX.[bit 9])
481H	1153	IA32_VMX_PINBASED_ CTLS	Unique	**Capability Reporting Register of Pin-based VM-execution Controls (R/O)** See Appendix A.3, "VM-Execution Controls" (If CPUID.01H:ECX.[bit 9])
482H	1154	IA32_VMX_PROCBASED_ CTLS	Unique	**Capability Reporting Register of Primary Processor-based VM-execution Controls (R/O)** See Appendix A.3, "VM-Execution Controls" (If CPUID.01H:ECX.[bit 9])
483H	1155	IA32_VMX_EXIT_CTLS	Unique	**Capability Reporting Register of VM-exit Controls (R/O)** See Appendix A.4, "VM-Exit Controls" (If CPUID.01H:ECX.[bit 9])
484H	1156	IA32_VMX_ENTRY_CTLS	Unique	**Capability Reporting Register of VM-entry Controls (R/O)** See Appendix A.5, "VM-Entry Controls" (If CPUID.01H:ECX.[bit 9])
485H	1157	IA32_VMX_MISC	Unique	**Reporting Register of Miscellaneous VMX Capabilities (R/O)** See Appendix A.6, "Miscellaneous Data" (If CPUID.01H:ECX.[bit 9])
486H	1158	IA32_VMX_CR0_FIXED0	Unique	**Capability Reporting Register of CR0 Bits Fixed to 0 (R/O)** See Appendix A.7, "VMX-Fixed Bits in CR0" (If CPUID.01H:ECX.[bit 9])
487H	1159	IA32_VMX_CR0_FIXED1	Unique	**Capability Reporting Register of CR0 Bits Fixed to 1 (R/O)** See Appendix A.7, "VMX-Fixed Bits in CR0" (If CPUID.01H:ECX.[bit 9])
488H	1160	IA32_VMX_CR4_FIXED0	Unique	**Capability Reporting Register of CR4 Bits Fixed to 0 (R/O)** See Appendix A.8, "VMX-Fixed Bits in CR4" (If CPUID.01H:ECX.[bit 9])

Table 2-47. MSRs in Intel® Core™ Solo, Intel® Core™ Duo Processors, and Dual-Core Intel® Xeon® Processor LV (Contd.)

Register Address		Register Name	Shared/ Unique	Bit Description
Hex	Dec			
489H	1161	IA32_VMX_CR4_FIXED1	Unique	**Capability Reporting Register of CR4 Bits Fixed to 1 (R/O)** See Appendix A.8, "VMX-Fixed Bits in CR4" (If CPUID.01H:ECX.[bit 9])
48AH	1162	IA32_VMX_VMCS_ENUM	Unique	**Capability Reporting Register of VMCS Field Enumeration (R/O)** See Appendix A.9, "VMCS Enumeration" (If CPUID.01H:ECX.[bit 9])
48BH	1163	IA32_VMX_PROCBASED_ CTLS2	Unique	**Capability Reporting Register of Secondary Processor-based VM-execution Controls (R/O)** See Appendix A.3, "VM-Execution Controls" (If CPUID.01H:ECX.[bit 9] and IA32_VMX_PROCBASED_CTLS[bit 63])
600H	1536	IA32_DS_AREA	Unique	**DS Save Area (R/W)** See Table 2-2. See Section 18.6.3.4, "Debug Store (DS) Mechanism."
		31:0		**DS Buffer Management Area** Linear address of the first byte of the DS buffer management area.
		63:32		Reserved.
C000_ 0080H		IA32_EFER	Unique	See Table 2-2.
		10:0		Reserved.
		11		**Execute Disable Bit Enable**
		63:12		Reserved.

2.20 MSRS IN THE PENTIUM M PROCESSOR

Model-specific registers (MSRs) for the Pentium M processor are similar to those described in Section 2.21 for P6 family processors. The following table describes new MSRs and MSRs whose behavior has changed on the Pentium M processor.

Table 2-48. MSRs in Pentium M Processors

Register Address		Register Name	Bit Description
Hex	Dec		
0H	0	P5_MC_ADDR	See Section 2.22, "MSRs in Pentium Processors."
1H	1	P5_MC_TYPE	See Section 2.22, "MSRs in Pentium Processors."
10H	16	IA32_TIME_STAMP_COUNTER	See Section 17.17, "Time-Stamp Counter," and see Table 2-2.
17H	23	IA32_PLATFORM_ID	**Platform ID (R)** See Table 2-2. The operating system can use this MSR to determine "slot" information for the processor and the proper microcode update to load.

Table 2-48. MSRs in Pentium M Processors (Contd.)

Register Address		Register Name	Bit Description
Hex	Dec		
2AH	42	MSR_EBL_CR_POWERON	**Processor Hard Power-On Configuration** **(R/W)** Enables and disables processor features. **(R)** Indicates current processor configuration.
		0	Reserved.
		1	**Data Error Checking Enable (R)** 0 = Disabled Always 0 on the Pentium M processor.
		2	**Response Error Checking Enable (R)** 0 = Disabled Always 0 on the Pentium M processor.
		3	**MCERR# Drive Enable (R)** 0 = Disabled Always 0 on the Pentium M processor.
		4	**Address Parity Enable (R)** 0 = Disabled Always 0 on the Pentium M processor.
		6:5	Reserved.
		7	**BINIT# Driver Enable (R)** 1 = Enabled; 0 = Disabled Always 0 on the Pentium M processor.
		8	**Output Tri-state Enabled (R/O)** 1 = Enabled; 0 = Disabled
		9	**Execute BIST (R/O)** 1 = Enabled; 0 = Disabled
		10	**MCERR# Observation Enabled (R/O)** 1 = Enabled; 0 = Disabled Always 0 on the Pentium M processor.
		11	Reserved.
		12	**BINIT# Observation Enabled (R/O)** 1 = Enabled; 0 = Disabled Always 0 on the Pentium M processor.
		13	Reserved.
		14	**1 MByte Power on Reset Vector (R/O)** 1 = 1 MByte; 0 = 4 GBytes Always 0 on the Pentium M processor.
		15	Reserved.
		17:16	**APIC Cluster ID (R/O)** Always 00B on the Pentium M processor.

Table 2-48. MSRs in Pentium M Processors (Contd.)

Register Address		Register Name	Bit Description
Hex	Dec		
		18	**System Bus Frequency (R/O)** 0 = 100 MHz 1 = Reserved Always 0 on the Pentium M processor.
		19	Reserved.
		21: 20	**Symmetric Arbitration ID (R/O)** Always 00B on the Pentium M processor.
		26:22	Clock Frequency Ratio (R/O)
40H	64	MSR_LASTBRANCH_0	**Last Branch Record 0 (R/W)** One of 8 last branch record registers on the last branch record stack: bits 31-0 hold the 'from' address and bits 63-32 hold the to address. See also: ▪ Last Branch Record Stack TOS at 1C9H ▪ Section 17.15, "Last Branch, Interrupt, and Exception Recording (Pentium M Processors)"
41H	65	MSR_LASTBRANCH_1	**Last Branch Record 1 (R/W)** See description of MSR_LASTBRANCH_0.
42H	66	MSR_LASTBRANCH_2	**Last Branch Record 2 (R/W)** See description of MSR_LASTBRANCH_0.
43H	67	MSR_LASTBRANCH_3	**Last Branch Record 3 (R/W)** See description of MSR_LASTBRANCH_0.
44H	68	MSR_LASTBRANCH_4	**Last Branch Record 4 (R/W)** See description of MSR_LASTBRANCH_0.
45H	69	MSR_LASTBRANCH_5	**Last Branch Record 5 (R/W)** See description of MSR_LASTBRANCH_0.
46H	70	MSR_LASTBRANCH_6	**Last Branch Record 6 (R/W)** See description of MSR_LASTBRANCH_0.
47H	71	MSR_LASTBRANCH_7	**Last Branch Record 7 (R/W)** See description of MSR_LASTBRANCH_0.
119H	281	MSR_BBL_CR_CTL	Control register Used to program L2 commands to be issued via cache configuration accesses mechanism. Also receives L2 lookup response.
		63:0	Reserved.
11EH	281	MSR_BBL_CR_CTL3	Control register 3 Used to configure the L2 Cache.
		0	**L2 Hardware Enabled (RO)** 1 = If the L2 is hardware-enabled 0 = Indicates if the L2 is hardware-disabled
		4:1	Reserved.

Table 2-48. MSRs in Pentium M Processors (Contd.)

Register Address		Register Name	Bit Description
Hex	Dec		
		5	**ECC Check Enable (RO)** This bit enables ECC checking on the cache data bus. ECC is always generated on write cycles. 0 = Disabled (default) 1 = Enabled For the Pentium M processor, ECC checking on the cache data bus is always enabled.
		7:6	Reserved.
		8	**L2 Enabled (R/W)** 1 = L2 cache has been initialized 0 = Disabled (default) Until this bit is set the processor will not respond to the WBINVD instruction or the assertion of the FLUSH# input.
		22:9	Reserved.
		23	**L2 Not Present (RO)** 0 = L2 Present 1 = L2 Not Present
		63:24	Reserved.
179H	377	IA32_MCG_CAP	Read-only register that provides information about the machine-check architecture of the processor.
		7:0	**Count (RO)** Indicates the number of hardware unit error reporting banks available in the processor.
		8	**IA32_MCG_CTL Present (RO)** 1 = Indicates that the processor implements the MSR_MCG_CTL register found at MSR 17BH. 0 = Not supported.
		63:9	Reserved.
17AH	378	IA32_MCG_STATUS	Global Machine Check Status
		0	**RIPV** When set, this bit indicates that the instruction addressed by the instruction pointer pushed on the stack (when the machine check was generated) can be used to restart the program. If this bit is cleared, the program cannot be reliably restarted.
		1	**EIPV** When set, this bit indicates that the instruction addressed by the instruction pointer pushed on the stack (when the machine check was generated) is directly associated with the error.
		2	**MCIP** When set, this bit indicates that a machine check has been generated. If a second machine check is detected while this bit is still set, the processor enters a shutdown state. Software should write this bit to 0 after processing a machine check exception.

Table 2-48. MSRs in Pentium M Processors (Contd.)

Register Address		Register Name	Bit Description
Hex	Dec		
		63:3	Reserved.
198H	408	IA32_PERF_STATUS	See Table 2-2.
199H	409	IA32_PERF_CTL	See Table 2-2.
19AH	410	IA32_CLOCK_MODULATION	**Clock Modulation (R/W).** See Table 2-2. See Section 14.7.3, "Software Controlled Clock Modulation."
19BH	411	IA32_THERM_INTERRUPT	**Thermal Interrupt Control (R/W)** See Table 2-2. See Section 14.7.2, "Thermal Monitor."
19CH	412	IA32_THERM_STATUS	**Thermal Monitor Status (R/W)** See Table 2-2. See Section 14.7.2, "Thermal Monitor."
19DH	413	MSR_THERM2_CTL	Thermal Monitor 2 Control
		15:0	Reserved.
		16	**TM_SELECT (R/W)** Mode of automatic thermal monitor: 0 = Thermal Monitor 1 (thermally-initiated on-die modulation of the stop-clock duty cycle) 1 = Thermal Monitor 2 (thermally-initiated frequency transitions) If bit 3 of the IA32_MISC_ENABLE register is cleared, TM_SELECT has no effect. Neither TM1 nor TM2 will be enabled.
		63:16	Reserved.
1A0H	416	IA32_MISC_ENABLE	**Enable Miscellaneous Processor Features (R/W)** Allows a variety of processor functions to be enabled and disabled.
		2:0	Reserved.
		3	**Automatic Thermal Control Circuit Enable (R/W)** 1 = Setting this bit enables the thermal control circuit (TCC) portion of the Intel Thermal Monitor feature. This allows processor clocks to be automatically modulated based on the processor's thermal sensor operation. 0 = Disabled (default). The automatic thermal control circuit enable bit determines if the thermal control circuit (TCC) will be activated when the processor's internal thermal sensor determines the processor is about to exceed its maximum operating temperature. When the TCC is activated and TM1 is enabled, the processors clocks will be forced to a 50% duty cycle. BIOS must enable this feature. The bit should not be confused with the on-demand thermal control circuit enable bit.
		6:4	Reserved.

Table 2-48. MSRs in Pentium M Processors (Contd.)

Register Address		Register Name	Bit Description
Hex	Dec		
		7	**Performance Monitoring Available (R)** 1 = Performance monitoring enabled 0 = Performance monitoring disabled
		9:8	Reserved.
		10	**FERR# Multiplexing Enable (R/W)** 1 = FERR# asserted by the processor to indicate a pending break event within the processor 0 = Indicates compatible FERR# signaling behavior This bit must be set to 1 to support XAPIC interrupt model usage.
			Branch Trace Storage Unavailable (RO) 1 = Processor doesn't support branch trace storage (BTS) 0 = BTS is supported
		12	**Processor Event Based Sampling Unavailable (RO)** 1 = Processor does not support processor event based sampling (PEBS); 0 = PEBS is supported. The Pentium M processor does not support PEBS.
		15:13	Reserved.
		16	**Enhanced Intel SpeedStep Technology Enable (R/W)** 1 = Enhanced Intel SpeedStep Technology enabled. On the Pentium M processor, this bit may be configured to be read-only.
		22:17	Reserved.
		23	**xTPR Message Disable (R/W)** When set to 1, xTPR messages are disabled. xTPR messages are optional messages that allow the processor to inform the chipset of its priority. The default is processor specific.
		63:24	Reserved.
1C9H	457	MSR_LASTBRANCH_TOS	**Last Branch Record Stack TOS (R/W)** Contains an index (bits 0-3) that points to the MSR containing the most recent branch record. See also: ▪ MSR_LASTBRANCH_0_FROM_IP (at 40H) ▪ Section 17.15, "Last Branch, Interrupt, and Exception Recording (Pentium M Processors)"
1D9H	473	MSR_DEBUGCTLB	**Debug Control (R/W)** Controls how several debug features are used. Bit definitions are discussed in the referenced section. See Section 17.15, "Last Branch, Interrupt, and Exception Recording (Pentium M Processors)."

Table 2-48. MSRs in Pentium M Processors (Contd.)

Register Address		Register Name	Bit Description
Hex	Dec		
1DDH	477	MSR_LER_TO_LIP	**Last Exception Record To Linear IP (R)** This area contains a pointer to the target of the last branch instruction that the processor executed prior to the last exception that was generated or the last interrupt that was handled. See Section 17.15, "Last Branch, Interrupt, and Exception Recording (Pentium M Processors)" and Section 17.16.2, "Last Branch and Last Exception MSRs."
1DEH	478	MSR_LER_FROM_LIP	**Last Exception Record From Linear IP (R)** Contains a pointer to the last branch instruction that the processor executed prior to the last exception that was generated or the last interrupt that was handled. See Section 17.15, "Last Branch, Interrupt, and Exception Recording (Pentium M Processors)" and Section 17.16.2, "Last Branch and Last Exception MSRs."
2FFH	767	IA32_MTRR_DEF_TYPE	**Default Memory Types (R/W)** Sets the memory type for the regions of physical memory that are not mapped by the MTRRs. See Section 11.11.2.1, "IA32_MTRR_DEF_TYPE MSR."
400H	1024	IA32_MC0_CTL	See Section 15.3.2.1, "IA32_MCi_CTL MSRs."
401H	1025	IA32_MC0_STATUS	See Section 15.3.2.2, "IA32_MCi_STATUS MSRS."
402H	1026	IA32_MC0_ADDR	See Section 14.3.2.3., "IA32_MCi_ADDR MSRs". The IA32_MC0_ADDR register is either not implemented or contains no address if the ADDRV flag in the IA32_MC0_STATUS register is clear. When not implemented in the processor, all reads and writes to this MSR will cause a general-protection exception.
404H	1028	IA32_MC1_CTL	See Section 15.3.2.1, "IA32_MCi_CTL MSRs."
405H	1029	IA32_MC1_STATUS	See Section 15.3.2.2, "IA32_MCi_STATUS MSRS."
406H	1030	IA32_MC1_ADDR	See Section 15.3.2.3, "IA32_MCi_ADDR MSRs." The IA32_MC1_ADDR register is either not implemented or contains no address if the ADDRV flag in the IA32_MC1_STATUS register is clear. When not implemented in the processor, all reads and writes to this MSR will cause a general-protection exception.
408H	1032	IA32_MC2_CTL	See Section 15.3.2.1, "IA32_MCi_CTL MSRs."
409H	1033	IA32_MC2_STATUS	See Chapter 15.3.2.2, "IA32_MCi_STATUS MSRS."
40AH	1034	IA32_MC2_ADDR	See Section 15.3.2.3, "IA32_MCi_ADDR MSRs." The IA32_MC2_ADDR register is either not implemented or contains no address if the ADDRV flag in the IA32_MC2_STATUS register is clear. When not implemented in the processor, all reads and writes to this MSR will cause a general-protection exception.
40CH	1036	MSR_MC4_CTL	See Section 15.3.2.1, "IA32_MCi_CTL MSRs."
40DH	1037	MSR_MC4_STATUS	See Section 15.3.2.2, "IA32_MCi_STATUS MSRS."

Table 2-48. MSRs in Pentium M Processors (Contd.)

Register Address		Register Name	Bit Description
Hex	Dec		
40EH	1038	MSR_MC4_ADDR	See Section 15.3.2.3, "IA32_MCi_ADDR MSRs." The MSR_MC4_ADDR register is either not implemented or contains no address if the ADDRV flag in the MSR_MC4_STATUS register is clear. When not implemented in the processor, all reads and writes to this MSR will cause a general-protection exception.
410H	1040	MSR_MC3_CTL	See Section 15.3.2.1, "IA32_MCi_CTL MSRs."
411H	1041	MSR_MC3_STATUS	See Section 15.3.2.2, "IA32_MCi_STATUS MSRS."
412H	1042	MSR_MC3_ADDR	See Section 15.3.2.3, "IA32_MCi_ADDR MSRs." The MSR_MC3_ADDR register is either not implemented or contains no address if the ADDRV flag in the MSR_MC3_STATUS register is clear. When not implemented in the processor, all reads and writes to this MSR will cause a general-protection exception.
600H	1536	IA32_DS_AREA	**DS Save Area (R/W)** See Table 2-2. Points to the DS buffer management area, which is used to manage the BTS and PEBS buffers. See Section 18.6.3.4, "Debug Store (DS) Mechanism."
		31:0	**DS Buffer Management Area** Linear address of the first byte of the DS buffer management area.
		63:32	Reserved.

2.21 MSRS IN THE P6 FAMILY PROCESSORS

The following MSRs are defined for the P6 family processors. The MSRs in this table that are shaded are available only in the Pentium II and Pentium III processors. Beginning with the Pentium 4 processor, some of the MSRs in this list have been designated as "architectural" and have had their names changed. See Table 2-2 for a list of the architectural MSRs.

Table 2-49. MSRs in the P6 Family Processors

Register Address		Register Name	Bit Description
Hex	Dec		
0H	0	P5_MC_ADDR	See Section 2.22, "MSRs in Pentium Processors."
1H	1	P5_MC_TYPE	See Section 2.22, "MSRs in Pentium Processors."
10H	16	TSC	See Section 17.17, "Time-Stamp Counter."
17H	23	IA32_PLATFORM_ID	**Platform ID (R)** The operating system can use this MSR to determine "slot" information for the processor and the proper microcode update to load.
		49:0	Reserved.

Table 2-49. MSRs in the P6 Family Processors (Contd.)

Register Address		Register Name	Bit Description
Hex	Dec		
		52:50	**Platform Id (R)** Contains information concerning the intended platform for the processor. 52 51 50 0 0 0 Processor Flag 0 0 0 1 Processor Flag 1 0 1 0 Processor Flag 2 0 1 1 Processor Flag 3 1 0 0 Processor Flag 4 1 0 1 Processor Flag 5 1 1 0 Processor Flag 6 1 1 1 Processor Flag 7
		56:53	L2 Cache Latency Read.
		59:57	Reserved.
		60	Clock Frequency Ratio Read.
		63:61	Reserved.
1BH	27	APIC_BASE	Section 10.4.4, "Local APIC Status and Location."
		7:0	Reserved.
		8	**Boot Strap Processor indicator Bit** 1 = BSP
		10:9	Reserved.
		11	**APIC Global Enable Bit - Permanent till reset** 1 = Enabled 0 = Disabled
		31:12	APIC Base Address.
		63:32	Reserved.
2AH	42	EBL_CR_POWERON	**Processor Hard Power-On Configuration** **(R/W)** Enables and disables processor features; **(R)** indicates current processor configuration.
		0	Reserved.[1]
		1	**Data Error Checking Enable (R/W)** 1 = Enabled 0 = Disabled
		2	**Response Error Checking Enable FRCERR Observation Enable (R/W)** 1 = Enabled 0 = Disabled
		3	**AERR# Drive Enable (R/W)** 1 = Enabled 0 = Disabled
		4	**BERR# Enable for Initiator Bus Requests (R/W)** 1 = Enabled 0 = Disabled
		5	Reserved.

Table 2-49. MSRs in the P6 Family Processors (Contd.)

Register Address		Register Name	Bit Description
Hex	Dec		
		6	**BERR# Driver Enable for Initiator Internal Errors (R/W)** 1 = Enabled 0 = Disabled
		7	**BINIT# Driver Enable (R/W)** 1 = Enabled 0 = Disabled
		8	**Output Tri-state Enabled (R)** 1 = Enabled 0 = Disabled
		9	**Execute BIST (R)** 1 = Enabled 0 = Disabled
		10	**AERR# Observation Enabled (R)** 1 = Enabled 0 = Disabled
		11	Reserved.
		12	**BINIT# Observation Enabled (R)** 1 = Enabled 0 = Disabled
		13	**In Order Queue Depth (R)** 1 = 1 0 = 8
		14	**1-MByte Power on Reset Vector (R)** 1 = 1MByte 0 = 4GBytes
		15	**FRC Mode Enable (R)** 1 = Enabled 0 = Disabled
		17:16	**APIC Cluster ID (R)**
		19:18	**System Bus Frequency (R)** 00 = 66MHz 10 = 100Mhz 01 = 133MHz 11 = Reserved
		21: 20	**Symmetric Arbitration ID (R)**
		25:22	**Clock Frequency Ratio (R)**
		26	**Low Power Mode Enable (R/W)**
		27	**Clock Frequency Ratio**
		63:28	Reserved.[1]
33H	51	TEST_CTL	**Test Control Register**

Table 2-49. MSRs in the P6 Family Processors (Contd.)

Register Address		Register Name	Bit Description
Hex	Dec		
		29:0	Reserved.
		30	**Streaming Buffer Disable**
		31	**Disable LOCK#** Assertion for split locked access.
79H	121	BIOS_UPDT_TRIG	BIOS Update Trigger Register.
88H	136	BBL_CR_D0[63:0]	Chunk 0 data register D[63:0]: used to write to and read from the L2
89H	137	BBL_CR_D1[63:0]	Chunk 1 data register D[63:0]: used to write to and read from the L2
8AH	138	BBL_CR_D2[63:0]	Chunk 2 data register D[63:0]: used to write to and read from the L2
8BH	139	BIOS_SIGN/BBL_CR_D3[63:0]	**BIOS Update Signature Register or Chunk 3 data register D[63:0]** Used to write to and read from the L2 depending on the usage model.
C1H	193	PerfCtr0 (PERFCTR0)	Performance Counter Register See Table 2-2.
C2H	194	PerfCtr1 (PERFCTR1)	Performance Counter Register See Table 2-2.
FEH	254	MTRRcap	Memory Type Range Registers
116H	278	BBL_CR_ADDR [63:0] BBL_CR_ADDR [63:32] BBL_CR_ADDR [31:3] BBL_CR_ADDR [2:0]	Address register: used to send specified address (A31-A3) to L2 during cache initialization accesses. Reserved, Address bits [35:3] Reserved Set to 0.
118H	280	BBL_CR_DECC[63:0]	Data ECC register D[7:0]: used to write ECC and read ECC to/from L2
119H	281	BBL_CR_CTL BL_CR_CTL[63:22] BBL_CR_CTL[21]	Control register: used to program L2 commands to be issued via cache configuration accesses mechanism. Also receives L2 lookup response Reserved Processor number[2] Disable = 1 Enable = 0 Reserved
		BBL_CR_CTL[20:19] BBL_CR_CTL[18] BBL_CR_CTL[17] BBL_CR_CTL[16] BBL_CR_CTL[15:14] BBL_CR_CTL[13:12] BBL_CR_CTL[11:10] BBL_CR_CTL[9:8] BBL_CR_CTL[7] BBL_CR_CTL[6:5]	User supplied ECC Reserved L2 Hit Reserved State from L2 Modified - 11,Exclusive - 10, Shared - 01, Invalid - 00 Way from L2 Way 0 - 00, Way 1 - 01, Way 2 - 10, Way 3 - 11 Way to L2 Reserved State to L2

Table 2-49. MSRs in the P6 Family Processors (Contd.)

Register Address		Register Name	Bit Description
Hex	Dec		
		BBL_CR_CTL[4:0] 01100 01110 01111 00010 00011 010 + MESI encode 111 + MESI encode 100 + MESI encode	L2 Command Data Read w/ LRU update (RLU) Tag Read w/ Data Read (TRR) Tag Inquire (TI) L2 Control Register Read (CR) L2 Control Register Write (CW) Tag Write w/ Data Read (TWR) Tag Write w/ Data Write (TWW) Tag Write (TW)
11AH	282	BBL_CR_TRIG	Trigger register: used to initiate a cache configuration accesses access, Write only with Data = 0.
11BH	283	BBL_CR_BUSY	Busy register: indicates when a cache configuration accesses L2 command is in progress. D[0] = 1 = BUSY
11EH	286	BBL_CR_CTL3 BBL_CR_CTL3[63:26] BBL_CR_CTL3[25] BBL_CR_CTL3[24] BBL_CR_CTL3[23] BBL_CR_CTL3[22:20] 111 110 101 100 011 010 001 000 BBL_CR_CTL3[19] BBL_CR_CTL3[18]	Control register 3: used to configure the L2 Cache Reserved Cache bus fraction (read only) Reserved L2 Hardware Disable (read only) L2 Physical Address Range support 64GBytes 32GBytes 16GBytes 8GBytes 4GBytes 2GBytes 1GBytes 512MBytes Reserved Cache State error checking enable (read/write)

Table 2-49. MSRs in the P6 Family Processors (Contd.)

Register Address		Register Name	Bit Description
Hex	Dec		
		BBL_CR_CTL3[17:13 00001 00010 00100 01000 10000 BBL_CR_CTL3[12:11] BBL_CR_CTL3[10:9] 00 01 10 11 BBL_CR_CTL3[8] BBL_CR_CTL3[7] BBL_CR_CTL3[6] BBL_CR_CTL3[5] BBL_CR_CTL3[4:1] BBL_CR_CTL3[0]	Cache size per bank (read/write) 256KBytes 512KBytes 1MByte 2MByte 4MBytes Number of L2 banks (read only) L2 Associativity (read only) Direct Mapped 2 Way 4 Way Reserved L2 Enabled (read/write) CRTN Parity Check Enable (read/write) Address Parity Check Enable (read/write) ECC Check Enable (read/write) L2 Cache Latency (read/write) L2 Configured (read/write)
174H	372	SYSENTER_CS_MSR	CS register target for CPL 0 code
175H	373	SYSENTER_ESP_MSR	Stack pointer for CPL 0 stack
176H	374	SYSENTER_EIP_MSR	CPL 0 code entry point
179H	377	MCG_CAP	Machine Check Global Control Register
17AH	378	MCG_STATUS	Machine Check Error Reporting Register - contains information related to a machine-check error if its VAL (valid) flag is set. Software is responsible for clearing IA32_MCi_STATUS MSRs by explicitly writing 0s to them; writing 1s to them causes a general-protection exception.
17BH	379	MCG_CTL	Machine Check Error Reporting Register - controls signaling of #MC for errors produced by a particular hardware unit (or group of hardware units).
186H	390	PerfEvtSel0 (EVNTSEL0)	Performance Event Select Register 0 (R/W)
		7:0	**Event Select** Refer to Performance Counter section for a list of event encodings.
		15:8	**UMASK (Unit Mask)** Unit mask register set to 0 to enable all count options.
		16	**USER** Controls the counting of events at Privilege levels of 1, 2, and 3.
		17	**OS** Controls the counting of events at Privilege level of 0.

Table 2-49. MSRs in the P6 Family Processors (Contd.)

Register Address		Register Name	Bit Description
Hex	Dec		
		18	**E** Occurrence/Duration Mode Select 1 = Occurrence 0 = Duration
		19	**PC** Enabled the signaling of performance counter overflow via BP0 pin
		20	**INT** Enables the signaling of counter overflow via input to APIC 1 = Enable 0 = Disable
		22	**ENABLE** Enables the counting of performance events in both counters 1 = Enable 0 = Disable
		23	**INV** Inverts the result of the CMASK condition 1 = Inverted 0 = Non-Inverted
		31:24	CMASK (Counter Mask).
187H	391	PerfEvtSel1 (EVNTSEL1)	Performance Event Select for Counter 1 (R/W)
		7:0	**Event Select** Refer to Performance Counter section for a list of event encodings.
		15:8	**UMASK (Unit Mask)** Unit mask register set to 0 to enable all count options.
		16	**USER** Controls the counting of events at Privilege levels of 1, 2, and 3.
		17	**OS** Controls the counting of events at Privilege level of 0
		18	**E** Occurrence/Duration Mode Select 1 = Occurrence 0 = Duration
		19	**PC** Enabled the signaling of performance counter overflow via BP0 pin.

Table 2-49. MSRs in the P6 Family Processors (Contd.)

Register Address		Register Name	Bit Description
Hex	Dec		
		20	**INT** Enables the signaling of counter overflow via input to APIC 1 = Enable 0 = Disable
		23	**INV** Inverts the result of the CMASK condition 1 = Inverted 0 = Non-Inverted
		31:24	**CMASK (Counter Mask)**
1D9H	473	DEBUGCTLMSR	Enables last branch, interrupt, and exception recording; taken branch breakpoints; the breakpoint reporting pins; and trace messages. This register can be written to using the WRMSR instruction, when operating at privilege level 0 or when in real-address mode.
		0	Enable/Disable Last Branch Records
		1	Branch Trap Flag
		2	Performance Monitoring/Break Point Pins
		3	Performance Monitoring/Break Point Pins
		4	Performance Monitoring/Break Point Pins
		5	Performance Monitoring/Break Point Pins
		6	Enable/Disable Execution Trace Messages
		31:7	Reserved
1DBH	475	LASTBRANCHFROMIP	32-bit register for recording the instruction pointers for the last branch, interrupt, or exception that the processor took prior to a debug exception being generated.
1DCH	476	LASTBRANCHTOIP	32-bit register for recording the instruction pointers for the last branch, interrupt, or exception that the processor took prior to a debug exception being generated.
1DDH	477	LASTINTFROMIP	Last INT from IP
1DEH	478	LASTINTTOIP	Last INT to IP
200H	512	MTRRphysBase0	Memory Type Range Registers
201H	513	MTRRphysMask0	Memory Type Range Registers
202H	514	MTRRphysBase1	Memory Type Range Registers
203H	515	MTRRphysMask1	Memory Type Range Registers
204H	516	MTRRphysBase2	Memory Type Range Registers
205H	517	MTRRphysMask2	Memory Type Range Registers
206H	518	MTRRphysBase3	Memory Type Range Registers
207H	519	MTRRphysMask3	Memory Type Range Registers
208H	520	MTRRphysBase4	Memory Type Range Registers
209H	521	MTRRphysMask4	Memory Type Range Registers
20AH	522	MTRRphysBase5	Memory Type Range Registers

Table 2-49. MSRs in the P6 Family Processors (Contd.)

Register Address		Register Name	Bit Description
Hex	Dec		
20BH	523	MTRRphysMask5	Memory Type Range Registers
20CH	524	MTRRphysBase6	Memory Type Range Registers
20DH	525	MTRRphysMask6	Memory Type Range Registers
20EH	526	MTRRphysBase7	Memory Type Range Registers
20FH	527	MTRRphysMask7	Memory Type Range Registers
250H	592	MTRRfix64K_00000	Memory Type Range Registers
258H	600	MTRRfix16K_80000	Memory Type Range Registers
259H	601	MTRRfix16K_A0000	Memory Type Range Registers
268H	616	MTRRfix4K_C0000	Memory Type Range Registers
269H	617	MTRRfix4K_C8000	Memory Type Range Registers
26AH	618	MTRRfix4K_D0000	Memory Type Range Registers
26BH	619	MTRRfix4K_D8000	Memory Type Range Registers
26CH	620	MTRRfix4K_E0000	Memory Type Range Registers
26DH	621	MTRRfix4K_E8000	Memory Type Range Registers
26EH	622	MTRRfix4K_F0000	Memory Type Range Registers
26FH	623	MTRRfix4K_F8000	Memory Type Range Registers
2FFH	767	MTRRdefType	Memory Type Range Registers
		2:0	Default memory type
		10	Fixed MTRR enable
		11	MTRR Enable
400H	1024	MC0_CTL	Machine Check Error Reporting Register - controls signaling of #MC for errors produced by a particular hardware unit (or group of hardware units).
401H	1025	MC0_STATUS	Machine Check Error Reporting Register - contains information related to a machine-check error if its VAL (valid) flag is set. Software is responsible for clearing IA32_MCi_STATUS MSRs by explicitly writing 0s to them; writing 1s to them causes a general-protection exception.
		15:0	MC_STATUS_MCACOD
		31:16	MC_STATUS_MSCOD
		57	MC_STATUS_DAM
		58	MC_STATUS_ADDRV
		59	MC_STATUS_MISCV
		60	MC_STATUS_EN. (Note: For MC0_STATUS only, this bit is hardcoded to 1.)
		61	MC_STATUS_UC
		62	MC_STATUS_O
		63	MC_STATUS_V
402H	1026	MC0_ADDR	
403H	1027	MC0_MISC	Defined in MCA architecture but not implemented in the P6 family processors.

Table 2-49. MSRs in the P6 Family Processors (Contd.)

Register Address		Register Name	Bit Description
Hex	Dec		
404H	1028	MC1_CTL	
405H	1029	MC1_STATUS	Bit definitions same as MC0_STATUS.
406H	1030	MC1_ADDR	
407H	1031	MC1_MISC	Defined in MCA architecture but not implemented in the P6 family processors.
408H	1032	MC2_CTL	
409H	1033	MC2_STATUS	Bit definitions same as MC0_STATUS.
40AH	1034	MC2_ADDR	
40BH	1035	MC2_MISC	Defined in MCA architecture but not implemented in the P6 family processors.
40CH	1036	MC4_CTL	
40DH	1037	MC4_STATUS	Bit definitions same as MC0_STATUS, except bits 0, 4, 57, and 61 are hardcoded to 1.
40EH	1038	MC4_ADDR	Defined in MCA architecture but not implemented in P6 Family processors.
40FH	1039	MC4_MISC	Defined in MCA architecture but not implemented in the P6 family processors.
410H	1040	MC3_CTL	
411H	1041	MC3_STATUS	Bit definitions same as MC0_STATUS.
412H	1042	MC3_ADDR	
413H	1043	MC3_MISC	Defined in MCA architecture but not implemented in the P6 family processors.

NOTES

1. Bit 0 of this register has been redefined several times, and is no longer used in P6 family processors.
2. The processor number feature may be disabled by setting bit 21 of the BBL_CR_CTL MSR (model-specific register address 119h) to "1". Once set, bit 21 of the BBL_CR_CTL may not be cleared. This bit is write-once. The processor number feature will be disabled until the processor is reset.
3. The Pentium III processor will prevent FSB frequency overclocking with a new shutdown mechanism. If the FSB frequency selected is greater than the internal FSB frequency the processor will shutdown. If the FSB selected is less than the internal FSB frequency the BIOS may choose to use bit 11 to implement its own shutdown policy.

2.22 MSRS IN PENTIUM PROCESSORS

The following MSRs are defined for the Pentium processors. The P5_MC_ADDR, P5_MC_TYPE, and TSC MSRs (named IA32_P5_MC_ADDR, IA32_P5_MC_TYPE, and IA32_TIME_STAMP_COUNTER in the Pentium 4 processor) are architectural; that is, code that accesses these registers will run on Pentium 4 and P6 family processors without generating exceptions (see Section 2.1, "Architectural MSRs"). The CESR, CTR0, and CTR1 MSRs are unique to Pentium processors; code that accesses these registers will generate exceptions on Pentium 4 and P6 family processors.

Table 2-50. MSRs in the Pentium Processor

Register Address		Register Name	Bit Description
Hex	Dec		
0H	0	P5_MC_ADDR	See Section 15.10.2, "Pentium Processor Machine-Check Exception Handling."
1H	1	P5_MC_TYPE	See Section 15.10.2, "Pentium Processor Machine-Check Exception Handling."
10H	16	TSC	See Section 17.17, "Time-Stamp Counter."
11H	17	CESR	See Section 18.6.9.1, "Control and Event Select Register (CESR)."
12H	18	CTR0	Section 18.6.9.3, "Events Counted."
13H	19	CTR1	Section 18.6.9.3, "Events Counted."

2.23 MSR INDEX

MSRs of recent processors are indexed here for convenience. IA32 MSRs are excluded from this index.

MSR Name and CPUID DisplayFamily_DisplayModel	Location
MSR_B0_PMON_EVNT_SEL2	
06_2EH	See Table 2-16
MSR_B0_PMON_EVNT_SEL3	
06_2EH	See Table 2-16
MSR_B0_PMON_MASK	
06_2EH	See Table 2-16
MSR_B0_PMON_MATCH	
06_2EH	See Table 2-16
MSR_B1_PMON_BOX_CTRL	
06_2EH	See Table 2-16
MSR_B1_PMON_BOX_OVF_CTRL	
06_2EH	See Table 2-16
MSR_B1_PMON_BOX_STATUS	
06_2EH	See Table 2-16
MSR_B1_PMON_CTR0	
06_2EH	See Table 2-16
MSR_B1_PMON_CTR1	
06_2EH	See Table 2-16
MSR_B1_PMON_CTR2	
06_2EH	See Table 2-16
MSR_B1_PMON_CTR3	
06_2EH	See Table 2-16
MSR_B1_PMON_EVNT_SEL0	
06_2EH	See Table 2-16
MSR_B1_PMON_EVNT_SEL1	
06_2EH	See Table 2-16
MSR_B1_PMON_EVNT_SEL2	
06_2EH	See Table 2-16
MSR_B1_PMON_EVNT_SEL3	
06_2EH	See Table 2-16
MSR_B1_PMON_MASK	
06_2EH	See Table 2-16
MSR_B1_PMON_MATCH	
06_2EH	See Table 2-16
MSR_BBL_CR_CTL	
06_09H	See Table 2-48
MSR_BBL_CR_CTL3	
06_0FH, 06_17H	See Table 2-3
06_1CH, 06_26H, 06_27H, 06_35H, 06_36H	See Table 2-4
06_37H, 06_4AH, 06_4DH, 06_5AH, 06_5DH	See Table 2-7
06_0EH	See Table 2-47
06_09H	See Table 2-48

MSR Name and CPUID DisplayFamily_DisplayModel	Location
MSR_BPU_CCCR0	
0FH	See Table 2-44
MSR_BPU_CCCR1	
0FH	See Table 2-44
MSR_BPU_CCCR2	
0FH	See Table 2-44
MSR_BPU_CCCR3	
0FH	See Table 2-44
MSR_BPU_COUNTER0	
0FH	See Table 2-44
MSR_BPU_COUNTER1	
0FH	See Table 2-44
MSR_BPU_COUNTER2	
0FH	See Table 2-44
MSR_BPU_COUNTER3	
0FH	See Table 2-44
MSR_BPU_ESCR0	
0FH	See Table 2-44
MSR_BPU_ESCR1	
0FH	See Table 2-44
MSR_BR_DETECT_COUNTER_CONFIG_i	
06_66H	See Table 2-40
MSR_BR_DETECT_CTRL	
06_66H	See Table 2-40
MSR_BR_DETECT_STATUS	
06_66H	See Table 2-40
MSR_BSU_ESCR0	
0FH	See Table 2-44
MSR_BSU_ESCR1	
0FH	See Table 2-44
MSR_C0_PMON_BOX_CTRL	
06_2EH	See Table 2-16
06_2DH	See Table 2-23
06_3FH	See Table 2-32
MSR_C0_PMON_BOX_FILTER	
06_2DH	See Table 2-23
MSR_C0_PMON_BOX_FILTER0	
06_3FH	See Table 2-32
MSR_C0_PMON_BOX_FILTER1	
06_3EH	See Table 2-27
06_3FH	See Table 2-32
MSR_C0_PMON_BOX_OVF_CTRL	

MSR Name and CPUID DisplayFamily_DisplayModel	Location
06_2EH	See Table 2-16
MSR_C0_PMON_BOX_STATUS	
06_2EH	See Table 2-16
06_3FH	See Table 2-32
MSR_C0_PMON_CTR0	
06_2EH	See Table 2-16
06_2DH	See Table 2-23
06_3FH	See Table 2-32
MSR_C0_PMON_CTR1	
06_2EH	See Table 2-16
06_2DH	See Table 2-23
06_3FH	See Table 2-32
MSR_C0_PMON_CTR2	
06_2EH	See Table 2-16
06_2DH	See Table 2-23
06_3FH	See Table 2-32
MSR_C0_PMON_CTR3	
06_2EH	See Table 2-16
06_2DH	See Table 2-23
06_3FH	See Table 2-32
MSR_C0_PMON_CTR4	
06_2EH	See Table 2-16
MSR_C0_PMON_CTR5	
06_2EH	See Table 2-16
MSR_C0_PMON_EVNT_SEL0	
06_2EH	See Table 2-16
06_2DH	See Table 2-23
06_3FH	See Table 2-32
MSR_C0_PMON_EVNT_SEL1	
06_2EH	See Table 2-16
06_2DH	See Table 2-23
06_3FH	See Table 2-32
MSR_C0_PMON_CTR1	
06_2EH	See Table 2-16
06_2DH	See Table 2-23
06_3FH	See Table 2-32
MSR_C0_PMON_EVNT_SEL2	
06_2EH	See Table 2-16
06_2DH	See Table 2-23
06_3FH	See Table 2-32
MSR_C0_PMON_CTR2	
06_2EH	See Table 2-16

MSR Name and CPUID DisplayFamily_DisplayModel	Location
06_2DH	See Table 2-23
06_3FH	See Table 2-32
MSR_C0_PMON_EVNT_SEL3	
06_2EH	See Table 2-16
06_2DH	See Table 2-23
06_3FH	See Table 2-32
MSR_C0_PMON_EVNT_SEL4	
06_2EH	See Table 2-16
MSR_C0_PMON_EVNT_SEL5	
06_2EH	See Table 2-16
MSR_C1_PMON_BOX_CTRL	
06_2EH	See Table 2-16
06_2DH	See Table 2-23
06_3FH	See Table 2-32
MSR_C1_PMON_BOX_FILTER	
06_2DH	See Table 2-23
MSR_C1_PMON_BOX_FILTER0	
06_3FH	See Table 2-32
MSR_C1_PMON_BOX_FILTER1	
06_3EH	See Table 2-27
06_3FH	See Table 2-32
MSR_C1_PMON_BOX_OVF_CTRL	
06_2EH	See Table 2-16
MSR_C1_PMON_BOX_STATUS	
06_2EH	See Table 2-16
06_3FH	See Table 2-32
MSR_C1_PMON_CTR0	
06_2EH	See Table 2-16
06_2DH	See Table 2-23
06_3FH	See Table 2-32
MSR_C1_PMON_CTR1	
06_2EH	See Table 2-16
06_2DH	See Table 2-23
06_3FH	See Table 2-32
MSR_C1_PMON_CTR2	
06_2EH	See Table 2-16
06_2DH	See Table 2-23
06_3FH	See Table 2-32
MSR_C1_PMON_CTR3	
06_2EH	See Table 2-16
06_2DH	See Table 2-23
06_3FH	See Table 2-32

MSR Name and CPUID DisplayFamily_DisplayModel	Location
MSR_C1_PMON_CTR4	
06_2EH	See Table 2-16
MSR_C1_PMON_CTR5	
06_2EH	See Table 2-16
MSR_C1_PMON_EVNT_SEL0	
06_2EH	See Table 2-16
06_2DH	See Table 2-23
06_3FH	See Table 2-32
MSR_C1_PMON_EVNT_SEL1	
06_2EH	See Table 2-16
06_2DH	See Table 2-23
06_3FH	See Table 2-32
MSR_C1_PMON_EVNT_SEL2	
06_2EH	See Table 2-16
06_2DH	See Table 2-23
06_3FH	See Table 2-32
MSR_C1_PMON_EVNT_SEL3	
06_2EH	See Table 2-16
06_2DH	See Table 2-23
06_3FH	See Table 2-32
MSR_C1_PMON_EVNT_SEL4	
06_2EH	See Table 2-16
MSR_C1_PMON_EVNT_SEL5	
06_2EH	See Table 2-16
MSR_C10_PMON_BOX_FILTER	
06_3EH	See Table 2-27
MSR_C10_PMON_BOX_FILTER0	
06_3FH	See Table 2-32
MSR_C10_PMON_BOX_FILTER1	
06_3EH	See Table 2-27
06_3FH	See Table 2-32
MSR_C11_PMON_BOX_FILTER	
06_3EH	See Table 2-27
MSR_C11_PMON_BOX_FILTER0	
06_3FH	See Table 2-32
MSR_C11_PMON_BOX_FILTER1	
06_3EH	See Table 2-27
06_3FH	See Table 2-32
MSR_C12_PMON_BOX_FILTER	
06_3EH	See Table 2-27
MSR_C12_PMON_BOX_FILTER0	
06_3FH	See Table 2-32

MSR Name and CPUID DisplayFamily_DisplayModel	Location
MSR_C12_PMON_BOX_FILTER1	
06_3EH	See Table 2-27
06_3FH	See Table 2-32
MSR_C13_PMON_BOX_FILTER	
06_3EH	See Table 2-27
MSR_C13_PMON_BOX_FILTER0	
06_3FH	See Table 2-32
MSR_C13_PMON_BOX_FILTER1	
06_3EH	See Table 2-27
06_3FH	See Table 2-32
MSR_C14_PMON_BOX_FILTER	
06_3EH	See Table 2-27
MSR_C14_PMON_BOX_FILTER0	
06_3FH	See Table 2-32
MSR_C14_PMON_BOX_FILTER1	
06_3EH	See Table 2-27
06_3FH	See Table 2-32
MSR_C15_PMON_BOX_CTL	
06_3FH	See Table 2-32
MSR_C15_PMON_BOX_FILTER0	
06_3FH	See Table 2-32
MSR_C15_PMON_BOX_FILTER1	
06_3FH	See Table 2-32
MSR_C15_PMON_BOX_STATUS	
06_3FH	See Table 2-32
MSR_C15_PMON_CTR0	
06_3FH	See Table 2-32
MSR_C15_PMON_CTR1	
06_3FH	See Table 2-32
MSR_C15_PMON_CTR2	
06_3FH	See Table 2-32
MSR_C15_PMON_CTR3	
06_3FH	See Table 2-32
MSR_C15_PMON_EVNTSEL0	
06_3FH	See Table 2-32
MSR_C15_PMON_EVNTSEL1	
06_3FH	See Table 2-32
MSR_C15_PMON_EVNTSEL2	
06_3FH	See Table 2-32
MSR_C15_PMON_EVNTSEL3	
06_3FH	See Table 2-32
MSR_C16_PMON_BOX_CTL	

MSR Name and CPUID DisplayFamily_DisplayModel	Location
06_3FH	See Table 2-32
MSR_C16_PMON_BOX_FILTER0	
06_3FH	See Table 2-32
MSR_C16_PMON_BOX_FILTER1	
06_3FH	See Table 2-32
MSR_C16_PMON_BOX_STATUS	
06_3FH	See Table 2-32
MSR_C16_PMON_CTR0	
06_3FH	See Table 2-32
MSR_C16_PMON_CTR3	
06_3FH	See Table 2-32
MSR_C16_PMON_CTR2	
06_3FH	See Table 2-32
MSR_C16_PMON_CTR3	
06_3FH	See Table 2-32
MSR_C16_PMON_EVNTSEL0	
06_3FH	See Table 2-32
MSR_C16_PMON_EVNTSEL1	
06_3FH	See Table 2-32
MSR_C16_PMON_EVNTSEL2	
06_3FH	See Table 2-32
MSR_C16_PMON_EVNTSEL3	
06_3FH	See Table 2-32
MSR_C17_PMON_BOX_CTL	
06_3FH	See Table 2-32
MSR_C17_PMON_BOX_FILTER0	
06_3FH	See Table 2-32
MSR_C17_PMON_BOX_FILTER1	
06_3FH	See Table 2-32
MSR_C17_PMON_BOX_STATUS	
06_3FH	See Table 2-32
MSR_C17_PMON_CTR0	
06_3FH	See Table 2-32
MSR_C17_PMON_CTR1	
06_3FH	See Table 2-32
MSR_C17_PMON_CTR2	
06_3FH	See Table 2-32
MSR_C17_PMON_CTR3	
06_3FH	See Table 2-32
MSR_C17_PMON_EVNTSEL0	
06_3FH	See Table 2-32
MSR_C17_PMON_EVNTSEL1	

MSR Name and CPUID DisplayFamily_DisplayModel	Location
06_3FH	See Table 2-32
MSR_C17_PMON_EVNTSEL2	
06_3FH	See Table 2-32
MSR_C17_PMON_EVNTSEL3	
06_3FH	See Table 2-32
MSR_C2_PMON_BOX_CTRL	
06_2EH	See Table 2-16
06_2DH	See Table 2-23
06_3FH	See Table 2-32
MSR_C2_PMON_BOX_FILTER	
06_2DH	See Table 2-23
MSR_C2_PMON_BOX_FILTER0	
06_3FH	See Table 2-32
MSR_C2_PMON_BOX_FILTER1	
06_3EH	See Table 2-27
06_3FH	See Table 2-32
MSR_C2_PMON_BOX_OVF_CTRL	
06_2EH	See Table 2-16
MSR_C2_PMON_BOX_STATUS	
06_2EH	See Table 2-16
06_3FH	See Table 2-32
MSR_C2_PMON_CTR0	
06_2EH	See Table 2-16
06_2DH	See Table 2-23
06_3FH	See Table 2-32
MSR_C2_PMON_CTR1	
06_2EH	See Table 2-16
06_2DH	See Table 2-23
06_3FH	See Table 2-32
MSR_C2_PMON_CTR2	
06_2EH	See Table 2-16
06_2DH	See Table 2-23
06_3FH	See Table 2-32
MSR_C2_PMON_CTR3	
06_2EH	See Table 2-16
06_2DH	See Table 2-23
06_3FH	See Table 2-32
MSR_C2_PMON_CTR4	
06_2EH	See Table 2-16
MSR_C2_PMON_CTR5	
06_2EH	See Table 2-16
MSR_C2_PMON_EVNT_SEL0	

MSR Name and CPUID DisplayFamily_DisplayModel	Location
06_2EH	See Table 2-16
06_2DH	See Table 2-23
06_3FH	See Table 2-32
MSR_C2_PMON_EVNT_SEL1	
06_2EH	See Table 2-16
06_2DH	See Table 2-23
06_3FH	See Table 2-32
MSR_C2_PMON_EVNT_SEL2	
06_2EH	See Table 2-16
06_2DH	See Table 2-23
06_3FH	See Table 2-32
MSR_C2_PMON_EVNT_SEL3	
06_2EH	See Table 2-16
06_2DH	See Table 2-23
06_3FH	See Table 2-32
MSR_C2_PMON_EVNT_SEL4	
06_2EH	See Table 2-16
MSR_C2_PMON_EVNT_SEL5	
06_2EH	See Table 2-16
MSR_C3_PMON_BOX_CTRL	
06_2EH	See Table 2-16
06_2DH	See Table 2-23
06_3FH	See Table 2-32
MSR_C3_PMON_BOX_FILTER	
06_2DH	See Table 2-23
MSR_C3_PMON_BOX_FILTER0	
06_3FH	See Table 2-32
MSR_C3_PMON_BOX_FILTER1	
06_3EH	See Table 2-27
06_3FH	See Table 2-32
MSR_C3_PMON_BOX_OVF_CTRL	
06_2EH	See Table 2-16
MSR_C3_PMON_BOX_STATUS	
06_2EH	See Table 2-16
06_3FH	See Table 2-32
MSR_C3_PMON_CTR0	
06_2EH	See Table 2-16
06_2DH	See Table 2-23
06_3FH	See Table 2-32
MSR_C3_PMON_CTR1	
06_2EH	See Table 2-16
06_2DH	See Table 2-23

MSR Name and CPUID DisplayFamily_DisplayModel	Location
06_3FH	See Table 2-32
MSR_C3_PMON_CTR2	
06_2EH	See Table 2-16
06_2DH	See Table 2-23
06_3FH	See Table 2-32
MSR_C3_PMON_CTR3	
06_2EH	See Table 2-16
06_2DH	See Table 2-23
06_3FH	See Table 2-32
MSR_C3_PMON_CTR4	
06_2EH	See Table 2-16
MSR_C3_PMON_CTR5	
06_2EH	See Table 2-16
MSR_C3_PMON_EVNT_SEL0	
06_2EH	See Table 2-16
06_2DH	See Table 2-23
06_3FH	See Table 2-32
MSR_C3_PMON_EVNT_SEL1	
06_2EH	See Table 2-16
06_2DH	See Table 2-23
06_3FH	See Table 2-32
MSR_C3_PMON_EVNT_SEL2	
06_2EH	See Table 2-16
06_2DH	See Table 2-23
06_3FH	See Table 2-32
MSR_C3_PMON_EVNT_SEL3	
06_2EH	See Table 2-16
06_2DH	See Table 2-23
06_3FH	See Table 2-32
MSR_C3_PMON_EVNT_SEL4	
06_2EH	See Table 2-16
MSR_C3_PMON_EVNT_SEL5	
06_2EH	See Table 2-16
MSR_C4_PMON_BOX_CTRL	
06_2EH	See Table 2-16
06_2DH	See Table 2-23
06_3FH	See Table 2-32
MSR_C4_PMON_BOX_FILTER	
06_2DH	See Table 2-23
MSR_C4_PMON_BOX_FILTER0	
06_3FH	See Table 2-32
MSR_C4_PMON_BOX_FILTER1	

MSR Name and CPUID DisplayFamily_DisplayModel	Location
06_3EH	See Table 2-27
06_3FH	See Table 2-32
MSR_C4_PMON_BOX_OVF_CTRL	
06_2EH	See Table 2-16
MSR_C4_PMON_BOX_STATUS	
06_2EH	See Table 2-16
06_3FH	See Table 2-32
MSR_C4_PMON_CTR0	
06_2EH	See Table 2-16
06_2DH	See Table 2-23
06_3FH	See Table 2-32
MSR_C4_PMON_CTR1	
06_2EH	See Table 2-16
06_2DH	See Table 2-23
06_3FH	See Table 2-32
MSR_C4_PMON_CTR2	
06_2EH	See Table 2-16
06_2DH	See Table 2-23
06_3FH	See Table 2-32
MSR_C4_PMON_CTR3	
06_2EH	See Table 2-16
06_2DH	See Table 2-23
06_3FH	See Table 2-32
MSR_C4_PMON_CTR4	
06_2EH	See Table 2-16
MSR_C4_PMON_CTR5	
06_2EH	See Table 2-16
MSR_C4_PMON_EVNT_SEL0	
06_2EH	See Table 2-16
06_2DH	See Table 2-23
06_3FH	See Table 2-32
MSR_C4_PMON_EVNT_SEL1	
06_2EH	See Table 2-16
06_2DH	See Table 2-23
06_3FH	See Table 2-32
MSR_C4_PMON_EVNT_SEL2	
06_2EH	See Table 2-16
06_2DH	See Table 2-23
06_3FH	See Table 2-32
MSR_C4_PMON_EVNT_SEL3	
06_2EH	See Table 2-16
06_2DH	See Table 2-23

MSR Name and CPUID DisplayFamily_DisplayModel	Location
06_3FH	See Table 2-32
MSR_C4_PMON_EVNT_SEL4	
06_2EH	See Table 2-16
MSR_C4_PMON_EVNT_SEL5	
06_2EH	See Table 2-16
MSR_C5_PMON_BOX_CTRL	
06_2EH	See Table 2-16
06_2DH	See Table 2-23
06_3FH	See Table 2-32
MSR_C5_PMON_BOX_FILTER	
06_2DH	See Table 2-23
MSR_C5_PMON_BOX_FILTER0	
06_3FH	See Table 2-32
MSR_C5_PMON_BOX_FILTER1	
06_3EH	See Table 2-27
06_3FH	See Table 2-32
MSR_C5_PMON_BOX_OVF_CTRL	
06_2EH	See Table 2-16
MSR_C5_PMON_BOX_STATUS	
06_2EH	See Table 2-16
06_3FH	See Table 2-32
MSR_C5_PMON_CTR0	
06_2EH	See Table 2-16
06_2DH	See Table 2-23
06_3FH	See Table 2-32
MSR_C5_PMON_CTR1	
06_2EH	See Table 2-16
06_2DH	See Table 2-23
06_3FH	See Table 2-32
MSR_C5_PMON_CTR2	
06_2EH	See Table 2-16
06_2DH	See Table 2-23
06_3FH	See Table 2-32
MSR_C5_PMON_CTR3	
06_2EH	See Table 2-16
06_2DH	See Table 2-23
06_3FH	See Table 2-32
MSR_C5_PMON_CTR4	
06_2EH	See Table 2-16
MSR_C5_PMON_CTR5	
06_2EH	See Table 2-16
MSR_C5_PMON_EVNT_SEL0	

MSR Name and CPUID DisplayFamily_DisplayModel	Location
06_2EH	See Table 2-16
06_2DH	See Table 2-23
06_3FH	See Table 2-32
MSR_C5_PMON_EVNT_SEL1	
06_2EH	See Table 2-16
06_2DH	See Table 2-23
06_3FH	See Table 2-32
MSR_C5_PMON_EVNT_SEL2	
06_2EH	See Table 2-16
06_2DH	See Table 2-23
06_3FH	See Table 2-32
MSR_C5_PMON_EVNT_SEL3	
06_2EH	See Table 2-16
06_2DH	See Table 2-23
06_3FH	See Table 2-32
MSR_C5_PMON_EVNT_SEL4	
06_2EH	See Table 2-16
MSR_C5_PMON_EVNT_SEL5	
06_2EH	See Table 2-16
MSR_C6_PMON_BOX_CTRL	
06_2EH	See Table 2-16
06_2DH	See Table 2-23
06_3FH	See Table 2-32
MSR_C6_PMON_BOX_FILTER	
06_2DH	See Table 2-23
MSR_C6_PMON_BOX_FILTER0	
06_3FH	See Table 2-32
MSR_C6_PMON_BOX_FILTER1	
06_3EH	See Table 2-27
06_3FH	See Table 2-32
MSR_C6_PMON_BOX_OVF_CTRL	
06_2EH	See Table 2-16
MSR_C6_PMON_BOX_STATUS	
06_2EH	See Table 2-16
06_3FH	See Table 2-32
MSR_C6_PMON_CTR0	
06_2EH	See Table 2-16
06_2DH	See Table 2-23
06_3FH	See Table 2-32
MSR_C6_PMON_CTR1	
06_2EH	See Table 2-16
06_2DH	See Table 2-23

MSR Name and CPUID DisplayFamily_DisplayModel	Location
06_3FH	See Table 2-32
MSR_C6_PMON_CTR2	
06_2EH	See Table 2-16
06_2DH	See Table 2-23
06_3FH	See Table 2-32
MSR_C6_PMON_CTR3	
06_2EH	See Table 2-16
06_2DH	See Table 2-23
06_3FH	See Table 2-32
MSR_C6_PMON_CTR4	
06_2EH	See Table 2-16
MSR_C6_PMON_CTR5	
06_2EH	See Table 2-16
MSR_C6_PMON_EVNT_SEL0	
06_2EH	See Table 2-16
06_2DH	See Table 2-23
06_3FH	See Table 2-32
MSR_C6_PMON_EVNT_SEL1	
06_2EH	See Table 2-16
06_2DH	See Table 2-23
06_3FH	See Table 2-32
MSR_C6_PMON_EVNT_SEL2	
06_2EH	See Table 2-16
06_2DH	See Table 2-23
06_3FH	See Table 2-32
MSR_C6_PMON_EVNT_SEL3	
06_2EH	See Table 2-16
06_2DH	See Table 2-23
06_3FH	See Table 2-32
MSR_C6_PMON_EVNT_SEL4	
06_2EH	See Table 2-16
MSR_C6_PMON_EVNT_SEL5	
06_2EH	See Table 2-16
MSR_C7_PMON_BOX_CTRL	
06_2EH	See Table 2-16
06_2DH	See Table 2-23
06_3FH	See Table 2-32
MSR_C7_PMON_BOX_FILTER	
06_2DH	See Table 2-23
MSR_C7_PMON_BOX_FILTER0	
06_3FH	See Table 2-32
MSR_C7_PMON_BOX_FILTER1	

MSR Name and CPUID DisplayFamily_DisplayModel	Location
06_3EH	See Table 2-27
06_3FH	See Table 2-32
MSR_C7_PMON_BOX_OVF_CTRL	
06_2EH	See Table 2-16
MSR_C7_PMON_BOX_STATUS	
06_2EH	See Table 2-16
06_3FH	See Table 2-32
MSR_C7_PMON_CTR0	
06_2EH	See Table 2-16
06_2DH	See Table 2-23
06_3FH	See Table 2-32
MSR_C7_PMON_CTR1	
06_2EH	See Table 2-16
06_2DH	See Table 2-23
06_3FH	See Table 2-32
MSR_C7_PMON_CTR2	
06_2EH	See Table 2-16
06_2DH	See Table 2-23
06_3FH	See Table 2-32
MSR_C7_PMON_CTR3	
06_2EH	See Table 2-16
06_2DH	See Table 2-23
06_3FH	See Table 2-32
MSR_C7_PMON_CTR4	
06_2EH	See Table 2-16
MSR_C7_PMON_CTR5	
06_2EH	See Table 2-16
MSR_C7_PMON_EVNT_SEL0	
06_2EH	See Table 2-16
06_2DH	See Table 2-23
06_3FH	See Table 2-32
MSR_C7_PMON_EVNT_SEL1	
06_2EH	See Table 2-16
06_2DH	See Table 2-23
06_3FH	See Table 2-32
MSR_C7_PMON_EVNT_SEL2	
06_2EH	See Table 2-16
06_2DH	See Table 2-23
06_3FH	See Table 2-32
MSR_C7_PMON_EVNT_SEL3	
06_2EH	See Table 2-16
06_2DH	See Table 2-23

MSR Name and CPUID DisplayFamily_DisplayModel	Location
06_3FH	See Table 2-32
MSR_C7_PMON_EVNT_SEL4	
06_2EH	See Table 2-16
MSR_C7_PMON_EVNT_SEL5	
06_2EH	See Table 2-16
MSR_C8_PMON_BOX_CTRL	
06_2FH	See Table 2-18
06_3EH	See Table 2-27
06_3FH	See Table 2-32
MSR_C8_PMON_BOX_FILTER	
06_3EH	See Table 2-27
MSR_C8_PMON_BOX_FILTER0	
06_3FH	See Table 2-32
MSR_C8_PMON_BOX_FILTER1	
06_3EH	See Table 2-27
06_3FH	See Table 2-32
MSR_C8_PMON_BOX_OVF_CTRL	
06_2FH	See Table 2-18
MSR_C8_PMON_BOX_STATUS	
06_2FH	See Table 2-18
06_3FH	See Table 2-32
MSR_C8_PMON_CTR0	
06_2FH	See Table 2-18
06_3EH	See Table 2-27
06_3FH	See Table 2-32
MSR_C8_PMON_CTR1	
06_2FH	See Table 2-18
06_3EH	See Table 2-27
06_3FH	See Table 2-32
MSR_C8_PMON_CTR2	
06_2FH	See Table 2-18
06_3EH	See Table 2-27
06_3FH	See Table 2-32
MSR_C8_PMON_CTR3	
06_2FH	See Table 2-18
06_3EH	See Table 2-27
06_3FH	See Table 2-32
MSR_C8_PMON_CTR4	
06_2FH	See Table 2-18
MSR_C8_PMON_CTR5	
06_2FH	See Table 2-18
MSR_C8_PMON_EVNT_SEL0	

MSR Name and CPUID DisplayFamily_DisplayModel	Location
06_2FH	See Table 2-18
06_3EH	See Table 2-27
06_3FH	See Table 2-32
MSR_C8_PMON_EVNT_SEL1	
06_2FH	See Table 2-18
06_3EH	See Table 2-27
06_3FH	See Table 2-32
MSR_C8_PMON_EVNT_SEL2	
06_2FH	See Table 2-18
06_3EH	See Table 2-27
06_3FH	See Table 2-32
MSR_C8_PMON_EVNT_SEL3	
06_2FH	See Table 2-18
06_3EH	See Table 2-27
06_3FH	See Table 2-32
MSR_C8_PMON_EVNT_SEL4	
06_2FH	See Table 2-18
MSR_C8_PMON_EVNT_SEL5	
06_2FH	See Table 2-18
MSR_C9_PMON_BOX_CTRL	
06_2FH	See Table 2-18
06_3EH	See Table 2-27
06_3FH	See Table 2-32
MSR_C9_PMON_BOX_FILTER	
06_3EH	See Table 2-27
MSR_C9_PMON_BOX_FILTER0	
06_3FH	See Table 2-32
MSR_C9_PMON_BOX_FILTER1	
06_3EH	See Table 2-27
06_3FH	See Table 2-32
MSR_C9_PMON_BOX_OVF_CTRL	
06_2FH	See Table 2-18
MSR_C9_PMON_BOX_STATUS	
06_2FH	See Table 2-18
06_3FH	See Table 2-32
MSR_C9_PMON_CTR0	
06_2FH	See Table 2-18
06_3EH	See Table 2-27
06_3FH	See Table 2-32
MSR_C9_PMON_CTR1	
06_2FH	See Table 2-18
06_3EH	See Table 2-27

MSR Name and CPUID DisplayFamily_DisplayModel	Location
06_3FH	See Table 2-32
MSR_C9_PMON_CTR2	
06_2FH	See Table 2-18
06_3EH	See Table 2-27
06_3FH	See Table 2-32
MSR_C9_PMON_CTR3	
06_2FH	See Table 2-18
06_3EH	See Table 2-27
06_3FH	See Table 2-32
MSR_C9_PMON_CTR4	
06_2FH	See Table 2-18
MSR_C9_PMON_CTR5	
06_2FH	See Table 2-18
MSR_C9_PMON_EVNT_SEL0	
06_2FH	See Table 2-18
06_3EH	See Table 2-27
06_3FH	See Table 2-32
MSR_C9_PMON_EVNT_SEL1	
06_2FH	See Table 2-18
06_3EH	See Table 2-27
06_3FH	See Table 2-32
MSR_C9_PMON_EVNT_SEL2	
06_2FH	See Table 2-18
06_3EH	See Table 2-27
06_3FH	See Table 2-32
MSR_C9_PMON_EVNT_SEL3	
06_2FH	See Table 2-18
06_3EH	See Table 2-27
06_3FH	See Table 2-32
MSR_C9_PMON_EVNT_SEL4	
06_2FH	See Table 2-18
MSR_C9_PMON_EVNT_SEL5	
06_2FH	See Table 2-18
MSR_CC6_DEMOTION_POLICY_CONFIG	
06_37H	See Table 2-9
MSR_CONFIG_TDP_CONTROL	
06_3AH	See Table 2-24
06_3CH, 06_45H, 06_46H	See Table 2-28
06_57H	See Table 2-42
MSR_CONFIG_TDP_LEVEL1	
06_3AH	See Table 2-24
06_3CH, 06_45H, 06_46H	See Table 2-28

MSR Name and CPUID DisplayFamily_DisplayModel	Location
06_57H	See Table 2-42
MSR_CONFIG_TDP_LEVEL2	
06_3AH	See Table 2-24
06_3CH, 06_45H, 06_46H	See Table 2-28
06_57H	See Table 2-42
MSR_CONFIG_TDP_NOMINAL	
06_3AH	See Table 2-24
06_3CH, 06_45H, 06_46H	See Table 2-28
06_57H	See Table 2-42
MSR_CORE_C1_RESIDENCY	
06_37H, 06_4AH, 06_4DH, 06_5AH, 06_5DH, 06_5CH, 06_7AH	See Table 2-6
06_66H	See Table 2-40
MSR_CORE_C3_RESIDENCY	
06_5CH, 06_7AH	See Table 2-12
06_1AH, 06_1EH, 06_1FH, 06_2EH, 06_25H, 06_2CH, 06_2FH	See Table 2-14
06_2AH, 06_2DH, 06_3AH, 06_3CH, 06_3EH, 06_3FH, 06_45H, 06_46H	See Table 2-19
MSR_CORE_C6_RESIDENCY	
06_37H, 06_4AH, 06_4DH, 06_5AH, 06_5DH, 06_5CH, 06_7AH	See Table 2-6
06_1AH, 06_1EH, 06_1FH, 06_2EH, 06_25H, 06_2CH, 06_2FH	See Table 2-14
06_2AH, 06_2DH, 06_3AH, 06_3CH, 06_3EH, 06_3FH, 06_45H, 06_46H	See Table 2-19
06_57H	See Table 2-42
MSR_CORE_C7_RESIDENCY	
06_2AH, 06_2DH, 06_3AH, 06_3CH, 06_3EH, 06_3FH, 06_45H, 06_46H	See Table 2-19
MSR_CORE_GFXE_OVERLAP_C0	
06_4EH, 06_5EH, 06_55H, 06_8EH, 06_9EH, 06_66H	See Table 2-38
MSR_CORE_HDC_RESIDENCY	
06_4EH, 06_5EH, 06_55H, 06_8EH, 06_9EH, 06_66H	See Table 2-38
MSR_CORE_PERF_LIMIT_REASONS	
06_5CH, 06_7AH	See Table 2-12
06_3CH, 06_45H, 06_46H	See Table 2-29
06_3F	See Table 2-31
06_56H, 06_4FH	See Table 2-35
06_57H	See Table 2-42
MSR_CORE_THREAD_COUNT	
06_3FH	See Table 2-31
MSR_CRU_ESCR0	
0FH	See Table 2-44
MSR_CRU_ESCR1	
0FH	See Table 2-44
MSR_CRU_ESCR2	
0FH	See Table 2-44
MSR_CRU_ESCR3	

MSR Name and CPUID DisplayFamily_DisplayModel	Location
0FH	See Table 2-44
MSR_CRU_ESCR4	
0FH	See Table 2-44
MSR_CRU_ESCR5	
0FH	See Table 2-44
MSR_DAC_ESCR0	
0FH	See Table 2-44
MSR_DAC_ESCR1	
0FH	See Table 2-44
MSR_DRAM_ENERGY_ STATUS	
06_5CH, 06_7AH	See Table 2-12
06_2DH	See Table 2-22
06_3EH, 06_3FH	See Table 2-25
06_3CH, 06_45H, 06_46H	See Table 2-28
06_3F	See Table 2-31
06_56H, 06_4FH	See Table 2-35
06_57H	See Table 2-42
MSR_DRAM_PERF_STATUS	
06_5CH, 06_7AH	See Table 2-12
06_2DH	See Table 2-22
06_3EH, 06_3FH	See Table 2-25
06_3CH, 06_45H, 06_46H	See Table 2-28
06_3F	See Table 2-31
06_56H, 06_4FH	See Table 2-35
06_57H	See Table 2-42
MSR_DRAM_POWER_INFO	
06_5CH, 06_7AH	See Table 2-12
06_2DH	See Table 2-22
06_3EH, 06_3FH	See Table 2-25
06_3F	See Table 2-31
06_56H, 06_4FH	See Table 2-35
06_57H	See Table 2-42
MSR_DRAM_POWER_LIMIT	
06_5CH, 06_7AH	See Table 2-12
06_2DH	See Table 2-22
06_3EH, 06_3FH	See Table 2-25
06_3F	See Table 2-31
06_56H, 06_4FH	See Table 2-35
06_57H	See Table 2-42
MSR_EBC_FREQUENCY_ID	
0FH	See Table 2-44
MSR_EBC_HARD_POWERON	

MSR Name and CPUID DisplayFamily_DisplayModel	Location
0FH	See Table 2-44
MSR_EBC_SOFT_POWERON	
0FH	See Table 2-44
MSR_EBL_CR_POWERON	
06_0FH, 06_17H	See Table 2-3
06_1CH, 06_26H, 06_27H, 06_35H, 06_36H	See Table 2-4
06_37H, 06_4AH, 06_4DH, 06_5AH, 06_5DH, 06_5CH, 06_7AH.	See Table 2-6
06_0EH	See Table 2-47
06_09H	See Table 2-48
MSR_EFSB_DRDY0	
0F_03H, 0F_04H	See Table 2-45
MSR_EFSB_DRDY1	
0F_03H, 0F_04H	See Table 2-45
MSR_EMON_L3_CTR_CTL0	
06_0FH, 06_17H	See Table 2-3
0F_06H	See Table 2-46
MSR_EMON_L3_CTR_CTL1	
06_0FH, 06_17H	See Table 2-3
0F_06H	See Table 2-46
MSR_EMON_L3_CTR_CTL2	
06_0FH, 06_17H	See Table 2-3
0F_06H	See Table 2-46
MSR_EMON_L3_CTR_CTL3	
06_0FH, 06_17H	See Table 2-3
0F_06H	See Table 2-46
MSR_EMON_L3_CTR_CTL4	
06_0FH, 06_17H	See Table 2-3
0F_06H	See Table 2-46
MSR_EMON_L3_CTR_CTL5	
06_0FH, 06_17H	See Table 2-3
0F_06H	See Table 2-46
MSR_EMON_L3_CTR_CTL6	
06_0FH, 06_17H	See Table 2-3
0F_06H	See Table 2-46
MSR_EMON_L3_CTR_CTL7	
06_0FH, 06_17H	See Table 2-3
0F_06H	See Table 2-46
MSR_EMON_L3_GL_CTL	
06_0FH, 06_17H	See Table 2-3
MSR_ERROR_CONTROL	
06_2DH	See Table 2-22
06_3EH	See Table 2-25

MSR Name and CPUID DisplayFamily_DisplayModel	Location
06_3F	See Table 2-31
MSR_FEATURE_CONFIG	
06_37H, 06_4AH, 06_4DH, 06_5AH, 06_5DH, 06_5CH, 06_7AH.	See Table 2-6
06_25H, 06_2CH.	See Table 2-17
06_2FH	See Table 2-18
06_2AH, 06_2DH.	See Table 2-19
06_57H	See Table 2-42
MSR_FIRM_ESCR0	
0FH	See Table 2-44
MSR_FIRM_ESCR1	
0FH	See Table 2-44
MSR_FLAME_CCCR0	
0FH	See Table 2-44
MSR_FLAME_CCCR1	
0FH	See Table 2-44
MSR_FLAME_CCCR2	
0FH	See Table 2-44
MSR_FLAME_CCCR3	
0FH	See Table 2-44
MSR_FLAME_COUNTER0	
0FH	See Table 2-44
MSR_FLAME_COUNTER1	
0FH	See Table 2-44
MSR_FLAME_COUNTER2	
0FH	See Table 2-44
MSR_FLAME_COUNTER3	
0FH	See Table 2-44
MSR_FLAME_ESCR0	
0FH	See Table 2-44
MSR_FLAME_ESCR1	
0FH	See Table 2-44
MSR_FSB_ESCR0	
0FH	See Table 2-44
MSR_FSB_ESCR1	
0FH	See Table 2-44
MSR_FSB_FREQ	
06_0FH, 06_17H	See Table 2-3
06_1CH, 06_26H, 06_27H, 06_35H, 06_36H	See Table 2-4
06_37H, 06_4AH, 06_4DH, 06_5AH, 06_5DH	See Table 2-7
06_4CH	See Table 2-11
06_0EH	See Table 2-47
MSR_GQ_SNOOP_MESF	

MSR Name and CPUID DisplayFamily_DisplayModel	Location
06_1AH, 06_1EH, 06_1FH, 06_25H, 06_2CH	See Table 2-15
MSR_GRAPHICS_PERF_LIMIT_REASONS	
06_3CH, 06_45H, 06_46H	See Table 2-29
MSR_IFSB_BUSQ0	
0F_03H, 0F_04H	See Table 2-45
MSR_IFSB_BUSQ1	
0F_03H, 0F_04H	See Table 2-45
MSR_IFSB_CNTR7	
0F_03H, 0F_04H	See Table 2-45
MSR_IFSB_CTL6	
0F_03H, 0F_04H	See Table 2-45
MSR_IFSB_SNPQ0	
0F_03H, 0F_04H	See Table 2-45
MSR_IFSB_SNPQ1	
0F_03H, 0F_04H	See Table 2-45
MSR_IQ_CCCR0	
0FH	See Table 2-44
MSR_IQ_CCCR1	
0FH	See Table 2-44
MSR_IQ_CCCR2	
0FH	See Table 2-44
MSR_IQ_CCCR3	
0FH	See Table 2-44
MSR_IQ_CCCR4	
0FH	See Table 2-44
MSR_IQ_CCCR5	
0FH	See Table 2-44
MSR_IQ_COUNTER0	
0FH	See Table 2-44
MSR_IQ_COUNTER1	
0FH	See Table 2-44
MSR_IQ_COUNTER2	
0FH	See Table 2-44
MSR_IQ_COUNTER3	
0FH	See Table 2-44
MSR_IQ_COUNTER4	
0FH	See Table 2-44
MSR_IQ_COUNTER5	
0FH	See Table 2-44
MSR_IQ_ESCR0	
0FH	See Table 2-44
MSR_IQ_ESCR1	

MSR Name and CPUID DisplayFamily_DisplayModel	Location
0FH	See Table 2-44
MSR_IS_ESCR0	
0FH	See Table 2-44
MSR_IS_ESCR1	
0FH	See Table 2-44
MSR_ITLB_ESCR0	
0FH	See Table 2-44
MSR_ITLB_ESCR1	
0FH	See Table 2-44
MSR_IX_ESCR0	
0FH	See Table 2-44
MSR_IX_ESCR1	
0FH	See Table 2-44
MSR_LASTBRANCH_0	
0FH	See Table 2-44
06_0EH	See Table 2-47
06_09H	See Table 2-48
MSR_LASTBRANCH_0_FROM_IP	
06_0FH, 06_17H	See Table 2-3
06_1CH, 06_26H, 06_27H, 06_35H, 06_36H	See Table 2-4
06_37H, 06_4AH, 06_4DH, 06_5AH, 06_5DH	See Table 2-7
06_5CH	See Table 2-12
06_7AH	See Table 2-13
06_1AH, 06_1EH, 06_1FH, 06_2EH	See Table 2-14
06_2AH, 06_2DH	See Table 2-19
0FH	See Table 2-44
MSR_LASTBRANCH_0_TO_IP	
06_0FH, 06_17H	See Table 2-3
06_1CH, 06_26H, 06_27H, 06_35H, 06_36H	See Table 2-4
06_37H, 06_4AH, 06_4DH, 06_5AH, 06_5DH	See Table 2-7
06_5CH	See Table 2-12
06_7AH	See Table 2-13
06_1AH, 06_1EH, 06_1FH, 06_2EH	See Table 2-14
06_2AH, 06_2DH	See Table 2-19
0FH	See Table 2-44
MSR_LASTBRANCH_1_FROM_IP	
06_0FH, 06_17H	See Table 2-3
06_1CH, 06_26H, 06_27H, 06_35H, 06_36H	See Table 2-4
06_37H, 06_4AH, 06_4DH, 06_5AH, 06_5DH	See Table 2-7
06_5CH, 06_7AH	See Table 2-12
06_1AH, 06_1EH, 06_1FH, 06_2EH	See Table 2-14
06_2AH, 06_2DH	See Table 2-19

MSR Name and CPUID DisplayFamily_DisplayModel	Location
0FH	See Table 2-44
MSR_LASTBRANCH_1_TO_IP	
06_0FH, 06_17H	See Table 2-3
06_1CH, 06_26H, 06_27H, 06_35H, 06_36H	See Table 2-4
06_37H, 06_4AH, 06_4DH, 06_5AH, 06_5DH	See Table 2-7
06_5CH, 06_7AH	See Table 2-12
06_1AH, 06_1EH, 06_1FH, 06_2EH	See Table 2-14
06_2AH, 06_2DH	See Table 2-19
0FH	See Table 2-44
MSR_LASTBRANCH_10_FROM_IP	
06_5CH, 06_7AH	See Table 2-12
06_1AH, 06_1EH, 06_1FH, 06_2EH	See Table 2-14
06_2AH, 06_2DH	See Table 2-19
0FH	See Table 2-44
MSR_LASTBRANCH_10_TO_IP	
06_5CH, 06_7AH	See Table 2-12
06_1AH, 06_1EH, 06_1FH, 06_2EH	See Table 2-14
06_2AH, 06_2DH	See Table 2-19
0FH	See Table 2-44
MSR_LASTBRANCH_11_FROM_IP	
06_5CH, 06_7AH	See Table 2-12
06_1AH, 06_1EH, 06_1FH, 06_2EH	See Table 2-14
06_2AH, 06_2DH	See Table 2-19
0FH	See Table 2-44
MSR_LASTBRANCH_11_TO_IP	
06_5CH, 06_7AH	See Table 2-12
06_1AH, 06_1EH, 06_1FH, 06_2EH	See Table 2-14
06_2AH, 06_2DH	See Table 2-19
0FH	See Table 2-44
MSR_LASTBRANCH_12_FROM_IP	
06_5CH, 06_7AH	See Table 2-12
06_1AH, 06_1EH, 06_1FH, 06_2EH	See Table 2-14
06_2AH, 06_2DH	See Table 2-19
0FH	See Table 2-44
MSR_LASTBRANCH_12_TO_IP	
06_5CH, 06_7AH	See Table 2-12
06_1AH, 06_1EH, 06_1FH, 06_2EH	See Table 2-14
06_2AH, 06_2DH	See Table 2-19
0FH	See Table 2-44
MSR_LASTBRANCH_13_FROM_IP	
06_5CH, 06_7AH	See Table 2-12
06_1AH, 06_1EH, 06_1FH, 06_2EH	See Table 2-14

MSR Name and CPUID DisplayFamily_DisplayModel	Location
06_2AH, 06_2DH	See Table 2-19
0FH	See Table 2-44
MSR_LASTBRANCH_13_TO_IP	
06_5CH, 06_7AH	See Table 2-12
06_1AH, 06_1EH, 06_1FH, 06_2EH	See Table 2-14
06_2AH, 06_2DH	See Table 2-19
0FH	See Table 2-44
MSR_LASTBRANCH_14_FROM_IP	
06_5CH, 06_7AH	See Table 2-12
06_1AH, 06_1EH, 06_1FH, 06_2EH	See Table 2-14
06_2AH, 06_2DH	See Table 2-19
0FH	See Table 2-44
MSR_LASTBRANCH_14_TO_IP	
06_5CH, 06_7AH	See Table 2-12
06_1AH, 06_1EH, 06_1FH, 06_2EH	See Table 2-14
06_2AH, 06_2DH	See Table 2-19
0FH	See Table 2-44
MSR_LASTBRANCH_15_FROM_IP	
06_5CH, 06_7AH	See Table 2-12
06_1AH, 06_1EH, 06_1FH, 06_2EH	See Table 2-14
06_2AH, 06_2DH	See Table 2-19
0FH	See Table 2-44
MSR_LASTBRANCH_15_TO_IP	
06_5CH, 06_7AH	See Table 2-12
06_1AH, 06_1EH, 06_1FH, 06_2EH	See Table 2-14
06_2AH, 06_2DH	See Table 2-19
0FH.	See Table 2-44
MSR_LASTBRANCH_16_FROM_IP	
06_5CH, 06_7AH	See Table 2-12
06_4EH, 06_5EH, 06_55H, 06_8EH, 06_9EH, 06_66H.	See Table 2-38
MSR_LASTBRANCH_16_TO_IP	
06_5CH, 06_7AH	See Table 2-12
06_4EH, 06_5EH, 06_55H, 06_8EH, 06_9EH, 06_66H.	See Table 2-38
MSR_LASTBRANCH_17_FROM_IP	
06_5CH, 06_7AH	See Table 2-12
06_4EH, 06_5EH, 06_55H, 06_8EH, 06_9EH, 06_66H.	See Table 2-38
MSR_LASTBRANCH_17_TO_IP	
06_5CH, 06_7AH	See Table 2-12
06_4EH, 06_5EH, 06_55H, 06_8EH, 06_9EH, 06_66H.	See Table 2-38
MSR_LASTBRANCH_18_FROM_IP	
06_5CH, 06_7AH	See Table 2-12
06_4EH, 06_5EH, 06_55H, 06_8EH, 06_9EH, 06_66H.	See Table 2-38

MSR Name and CPUID DisplayFamily_DisplayModel	Location
MSR_LASTBRANCH_18_TO_IP	
06_5CH, 06_7AH	See Table 2-12
06_4EH, 06_5EH, 06_55H, 06_8EH, 06_9EH, 06_66H	See Table 2-38
MSR_LASTBRANCH_19_FROM_IP	
06_5CH, 06_7AH	See Table 2-12
06_4EH, 06_5EH, 06_55H, 06_8EH, 06_9EH, 06_66H	See Table 2-38
MSR_LASTBRANCH_19_TO_IP	
06_5CH, 06_7AH	See Table 2-12
06_4EH, 06_5EH, 06_55H, 06_8EH, 06_9EH, 06_66H	See Table 2-38
MSR_LASTBRANCH_2	
0FH	See Table 2-44
06_0EH	See Table 2-47
06_09H	See Table 2-48
MSR_LASTBRANCH_2_FROM_IP	
06_0FH, 06_17H	See Table 2-3
06_1CH, 06_26H, 06_27H, 06_35H, 06_36H	See Table 2-4
06_37H, 06_4AH, 06_4DH, 06_5AH, 06_5DH	See Table 2-7
06_5CH, 06_7AH	See Table 2-12
06_1AH, 06_1EH, 06_1FH, 06_2EH	See Table 2-14
06_2AH, 06_2DH	See Table 2-19
0FH	See Table 2-44
MSR_LASTBRANCH_2_TO_IP	
06_0FH, 06_17H	See Table 2-3
06_1CH, 06_26H, 06_27H, 06_35H, 06_36H	See Table 2-4
06_37H, 06_4AH, 06_4DH, 06_5AH, 06_5DH	See Table 2-7
06_5CH, 06_7AH	See Table 2-12
06_1AH, 06_1EH, 06_1FH, 06_2EH	See Table 2-14
06_2AH, 06_2DH	See Table 2-19
0FH	See Table 2-44
MSR_LASTBRANCH_20_FROM_IP	
06_5CH, 06_7AH	See Table 2-12
06_4EH, 06_5EH, 06_55H, 06_8EH, 06_9EH, 06_66H	See Table 2-38
MSR_LASTBRANCH_20_TO_IP	
06_5CH, 06_7AH	See Table 2-12
06_4EH, 06_5EH, 06_55H, 06_8EH, 06_9EH, 06_66H	See Table 2-38
MSR_LASTBRANCH_21_FROM_IP	
06_5CH, 06_7AH	See Table 2-12
06_4EH, 06_5EH, 06_55H, 06_8EH, 06_9EH, 06_66H	See Table 2-38
MSR_LASTBRANCH_21_TO_IP	
06_5CH, 06_7AH	See Table 2-12
06_4EH, 06_5EH, 06_55H, 06_8EH, 06_9EH, 06_66H	See Table 2-38
MSR_LASTBRANCH_22_FROM_IP	

MSR Name and CPUID DisplayFamily_DisplayModel	Location
06_5CH, 06_7AH	See Table 2-12
06_4EH, 06_5EH, 06_55H, 06_8EH, 06_9EH, 06_66H	See Table 2-38
MSR_LASTBRANCH_22_TO_IP	
06_5CH, 06_7AH	See Table 2-12
06_4EH, 06_5EH, 06_55H, 06_8EH, 06_9EH, 06_66H	See Table 2-38
MSR_LASTBRANCH_23_FROM_IP	
06_5CH, 06_7AH	See Table 2-12
06_4EH, 06_5EH, 06_55H, 06_8EH, 06_9EH, 06_66H	See Table 2-38
MSR_LASTBRANCH_23_TO_IP	
06_5CH, 06_7AH	See Table 2-12
06_4EH, 06_5EH, 06_55H, 06_8EH, 06_9EH, 06_66H	See Table 2-38
MSR_LASTBRANCH_24_FROM_IP	
06_5CH, 06_7AH	See Table 2-12
06_4EH, 06_5EH, 06_55H, 06_8EH, 06_9EH, 06_66H	See Table 2-38
MSR_LASTBRANCH_24_TO_IP	
06_5CH, 06_7AH	See Table 2-12
06_4EH, 06_5EH, 06_55H, 06_8EH, 06_9EH, 06_66H	See Table 2-38
MSR_LASTBRANCH_25_FROM_IP	
06_5CH, 06_7AH	See Table 2-12
06_4EH, 06_5EH, 06_55H, 06_8EH, 06_9EH, 06_66H	See Table 2-38
MSR_LASTBRANCH_25_TO_IP	
06_5CH, 06_7AH	See Table 2-12
06_4EH, 06_5EH, 06_55H, 06_8EH, 06_9EH, 06_66H	See Table 2-38
MSR_LASTBRANCH_26_FROM_IP	
06_5CH, 06_7AH	See Table 2-12
06_4EH, 06_5EH, 06_55H, 06_8EH, 06_9EH, 06_66H	See Table 2-38
MSR_LASTBRANCH_26_TO_IP	
06_5CH, 06_7AH	See Table 2-12
06_4EH, 06_5EH, 06_55H, 06_8EH, 06_9EH, 06_66H	See Table 2-38
MSR_LASTBRANCH_27_FROM_IP	
06_5CH, 06_7AH	See Table 2-12
06_4EH, 06_5EH, 06_55H, 06_8EH, 06_9EH, 06_66H	See Table 2-38
MSR_LASTBRANCH_27_TO_IP	
06_5CH, 06_7AH	See Table 2-12
06_4EH, 06_5EH, 06_55H, 06_8EH, 06_9EH, 06_66H	See Table 2-38
MSR_LASTBRANCH_28_FROM_IP	
06_5CH, 06_7AH	See Table 2-12
06_4EH, 06_5EH, 06_55H, 06_8EH, 06_9EH, 06_66H	See Table 2-38
MSR_LASTBRANCH_28_TO_IP	
06_5CH, 06_7AH	See Table 2-12
06_4EH, 06_5EH, 06_55H, 06_8EH, 06_9EH, 06_66H	See Table 2-38
MSR_LASTBRANCH_29_FROM_IP	

MSR Name and CPUID DisplayFamily_DisplayModel	Location
06_5CH, 06_7AH	See Table 2-12
06_4EH, 06_5EH, 06_55H, 06_8EH, 06_9EH, 06_66H	See Table 2-38
MSR_LASTBRANCH_29_TO_IP	
06_5CH, 06_7AH	See Table 2-12
06_4EH, 06_5EH, 06_55H, 06_8EH, 06_9EH, 06_66H	See Table 2-38
MSR_LASTBRANCH_3	
0FH	See Table 2-44
06_0EH	See Table 2-47
06_09H	See Table 2-48
MSR_LASTBRANCH_3_FROM_IP	
06_0FH, 06_17H	See Table 2-3
06_1CH, 06_26H, 06_27H, 06_35H, 06_36H	See Table 2-4
06_37H, 06_4AH, 06_4DH, 06_5AH, 06_5DH	See Table 2-7
06_5CH, 06_7AH	See Table 2-12
06_1AH, 06_1EH, 06_1FH, 06_2EH	See Table 2-14
06_2AH, 06_2DH	See Table 2-19
0FH	See Table 2-44
MSR_LASTBRANCH_3_TO_IP	
06_0FH, 06_17H	See Table 2-3
06_1CH, 06_26H, 06_27H, 06_35H, 06_36H	See Table 2-4
06_37H, 06_4AH, 06_4DH, 06_5AH, 06_5DH	See Table 2-7
06_5CH, 06_7AH	See Table 2-12
06_1AH, 06_1EH, 06_1FH, 06_2EH	See Table 2-14
06_2AH, 06_2DH	See Table 2-19
0FH	See Table 2-44
MSR_LASTBRANCH_30_FROM_IP	
06_5CH, 06_7AH	See Table 2-12
06_4EH, 06_5EH, 06_55H, 06_8EH, 06_9EH, 06_66H	See Table 2-38
MSR_LASTBRANCH_30_TO_IP	
06_5CH, 06_7AH	See Table 2-12
06_4EH, 06_5EH, 06_55H, 06_8EH, 06_9EH, 06_66H	See Table 2-38
MSR_LASTBRANCH_31_FROM_IP	
06_5CH, 06_7AH	See Table 2-12
06_4EH, 06_5EH, 06_55H, 06_8EH, 06_9EH, 06_66H	See Table 2-38
MSR_LASTBRANCH_31_TO_IP	
06_5CH, 06_7AH	See Table 2-12
06_4EH, 06_5EH, 06_55H, 06_8EH, 06_9EH, 06_66H	See Table 2-38
MSR_LASTBRANCH_4	
06_0EH	See Table 2-47
06_09H	See Table 2-48
MSR_LASTBRANCH_4_FROM_IP	
06_1CH, 06_26H, 06_27H, 06_35H, 06_36H	See Table 2-4

MSR Name and CPUID DisplayFamily_DisplayModel	Location
06_37H, 06_4AH, 06_4DH, 06_5AH, 06_5DH	See Table 2-7
06_5CH, 06_7AH	See Table 2-12
06_1AH, 06_1EH, 06_1FH, 06_2EH	See Table 2-14
06_2AH, 06_2DH	See Table 2-19
0FH	See Table 2-44
MSR_LASTBRANCH_4_TO_IP	
06_1CH, 06_26H, 06_27H, 06_35H, 06_36H	See Table 2-4
06_37H, 06_4AH, 06_4DH, 06_5AH, 06_5DH	See Table 2-7
06_5CH, 06_7AH	See Table 2-12
06_1AH, 06_1EH, 06_1FH, 06_2EH	See Table 2-14
06_2AH, 06_2DH	See Table 2-19
0FH	See Table 2-44
MSR_LASTBRANCH_5	
06_0EH	See Table 2-47
06_09H	See Table 2-48
MSR_LASTBRANCH_5_FROM_IP	
06_1CH, 06_26H, 06_27H, 06_35H, 06_36H	See Table 2-4
06_37H, 06_4AH, 06_4DH, 06_5AH, 06_5DH	See Table 2-7
06_5CH, 06_7AH	See Table 2-12
06_1AH, 06_1EH, 06_1FH, 06_2EH	See Table 2-14
06_2AH, 06_2DH	See Table 2-19
0FH	See Table 2-44
MSR_LASTBRANCH_5_TO_IP	
06_1CH, 06_26H, 06_27H, 06_35H, 06_36H	See Table 2-4
06_37H, 06_4AH, 06_4DH, 06_5AH, 06_5DH	See Table 2-7
06_5CH, 06_7AH	See Table 2-12
06_1AH, 06_1EH, 06_1FH, 06_2EH	See Table 2-14
06_2AH, 06_2DH	See Table 2-19
0FH	See Table 2-44
MSR_LASTBRANCH_6	
06_0EH	See Table 2-47
06_09H	See Table 2-48
MSR_LASTBRANCH_6_FROM_IP	
06_1CH, 06_26H, 06_27H, 06_35H, 06_36H	See Table 2-4
06_37H, 06_4AH, 06_4DH, 06_5AH, 06_5DH	See Table 2-7
06_5CH, 06_7AH	See Table 2-12
06_1AH, 06_1EH, 06_1FH, 06_2EH	See Table 2-14
06_2AH, 06_2DH	See Table 2-19
0FH	See Table 2-44
MSR_LASTBRANCH_6_TO_IP	
06_1CH, 06_26H, 06_27H, 06_35H, 06_36H	See Table 2-4
06_37H, 06_4AH, 06_4DH, 06_5AH, 06_5DH	See Table 2-7

MSR Name and CPUID DisplayFamily_DisplayModel	Location
06_5CH, 06_7AH	See Table 2-12
06_1AH, 06_1EH, 06_1FH, 06_2EH	See Table 2-14
06_2AH, 06_2DH	See Table 2-19
0FH	See Table 2-44
MSR_LASTBRANCH_7	
06_0EH	See Table 2-47
06_09H	See Table 2-48
MSR_LASTBRANCH_7_FROM_IP	
06_1CH, 06_26H, 06_27H, 06_35H, 06_36H	See Table 2-4
06_37H, 06_4AH, 06_4DH, 06_5AH, 06_5DH	See Table 2-7
06_5CH, 06_7AH	See Table 2-12
06_1AH, 06_1EH, 06_1FH, 06_2EH	See Table 2-14
06_2AH, 06_2DH	See Table 2-19
0FH	See Table 2-44
MSR_LASTBRANCH_7_TO_IP	
06_1CH, 06_26H, 06_27H, 06_35H, 06_36H	See Table 2-4
06_37H, 06_4AH, 06_4DH, 06_5AH, 06_5DH	See Table 2-7
06_5CH, 06_7AH	See Table 2-12
06_1AH, 06_1EH, 06_1FH, 06_2EH	See Table 2-14
06_2AH, 06_2DH	See Table 2-19
0FH	See Table 2-44
MSR_LASTBRANCH_8_FROM_IP	
06_5CH, 06_7AH	See Table 2-12
06_1AH, 06_1EH, 06_1FH, 06_2EH	See Table 2-14
06_2AH, 06_2DH	See Table 2-19
0FH	See Table 2-44
MSR_LASTBRANCH_8_TO_IP	
06_5CH, 06_7AH	See Table 2-12
06_1AH, 06_1EH, 06_1FH, 06_2EH	See Table 2-14
06_2AH, 06_2DH	See Table 2-19
0FH	See Table 2-44
MSR_LASTBRANCH_9_FROM_IP	
06_5CH, 06_7AH	See Table 2-12
06_1AH, 06_1EH, 06_1FH, 06_2EH	See Table 2-14
06_2AH, 06_2DH	See Table 2-19
0FH	See Table 2-44
MSR_LASTBRANCH_9_TO_IP	
06_5CH, 06_7AH	See Table 2-12
06_1AH, 06_1EH, 06_1FH, 06_2EH	See Table 2-14
06_2AH, 06_2DH	See Table 2-19
0FH	See Table 2-44
MSR_LASTBRANCH_TOS	

MSR Name and CPUID DisplayFamily_DisplayModel	Location
06_0FH, 06_17H	See Table 2-3
06_1CH, 06_26H, 06_27H, 06_35H, 06_36H	See Table 2-4
06_37H, 06_4AH, 06_4DH, 06_5AH, 06_5DH	See Table 2-7
06_5CH, 06_7AH	See Table 2-12
06_1AH, 06_1EH, 06_1FH, 06_2EH	See Table 2-14
06_2AH, 06_2DH	See Table 2-19
06_4EH, 06_5EH, 06_55H, 06_8EH, 06_9EH, 06_66H	See Table 2-38
06_57H	See Table 2-42
06_0EH	See Table 2-47
06_09H	See Table 2-48
MSR_LASTBRANCH_INFO_0	
06_7AH	See Table 2-13
MSR_LBR_INFO_1	
06_4EH, 06_5EH, 06_55H, 06_8EH, 06_9EH, 06_66H	See Table 2-38
06_7AH	See Table 2-13
MSR_LBR_INFO_10	
06_4EH, 06_5EH, 06_55H, 06_8EH, 06_9EH, 06_66H	See Table 2-38
06_7AH	See Table 2-13
MSR_LBR_INFO_11	
06_4EH, 06_5EH, 06_55H, 06_8EH, 06_9EH, 06_66H	See Table 2-38
06_7AH	See Table 2-13
MSR_LBR_INFO_12	
06_4EH, 06_5EH, 06_55H, 06_8EH, 06_9EH, 06_66H	See Table 2-38
06_7AH	See Table 2-13
MSR_LBR_INFO_13	
06_4EH, 06_5EH, 06_55H, 06_8EH, 06_9EH, 06_66H	See Table 2-38
06_7AH	See Table 2-13
MSR_LBR_INFO_14	
06_4EH, 06_5EH, 06_55H, 06_8EH, 06_9EH, 06_66H	See Table 2-38
06_7AH	See Table 2-13
MSR_LBR_INFO_15	
06_4EH, 06_5EH, 06_55H, 06_8EH, 06_9EH, 06_66H	See Table 2-38
06_7AH	See Table 2-13
MSR_LBR_INFO_16	
06_4EH, 06_5EH, 06_55H, 06_8EH, 06_9EH, 06_66H	See Table 2-38
06_7AH	See Table 2-13
MSR_LBR_INFO_17	
06_4EH, 06_5EH, 06_55H, 06_8EH, 06_9EH, 06_66H	See Table 2-38
06_7AH	See Table 2-13
MSR_LBR_INFO_18	
06_4EH, 06_5EH, 06_55H, 06_8EH, 06_9EH, 06_66H	See Table 2-38
06_7AH	See Table 2-13

MSR Name and CPUID DisplayFamily_DisplayModel	Location
MSR_LBR_INFO_19	
06_4EH, 06_5EH, 06_55H, 06_8EH, 06_9EH, 06_66H	See Table 2-38
06_7AH	See Table 2-13
MSR_LBR_INFO_2	
06_4EH, 06_5EH, 06_55H, 06_8EH, 06_9EH, 06_66H	See Table 2-38
06_7AH	See Table 2-13
MSR_LBR_INFO_20	
06_4EH, 06_5EH, 06_55H, 06_8EH, 06_9EH, 06_66H	See Table 2-38
06_7AH	See Table 2-13
MSR_LBR_INFO_21	
06_4EH, 06_5EH, 06_55H, 06_8EH, 06_9EH, 06_66H	See Table 2-38
06_7AH	See Table 2-13
MSR_LBR_INFO_22	
06_4EH, 06_5EH, 06_55H, 06_8EH, 06_9EH, 06_66H	See Table 2-38
06_7AH	See Table 2-13
MSR_LBR_INFO_23	
06_4EH, 06_5EH, 06_55H, 06_8EH, 06_9EH, 06_66H	See Table 2-38
06_7AH	See Table 2-13
MSR_LBR_INFO_24	
06_4EH, 06_5EH, 06_55H, 06_8EH, 06_9EH, 06_66H	See Table 2-38
06_7AH	See Table 2-13
MSR_LBR_INFO_25	
06_4EH, 06_5EH, 06_55H, 06_8EH, 06_9EH, 06_66H	See Table 2-38
06_7AH	See Table 2-13
MSR_LBR_INFO_26	
06_4EH, 06_5EH, 06_55H, 06_8EH, 06_9EH, 06_66H	See Table 2-38
06_7AH	See Table 2-13
MSR_LBR_INFO_27	
06_4EH, 06_5EH, 06_55H, 06_8EH, 06_9EH, 06_66H	See Table 2-38
06_7AH	See Table 2-13
MSR_LBR_INFO_28	
06_4EH, 06_5EH, 06_55H, 06_8EH, 06_9EH, 06_66H	See Table 2-38
06_7AH	See Table 2-13
MSR_LBR_INFO_29	
06_4EH, 06_5EH, 06_55H, 06_8EH, 06_9EH, 06_66H	See Table 2-38
06_7AH	See Table 2-13
MSR_LBR_INFO_3	
06_4EH, 06_5EH, 06_55H, 06_8EH, 06_9EH, 06_66H	See Table 2-38
06_7AH	See Table 2-13
MSR_LBR_INFO_30	
06_4EH, 06_5EH, 06_55H, 06_8EH, 06_9EH, 06_66H	See Table 2-38
06_7AH	See Table 2-13

MSR Name and CPUID DisplayFamily_DisplayModel	Location
MSR_LBR_INFO_31	
06_4EH, 06_5EH, 06_55H, 06_8EH, 06_9EH, 06_66H	See Table 2-38
06_7AH	See Table 2-13
MSR_LBR_INFO_4	
06_4EH, 06_5EH, 06_55H, 06_8EH, 06_9EH, 06_66H	See Table 2-38
06_7AH	See Table 2-13
MSR_LBR_INFO_5	
06_4EH, 06_5EH, 06_55H, 06_8EH, 06_9EH, 06_66H	See Table 2-38
06_7AH	See Table 2-13
MSR_LBR_INFO_6	
06_4EH, 06_5EH, 06_55H, 06_8EH, 06_9EH, 06_66H	See Table 2-38
06_7AH	See Table 2-13
MSR_LBR_INFO_7	
06_4EH, 06_5EH, 06_55H, 06_8EH, 06_9EH, 06_66H	See Table 2-38
06_7AH	See Table 2-13
MSR_LBR_INFO_8	
06_4EH, 06_5EH, 06_55H, 06_8EH, 06_9EH, 06_66H	See Table 2-38
06_7AH	See Table 2-13
MSR_LBR_INFO_9	
06_4EH, 06_5EH, 06_55H, 06_8EH, 06_9EH, 06_66H	See Table 2-38
06_7AH	See Table 2-13
MSR_LBR_SELECT	
06_37H, 06_4AH, 06_4DH, 06_5AH, 06_5DH	See Table 2-7
06_5CH, 06_7AH	See Table 2-12
06_1AH, 06_1EH, 06_1FH, 06_2EH	See Table 2-14
06_2AH, 06_2DH	See Table 2-19
06_3CH, 06_45H, 06_46H	See Table 2-28
06_57H	See Table 2-42
MSR_LER_FROM_LIP	
06_0FH, 06_17H	See Table 2-3
06_1CH, 06_26H, 06_27H, 06_35H, 06_36H	See Table 2-4
06_37H, 06_4AH, 06_4DH, 06_5AH, 06_5DH, 06_5CH, 06_7AH	See Table 2-6
06_1AH, 06_1EH, 06_1FH, 06_2EH	See Table 2-14
06_2AH, 06_2DH	See Table 2-19
06_57H	See Table 2-42
0FH	See Table 2-44
06_0EH	See Table 2-47
06_09H	See Table 2-48
MSR_LER_TO_LIP	
06_0FH, 06_17H	See Table 2-3
06_1CH, 06_26H, 06_27H, 06_35H, 06_36H	See Table 2-4
06_37H, 06_4AH, 06_4DH, 06_5AH, 06_5DH, 06_5CH, 06_7AH	See Table 2-6

MSR Name and CPUID DisplayFamily_DisplayModel	Location
06_1AH, 06_1EH, 06_1FH, 06_2EH	See Table 2-14
06_2AH, 06_2DH	See Table 2-19
06_57H	See Table 2-42
0FH	See Table 2-44
06_0EH	See Table 2-47
06_09H	See Table 2-48
MSR_M0_PMON_ADDR_MASK	
06_2EH	See Table 2-16
MSR_M0_PMON_ADDR_MATCH	
06_2EH	See Table 2-16
MSR_M0_PMON_BOX_CTRL	
06_2EH	See Table 2-16
MSR_M0_PMON_BOX_OVF_CTRL	
06_2EH	See Table 2-16
MSR_M0_PMON_BOX_STATUS	
06_2EH	See Table 2-16
MSR_M0_PMON_CTR0	
06_2EH	See Table 2-16
MSR_M0_PMON_CTR1	
06_2EH	See Table 2-16
MSR_M0_PMON_CTR2	
06_2EH	See Table 2-16
MSR_M0_PMON_CTR3	
06_2EH	See Table 2-16
MSR_M0_PMON_CTR4	
06_2EH	See Table 2-16
MSR_M0_PMON_CTR5	
06_2EH	See Table 2-16
MSR_M0_PMON_DSP	
06_2EH	See Table 2-16
MSR_M0_PMON_EVNT_SEL0	
06_2EH	See Table 2-16
MSR_M0_PMON_EVNT_SEL1	
06_2EH	See Table 2-16
MSR_M0_PMON_EVNT_SEL2	
06_2EH	See Table 2-16
MSR_M0_PMON_EVNT_SEL3	
06_2EH	See Table 2-16
MSR_M0_PMON_EVNT_SEL4	
06_2EH	See Table 2-16
MSR_M0_PMON_EVNT_SEL5	
06_2EH	See Table 2-16

MSR Name and CPUID DisplayFamily_DisplayModel	Location
MSR_M0_PMON_ISS	
06_2EH	See Table 2-16
MSR_M0_PMON_MAP	
06_2EH	See Table 2-16
MSR_M0_PMON_MM_CONFIG	
06_2EH	See Table 2-16
MSR_M0_PMON_MSC_THR	
06_2EH	See Table 2-16
MSR_M0_PMON_PGT	
06_2EH	See Table 2-16
MSR_M0_PMON_PLD	
06_2EH	See Table 2-16
MSR_M0_PMON_TIMESTAMP	
06_2EH	See Table 2-16
MSR_M0_PMON_ZDP	
06_2EH	See Table 2-16
MSR_M1_PMON_ADDR_MASK	
06_2EH	See Table 2-16
MSR_M1_PMON_ADDR_MATCH	
06_2EH	See Table 2-16
MSR_M1_PMON_BOX_CTRL	
06_2EH	See Table 2-16
MSR_M1_PMON_BOX_OVF_CTRL	
06_2EH	See Table 2-16
MSR_M1_PMON_BOX_STATUS	
06_2EH	See Table 2-16
MSR_M1_PMON_CTR0	
06_2EH	See Table 2-16
MSR_M1_PMON_CTR1	
06_2EH	See Table 2-16
MSR_M1_PMON_CTR2	
06_2EH	See Table 2-16
MSR_M1_PMON_CTR3	
06_2EH	See Table 2-16
MSR_M1_PMON_CTR4	
06_2EH	See Table 2-16
MSR_M1_PMON_CTR5	
06_2EH	See Table 2-16
MSR_M1_PMON_DSP	
06_2EH	See Table 2-16
MSR_M1_PMON_EVNT_SEL0	
06_2EH	See Table 2-16

MSR Name and CPUID DisplayFamily_DisplayModel	Location
MSR_M1_PMON_EVNT_SEL1	
06_2EH	See Table 2-16
MSR_M1_PMON_EVNT_SEL2	
06_2EH	See Table 2-16
MSR_M1_PMON_EVNT_SEL3	
06_2EH	See Table 2-16
MSR_M1_PMON_EVNT_SEL4	
06_2EH	See Table 2-16
MSR_M1_PMON_EVNT_SEL5	
06_2EH	See Table 2-16
MSR_M1_PMON_ISS	
06_2EH	See Table 2-16
MSR_M1_PMON_MAP	
06_2EH	See Table 2-16
MSR_M1_PMON_MM_CONFIG	
06_2EH	See Table 2-16
MSR_M1_PMON_MSC_THR	
06_2EH	See Table 2-16
MSR_M1_PMON_PGT	
06_2EH	See Table 2-16
MSR_M1_PMON_PLD	
06_2EH	See Table 2-16
MSR_M1_PMON_TIMESTAMP	
06_2EH	See Table 2-16
MSR_M1_PMON_ZDP	
06_2EH	See Table 2-16
IA32_MC0_MISC / MSR_MC0_MISC	
06_1AH, 06_1EH, 06_1FH, 06_2EH	See Table 2-14
MSR_MC0_RESIDENCY	
06_57H	See Table 2-42
IA32_MC1_MISC / MSR_MC1_MISC	
06_1AH, 06_1EH, 06_1FH, 06_2EH	See Table 2-14
IA32_MC10_ADDR / MSR_MC10_ADDR	
06_2EH	See Table 2-16
06_2DH	See Table 2-22
06_3EH	See Table 2-25
06_3F	See Table 2-31
06_56H, 06_4FH	See Table 2-36
06_4FH	See Table 2-37
IA32_MC10_CTL / MSR_MC10_CTL	
06_2EH	See Table 2-16
06_2DH	See Table 2-22

MSR Name and CPUID DisplayFamily_DisplayModel	Location
06_3EH	See Table 2-25
06_3F	See Table 2-31
06_56H, 06_4FH	See Table 2-36
06_4FH	See Table 2-37
IA32_MC10_MISC / MSR_MC10_MISC	
06_2EH	See Table 2-16
06_2DH	See Table 2-22
06_3EH	See Table 2-25
06_3F	See Table 2-31
06_56H, 06_4FH	See Table 2-36
06_4FH	See Table 2-37
IA32_MC10_STATUS / MSR_MC10_STATUS	
06_2EH	See Table 2-16
06_2DH	See Table 2-22
06_3EH	See Table 2-25
06_3F	See Table 2-31
06_56H, 06_4FH	See Table 2-36
06_4FH	See Table 2-37
IA32_MC11_ADDR / MSR_MC11_ADDR	
06_2EH	See Table 2-16
06_2DH	See Table 2-22
06_3EH	See Table 2-25
06_3F	See Table 2-31
06_4FH	See Table 2-37
IA32_MC11_CTL / MSR_MC11_CTL	
06_2EH	See Table 2-16
06_2DH	See Table 2-22
06_3EH	See Table 2-25
06_3F	See Table 2-31
06_4FH	See Table 2-37
IA32_MC11_MISC / MSR_MC11_MISC	
06_2EH	See Table 2-16
06_2DH	See Table 2-22
06_3EH	See Table 2-25
06_3F	See Table 2-31
06_4FH	See Table 2-37
IA32_MC11_STATUS / MSR_MC11_STATUS	
06_2EH	See Table 2-16
06_2DH	See Table 2-22
06_3EH	See Table 2-25
06_3F	See Table 2-31
06_4FH	See Table 2-37

MSR Name and CPUID DisplayFamily_DisplayModel	Location
IA32_MC12_ADDR / MSR_MC12_ADDR	
06_2EH	See Table 2-16
06_2DH	See Table 2-22
06_3EH	See Table 2-25
06_3F	See Table 2-31
06_4FH	See Table 2-37
IA32_MC12_CTL / MSR_MC12_CTL	
06_2EH	See Table 2-16
06_2DH	See Table 2-22
06_3EH	See Table 2-25
06_3F	See Table 2-31
06_4FH	See Table 2-37
IA32_MC12_MISC / MSR_MC12_MISC	
06_2EH	See Table 2-16
06_2DH	See Table 2-22
06_3EH	See Table 2-25
06_3F	See Table 2-31
06_4FH	See Table 2-37
IA32_MC12_STATUS / MSR_MC12_STATUS	
06_2EH	See Table 2-16
06_2DH	See Table 2-22
06_3EH	See Table 2-25
06_3F	See Table 2-31
06_4FH	See Table 2-37
IA32_MC13_ADDR / MSR_MC13_ADDR	
06_2EH	See Table 2-16
06_2DH	See Table 2-22
06_3EH	See Table 2-25
06_3F	See Table 2-31
06_4FH	See Table 2-37
IA32_MC13_CTL / MSR_MC13_CTL	
06_2EH	See Table 2-16
06_2DH	See Table 2-22
06_3EH	See Table 2-25
06_3F	See Table 2-31
06_4FH	See Table 2-37
IA32_MC13_MISC / MSR_MC13_MISC	
06_2EH	See Table 2-16
06_2DH	See Table 2-22
06_3EH	See Table 2-25
06_3F	See Table 2-31
06_4FH	See Table 2-37

MSR Name and CPUID DisplayFamily_DisplayModel	Location
IA32_MC13_STATUS / MSR_MC13_STATUS	
06_2EH	See Table 2-16
06_2DH	See Table 2-22
06_3EH	See Table 2-25
06_3F	See Table 2-31
06_4FH	See Table 2-37
IA32_MC14_ADDR / MSR_MC14_ADDR	
06_2EH	See Table 2-16
06_2DH	See Table 2-22
06_3EH	See Table 2-25
06_3F	See Table 2-31
06_4FH	See Table 2-37
IA32_MC14_CTL / MSR_MC14_CTL	
06_2EH	See Table 2-16
06_2DH	See Table 2-22
06_3EH	See Table 2-25
06_3F	See Table 2-31
06_4FH	See Table 2-37
IA32_MC14_MISC / MSR_MC14_MISC	
06_2EH	See Table 2-16
06_2DH	See Table 2-22
06_3EH	See Table 2-25
06_3F	See Table 2-31
06_4FH	See Table 2-37
IA32_MC14_STATUS / MSR_MC14_STATUS	
06_2EH	See Table 2-16
06_2DH	See Table 2-22
06_3EH	See Table 2-25
06_3F	See Table 2-31
06_4FH	See Table 2-37
IA32_MC15_ADDR / MSR_MC15_ADDR	
06_2EH	See Table 2-16
06_2DH	See Table 2-22
06_3EH	See Table 2-25
06_3F	See Table 2-31
06_4FH	See Table 2-37
IA32_MC15_CTL / MSR_MC15_CTL	
06_2EH	See Table 2-16
06_2DH	See Table 2-22
06_3EH	See Table 2-25
06_3F	See Table 2-31
06_4FH	See Table 2-37

MSR Name and CPUID DisplayFamily_DisplayModel	Location
IA32_MC15_MISC / MSR_MC15_MISC	
06_2EH	See Table 2-16
06_2DH	See Table 2-22
06_3EH	See Table 2-25
06_3F	See Table 2-31
06_4FH	See Table 2-37
IA32_MC15_STATUS / MSR_MC15_STATUS	
06_2EH	See Table 2-16
06_2DH	See Table 2-22
06_3EH	See Table 2-25
06_3F	See Table 2-31
06_4FH	See Table 2-37
IA32_MC16_ADDR / MSR_MC16_ADDR	
06_2EH	See Table 2-16
06_2DH	See Table 2-22
06_3EH	See Table 2-25
06_3F	See Table 2-31
06_4FH	See Table 2-37
IA32_MC16_CTL / MSR_MC16_CTL	
06_2EH	See Table 2-16
06_2DH	See Table 2-22
06_3EH	See Table 2-25
06_3F	See Table 2-31
06_4FH	See Table 2-37
IA32_MC16_MISC / MSR_MC16_MISC	
06_2EH	See Table 2-16
06_2DH	See Table 2-22
06_3EH	See Table 2-25
06_3F	See Table 2-31
06_4FH	See Table 2-37
IA32_MC16_STATUS / MSR_MC16_STATUS	
06_2EH	See Table 2-16
06_2DH	See Table 2-22
06_3EH	See Table 2-25
06_3F	See Table 2-31
06_4FH	See Table 2-37
IA32_MC17_ADDR / MSR_MC17_ADDR	
06_2EH	See Table 2-16
06_2DH	See Table 2-22
06_3EH	See Table 2-25
06_3F	See Table 2-31
06_56H, 06_4FH	See Table 2-36

MSR Name and CPUID DisplayFamily_DisplayModel	Location
06_4FH	See Table 2-37
IA32_MC17_CTL / MSR_MC17_CTL	
06_2EH	See Table 2-16
06_2DH	See Table 2-22
06_3EH	See Table 2-25
06_3F	See Table 2-31
06_56H, 06_4FH	See Table 2-36
06_4FH	See Table 2-37
IA32_MC17_MISC / MSR_MC17_MISC	
06_2EH	See Table 2-16
06_2DH	See Table 2-22
06_3EH	See Table 2-25
06_3F	See Table 2-31
06_56H, 06_4FH	See Table 2-36
06_4FH	See Table 2-37
IA32_MC17_STATUS / MSR_MC17_STATUS	
06_2EH	See Table 2-16
06_2DH	See Table 2-22
06_3EH	See Table 2-25
06_3F	See Table 2-31
06_56H, 06_4FH	See Table 2-36
06_4FH	See Table 2-37
IA32_MC18_ADDR / MSR_MC18_ADDR	
06_2EH	See Table 2-16
06_2DH	See Table 2-22
06_3EH	See Table 2-25
06_3F	See Table 2-31
06_56H, 06_4FH	See Table 2-36
06_4FH	See Table 2-37
IA32_MC18_CTL / MSR_MC18_CTL	
06_2EH	See Table 2-16
06_2DH	See Table 2-22
06_3EH	See Table 2-25
06_3F	See Table 2-31
06_56H, 06_4FH	See Table 2-36
06_4FH	See Table 2-37
IA32_MC18_MISC / MSR_MC18_MISC	
06_2EH	See Table 2-16
06_2DH	See Table 2-22
06_3EH	See Table 2-25
06_3F	See Table 2-31
06_56H, 06_4FH	See Table 2-36

MSR Name and CPUID DisplayFamily_DisplayModel	Location
06_4FH	See Table 2-37
IA32_MC18_STATUS / MSR_MC18_STATUS	
06_2EH	See Table 2-16
06_2DH	See Table 2-22
06_3EH	See Table 2-25
06_3F	See Table 2-31
06_56H, 06_4FH	See Table 2-36
06_4FH	See Table 2-37
IA32_MC19_ADDR / MSR_MC19_ADDR	
06_2EH	See Table 2-16
06_2DH	See Table 2-22
06_3EH	See Table 2-25
06_3F	See Table 2-31
06_56H, 06_4FH	See Table 2-36
06_4FH	See Table 2-37
IA32_MC19_CTL / MSR_MC19_CTL	
06_2EH	See Table 2-16
06_2DH	See Table 2-22
06_3EH	See Table 2-25
06_3F	See Table 2-31
06_56H, 06_4FH	See Table 2-36
06_4FH	See Table 2-37
IA32_MC19_MISC / MSR_MC19_MISC	
06_2EH	See Table 2-16
06_2DH	See Table 2-22
06_3EH	See Table 2-25
06_3F	See Table 2-31
06_56H, 06_4FH	See Table 2-36
06_4FH	See Table 2-37
IA32_MC19_STATUS / MSR_MC19_STATUS	
06_2EH	See Table 2-16
06_2DH	See Table 2-22
06_3EH	See Table 2-25
06_3F	See Table 2-31
06_56H, 06_4FH	See Table 2-36
06_4FH	See Table 2-37
IA32_MC2_MISC / MSR_MC2_MISC	
06_1AH, 06_1EH, 06_1FH, 06_2EH	See Table 2-14
IA32_MC20_ADDR / MSR_MC20_ADDR	
06_2EH	See Table 2-16
06_3EH	See Table 2-25
06_3F	See Table 2-31

MSR Name and CPUID DisplayFamily_DisplayModel	Location
06_4FH	See Table 2-37
IA32_MC20_CTL / MSR_MC20_CTL	
06_2EH	See Table 2-16
06_3EH	See Table 2-25
06_3F	See Table 2-31
06_4FH	See Table 2-37
IA32_MC20_MISC / MSR_MC20_MISC	
06_2EH	See Table 2-16
06_3EH	See Table 2-25
06_3F	See Table 2-31
06_4FH	See Table 2-37
IA32_MC20_STATUS / MSR_MC20_STATUS	
06_2EH	See Table 2-16
06_3EH	See Table 2-25
06_3F	See Table 2-31
06_4FH	See Table 2-37
IA32_MC21_ADDR / MSR_MC21_ADDR	
06_2EH	See Table 2-16
06_3EH	See Table 2-25
06_3F	See Table 2-31
06_4F	See Table 2-37
IA32_MC21_CTL / MSR_MC21_CTL	
06_2EH	See Table 2-16
06_3EH	See Table 2-25
06_3F	See Table 2-31
06_4F	See Table 2-37
IA32_MC21_MISC / MSR_MC21_MISC	
06_2EH	See Table 2-16
06_3EH	See Table 2-25
06_3F	See Table 2-31
06_4F	See Table 2-37
IA32_MC21_STATUS / MSR_MC21_STATUS	
06_2EH	See Table 2-16
06_3EH	See Table 2-25
06_3F	See Table 2-31
06_4F	See Table 2-37
IA32_MC22_ADDR / MSR_MC22_ADDR	
06_3EH	See Table 2-25
IA32_MC22_CTL / MSR_MC22_CTL	
06_3EH	See Table 2-25
IA32_MC22_MISC / MSR_MC22_MISC	
06_3EH	See Table 2-25

MSR Name and CPUID DisplayFamily_DisplayModel	Location
IA32_MC22_STATUS / MSR_MC22_STATUS	
06_3EH	See Table 2-25
IA32_MC23_ADDR / MSR_MC23_ADDR	
06_3EH	See Table 2-25
IA32_MC23_CTL / MSR_MC23_CTL	
06_3EH	See Table 2-25
IA32_MC23_MISC / MSR_MC23_MISC	
06_3EH	See Table 2-25
IA32_MC23_STATUS / MSR_MC23_STATUS	
06_3EH	See Table 2-25
IA32_MC24_ADDR / MSR_MC24_ADDR	
06_3EH	See Table 2-25
IA32_MC24_CTL / MSR_MC24_CTL	
06_3EH	See Table 2-25
IA32_MC24_MISC / MSR_MC24_MISC	
06_3EH	See Table 2-25
IA32_MC24_STATUS / MSR_MC24_STATUS	
06_3EH	See Table 2-25
IA32_MC25_ADDR / MSR_MC25_ADDR	
06_3EH	See Table 2-25
IA32_MC25_CTL / MSR_MC25_CTL	
06_3EH	See Table 2-25
IA32_MC25_MISC / MSR_MC25_MISC	
06_3EH	See Table 2-25
IA32_MC25_STATUS / MSR_MC25_STATUS	
06_3EH	See Table 2-25
IA32_MC26_ADDR / MSR_MC26_ADDR	
06_3EH	See Table 2-25
IA32_MC26_CTL / MSR_MC26_CTL	
06_3EH	See Table 2-25
IA32_MC26_MISC / MSR_MC26_MISC	
06_3EH	See Table 2-25
IA32_MC26_STATUS / MSR_MC26_STATUS	
06_3EH	See Table 2-25
IA32_MC27_ADDR / MSR_MC27_ADDR	
06_3EH	See Table 2-25
IA32_MC27_CTL / MSR_MC27_CTL	
06_3EH	See Table 2-25
IA32_MC27_MISC / MSR_MC27_MISC	
06_3EH	See Table 2-25
IA32_MC27_STATUS / MSR_MC27_STATUS	
06_3EH	See Table 2-25

MSR Name and CPUID DisplayFamily_DisplayModel	Location
IA32_MC28_ADDR / MSR_MC28_ADDR	
06_3EH	See Table 2-25
IA32_MC28_CTL / MSR_MC28_CTL	
06_3EH	See Table 2-25
IA32_MC28_MISC / MSR_MC28_MISC	
06_3EH	See Table 2-25
IA32_MC28_STATUS / MSR_MC28_STATUS	
06_3EH	See Table 2-25
IA32_MC29_ADDR / MSR_MC29_ADDR	
06_3EH	See Table 2-26
IA32_MC29_CTL / MSR_MC29_CTL	
06_3EH	See Table 2-26
IA32_MC29_MISC / MSR_MC29_MISC	
06_3EH	See Table 2-26
IA32_MC29_STATUS / MSR_MC29_STATUS	
06_3EH	See Table 2-26
IA32_MC3_ADDR / MSR_MC3_ADDR	
06_0FH, 06_17H	See Table 2-3
06_1CH, 06_26H, 06_27H, 06_35H, 06_36H	See Table 2-4
06_37H, 06_4AH, 06_4DH, 06_5AH, 06_5DH, 06_5CH, 06_7AH	See Table 2-6
06_1AH, 06_1EH, 06_1FH, 06_2EH	See Table 2-14
06_57H	See Table 2-42
06_0EH	See Table 2-47
06_09H	See Table 2-48
IA32_MC3_CTL / MSR_MC3_CTL	
06_0FH, 06_17H	See Table 2-3
06_1CH, 06_26H, 06_27H, 06_35H, 06_36H	See Table 2-4
06_37H, 06_4AH, 06_4DH, 06_5AH, 06_5DH, 06_5CH, 06_7AH	See Table 2-6
06_1AH, 06_1EH, 06_1FH, 06_2EH	See Table 2-14
06_57H	See Table 2-42
06_0EH	See Table 2-47
06_09H	See Table 2-48
IA32_MC3_MISC / MSR_MC3_MISC	
06_0FH, 06_17H	See Table 2-3
06_1AH, 06_1EH, 06_1FH, 06_2EH	See Table 2-14
06_0EH	See Table 2-47
IA32_MC3_STATUS / MSR_MC3_STATUS	
06_0FH, 06_17H	See Table 2-3
06_1CH, 06_26H, 06_27H, 06_35H, 06_36H	See Table 2-4
06_37H, 06_4AH, 06_4DH, 06_5AH, 06_5DH, 06_5CH, 06_7AH	See Table 2-6
06_1AH, 06_1EH, 06_1FH, 06_2EH	See Table 2-14
06_57H	See Table 2-42

MSR Name and CPUID DisplayFamily_DisplayModel	Location
06_0EH	See Table 2-47
06_09H	See Table 2-48
IA32_MC30_ADDR / MSR_MC30_ADDR	
06_3EH	See Table 2-26
IA32_MC30_CTL / MSR_MC30_CTL	
06_3EH	See Table 2-26
IA32_MC30_MISC / MSR_MC30_MISC	
06_3EH	See Table 2-26
IA32_MC30_STATUS / MSR_MC30_STATUS	
06_3EH	See Table 2-26
IA32_MC31_ADDR / MSR_MC31_ADDR	
06_3EH	See Table 2-26
IA32_MC31_CTL / MSR_MC31_CTL	
06_3EH	See Table 2-26
IA32_MC31_MISC / MSR_MC31_MISC	
06_3EH	See Table 2-26
IA32_MC31_STATUS / MSR_MC31_STATUS	
06_3EH	See Table 2-26
IA32_MC4_ADDR / MSR_MC4_ADDR	
06_0FH, 06_17H	See Table 2-3
06_1CH, 06_26H, 06_27H, 06_35H, 06_36H	See Table 2-4
06_37H, 06_4AH, 06_4DH, 06_5AH, 06_5DH, 06_5CH, 06_7AH	See Table 2-6
06_1AH, 06_1EH, 06_1FH, 06_2EH	See Table 2-14
06_57H	See Table 2-42
06_0EH	See Table 2-47
06_09H	See Table 2-48
IA32_MC4_CTL / MSR_MC4_CTL	
06_0FH, 06_17H	See Table 2-3
06_1CH, 06_26H, 06_27H, 06_35H, 06_36H	See Table 2-4
06_37H, 06_4AH, 06_4DH, 06_5AH, 06_5DH, 06_5CH, 06_7AH	See Table 2-6
06_1AH, 06_1EH, 06_1FH, 06_2EH	See Table 2-14
06_57H	See Table 2-42
06_0EH	See Table 2-47
06_09H	See Table 2-48
IA32_MC4_CTL2 / MSR_MC4_CTL2	
06_2AH, 06_2DH	See Table 2-19
IA32_MC4_STATUS / MSR_MC4_STATUS	
06_0FH, 06_17H	See Table 2-3
06_1CH, 06_26H, 06_27H, 06_35H, 06_36H	See Table 2-4
06_37H, 06_4AH, 06_4DH, 06_5AH, 06_5DH, 06_5CH, 06_7AH	See Table 2-6
06_1AH, 06_1EH, 06_1FH, 06_2EH	See Table 2-14
06_57H	See Table 2-42

MSR Name and CPUID DisplayFamily_DisplayModel	Location
06_0EH	See Table 2-47
06_09H	See Table 2-48
MSR_MC5_ADDR / MSR_MC5_ADDR	
06_0FH, 06_17H	See Table 2-3
06_37H, 06_4AH, 06_4DH, 06_5AH, 06_5DH, 06_5CH, 06_7AH	See Table 2-6
06_1AH, 06_1EH, 06_1FH, 06_2EH	See Table 2-14
06_2DH	See Table 2-22
06_3EH	See Table 2-25
06_3FH	See Table 2-31
06_4FH	See Table 2-37
06_57H	See Table 2-42
06_0EH	See Table 2-47
IA32_MC5_CTL / MSR_MC5_CTL	
06_0FH, 06_17H	See Table 2-3
06_37H, 06_4AH, 06_4DH, 06_5AH, 06_5DH, 06_5CH, 06_7AH	See Table 2-6
06_1AH, 06_1EH, 06_1FH, 06_2EH	See Table 2-14
06_2DH	See Table 2-22
06_3EH	See Table 2-25
06_3FH	See Table 2-31
06_4FH	See Table 2-37
06_57H	See Table 2-42
06_0EH	See Table 2-47
IA32_MC5_MISC / MSR_MC5_MISC	
06_0FH, 06_17H	See Table 2-3
06_1AH, 06_1EH, 06_1FH, 06_2EH	See Table 2-14
06_2DH	See Table 2-22
06_3EH	See Table 2-25
06_3FH	See Table 2-31
06_4FH	See Table 2-37
06_0EH	See Table 2-47
IA32_MC5_STATUS / MSR_MC5_STATUS	
06_0FH, 06_17H	See Table 2-3
06_37H, 06_4AH, 06_4DH, 06_5AH, 06_5DH, 06_5CH, 06_7AH	See Table 2-6
06_1AH, 06_1EH, 06_1FH, 06_2EH	See Table 2-14
06_2DH	See Table 2-22
06_3EH	See Table 2-25
06_3FH	See Table 2-31
06_4FH	See Table 2-37
06_57H	See Table 2-42
06_0EH	See Table 2-47
IA32_MC6_ADDR / MSR_MC6_ADDR	
06_1AH, 06_1EH, 06_1FH, 06_2EH	See Table 2-14

MSR Name and CPUID DisplayFamily_DisplayModel	Location
06_2DH	See Table 2-22
06_3EH	See Table 2-25
06_3F	See Table 2-31
06_56H, 06_4FH	See Table 2-36
06_4FH	See Table 2-37
IA32_MC6_CTL / MSR_MC6_CTL	
06_1AH, 06_1EH, 06_1FH, 06_2EH	See Table 2-14
06_2DH	See Table 2-22
06_3EH	See Table 2-25
06_3F	See Table 2-31
06_56H, 06_4FH	See Table 2-36
06_4FH	See Table 2-37
MSR_MC6_DEMOTION_POLICY_CONFIG	
06_37H	See Table 2-9
IA32_MC6_MISC / MSR_MC6_MISC	
06_1AH, 06_1EH, 06_1FH, 06_2EH	See Table 2-14
06_2DH	See Table 2-22
06_3EH	See Table 2-25
06_3F	See Table 2-31
06_56H, 06_4FH	See Table 2-36
06_4FH	See Table 2-37
MSR_MC6_RESIDENCY_COUNTER	
06_37H, 06_4AH, 06_4DH, 06_5AH, 06_5DH	See Table 2-7
06_37H	See Table 2-9
06_57H	See Table 2-42
IA32_MC6_STATUS / MSR_MC6_STATUS	
06_0FH, 06_17H	See Table 2-3
06_1AH, 06_1EH, 06_1FH, 06_2EH	See Table 2-14
06_2DH	See Table 2-22
06_3EH	See Table 2-25
06_3FH	See Table 2-31
06_56H, 06_4FH	See Table 2-36
06_4FH	See Table 2-37
IA32_MC7_ADDR / MSR_MC7_ADDR	
06_1AH, 06_1EH, 06_1FH, 06_2EH	See Table 2-14
06_2DH	See Table 2-22
06_3EH	See Table 2-25
06_3F	See Table 2-31
06_56H, 06_4FH	See Table 2-36
06_4FH	See Table 2-37
IA32_MC7_CTL / MSR_MC7_CTL	
06_1AH, 06_1EH, 06_1FH, 06_2EH	See Table 2-14

MSR Name and CPUID DisplayFamily_DisplayModel	Location
06_2DH	See Table 2-22
06_3EH	See Table 2-25
06_3F	See Table 2-31
06_56H, 06_4FH	See Table 2-36
06_4FH	See Table 2-37
IA32_MC7_MISC / MSR_MC7_MISC	
06_1AH, 06_1EH, 06_1FH, 06_2EH	See Table 2-14
06_2DH	See Table 2-22
06_3EH	See Table 2-25
06_3F	See Table 2-31
06_56H, 06_4FH	See Table 2-36
06_4FH	See Table 2-37
IA32_MC7_STATUS / MSR_MC7_STATUS	
06_1AH, 06_1EH, 06_1FH, 06_2EH	See Table 2-14
06_2DH	See Table 2-22
06_3EH	See Table 2-25
06_3F	See Table 2-31
06_56H, 06_4FH	See Table 2-36
06_4FH	See Table 2-37
IA32_MC8_ADDR / MSR_MC8_ADDR	
06_1AH, 06_1EH, 06_1FH, 06_2EH	See Table 2-14
06_2DH	See Table 2-22
06_3EH	See Table 2-25
06_3F	See Table 2-31
06_4FH	See Table 2-37
IA32_MC8_CTL / MSR_MC8_CTL	
06_1AH, 06_1EH, 06_1FH, 06_2EH	See Table 2-14
06_2DH	See Table 2-22
06_3EH	See Table 2-25
06_3F	See Table 2-31
06_4FH	See Table 2-37
IA32_MC8_MISC / MSR_MC8_MISC	
06_1AH, 06_1EH, 06_1FH, 06_2EH	See Table 2-14
06_2DH	See Table 2-22
06_3EH	See Table 2-25
06_3F	See Table 2-31
06_4FH	See Table 2-37
IA32_MC8_STATUS / MSR_MC8_STATUS	
06_1AH, 06_1EH, 06_1FH, 06_2EH	See Table 2-14
06_2DH	See Table 2-22
06_3EH	See Table 2-25
06_3F	See Table 2-31

MSR Name and CPUID DisplayFamily_DisplayModel	Location
06_4FH	See Table 2-37
IA32_MC9_ADDR / MSR_MC9_ADDR	
06_2EH	See Table 2-16
06_2DH	See Table 2-22
06_3EH	See Table 2-25
06_3F	See Table 2-31
06_56H, 06_4FH	See Table 2-36
06_4FH	See Table 2-37
IA32_MC9_CTL / MSR_MC9_CTL	
06_2EH	See Table 2-16
06_2DH	See Table 2-22
06_3EH	See Table 2-25
06_3F	See Table 2-31
06_56H, 06_4FH	See Table 2-36
06_4FH	See Table 2-37
IA32_MC9_MISC / MSR_MC9_MISC	
06_2EH	See Table 2-16
06_2DH	See Table 2-22
06_3EH	See Table 2-25
06_3F	See Table 2-31
06_56H, 06_4FH	See Table 2-36
06_4FH	See Table 2-37
IA32_MC9_STATUS / MSR_MC9_STATUS	
06_2EH	See Table 2-16
06_2DH	See Table 2-22
06_3EH	See Table 2-25
06_3F	See Table 2-31
06_56H, 06_4FH	See Table 2-36
06_4FH	See Table 2-37
MSR_MCG_MISC	
0FH	See Table 2-44
MSR_MCG_R10	
0FH	See Table 2-44
MSR_MCG_R11	
0FH	See Table 2-44
MSR_MCG_R12	
0FH	See Table 2-44
MSR_MCG_R13	
0FH	See Table 2-44
MSR_MCG_R14	
0FH	See Table 2-44
MSR_MCG_R15	

MSR Name and CPUID DisplayFamily_DisplayModel	Location
0FH	See Table 2-44
MSR_MCG_R8	
0FH	See Table 2-44
MSR_MCG_R9	
0FH	See Table 2-44
MSR_MCG_RAX	
0FH	See Table 2-44
MSR_MCG_RBP	
0FH	See Table 2-44
MSR_MCG_RBX	
0FH	See Table 2-44
MSR_MCG_RCX	
0FH	See Table 2-44
MSR_MCG_RDI	
0FH	See Table 2-44
MSR_MCG_RDX	
0FH	See Table 2-44
MSR_MCG_RESERVED1 - MSR_MCG_RESERVED5	
0FH	See Table 2-44
MSR_MCG_RFLAGS	
0FH	See Table 2-44
MSR_MCG_RIP	
0FH	See Table 2-44
MSR_MCG_RSI	
0FH	See Table 2-44
MSR_MCG_RSP	
0FH	See Table 2-44
MSR_MISC_FEATURE_CONTROL	
06_5CH, 06_7AH	See Table 2-12
06_1AH, 06_1EH, 06_1FH, 06_2EH	See Table 2-14
06_2AH, 06_2DH	See Table 2-19
MSR_MISC_PWR_MGMT	
06_5CH, 06_7AH	See Table 2-12
06_1AH, 06_1EH, 06_1FH, 06_2EH	See Table 2-14
06_2AH, 06_2DH	See Table 2-19
MSR_MOB_ESCR0	
0FH	See Table 2-44
MSR_MOB_ESCR1	
0FH	See Table 2-44
MSR_MS_CCCR0	
0FH	See Table 2-44
MSR_MS_CCCR1	

MSR Name and CPUID DisplayFamily_DisplayModel	Location
0FH	See Table 2-44
MSR_MS_CCCR2	
0FH	See Table 2-44
MSR_MS_CCCR3	
0FH	See Table 2-44
MSR_MS_COUNTER0	
0FH	See Table 2-44
MSR_MS_COUNTER1	
0FH	See Table 2-44
MSR_MS_COUNTER2	
0FH	See Table 2-44
MSR_MS_COUNTER3	
0FH	See Table 2-44
MSR_MS_ESCR0	
0FH	See Table 2-44
MSR_MS_ESCR1	
0FH	See Table 2-44
MSR_MTRRCAP	
06_4EH, 06_5EH, 06_55H, 06_8EH, 06_9EH, 06_66H	See Table 2-38
MSR_OFFCORE_RSP_0	
06_37H, 06_4AH, 06_4DH, 06_5AH, 06_5DH, 06_5CH, 06_7AH	See Table 2-6
06_1AH, 06_1EH, 06_1FH, 06_2EH	See Table 2-14
06_2AH, 06_2DH	See Table 2-19
06_57H	See Table 2-42
MSR_OFFCORE_RSP_1	
06_37H, 06_4AH, 06_4DH, 06_5AH, 06_5DH, 06_5CH, 06_7AH	See Table 2-6
06_25H, 06_2CH	See Table 2-17
06_2FH	See Table 2-18
06_2AH, 06_2DH	See Table 2-19
06_57H	See Table 2-42
MSR_PCIE_PLL_RATIO	
06_3FH	See Table 2-31
MSR_PCU_PMON_BOX_CTL	
06_2DH	See Table 2-23
06_3FH	See Table 2-32
MSR_PCU_PMON_BOX_FILTER	
06_2DH	See Table 2-23
06_3FH	See Table 2-32
MSR_PCU_PMON_BOX_STATUS	
06_3EH	See Table 2-27
06_3FH	See Table 2-32
MSR_PCU_PMON_CTR0	

MSR Name and CPUID DisplayFamily_DisplayModel	Location
06_2DH	See Table 2-23
06_3FH	See Table 2-32
MSR_PCU_PMON_CTR1	
06_2DH	See Table 2-23
06_3FH	See Table 2-32
MSR_PCU_PMON_CTR2	
06_2DH	See Table 2-23
06_3FH	See Table 2-32
MSR_PCU_PMON_CTR3	
06_2DH	See Table 2-23
06_3FH	See Table 2-32
MSR_PCU_PMON_EVNTSEL0	
06_2DH	See Table 2-23
06_3FH	See Table 2-32
MSR_PCU_PMON_EVNTSEL1	
06_2DH	See Table 2-23
06_3FH	See Table 2-32
MSR_PCU_PMON_EVNTSEL2	
06_2DH	See Table 2-23
06_3FH	See Table 2-32
MSR_PCU_PMON_EVNTSEL3	
06_2DH	See Table 2-23
06_3FH	See Table 2-32
MSR_PEBS_ENABLE	
06_0FH, 06_17H	See Table 2-3
06_1CH, 06_26H, 06_27H, 06_35H, 06_36H	See Table 2-4
06_37H, 06_4AH, 06_4DH, 06_5AH, 06_5DH	See Table 2-7
06_5CH	See Table 2-12
06_7AH	See Table 2-13
06_1AH, 06_1EH, 06_1FH, 06_2EH	See Table 2-14
06_2AH, 06_2DH	See Table 2-19
06_3EH	See Table 2-26
06_57H	See Table 2-42
0FH	See Table 2-44
MSR_PEBS_FRONTEND	
06_4EH, 06_5EH, 06_55H, 06_8EH, 06_9EH, 06_66H	See Table 2-38
MSR_PEBS_LD_LAT	
06_1AH, 06_1EH, 06_1FH, 06_2EH	See Table 2-14
06_2AH, 06_2DH	See Table 2-19
MSR_PEBS_MATRIX_VERT	
0FH	See Table 2-44
MSR_PEBS_NUM_ALT	

MSR Name and CPUID DisplayFamily_DisplayModel	Location
06_2DH	See Table 2-22
MSR_PERF_CAPABILITIES	
06_0FH, 06_17H	See Table 2-3
MSR_PERF_FIXED_CTR_CTRL	
06_0FH, 06_17H	See Table 2-3
MSR_PERF_FIXED_CTR0	
06_0FH, 06_17H	See Table 2-3
MSR_PERF_FIXED_CTR1	
06_0FH, 06_17H	See Table 2-3
MSR_PERF_FIXED_CTR2	
06_0FH, 06_17H	See Table 2-3
MSR_PERF_GLOBAL_CTRL	
06_0FH, 06_17H	See Table 2-3
MSR_PERF_GLOBAL_OVF_CTRL	
06_0FH, 06_17H	See Table 2-3
06_1AH, 06_1EH, 06_1FH, 06_2EH	See Table 2-14
MSR_PERF_GLOBAL_STATUS	
06_0FH, 06_17H	See Table 2-3
06_1AH, 06_1EH, 06_1FH, 06_2EH	See Table 2-14
MSR_PERF_STATUS	
06_0FH, 06_17H	See Table 2-3
06_1CH, 06_26H, 06_27H, 06_35H, 06_36H	See Table 2-4
06_2AH, 06_2DH	See Table 2-19
MSR_PKG_C10_RESIDENCY	
06_5CH, 06_7AH	See Table 2-12
06_45H	See Table 2-29 and Table 2-30
06_4FH	See Table 2-37
MSR_PKG_C2_RESIDENCY	
06_27H	See Table 2-5
06_5CH, 06_7AH	See Table 2-12
06_2AH, 06_2DH, 06_3AH, 06_3CH, 06_3EH, 06_3FH, 06_45H, 06_46H	See Table 2-19
06_57H	See Table 2-42
MSR_PKG_C3_RESIDENCY	
06_5CH, 06_7AH	See Table 2-12
06_1AH, 06_1EH, 06_1FH, 06_2EH, 06_25H, 06_2CH, 06_2FH	See Table 2-14
06_2AH, 06_2DH, 06_3AH, 06_3CH, 06_3EH, 06_3FH, 06_45H, 06_46H	See Table 2-19
06_66H	See Table 2-40
06_57H	See Table 2-42
MSR_PKG_C4_RESIDENCY	
06_27H	See Table 2-5
MSR_PKG_C6_RESIDENCY	

MSR Name and CPUID DisplayFamily_DisplayModel	Location
06_27H	See Table 2-5
06_37H, 06_4AH, 06_4DH, 06_5AH, 06_5DH	See Table 2-7
06_5CH, 06_7AH	See Table 2-12
06_1AH, 06_1EH, 06_1FH, 06_2EH, 06_25H, 06_2CH, 06_2FH	See Table 2-14
06_2AH, 06_2DH, 06_3AH, 06_3CH, 06_3EH, 06_3FH, 06_45H, 06_46H	See Table 2-19
06_57H	See Table 2-42
MSR_PKG_C7_RESIDENCY	
06_1AH, 06_1EH, 06_1FH, 06_2EH, 06_25H, 06_2CH, 06_2FH	See Table 2-14
06_2AH, 06_2DH, 06_3AH, 06_3CH, 06_3EH, 06_3FH, 06_45H, 06_46H	See Table 2-19
06_57H	See Table 2-42
MSR_PKG_C8_RESIDENCY	
06_45H	See Table 2-30
06_4FH	See Table 2-37
MSR_PKG_C9_RESIDENCY	
06_45H	See Table 2-30
06_4FH	See Table 2-37
MSR_PKG_CST_CONFIG_CONTROL	
06_37H, 06_4AH, 06_4DH, 06_5AH, 06_5DH	See Table 2-7
06_4CH	See Table 2-11
06_5CH, 06_7AH	See Table 2-12
06_1AH, 06_1EH, 06_1FH, 06_2EH	See Table 2-14
06_2AH, 06_2DH	See Table 2-19
06_3AH	See Table 2-24
06_3EH	See Table 2-25
06_3CH, 06_45H, 06_46H	See Table 2-29
06_45H	See Table 2-30
06_3F	See Table 2-31
06_3DH	See Table 2-34
06_56H, 06_4FH	See Table 2-35
06_57H	See Table 2-42
MSR_PKG_ENERGY_STATUS	
06_37H, 06_4AH, 06_5AH, 06_5DH	See Table 2-8
06_5CH, 06_7AH	See Table 2-12
06_2AH, 06_2DH, 06_3AH, 06_3CH, 06_3EH, 06_3FH, 06_45H, 06_46H	See Table 2-19
MSR_PKG_HDC_CONFIG	
06_4EH, 06_5EH, 06_55H, 06_8EH, 06_9EH, 06_66H	See Table 2-38
MSR_PKG_HDC_DEEP_RESIDENCY	
06_4EH, 06_5EH, 06_55H, 06_8EH, 06_9EH, 06_66H	See Table 2-38
MSR_PKG_HDC_SHALLOW_RESIDENCY	
06_4EH, 06_5EH, 06_55H, 06_8EH, 06_9EH, 06_66H	See Table 2-38
MSR_PKG_PERF_STATUS	
06_5CH, 06_7AH	See Table 2-12

MSR Name and CPUID DisplayFamily_DisplayModel	Location
06_2DH	See Table 2-22
06_3EH, 06_3FH	See Table 2-25
06_3CH, 06_45H, 06_46H	See Table 2-29
06_57H	See Table 2-42
MSR_PKG_POWER_INFO	
06_4DH	See Table 2-10
06_5CH, 06_7AH	See Table 2-12
06_2AH, 06_2DH, 06_3AH, 06_3CH, 06_3EH, 06_3FH, 06_45H, 06_46H	See Table 2-19
06_57H	See Table 2-42
MSR_PKG_POWER_LIMIT	
06_37H, 06_4AH, 06_5AH, 06_5DH	See Table 2-8
06_4DH	See Table 2-10
06_5CH, 06_7AH	See Table 2-12
06_2AH, 06_2DH, 06_3AH, 06_3CH, 06_3EH, 06_3FH, 06_45H, 06_46H	See Table 2-19
06_57H	See Table 2-42
MSR_PKGC_IRTL1	
06_5CH, 06_7AH	See Table 2-12
06_3CH, 06_45H, 06_46H.	See Table 2-28
MSR_PKGC_IRTL2	
06_5CH, 06_7AH	See Table 2-12
06_3CH, 06_45H, 06_46H.	See Table 2-28
MSR_PKGC3_IRTL	
06_5CH, 06_7AH	See Table 2-12
06_2AH, 06_2DH	See Table 2-19
MSR_PKGC6_IRTL	
06_2AH, 06_2DH	See Table 2-19
MSR_PKGC7_IRTL	
06_2AH	See Table 2-20
MSR_PLATFORM_BRV	
0FH	See Table 2-44
MSR_PLATFORM_ENERGY_COUNTER	
06_4EH, 06_5EH, 06_55H, 06_8EH, 06_9EH, 06_66H	See Table 2-38
MSR_PLATFORM_ID	
06_0FH, 06_17H	See Table 2-3
06_1CH, 06_26H, 06_27H, 06_35H, 06_36H	See Table 2-4
06_37H, 06_4AH, 06_4DH, 06_5AH, 06_5DH	See Table 2-7
06_5CH, 06_7AH	See Table 2-12
06_1AH, 06_1EH, 06_1FH, 06_2EH	See Table 2-14
MSR_PLATFORM_INFO	
06_5CH, 06_7AH	See Table 2-12
06_1AH, 06_1EH, 06_1FH, 06_2EH	See Table 2-14
06_2AH, 06_2DH	See Table 2-19

MSR Name and CPUID DisplayFamily_DisplayModel	Location
06_3AH	See Table 2-24
06_3EH	See Table 2-25
06_3CH, 06_45H, 06_46H	See Table 2-28 and Table 2-29
06_56H, 06_4FH	See Table 2-35
06_57H	See Table 2-42
MSR_PLATFORM_POWER_LIMIT	
06_4EH, 06_5EH, 06_55H, 06_8EH, 06_9EH, 06_66H	See Table 2-38
MSR_PMG_IO_CAPTURE_BASE	
06_37H, 06_4AH, 06_4DH, 06_5AH, 06_5DH, 06_5CH, 06_7AH	See Table 2-6
06_4CH	See Table 2-11
06_1AH, 06_1EH, 06_1FH, 06_2EH	See Table 2-14
06_2AH, 06_2DH	See Table 2-19
06_3AH	See Table 2-24
06_3EH	See Table 2-25
06_57H	See Table 2-42
MSR_PMH_ESCR0	
0FH	See Table 2-44
MSR_PMH_ESCR1	
0FH	See Table 2-44
MSR_PMON_GLOBAL_CONFIG	
06_3EH	See Table 2-27
06_3FH	See Table 2-32
MSR_PMON_GLOBAL_CTL	
06_3EH	See Table 2-27
06_3FH	See Table 2-32
MSR_PMON_GLOBAL_STATUS	
06_3EH	See Table 2-27
06_3FH	See Table 2-32
MSR_POWER_CTL	
06_5CH, 06_7AH	See Table 2-12
06_1AH, 06_1EH, 06_1FH, 06_2EH	See Table 2-14
06_2AH, 06_2DH	See Table 2-19
MSR_PP0_ENERGY_STATUS	
06_37H, 06_4AH, 06_5AH, 06_5DH	See Table 2-8
06_5CH, 06_7AH	See Table 2-12
06_2AH, 06_2DH, 06_3AH, 06_3CH, 06_3EH, 06_3FH, 06_45H, 06_46H	See Table 2-19
06_57H	See Table 2-42
MSR_PP0_POLICY	
06_2AH, 06_45H	See Table 2-20
MSR_PP0_POWER_LIMIT	
06_4CH	See Table 2-11

MSR Name and CPUID DisplayFamily_DisplayModel	Location
06_2AH, 06_2DH, 06_3AH, 06_3CH, 06_3EH, 06_3FH, 06_45H, 06_46H	See Table 2-19
06_57H	See Table 2-42
MSR_PP1_ENERGY_STATUS	
06_5CH, 06_7AH	See Table 2-12
06_2AH, 06_45H	See Table 2-20
06_3CH, 06_45H, 06_46H	See Table 2-29
MSR_PP1_POLICY	
06_2AH, 06_45H	See Table 2-20
06_3CH, 06_45H, 06_46H	See Table 2-29
MSR_PP1_POWER_LIMIT	
06_2AH, 06_45H	See Table 2-20
06_3CH, 06_45H, 06_46H	See Table 2-29
MSR_PPERF	
06_4EH, 06_5EH, 06_55H, 06_8EH, 06_9EH, 06_66H	See Table 2-38
MSR_PPIN	
06_3EH	See Table 2-25
06_56H, 06_4FH	See Table 2-35
MSR_PPIN_CTL	
06_3EH	See Table 2-25
06_56H, 06_4FH	See Table 2-35
MSR_R0_PMON_BOX_CTRL	
06_2EH	See Table 2-16
MSR_R0_PMON_BOX_OVF_CTRL	
06_2EH	See Table 2-16
MSR_R0_PMON_BOX_STATUS	
06_2EH	See Table 2-16
MSR_R0_PMON_CTR0	
06_2EH	See Table 2-16
MSR_R0_PMON_CTR1	
06_2EH	See Table 2-16
MSR_R0_PMON_CTR2	
06_2EH	See Table 2-16
MSR_R0_PMON_CTR3	
06_2EH	See Table 2-16
MSR_R0_PMON_CTR4	
06_2EH	See Table 2-16
MSR_R0_PMON_CTR5	
06_2EH	See Table 2-16
MSR_R0_PMON_CTR6	
06_2EH	See Table 2-16
MSR_R0_PMON_CTR7	
06_2EH	See Table 2-16

MSR Name and CPUID DisplayFamily_DisplayModel	Location
MSR_R0_PMON_EVNT_SEL0	
06_2EH	See Table 2-16
MSR_R0_PMON_EVNT_SEL1	
06_2EH	See Table 2-16
MSR_R0_PMON_EVNT_SEL2	
06_2EH	See Table 2-16
MSR_R0_PMON_EVNT_SEL3	
06_2EH	See Table 2-16
MSR_R0_PMON_EVNT_SEL4	
06_2EH	See Table 2-16
MSR_R0_PMON_EVNT_SEL5	
06_2EH	See Table 2-16
MSR_R0_PMON_EVNT_SEL6	
06_2EH	See Table 2-16
MSR_R0_PMON_EVNT_SEL7	
06_2EH	See Table 2-16
MSR_R0_PMON_IPERF0_P0	
06_2EH	See Table 2-16
MSR_R0_PMON_IPERF0_P1	
06_2EH	See Table 2-16
MSR_R0_PMON_IPERF0_P2	
06_2EH	See Table 2-16
MSR_R0_PMON_IPERF0_P3	
06_2EH	See Table 2-16
MSR_R0_PMON_IPERF0_P4	
06_2EH	See Table 2-16
MSR_R0_PMON_IPERF0_P5	
06_2EH	See Table 2-16
MSR_R0_PMON_IPERF0_P6	
06_2EH	See Table 2-16
MSR_R0_PMON_IPERF0_P7	
06_2EH	See Table 2-16
MSR_R0_PMON_QLX_P0	
06_2EH	See Table 2-16
MSR_R0_PMON_QLX_P1	
06_2EH	See Table 2-16
MSR_R0_PMON_QLX_P2	
06_2EH	See Table 2-16
MSR_R0_PMON_QLX_P3	
06_2EH	See Table 2-16
MSR_R1_PMON_BOX_CTRL	
06_2EH	See Table 2-16

MSR Name and CPUID DisplayFamily_DisplayModel	Location
MSR_R1_PMON_BOX_OVF_CTRL	
06_2EH	See Table 2-16
MSR_R1_PMON_BOX_STATUS	
06_2EH	See Table 2-16
MSR_R1_PMON_CTR10	
06_2EH	See Table 2-16
MSR_R1_PMON_CTR11	
06_2EH	See Table 2-16
MSR_R1_PMON_CTR12	
06_2EH	See Table 2-16
MSR_R1_PMON_CTR13	
06_2EH	See Table 2-16
MSR_R1_PMON_CTR14	
06_2EH	See Table 2-16
MSR_R1_PMON_CTR15	
06_2EH	See Table 2-16
MSR_R1_PMON_CTR8	
06_2EH	See Table 2-16
MSR_R1_PMON_CTR9	
06_2EH	See Table 2-16
MSR_R1_PMON_EVNT_SEL10	
06_2EH	See Table 2-16
MSR_R1_PMON_EVNT_SEL11	
06_2EH	See Table 2-16
MSR_R1_PMON_EVNT_SEL12	
06_2EH	See Table 2-16
MSR_R1_PMON_EVNT_SEL13	
06_2EH	See Table 2-16
MSR_R1_PMON_EVNT_SEL14	
06_2EH	See Table 2-16
MSR_R1_PMON_EVNT_SEL15	
06_2EH	See Table 2-16
MSR_R1_PMON_EVNT_SEL8	
06_2EH	See Table 2-16
MSR_R1_PMON_EVNT_SEL9	
06_2EH	See Table 2-16
MSR_R1_PMON_IPERF1_P10	
06_2EH	See Table 2-16
MSR_R1_PMON_IPERF1_P11	
06_2EH	See Table 2-16
MSR_R1_PMON_IPERF1_P12	
06_2EH	See Table 2-16

MSR Name and CPUID DisplayFamily_DisplayModel	Location
MSR_R1_PMON_IPERF1_P13	
06_2EH	See Table 2-16
MSR_R1_PMON_IPERF1_P14	
06_2EH	See Table 2-16
MSR_R1_PMON_IPERF1_P15	
06_2EH	See Table 2-16
MSR_R1_PMON_IPERF1_P8	
06_2EH	See Table 2-16
MSR_R1_PMON_IPERF1_P9	
06_2EH	See Table 2-16
MSR_R1_PMON_QLX_P4	
06_2EH	See Table 2-16
MSR_R1_PMON_QLX_P5	
06_2EH	See Table 2-16
MSR_R1_PMON_QLX_P6	
06_2EH	See Table 2-16
MSR_R1_PMON_QLX_P7	
06_2EH	See Table 2-16
MSR_RAPL_POWER_UNIT	
06_37H, 06_4AH, 06_5AH, 06_5DH	See Table 2-8
06_4DH	See Table 2-10
06_5CH, 06_7AH	See Table 2-12
06_2AH, 06_2DH, 06_3AH, 06_3CH, 06_3EH, 06_3FH, 06_45H, 06_46H	See Table 2-19
06_3FH	See Table 2-31
06_56H, 06_4FH	See Table 2-35
06_57H	See Table 2-42
MSR_RAT_ESCR0	
0FH	See Table 2-44
MSR_RAT_ESCR1	
0FH	See Table 2-44
MSR_RING_PERF_LIMIT_REASONS	
06_3CH, 06_45H, 06_46H	See Table 2-29
MSR_S0_PMON_BOX_CTRL	
06_2EH	See Table 2-16
06_3FH	See Table 2-32
MSR_S0_PMON_BOX_FILTER	
06_3FH	See Table 2-32
MSR_S0_PMON_BOX_OVF_CTRL	
06_2EH	See Table 2-16
MSR_S0_PMON_BOX_STATUS	
06_2EH	See Table 2-16
MSR_S0_PMON_CTR0	

MSR Name and CPUID DisplayFamily_DisplayModel	Location
06_2EH	See Table 2-16
06_3FH	See Table 2-32
MSR_S0_PMON_CTR1	
06_2EH	See Table 2-16
06_3FH	See Table 2-32
MSR_S0_PMON_CTR2	
06_2EH	See Table 2-16
06_3FH	See Table 2-32
MSR_S0_PMON_CTR3	
06_2EH	See Table 2-16
06_3FH	See Table 2-32
MSR_S0_PMON_EVNT_SEL0	
06_2EH	See Table 2-16
06_3FH	See Table 2-32
MSR_S0_PMON_EVNT_SEL1	
06_2EH	See Table 2-16
06_3FH	See Table 2-32
MSR_S0_PMON_EVNT_SEL2	
06_2EH	See Table 2-16
06_3FH	See Table 2-32
MSR_S0_PMON_EVNT_SEL3	
06_2EH	See Table 2-16
06_3FH	See Table 2-32
MSR_S0_PMON_MASK	
06_2EH	See Table 2-16
MSR_S0_PMON_MATCH	
06_2EH	See Table 2-16
MSR_S1_PMON_BOX_CTRL	
06_2EH	See Table 2-16
06_3FH	See Table 2-32
MSR_S1_PMON_BOX_FILTER	
06_3FH	See Table 2-32
MSR_S1_PMON_BOX_OVF_CTRL	
06_2EH	See Table 2-16
MSR_S1_PMON_BOX_STATUS	
06_2EH	See Table 2-16
MSR_S1_PMON_CTR0	
06_2EH	See Table 2-16
06_3FH	See Table 2-32
MSR_S1_PMON_CTR1	
06_2EH	See Table 2-16
06_3FH	See Table 2-32

MSR Name and CPUID DisplayFamily_DisplayModel	Location
MSR_S1_PMON_CTR2	
06_2EH	See Table 2-16
06_3FH	See Table 2-32
MSR_S1_PMON_CTR3	
06_2EH	See Table 2-16
06_3FH	See Table 2-32
MSR_S1_PMON_EVNT_SEL0	
06_2EH	See Table 2-16
06_3FH	See Table 2-32
MSR_S1_PMON_EVNT_SEL1	
06_2EH	See Table 2-16
06_3FH	See Table 2-32
MSR_S1_PMON_EVNT_SEL2	
06_2EH	See Table 2-16
06_3FH	See Table 2-32
MSR_S1_PMON_EVNT_SEL3	
06_2EH	See Table 2-16
06_3FH	See Table 2-32
MSR_S1_PMON_MASK	
06_2EH	See Table 2-16
MSR_S1_PMON_MATCH	
06_2EH	See Table 2-16
MSR_S2_PMON_BOX_CTL	
06_3FH	See Table 2-32
MSR_S2_PMON_BOX_FILTER	
06_3FH	See Table 2-32
MSR_S2_PMON_CTR0	
06_3FH	See Table 2-32
MSR_S2_PMON_CTR1	
06_3FH	See Table 2-32
MSR_S2_PMON_CTR2	
06_3FH	See Table 2-32
MSR_S2_PMON_CTR3	
06_3FH	See Table 2-32
MSR_S2_PMON_EVNTSEL0	
06_3FH	See Table 2-32
MSR_S2_PMON_EVNTSEL1	
06_3FH	See Table 2-32
MSR_S2_PMON_EVNTSEL2	
06_3FH	See Table 2-32
MSR_S2_PMON_EVNTSEL3	
06_3FH	See Table 2-32

MSR Name and CPUID DisplayFamily_DisplayModel	Location
MSR_S3_PMON_BOX_CTL	
06_3FH	See Table 2-32
MSR_S3_PMON_BOX_FILTER	
06_3FH	See Table 2-32
MSR_S3_PMON_CTR0	
06_3FH	See Table 2-32
MSR_S3_PMON_CTR1	
06_3FH	See Table 2-32
MSR_S3_PMON_CTR2	
06_3FH	See Table 2-32
MSR_S3_PMON_CTR3	
06_3FH	See Table 2-32
MSR_S3_PMON_EVNTSEL0	
06_3FH	See Table 2-32
MSR_S3_PMON_EVNTSEL1	
06_3FH	See Table 2-32
MSR_S3_PMON_EVNTSEL2	
06_3FH	See Table 2-32
MSR_S3_PMON_EVNTSEL3	
06_3FH	See Table 2-32
MSR_SAAT_ESCR0	
0FH	See Table 2-44
MSR_SAAT_ESCR1	
0FH	See Table 2-44
MSR_SGXOWNEREPOCH0	
06_5CH, 06_7AH	See Table 2-12
06_4EH, 06_5EH, 06_55H, 06_8EH, 06_9EH, 06_66H	See Table 2-38
MSR_SGXOWNEREPOCH1	
06_5CH, 06_7AH	See Table 2-12
06_4EH, 06_5EH, 06_55H, 06_8EH, 06_9EH, 06_66H	See Table 2-38
MSR_SMI_COUNT	
06_37H, 06_4AH, 06_4DH, 06_5AH, 06_5DH, 06_5CH, 06_7AH	See Table 2-6
06_1AH, 06_1EH, 06_1FH, 06_2EH	See Table 2-14
06_2AH, 06_2DH	See Table 2-19
06_57H	See Table 2-42
MSR_SMM_BLOCKED	
06_5CH, 06_7AH	See Table 2-12
06_3CH, 06_45H, 06_46H	See Table 2-29
MSR_SMM_DELAYED	
06_5CH, 06_7AH	See Table 2-12
06_3CH, 06_45H, 06_46H	See Table 2-29
MSR_SMM_FEATURE_CONTROL	

MSR Name and CPUID DisplayFamily_DisplayModel	Location
06_5CH, 06_7AH	See Table 2-12
06_3CH, 06_45H, 06_46H	See Table 2-29
MSR_SMM_MCA_CAP	
06_5CH, 06_7AH	See Table 2-12
06_3CH, 06_45H, 06_46H	See Table 2-29
06_3FH	See Table 2-31
06_56H, 06_4FH	See Table 2-35
06_57H	See Table 2-42
MSR_SMRR_PHYSBASE	
06_0FH, 06_17H	See Table 2-3
MSR_SMRR_PHYSMASK	
06_0FH, 06_17H	See Table 2-3
MSR_SSU_ESCR0	
0FH	See Table 2-44
MSR_TBPU_ESCR0	
0FH	See Table 2-44
MSR_TBPU_ESCR1	
0FH	See Table 2-44
MSR_TC_ESCR0	
0FH	See Table 2-44
MSR_TC_ESCR1	
0FH	See Table 2-44
MSR_TC_PRECISE_EVENT	
0FH	See Table 2-44
MSR_TEMPERATURE_TARGET	
06_37H, 06_4AH, 06_4DH, 06_5AH, 06_5DH, 06_5CH, 06_7AH	See Table 2-6
06_1AH, 06_1EH, 06_1FH, 06_2EH	See Table 2-14
06_2AH, 06_2DH	See Table 2-19
06_3EH	See Table 2-25
06_56H, 06_4FH	See Table 2-35
06_57H	See Table 2-42
MSR_THERM2_CTL	
06_0FH, 06_17H	See Table 2-3
06_1CH, 06_26H, 06_27H, 06_35H, 06_36H	See Table 2-4
0FH	See Table 2-44
06_0EH	See Table 2-47
06_09H	See Table 2-48
MSR_THREAD_ID_INFO	
06_3FH	See Table 2-31
MSR_TURBO_ACTIVATION_RATIO	
06_5CH, 06_7AH	See Table 2-12
06_3AH	See Table 2-24

MSR Name and CPUID DisplayFamily_DisplayModel	Location
06_3CH, 06_45H, 06_46H	See Table 2-28
06_57H	See Table 2-42
MSR_TURBO_GROUP_CORECNT	
06_5CH, 06_7AH	See Table 2-12
MSR_TURBO_POWER_CURRENT_LIMIT	
06_1AH, 06_1EH, 06_1FH, 06_2EH	See Table 2-14
MSR_TURBO_RATIO_LIMIT	
06_37H, 06_4AH, 06_4DH, 06_5AH, 06_5DH, 06_5CH, 06_7AH	See Table 2-6
06_4DH	See Table 2-10
06_5CH, 06_7AH	See Table 2-12
06_1AH, 06_1EH, 06_1FH, 06_2EH, 06_25H, 06_2CH	See Table 2-14
06_1AH, 06_1EH, 06_1FH, 06_25H, 06_2CH	See Table 2-15
06_2EH	See Table 2-16
06_25H, 06_2CH	See Table 2-17
06_2FH	See Table 2-18
06_2AH, 06_45H	See Table 2-20
06_2DH	See Table 2-22
06_3EH	See Table 2-25 and Table 2-26
06_3CH, 06_45H, 06_46H	See Table 2-29
06_3FH	See Table 2-31
06_3DH	See Table 2-34
06_56H, 06_4FH	See Table 2-35
06_55H	See Table 2-41
06_57H	See Table 2-42
MSR_TURBO_RATIO_LIMIT1	
06_3EH	See Table 2-25 and Table 2-26
06_3FH	See Table 2-31
06_56H, 06_4FH	See Table 2-35
MSR_TURBO_RATIO_LIMIT2	
06_3FH	See Table 2-31
MSR_TURBO_RATIO_LIMIT3	
06_56H	See Table 2-36
06_4FH	See Table 2-37
MSR_TURBO_RATIO_LIMIT_CORES	
06_55H	See Table 2-41
MSR_U_PMON_BOX_STATUS	
06_3EH	See Table 2-27
06_3FH	See Table 2-32
MSR_U_PMON_CTR	
06_2EH	See Table 2-16

MSR Name and CPUID DisplayFamily_DisplayModel	Location
MSR_U_PMON_CTR0	
06_2DH	See Table 2-23
06_3FH	See Table 2-32
MSR_U_PMON_CTR1	
06_2DH	See Table 2-23
06_3FH	See Table 2-32
MSR_U_PMON_EVNT_SEL	
06_2EH	See Table 2-16
MSR_U_PMON_EVNTSEL0	
06_2DH	See Table 2-23
06_3FH	See Table 2-32
MSR_U_PMON_EVNTSEL1	
06_2DH	See Table 2-23
06_3FH	See Table 2-32
MSR_U_PMON_GLOBAL_CTRL	
06_2EH	See Table 2-16
MSR_U_PMON_GLOBAL_OVF_CTRL	
06_2EH	See Table 2-16
MSR_U_PMON_GLOBAL_STATUS	
06_2EH	See Table 2-16
MSR_U_PMON_UCLK_FIXED_CTL	
06_2DH	See Table 2-23
06_3FH	See Table 2-32
MSR_U_PMON_UCLK_FIXED_CTR	
06_2DH	See Table 2-23
06_3FH	See Table 2-32
MSR_U2L_ESCR0	
0FH	See Table 2-44
MSR_U2L_ESCR1	
0FH	See Table 2-44
MSR_UNC_ARB_PERFCTR0	
06_2AH	See Table 2-21
06_3CH, 06_45H, 06_46H	See Table 2-29
06_4EH, 06_5EH	See Table 2-39
MSR_UNC_ARB_PERFCTR1	
06_2AH	See Table 2-21
06_3CH, 06_45H, 06_46H	See Table 2-29
06_4EH, 06_5EH	See Table 2-39
MSR_UNC_ARB_PERFEVTSEL0	
06_2AH	See Table 2-21
06_3CH, 06_45H, 06_46H	See Table 2-29
06_4EH, 06_5EH	See Table 2-39

MSR Name and CPUID DisplayFamily_DisplayModel	Location
MSR_UNC_ARB_PERFEVTSEL1	
06_2AH	See Table 2-21
06_3CH, 06_45H, 06_46H	See Table 2-29
06_4EH, 06_5EH	See Table 2-39
MSR_UNC_CBO_0_PERFCTR0	
06_2AH	See Table 2-21
06_3CH, 06_45H, 06_46H	See Table 2-29
06_4EH, 06_5EH	See Table 2-39
MSR_UNC_CBO_0_PERFCTR1	
06_2AH	See Table 2-21
06_3CH, 06_45H, 06_46H	See Table 2-29
06_4EH, 06_5EH	See Table 2-39
MSR_UNC_CBO_0_PERFCTR2	
06_2AH	See Table 2-21
MSR_UNC_CBO_0_PERFCTR3	
06_2AH	See Table 2-21
MSR_UNC_CBO_0_PERFEVTSEL0	
06_2AH	See Table 2-21
06_3CH, 06_45H, 06_46H	See Table 2-29
06_4EH, 06_5EH	See Table 2-39
MSR_UNC_CBO_0_PERFEVTSEL1	
06_2AH	See Table 2-21
06_3CH, 06_45H, 06_46H	See Table 2-29
06_4EH, 06_5EH	See Table 2-39
MSR_UNC_CBO_0_PERFEVTSEL2	
06_2AH	See Table 2-21
MSR_UNC_CBO_0_PERFEVTSEL3	
06_2AH	See Table 2-21
MSR_UNC_CBO_0_UNIT_STATUS	
06_2AH	See Table 2-21
MSR_UNC_CBO_1_PERFCTR0	
06_2AH	See Table 2-21
06_3CH, 06_45H, 06_46H	See Table 2-29
06_4EH, 06_5EH	See Table 2-39
MSR_UNC_CBO_1_PERFCTR1	
06_2AH	See Table 2-21
06_3CH, 06_45H, 06_46H	See Table 2-29
06_4EH, 06_5EH	See Table 2-39
MSR_UNC_CBO_1_PERFCTR2	
06_2AH	See Table 2-21
MSR_UNC_CBO_1_PERFCTR3	
06_2AH	See Table 2-21

MSR Name and CPUID DisplayFamily_DisplayModel	Location
MSR_UNC_CBO_1_PERFEVTSEL0	
06_2AH	See Table 2-21
06_3CH, 06_45H, 06_46H	See Table 2-29
06_4EH, 06_5EH	See Table 2-39
MSR_UNC_CBO_1_PERFEVTSEL1	
06_2AH	See Table 2-21
06_3CH, 06_45H, 06_46H	See Table 2-29
06_4EH, 06_5EH	See Table 2-39
MSR_UNC_CBO_1_PERFEVTSEL2	
06_2AH	See Table 2-21
MSR_UNC_CBO_1_PERFEVTSEL3	
06_2AH	See Table 2-21
MSR_UNC_CBO_1_UNIT_STATUS	
06_2AH	See Table 2-21
MSR_UNC_CBO_2_PERFCTR0	
06_2AH	See Table 2-21
06_3CH, 06_45H, 06_46H	See Table 2-29
06_4EH, 06_5EH	See Table 2-39
MSR_UNC_CBO_2_PERFCTR1	
06_2AH	See Table 2-21
06_3CH, 06_45H, 06_46H	See Table 2-29
06_4EH, 06_5EH	See Table 2-39
MSR_UNC_CBO_2_PERFCTR2	
06_2AH	See Table 2-21
MSR_UNC_CBO_2_PERFCTR3	
06_2AH	See Table 2-21
MSR_UNC_CBO_2_PERFEVTSEL0	
06_2AH	See Table 2-21
06_3CH, 06_45H, 06_46H	See Table 2-29
06_4EH, 06_5EH	See Table 2-39
MSR_UNC_CBO_2_PERFEVTSEL1	
06_2AH	See Table 2-21
06_3CH, 06_45H, 06_46H	See Table 2-29
06_4EH, 06_5EH	See Table 2-39
MSR_UNC_CBO_2_PERFEVTSEL2	
06_2AH	See Table 2-21
MSR_UNC_CBO_2_PERFEVTSEL3	
06_2AH	See Table 2-21
MSR_UNC_CBO_2_UNIT_STATUS	
06_2AH	See Table 2-21
MSR_UNC_CBO_3_PERFCTR0	
06_2AH	See Table 2-21

MSR Name and CPUID DisplayFamily_DisplayModel	Location
06_3CH, 06_45H, 06_46H	See Table 2-29
06_4EH, 06_5EH	See Table 2-39
MSR_UNC_CBO_3_PERFCTR1	
06_2AH	See Table 2-21
06_3CH, 06_45H, 06_46H	See Table 2-29
06_4EH, 06_5EH	See Table 2-39
MSR_UNC_CBO_3_PERFCTR2	
06_2AH	See Table 2-21
MSR_UNC_CBO_3_PERFCTR3	
06_2AH	See Table 2-21
MSR_UNC_CBO_3_PERFEVTSEL0	
06_2AH	See Table 2-21
06_3CH, 06_45H, 06_46H	See Table 2-29
06_4EH, 06_5EH	See Table 2-39
MSR_UNC_CBO_3_PERFEVTSEL1	
06_2AH	See Table 2-21
06_3CH, 06_45H, 06_46H	See Table 2-29
06_4EH, 06_5EH	See Table 2-39
MSR_UNC_CBO_3_PERFEVTSEL2	
06_2AH	See Table 2-21
MSR_UNC_CBO_3_PERFEVTSEL3	
06_2AH	See Table 2-21
MSR_UNC_CBO_3_UNIT_STATUS	
06_2AH	See Table 2-21
MSR_UNC_CBO_4_PERFCTR0	
06_2AH	See Table 2-21
MSR_UNC_CBO_4_PERFCTR1	
06_2AH	See Table 2-21
MSR_UNC_CBO_4_PERFCTR2	
06_2AH	See Table 2-21
MSR_UNC_CBO_4_PERFCTR3	
06_2AH	See Table 2-21
MSR_UNC_CBO_4_PERFEVTSEL0	
06_2AH	See Table 2-21
MSR_UNC_CBO_4_PERFEVTSEL1	
06_2AH	See Table 2-21
MSR_UNC_CBO_4_PERFEVTSEL2	
06_2AH	See Table 2-21
MSR_UNC_CBO_4_PERFEVTSEL3	
06_2AH	See Table 2-21
MSR_UNC_CBO_4_UNIT_STATUS	
06_2AH	See Table 2-21

MSR Name and CPUID DisplayFamily_DisplayModel	Location
MSR_UNC_CBO_CONFIG	
06_2AH	See Table 2-21
06_3CH, 06_45H, 06_46H	See Table 2-29
06_4EH, 06_5EH	See Table 2-39
MSR_UNC_PERF_FIXED_CTR	
06_2AH	See Table 2-21
06_3CH, 06_45H, 06_46H	See Table 2-29
06_4EH, 06_5EH	See Table 2-39
MSR_UNC_PERF_FIXED_CTRL	
06_2AH	See Table 2-21
06_3CH, 06_45H, 06_46H	See Table 2-29
06_4EH, 06_5EH	See Table 2-39
MSR_UNC_PERF_GLOBAL_CTRL	
06_2AH	See Table 2-21
06_3CH, 06_45H, 06_46H	See Table 2-29
06_4EH, 06_5EH	See Table 2-39
MSR_UNC_PERF_GLOBAL_STATUS	
06_2AH	See Table 2-21
06_3CH, 06_45H, 06_46H	See Table 2-29
06_4EH, 06_5EH	See Table 2-39
MSR_UNCORE_ADDR_OPCODE_MATCH	
06_1AH, 06_1EH, 06_1FH, 06_25H, 06_2CH	See Table 2-15
MSR_UNCORE_FIXED_CTR_CTRL	
06_1AH, 06_1EH, 06_1FH, 06_25H, 06_2CH	See Table 2-15
MSR_UNCORE_FIXED_CTR0	
06_1AH, 06_1EH, 06_1FH, 06_25H, 06_2CH	See Table 2-15
MSR_UNCORE_PERF_GLOBAL_CTRL	
06_1AH, 06_1EH, 06_1FH, 06_25H, 06_2CH	See Table 2-15
MSR_UNCORE_PERF_GLOBAL_OVF_CTRL	
06_1AH, 06_1EH, 06_1FH, 06_25H, 06_2CH	See Table 2-15
MSR_UNCORE_PERF_GLOBAL_STATUS	
06_1AH, 06_1EH, 06_1FH, 06_25H, 06_2CH	See Table 2-15
MSR_UNCORE_PERFEVTSEL0	
06_1AH, 06_1EH, 06_1FH, 06_25H, 06_2CH	See Table 2-15
MSR_UNCORE_PERFEVTSEL1	
06_1AH, 06_1EH, 06_1FH, 06_25H, 06_2CH	See Table 2-15
MSR_UNCORE_PERFEVTSEL2	
06_1AH, 06_1EH, 06_1FH, 06_25H, 06_2CH	See Table 2-15
MSR_UNCORE_PERFEVTSEL3	
06_1AH, 06_1EH, 06_1FH, 06_25H, 06_2CH	See Table 2-15
MSR_UNCORE_PERFEVTSEL4	
06_1AH, 06_1EH, 06_1FH, 06_25H, 06_2CH	See Table 2-15

MSR Name and CPUID DisplayFamily_DisplayModel	Location
MSR_UNCORE_PERFEVTSEL5	
06_1AH, 06_1EH, 06_1FH, 06_25H, 06_2CH	See Table 2-15
MSR_UNCORE_PERFEVTSEL6	
06_1AH, 06_1EH, 06_1FH, 06_25H, 06_2CH	See Table 2-15
MSR_UNCORE_PERFEVTSEL7	
06_1AH, 06_1EH, 06_1FH, 06_25H, 06_2CH	See Table 2-15
MSR_UNCORE_PMC0	
06_1AH, 06_1EH, 06_1FH, 06_25H, 06_2CH	See Table 2-15
MSR_UNCORE_PMC1	
06_1AH, 06_1EH, 06_1FH, 06_25H, 06_2CH	See Table 2-15
MSR_UNCORE_PMC2	
06_1AH, 06_1EH, 06_1FH, 06_25H, 06_2CH	See Table 2-15
MSR_UNCORE_PMC3	
06_1AH, 06_1EH, 06_1FH, 06_25H, 06_2CH	See Table 2-15
MSR_UNCORE_PMC4	
06_1AH, 06_1EH, 06_1FH, 06_25H, 06_2CH	See Table 2-15
MSR_UNCORE_PMC5	
06_1AH, 06_1EH, 06_1FH, 06_25H, 06_2CH	See Table 2-15
06_2EH	See Table 2-16
MSR_UNCORE_PMC6	
06_1AH, 06_1EH, 06_1FH, 06_25H, 06_2CH	See Table 2-15
MSR_UNCORE_PMC7	
06_1AH, 06_1EH, 06_1FH, 06_25H, 06_2CH	See Table 2-15
MSR_UNCORE_PRMRR_BASE	
06_4EH, 06_5EH, 06_55H, 06_8EH, 06_9EH, 06_66H.	See Table 2-38
MSR_UNCORE_PRMRR_MASK	
06_4EH, 06_5EH, 06_55H, 06_8EH, 06_9EH, 06_66H.	See Table 2-38
MSR_W_PMON_BOX_CTRL	
06_2EH	See Table 2-16
MSR_W_PMON_BOX_OVF_CTRL	
06_2EH	See Table 2-16
MSR_W_PMON_BOX_STATUS	
06_2EH	See Table 2-16
MSR_W_PMON_CTR0	
06_2EH	See Table 2-16
MSR_W_PMON_CTR1	
06_2EH	See Table 2-16
MSR_W_PMON_CTR2	
06_2EH	See Table 2-16
MSR_W_PMON_CTR3	
06_2EH	See Table 2-16
MSR_W_PMON_EVNT_SEL0	

MSR Name and CPUID DisplayFamily_DisplayModel	Location
P6 Family	See Table 2-49
MTRRfix64K_00000	
06_0EH	See Table 2-47
P6 Family	See Table 2-49
MTRRphysBase0	
06_0EH	See Table 2-47
P6 Family	See Table 2-49
MTRRphysBase1	
06_0EH	See Table 2-47
P6 Family	See Table 2-49
MTRRphysBase2	
06_0EH	See Table 2-47
P6 Family	See Table 2-49
MTRRphysBase3	
06_0EH	See Table 2-47
P6 Family	See Table 2-49
MTRRphysBase4	
06_0EH	See Table 2-47
P6 Family	See Table 2-49
MTRRphysBase5	
06_0EH	See Table 2-47
P6 Family	See Table 2-49
MTRRphysBase6	
06_0EH	See Table 2-47
P6 Family	See Table 2-49
MTRRphysBase7	
06_0EH	See Table 2-47
P6 Family	See Table 2-49
MTRRphysMask0	
06_0EH	See Table 2-47
P6 Family	See Table 2-49
MTRRphysMask1	
06_0EH	See Table 2-47
P6 Family	See Table 2-49
MTRRphysMask2	
06_0EH	See Table 2-47
P6 Family	See Table 2-49
MTRRphysMask3	
06_0EH	See Table 2-47
P6 Family	See Table 2-49
MTRRphysMask4	
06_0EH	See Table 2-47

MSR Name and CPUID DisplayFamily_DisplayModel	Location
P6 Family	See Table 2-49
MTRRphysMask5	
06_0EH	See Table 2-47
P6 Family	See Table 2-49
MTRRphysMask6	
06_0EH	See Table 2-47
P6 Family	See Table 2-49
MTRRphysMask7	
06_0EH	See Table 2-47
P6 Family	See Table 2-49